Improving Neural Architecture Search With Bayesian Optimization and Generalization Mechanisms

Vasco Ferrinho Lopes

Tese para obtenção do Grau de Doutor em
Engenharia Informática
(3º ciclo de estudos)

Orientador: Prof. Doutor Luís Filipe Barbosa de Almeida Alexandre

Covilhã, janeiro de 2024

Composição do Júri

Hugo Proença
Professor Catedrático — Presidente
Universidade da Beira Interior

Jaime S. Cardoso
Professor Catedrático — Arguente
Faculdade de Engenharia da Universidade do Porto

Nuno Lourenço
Professor Auxiliar — Arguente
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

João C. Neves
Professor Auxiliar — Arguente
Universidade da Beira Interior

Luís A. Alexandre
Professor Catedrático — Orientador
Universidade da Beira Interior

Provas realizadas a 21 dezembro 2023 com início às 14:30 horas.

Declaração de Integridade

Eu, Vasco Ferrinho Lopes, que abaixo assino, estudante com o número de inscrição D2667 do 3º Ciclo de Engenharia Informática da Faculdade de Engenharias, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 10/01/2024

Assinado por : **VASCO FERRINHO LOPES**
Num. de Identificação: 15037381
Data: 2024.01.10 17:55:43 +0000

CHAVE MÓVEL

Thesis prepared at NOVA LINCS and Soft Computing and Image Analysis Laboratory (SOCIA Lab), and submitted to Universidade da Beira Interior for defense in a public examination session.

This work has been supported by 'FCT - Fundação para a Ciência e Tecnologia' (Portugal) through the research grant '2020.04588.BD', by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT, through national funds, partially supported by CENTRO-01-0247-FEDER-113023 - DeepNeuronic, and partially supported by project 026653 (POCI-01-0247-FEDER-026653) INDTECH 4.0 – New technologies for smart manufacturing, cofinanced by the Portugal 2020 Program (PT 2020), Compete 2020 Program and the European Union through the European Regional Development Fund (ERDF).

Agradecimentos

The work that led to this results from the support and collaboration of many, to whom I want to take a moment to express my deepest appreciation. Throughout this journey, the people I was lucky to meet, that supported and motivated me, and the ones I had the pleasure to collaborate with have been my greatest strength and source of inspiration.

I would like to start by expressing my deepest gratitude to Professor Luís Alexandre. Throughout this challenging journey, his inexhaustible presence, support, and guidance have been detrimental to shaping the successful completion of my doctoral thesis. It has been an honor to work with Professor Luís and have the opportunity to learn from his vast knowledge and experience, which he is always available and eager to share without any compromises. His work ethic, dedication, consistent commitment, and personal values are all aspects that deeply impacted me and will continue to influence my career and personal growth.

During this thesis, I had an amazing opportunity to spend time at Huawei and Google, where I had the privilege of engaging in insightful research that allowed me to broaden my horizons. Collaborating with such talented professionals and immersing myself in dynamic environments provided invaluable experiences for personal and intellectual growth. I deeply thank Pedro Esperança, Fadhel Ayed, Sarah Parisot, and Hafiz Tiomoko Ali for the work opportunity and a fantastic time at Huawei developing and working with neural networks. At Google, I had a wonderful experience for which I am truly thankful. Jacek, Sofia, and everyone there are incredibly talented and amazing people. They provided me with an environment that I could only have dreamt of. I cannot fully express my appreciation for the opportunity and everything they did during my stay in Zurich. Their attention to detail and continuous quest for the best possible work are a testament to their success and made me a better researcher and programmer.

I cannot fail to mention my gratitude to my colleagues at SOCIALab and NOVALincs, especially the ones I collaborated more with – Saeid, Bruno, João N., Nuno, Tiago, and André, which allowed for a fantastic working place. Also, everyone at DeepNeuronic and HealthtechLisboa community, I can only thank you for the opportunity to embark on different opportunities and meet a world of opportunities. It has been a pleasure to work alongside you and learn so many different things. Having the opportunity to collaborate and learn from all of you has been one of the most enjoyable things of the last few years. A special token of appreciation to Bruno Degardin, who has been an amazing partner and very supportive throughout this journey. His support, creativity, and friendship lead to the creation of amazing work, which I look forward to continuing. Also, I want to thank Saeid and João N. for the discussions and continuous availability for revisions and thought-provoking questions that greatly improved this work.

I must express my profound gratitude to my family. Your unconditional love, support, and constant encouragement have been essential to my personal and academic fulfillment. Your words of encouragement and belief in me have been a driving force. I am who I am because of you and the values you have passed me through hard work and by never giving up. Thank you for everything.

To Diana, my life partner, I want to express my heartfelt appreciation for your unwavering support and motivation. You have been a constant source of strength and encouragement throughout this journey. Your belief in me has helped me overcome many challenges. Thank you for always being there by my side, cheering me on, your patience and tireless support. I am truly grateful for your presence in my life.

To all my friends who directly or indirectly contributed to this work and have been by my side throughout this journey. Your insights, discussions, and support have been fundamental to my academic and personal growth. Sharing this path with all of you has been truly enriching, and I am immensely thankful for the opportunity to learn from each of you.

List of Publications

During the course of this work, several research articles were written and submitted to publication in both conferences and international journals. More, during the PhD there was also the opportunity to collaborate in different research projects that lead to publications. This section lists both the publications that are included in this thesis and the ones resulting from collaborations and are that are not directly within the PhD scope.

Publications included in the thesis resulting from this doctoral research program:

1. Auto-Classifier: A Robust Defect Detector Based on an AutoML Head. **Vasco Lopes**, Luís A. Alexandre. In 27th International Conference on Neural Information Processing (ICONIP), 2020.

2. An AutoML-based Approach to Multimodal Image Sentiment Analysis. **Vasco Lopes**, António Gaspar, Luís A. Alexandre, João Cordeiro. In International Joint Conference on Neural Networks (IJCNN), 2021.

3. EPE-NAS: Efficient Performance Estimation Without Training for Neural Architecture Search. **Vasco Lopes**, Saeid Alirezazadeh, Luís A. Alexandre. In 30th International Conference on Artificial Neural Networks (ICANN), 2021. **Awarded 1st Springer & ENNS Best Paper Award**.

4. Guided Evolution for Neural Architecture Search. **Vasco Lopes**, Miguel Santos, Bruno Degardin, Luís A. Alexandre. In Advances in Neural Information Processing Systems 35 (NeurIPS) - New In ML, 2021.

5. Efficient Guided Evolution for Neural Architecture Search. **Vasco Lopes**, Miguel Santos, Bruno Degardin, Luís A. Alexandre. In The Genetic and Evolutionary Computation Conference (GECCO), 2022.

6. MANAS: Multi-agent Neural Architecture Search. **Vasco Lopes**, Fabio Maria Carlucci, Pedro M. Esperança, Marco Singh, Victor Gabillon, Antoine Yang, Hang Xu, Zewei Chen, Jun Wang. Springer Machine Learning, 2023.

7. Guided Evolutionary Neural Architecture Search With Efficient Performance Estimation. **Vasco Lopes**, Miguel Santos, Bruno Degardin, Luís A. Alexandre. In review at a journal.

8. Towards Less Constrained Macro-Neural Architecture Search. **Vasco Lopes**, Luís A. Alexandre. In second review stage at a journal.

9. Are Neural Architecture Search Benchmarks Well Designed? A Deeper Look Into Operation Importance. **Vasco Lopes**, Bruno Degardin, Luís A. Alexandre. In second review stage at a journal.

10. Designing Downstream Architectures with Neural Tangent Kernel and Large Feature Extractors. **Vasco Lopes**, Luís A. Alexandre. Submitted.

Other publications resulting from the doctoral research program that are not included in the thesis:

1. GANprintR: Improved Fakes and Evaluation of the State of the Art in Face Manipulation Detection. João C. Neves, Rubén Tolosana, Rubén Vera-Rodríguez, **Vasco Lopes**, Hugo Proença, Julian Fiérrez. IEEE Journal of Selected Topics in Signal Processing 14(5):1038-1048, 2020.
2. Node and Network Entropy–A Novel Mathematical Model for Pattern Analysis of Team Sports Behavior. Fernando M. L. Martins, Ricardo Gomes, **Vasco Lopes**, Frutuoso G. M. Silva, Rui Mendes. Mathematics 8(9):1543, 2020.
3. A Hybrid Method for Training Convolutional Neural Networks. **Vasco Lopes**, Paulo A. P. Fazendeiro. In Computing Conference, Intelligent Computing SAI (3):298-308, 2021.
4. Real-time 2D–3D door detection and state classification on a low-power device. João G. Ramôa, **Vasco Lopes**, Luís A. Alexandre, Sandra Mogo. SN Applied Sciences 3, 590, 2021.
5. REGINA–Reasoning Graph Convolutional Networks in Human Action Recognition. Bruno Degardin, **Vasco Lopes**, Hugo Proença. IEEE Transactions on Information Forensics and Security 16:5442-5451, 2021.
6. Mathematical Models to Measure the Variability of Nodes and Networks in Team Sports. Fernando M. L. Martins, Ricardo Gomes, **Vasco Lopes**, Frutuoso G. M. Silva, Rui Mendes. Entropy 23(8): 1072, 2021.
7. Generative Adversarial Graph Convolutional Networks for Human Action Synthesis. Bruno Degardin, João C. Neves, **Vasco Lopes**, João Brito, Ehsan Yaghoubi, Hugo Proença. In Winter Conference on Applications of Computer Vision (WACV), 2022.
8. GAN Fingerprints in Face Image Synthesis. João C. Neves, Rubén Tolosana, Rubén Vera-Rodríguez, **Vasco Lopes**, Hugo Proença, Julian Fiérrez. Multimedia Forensics, 175-204, 2022.
9. ATOM: Self-supervised human action recognition using atomic motion representation learning. Bruno Degardin, **Vasco Lopes**, Hugo Proença. Image and Vision Computing 137, 2023.
10. Fake It Till You Recognize It: Quality Assessment for Human Action Generative Models. Bruno Degardin, **Vasco Lopes**, Hugo Proença. Under review.

Resumo

Os avanços nos domínios da Inteligência Artificial (IA) e da Aprendizagem Automática (AA) permitiram obter resultados impressionantes em vários problemas. Estes avanços podem ser atribuídos, em grande medida, aos algoritmos de aprendizagem profunda, especialmente às Redes Neuronais Convolucionais (RNCs). O sucesso crescente das RNCs deve-se principalmente ao engenho e aos esforços de engenharia de especialistas que conceberam e otimizaram arquiteturas de redes neuronais poderosas, obtendo resultados sem precedentes numa vasta panóplia de tarefas. No entanto, a aplicação de um método de AA a um problema para o qual não foi especifícamente concebido traduz-se normalmente em resultados sub-óptimos, que, em casos extremos, pode levar a desempenhos medíocres, dificultando assim a sustentabilidade dos sistemas e a massificação da sua utilização por não-especialistas. A conceção de RNCs para problemas específicos é uma tarefa difícil, uma vez que muitas escolhas de conceção das redes não são independentes umas das outras. Assim, tornou-se imperativo automatizar este processo através da conceção e desenvolvimento de métodos de Pesquisa Automática de Arquiteturas (PAA).

As arquiteturas encontradas com base em métodos de PAA alcançaram desempenhos notáveis em várias tarefas, superando as redes concebidas por humanos. No entanto, os métodos de PAA ainda enfrentam vários problemas. A maioria destes métodos depende fortemente de pressupostos definidos por humanos que restringem a pesquisa, como a estrutura da arquitetura, o número de camadas, heurísticas para a definição de parâmetros e os espaços de pesquisa. Os espaços de pesquisa mais comuns consistem em módulos repetíveis (células), em vez de explorarem totalmente o espaço de pesquisa da arquitetura através da conceção de arquiteturas completas (pesquisa macro), necessitando assim de conhecimento humano, restringindo a pesquisa a definições pré-definidas e limitando a exploração de arquiteturas novas e diversas, devido à existência de regras pré-definidas. Mais ainda, a necessidade de grandes capacidades de computação é ainda inerente à maioria dos métodos PAA e apenas alguns podem efetuar uma pesquisa macro.

No plano desta tese, o objetivo principal foi desenvolver novas soluções para mitigar os problemas mencionados anteriormente. Em primeiro lugar, apresentamos uma análise exaustiva dos componentes, métodos e *benchmarks* de PAA. Para este último, realizamos um estudo sobre a importância das diferentes operações, avaliando a influência que o conjunto de operações dos espaços de pesquisa tem sobre o desempenho das arquiteturas geradas. De seguida, estudámos o comportamento de várias redes neuronais em diferentes problemas de classificação e propusemos dois novos métodos para melhorar as redes neuronais existentes com PAA: i) procurando um novo componente de classificação e ii) procurando um método de fusão que permita efetuar uma classificação multimodal. De seguida, procurámos melhorar o custo de pesquisa dos métodos de PAA, propondo uma estratégia de estimativa que classifica as arquiteturas na fase de inicialização através da análise da matriz Jacobiana e uma pesquisa evolutiva que gera arquiteturas

com base em operações de mutação, tirando partido da estimativa de custo zero para orientar eficazmente o processo de pesquisa. Para melhorar ainda mais as capacidades dos métodos de PAA, estendemos o estudo sobre arquiteturas em fase de inicialização, propondo um segundo método de custo zero, que analisa o *Neural Tangent Kernel* de uma arquitetura gerada para inferir o seu desempenho final caso esta seja treinada. Propusemos também um novo espaço de pesquisa que aproveita extratores de características pré-treinados (RNCs) e força a pesquisa apenas para uma pequena arquitetura que aprende uma nova tarefa baseado nas informações geradas pelas redes maiores. Com estes dois métodos, mostrámos que redes de grande dimensão podem ser eficientemente aproveitadas para aprender novas tarefas sem necessidade de qualquer afinação ou de grandes recursos computacionais. De forma a aprimorar os custos de pesquisa e de memória dos métodos de PAA, propusemos o MANAS. Este método formula o problema de otimização de PAA como um problema de multi-agentes. Este utiliza agentes independentes que procuram operações de forma distribuída, repartindo assim o espaço de pesquisa. Com o MANAS, mostrámos que tanto o custo de pesquisa como os recursos de memória podem ser significativamente reduzidos, melhorando em simultâneo, o desempenho final. Por fim, para levar a área de PAA a espaços e configurações de pesquisa menos restritos, propusemos o LCMNAS, um método de PAA que efetua uma macro-pesquisa sem depender de heurísticas predefinidas ou de espaços de pesquisa restritivos. O LCMNAS introduz três componentes para o melhorar a otimização de PAA: i) um método que aproveita as informações sobre arquiteturas existentes para autonomamente gerar espaços de pesquisa complexos com base em grafos pesados com parâmetros ocultos, ii) uma estratégia de pesquisa evolutiva que gera arquiteturas completas a partir do zero e iii) uma abordagem de estimação de desempenho misto que combina informações sobre arquiteturas na fase de inicialização e estimativas de fidelidade reduzida para inferir a capacidade de treino de uma arquitetura, assim como a sua capacidade de modelar funções complexas.

Os resultados obtidos pelos métodos propostos mostram que é possível melhorar os métodos de PAA no que respeita aos custos de pesquisa e de memória, bem como em termos de computação necessária, obtendo resultados competitivos. Todos os métodos propostos foram avaliados em vários espaços de pesquisa e em vários conjuntos de dados, mostrando desempenhos elevados, requerendo ao mesmo tempo apenas uma fração do tempo e das necessidades de computação dos métodos de PAA anteriores.

Palavras-chave

Pesquisa Automática de Arquitecturas, Automatização de Aprendizagem Automática, Aprendizagem Automática, Visão Computacional, Aprendizagem Profunda, Redes Neuronais Convolucionais, Otimização de Parâmetros.

Resumo Alargado

Os avanços nos domínios da Inteligência Artificial (IA) e da Aprendizagem Automática (AA) permitiram obter resultados impressionantes em vários problemas. Estes avanços podem ser atribuídos, em grande medida, aos algoritmos de aprendizagem profunda, especialmente às Redes Neuronais Convolucionais (RNCs). O sucesso crescente das RNCs deve-se principalmente ao engenho e aos esforços de engenharia de especialistas que conceberam e otimizaram arquiteturas de redes neuronais poderosas, obtendo resultados sem precedentes numa vasta panóplia de tarefas. No entanto, a aplicação de um método de AA a um problema para o qual não foi especifícamente concebido traduz-se normalmente em resultados sub-óptimos, que, em casos extremos, pode levar a desempenhos medíocres, dificultando assim a sustentabilidade dos sistemas e a massificação da sua utilização por não-especialistas. A conceção de RNCs para problemas específicos é uma tarefa difícil, uma vez que muitas escolhas de conceção das redes não são independentes umas das outras. Assim, tornou-se imperativo automatizar este processo através da conceção e desenvolvimento de métodos de Pesquisa Automática de Arquiteturas (PAA).

As arquiteturas encontradas com base em métodos de PAA alcançaram desempenhos notáveis em várias tarefas, superando as redes concebidas por humanos. No entanto, os métodos de PAA ainda enfrentam vários problemas. A maioria destes métodos depende fortemente de pressupostos definidos por humanos que restringem a pesquisa, como a estrutura da arquitetura, o número de camadas, heurísticas para a definição de parâmetros e os espaços de pesquisa. Os espaços de pesquisa mais comuns consistem em módulos repetíveis (células), em vez de explorarem totalmente o espaço de pesquisa da arquitetura através da conceção de arquiteturas completas (pesquisa macro), necessitando assim de conhecimento humano, restringindo a pesquisa a definições pré-definidas e limitando a exploração de arquiteturas novas e diversas, devido à existência de regras pré-definidas. Mais ainda, a necessidade de grandes capacidades de computação é ainda inerente à maioria dos métodos PAA e apenas alguns podem efetuar uma pesquisa macro. De forma a mitigar os problemas mencionados, esta tese apresenta várias contribuições descritas ao longo de oito capítulos.

O primeiro capítulo define o âmbito e o problema na qual esta tese se enquadra. Além disso, são também descritos os principais problemas de métodos de PAA, assim como os objetivos do presente trabalho de investigação. Por fim, são também apresentados os principais avanços tecnológicos fruto deste trabalho que contribuíram para uma melhoria dos domínios de AA e PAA.

No segundo capítulo apresentámos uma revisão abrangente dos conceitos que compõem um método de PAA e como os *benchmarks* podem ser um passo importante para se efetuarem avaliações comparativas de forma justa. Primeiro, introduzimos o problema de conceber RNCs e como os métodos de PAA automatizam esse processo com base em três

componentes: o espaço de pesquisa, a estratégia de pesquisa e a estratégia de estimativa de desempenho. Para cada um deles, analisamos diferentes métodos, as suas vantagens e como essas propostas influenciam um método de PAA. Baseado nisso, mostramos que os métodos de PAA são ainda dependentes de diversas escolhas de desenho, afetando em grande escala o desempenho final e a computação necessária para efetuar uma pesquisa por redes neuronais. Efetuar comparações justas é extremamente difícil, pois diferentes métodos têm diferentes protocolos de treino e espaços de pesquisa. Deste modo, apresentamos também uma revisão sobre *benchmarks* de PAA e analisamos detalhadamente os três mais populares de forma a analisar o impacto das operações no resultado das redes geradas. Com isto, descobrimos que as camadas convolucionais são essenciais para se gerar arquiteturas com elevado desempenho e descobrimos também que a distribuição dos intervalos de desempenho é enviesada, sugerindo que nestes *benchmarks* não é difícil encontrar arquiteturas com precisões mais próximas do limite superior, uma vez que vários padrões, como o seu posicionamento de operações numa célula e as combinações de certas operações tendem a produzir arquiteturas com desempenhos competitivos.

O terceiro capítulo é inicialmente dedicado ao estudo do funcionamento e do comportamento de diferentes redes neuronais na tarefa de deteção de defeitos de superfície e na análise multimodal de sentimentos, com base em texto e imagem. Com base nos resultados obtidos através da avaliação de várias redes neuronais, propomos dois métodos para a deteção de defeitos: CNN-Fusion, que funde as classificações de várias RNCs apenas numa classificação final, e o Auto-Classifier, que aproveita as capacidades de extração de características de uma RNC e complementa-as com base numa pesquisa automática por um novo componente de classificação. Para a análise de sentimentos de forma multimodal propomos um método que realiza uma análise individual do texto e de imagens, fundindo essas classificações com base num componente de fusão, automaticamente gerado utilizando mecanismos de PAA. Os resultados obtidos pelos três métodos propostos mostram que o aprimoramento de RNCs com PAA pode melhorar ainda mais as suas capacidades de classificação.

O quarto capítulo apresenta dois métodos: EPE-NAS, uma estratégia de estimativa de desempenho que avalia arquiteturas em fase de inicialização de forma a obter uma correlação com o seu desempenho, caso sejam treinadas, e um algoritmo evolucionário que aproveita o primeiro método para obter informações sobre o espaço de pesquisa de forma rápida para orientar a pesquisa eficientemente. Primeiro, mostramos que, ao aproveitar as informações sobre os gradientes na camada de saída de uma arquitetura em relação aos dados de entrada, um método capaz de estimar o desempenho pode inferir com precisão se a arquitetura gerada é boa em menos de um segundo, sendo capaz de avaliar milhares de arquiteturas em questão de minutos. De seguida, apresentamos a estratégia evolutiva que força uma pesquisa com base nas arquiteturas com melhor desempenho através da geração de descendentes, e uma exploração do espaço de pesquisa através da realização de mutações. O método proposto guia a evolução gerando várias arquiteturas e avaliando-

as na fase de inicialização utilizando o método de estimativa de desempenho. Após esta avaliação inicial, apenas a arquitetura com melhor classificação é treinada e mantida para a geração seguinte. A criação de múltiplas arquiteturas a partir de uma arquitetura existente na população em cada geração permite uma extração constante de conhecimento sobre o espaço de pesquisa sem comprometer o desempenho. Os resultados de ambos os métodos mostram que podem obter resultados excelentes, sendo extremamente eficientes relativamente ao custo da pesquisa. Além disso, neste capítulo, mostramos que usando uma simples pesquisa aleatória acoplada à estratégia de estimativa proposta, é possível gerar arquiteturas de alto desempenho em segundos, capazes de superar muitos dos métodos de PAA atuais.

No quinto capítulo, apresentamos duas abordagens eshorar os métodos de PAA. A primeira é um espaço de pesquisa que utiliza grandes redes neuronais como extratores de características. Isto permite aproveitar o poder de modelos pré-treinados, focando a pesquisa apenas numa pequena arquitetura que aprende a resolver uma nova tarefa. O segundo, é um novo mecanismo de estimativa de desempenho inspirado na utilização de *Neural Tangent Kernels* (NTK) e no alinhamento entre vetores próprios para avaliar arquiteturas em fase de inicialização. Ao incorporar grandes redes neuronais como extratores de características, o espaço de pesquisa proposto beneficia das capacidades eficientes e robustas de representação de dados destas redes, permitindo o desenho de pequenas arquiteturas que aprendem novas tarefas com base nas representações geradas. Além disso, a arquitetura gerada centra-se na aprendizagem de uma tarefa utilizando as representações criadas pelas redes neuronais de grande dimensão, eliminando a necessidade de as voltar a treinar ou afinar em conjuntos de dados mais pequenos. Quanto ao método de estimativa de desempenho, que avalia as arquiteturas com base no seu NTK e tirando partido do alinhamento entre os vetores próprios, o método proposto proporciona uma forma rápida e eficiente de avaliar arquiteturas sem necessidade de qualquer treino. Os resultados dos vários testes efetuados validam a eficácia dos métodos propostos. Mais ainda, o estudo da capacidade de generalização, utilizando diferentes espaços de pesquisa, com diferentes métodos de pesquisa e diferentes mecanismos de estimativa de desempenho, mostram o desempenho melhorado das arquiteturas geradas, onde estas obtêm resultados competitivos, sendo capazes de melhorar os vários métodos de PAA avaliados.

No sexto capítulo formulamos a PAA como um problema de otimização por múltiplos agentes e propomos o método MANAS, uma ferramenta de aprendizagem multi-agente e online. O método proposto tem duas implementações diferentes, sendo estas computacionalmente leves, necessitando de pouca memória gráfica. As implementações propostas diferem no que diz respeito à criação de arquiteturas e à atribuição de recompensas durante o treino. Ao formular PAA como um problema de otimização com base em múltiplos agentes, o método proposto é capaz de superar o estado da arte, reduzindo o consumo de memória numa ordem de grandeza, ao mesmo tempo que obtém excelentes desempenhos

em termos de precisão e de tempo de pesquisa. Além disso, neste capítulo, propomos a utilização de três conjuntos de dados que já foram exaustivamente utilizados em problemas de visão computacional, mas não em NAS: Sport-8, Caltech-101 e MIT-67. Nestes, mostramos empiricamente a eficácia do método proposto e avaliamos o desempenho de diferentes métodos de NAS, tais como pesquisa aleatória. Por fim, este capítulo, ao avaliar múltiplos métodos de NAS, corrobora as preocupações levantadas em trabalhos recentes que afirmam que alguns dos algoritmos de PAA mais utilizados obtêm frequentemente pequenos ganhos comparativamente a uma pesquisa aleatória de arquiteturas.

O sétimo capítulo propõe uma nova abordagem de PAA, capaz de efetuar macro e micro-pesquisa sem restrições. Para tal, concebemos três novos componentes para o método de PAA. Para o desenho do espaço de pesquisa, propomos um método capaz de gerar autonomamente espaços de pesquisa complexos através da criação de grafos pesados com propriedades ocultas a partir de RNCs existentes. A estratégia de pesquisa proposta efetua micro e macro-pesquisa por arquiteturas por meio de evolução sem necessitar de restrições definidas pelos utilizadores, como o esqueleto das arquiteturas, definições de arquiteturas iniciais ou heurísticas. De forma a avaliar rapidamente as arquiteturas geradas, propomos a utilização de uma estratégia de desempenho misto que combina informações sobre arquiteturas em fase de inicialização com o seu desempenho em um conjunto de validação após um treino parcial em um pequeno conjunto de dados. Os testes efetuados mostram que o método proposto é capaz de gerar excelentes arquiteturas na pesquisa baseada em células, superando os métodos atuais em onze conjuntos de dados diferentes, e em três conjuntos de dados quando efetuado uma macro-pesquisa, onde o método consegue gerar arquiteturas a partir do zero com um requisito mínimo de computação gráfica, alcançando erros de teste de 2,96% no CIFAR-10, 20,94% no CIFAR-100 e 43,35% no ImageNet16-120. Além disso, neste capítulo é também estudada a importância de diferentes componentes que formam um método de PAA, permitindo extrair conclusões sobre a escolha de desenho de arquiteturas. Os métodos propostos neste capítulo servem o objetivo de expandir as fronteiras de PAA para espaços menos restritos, onde a experiência humana para a conceção de estruturas para arquiteturas e espaços de pesquisa é reduzida, ao mesmo tempo que é capaz de gerar arquiteturas de uma forma eficiente, permitindo assim um passo em direção à utilização generalizada de métodos de PAA para diferentes problemas e conjuntos de dados.

Por último, os principais resultados deste trabalho de investigação são resumidos no capítulo oito. Além disso, são também apontados, com base no trabalho desenvolvido e nas contribuições desta tese, aqueles que acreditamos ser os passos futuros mais interessantes para a área de PAA.

Abstract

Advances in Artificial Intelligence (AI) and Machine Learning (ML) obtained impressive breakthroughs and remarkable results in various problems. These advances can be largely attributed to deep learning algorithms, especially Convolutional Neural Networks (CNNs). The ever-growing success of CNNs is mainly due to the ingenuity and engineering efforts of human experts who have designed and optimized powerful neural network architectures, which obtained unprecedented results in a vast panoply of tasks. However, applying a ML method to a problem for which it has not been explicitly tailor-made usually leads to sub-optimal results, which in extreme cases can even lead to poor performances, thus hindering the sustainability of a system and the wide-spread application of ML by non-experts. Designing tailor-made CNNs for specific problems is a difficult task, as many design choices depend on each other. Thus, it became logical to automate this process by designing and developing automated Neural Architecture Search (NAS) methods.

Architectures found with NAS achieve state-of-the-art performance in various tasks, outperforming human-designed networks. However, NAS methods still face several problems. Most heavily rely on human-defined assumptions constraining the search, such as the architecture's outer-skeletons, number of layers, parameter heuristics, and search spaces. Common search spaces consist of repeatable modules (cells) instead of fully exploring the architecture's search space by designing entire architectures (macro-search), which requires deep human expertise and restricts the search to pre-defined settings and narrows the exploration of new and diverse architectures by having forced rules. Also, considerable computation is still inherent to most NAS methods, and only a few can perform macro-search.

In this thesis, we focused on proposing novel solutions to mitigate the problems mentioned above. First, we provide a comprehensive review of NAS components, methods, and benchmarks. For the latter, we conduct a study on operation importance to evaluate how the operation pool of search spaces influences the performance of generated architectures. Following, we studied how different neural networks behave for different classification problems and proposed two novel methods to improve upon existing neural networks with NAS by i) searching for a new classification head and ii) searching for a fusion method that allows performing multimodal classification. We then looked into improving the search cost of NAS methods by proposing a zero-proxy estimation strategy that scores architectures at initialization stage through the analysis of the Jacobian matrix and an evolutionary strategy that generates architectures by performing operation mutation and by leveraging the zero-cost proxy estimation to efficiently guide the search process. To further improve the capabilities of NAS methods, we extend the analysis of architectures at initialization stage by proposing a second zero-cost proxy method, which looks at the Neural Tangent Kernel of a generated architecture to infer its final performance if trained. With this, we also propose a novel search space that leverages large pre-trained

feature extractors (CNNs) and forces the search only to a small middleware architecture that learns a downstream task. These two methods showed that large models can be efficiently leveraged to learn new tasks without requiring any fine-tuning or extensive computational resources. To further improve the search and memory costs of NAS methods, we proposed MANAS. This method frames NAS as a multi-agent optimization problem and uses independent agents that search for operations in a distributed manner. With MANAS, we showed that both the search cost and the memory resources can be heavily reduced while improving the final performance. Finally, to push NAS to less constrained search spaces and settings, we proposed LCMNAS, a NAS method that performs macro-search without relying on pre-defined heuristics or bounded search spaces. LCMNAS introduces three components for the NAS pipeline: i) a method that leverages information about well-known architectures to autonomously generate complex search spaces based on weighted directed graphs with hidden properties, ii) an evolutionary search strategy that generates complete architectures from scratch, and iii) a mixed-performance estimation approach that combines information about architectures at initialization stage and lower fidelity estimates to infer their trainability and capacity to model complex functions.

Results obtained by the proposed methods show that it is possible to improve NAS methods regarding search and memory costs, as well as computation requirements, while still obtaining state-of-the-art results. All proposed methods were evaluated in multiple search spaces and several data sets, showing improved performances while requiring only a fraction of previous NAS methods' time and computation needs.

Keywords

Neural Architecture Search, Automated Machine Learning, Machine Learning, Computer Vision, Deep Learning, Convolutional Neural Networks, Parameter Optimization.

Contents

1 Introduction **1**
1.1 Problem Definition and Research Objectives 1
1.2 Main Contributions 5
1.3 Thesis Outline 7

2 Related Work **9**
2.1 Introduction 9
2.2 Neural Architecture Search 10
2.2.1 Search Space 12
2.2.2 Search Strategy 16
2.2.3 Performance Estimation Strategy 20
2.3 Neural Architecture Search Benchmarks 22
2.3.1 Overview 23
2.3.2 Studied Benchmark's Design 26
2.3.3 Evaluation 28
2.3.4 Best Practice Suggestions 40
2.4 Conclusions 45

3 Improving Convolutional Neural Networks Through Method Fusion **47**
3.1 Introduction 47
3.2 Proposed Methods 47
3.2.1 Deep Neural Networks 48
3.2.2 Convolutional Neural Networks Fusion 52
3.2.3 Auto-Classifier 52
3.2.4 Auto-Fusion 54
3.3 Experiments 55
3.3.1 Data Sets 56
3.3.2 Results and Discussion on Detecting Defects 56
3.3.3 Results and Discussion on Multimodal Sentiment Analysis 59
3.4 Conclusions 62

4 Guided Neural Architecture Search Through Efficient Performance Estimation **65**
4.1 Introduction 65
4.2 Proposed Method 67

4.2.1 Zero-cost Proxy Performance Estimation . . . 67
4.2.2 Guided Evolution . . . 68
4.3 Experiments . . . 70
4.3.1 Results and Discussion . . . 71
4.3.2 Ablation Studies . . . 75
4.4 Conclusions . . . 78

5 Designing Architectures with Neural Tangent Kernel and Large Feature Extractors 81
5.1 Introduction . . . 81
5.2 Proposed method . . . 82
5.2.1 Performance Estimation Mechanism . . . 82
5.2.2 Leveraging Large Feature Extractors Search Space . . . 84
5.3 Experiments . . . 85
5.3.1 Search Spaces . . . 85
5.3.2 Implementation Details . . . 86
5.3.3 Results and Discussion On Performance Estimation . . . 87
5.3.4 Results and Discussion On Combining Large Vision Classifiers . . . 88
5.4 Conclusions . . . 91

6 Neural Architecture Search as a Multi-Agent Problem 93
6.1 Introduction . . . 93
6.2 Proposed Method . . . 95
6.2.1 Preliminary: Neural Architecture Search Cell Search . . . 95
6.2.2 Online Multi-agent Learning . . . 95
6.2.3 Adversarial Implementations . . . 99
6.3 Experiments . . . 101
6.3.1 Search Space and Search Protocols . . . 102
6.3.2 Data Sets . . . 102
6.3.3 Results and Discussion . . . 103
6.3.4 Ablation Studies . . . 105
6.4 Conclusions . . . 107

7 Towards Less Constrained Macro Neural Architecture Search 109
7.1 Introduction . . . 109
7.2 Proposed Method . . . 111
7.2.1 Search Space . . . 111
7.2.2 Search Strategy . . . 113
7.2.3 Performance Estimation Mechanism . . . 113
7.3 Designing Entire Architectures: Experiments and Results Analysis . . . 114
7.3.1 Data Sets . . . 114
7.3.2 Final Training . . . 115
7.3.3 Search Space . . . 116

7.3.4 Results and Discussion 116
7.3.5 Ablation Studies 120
7.4 Designing Cells: Experiments and Results Analysis 122
7.4.1 Results and Discussion 122
7.4.2 Ablation Studies 124
7.5 Conclusions 126

8 Conclusion **129**
8.1 Conclusion 129
8.2 Future Directions 131

Bibliografia **135**

List of Figures

1.1 Comparison between traditional ML workflow and the workflow of an AutoML method. 1

1.2 Main fields of research within Automated Machine Learning (AutoML). . . 2

2.1 Examples of some of the most influential neural networks for CV applications and associated main contributions. 10

2.2 General flow of a Neural Architecture Search (NAS) approach, where a controller generates an architecture A from a predefined space of possible architectures, $\mathcal{A}$. The generated architecture is then evaluated, and its performance is used as a reward to update the controller. In this thesis, we propose several methods for all three NAS components: search space, search strategy and the performance estimation strategy. 11

2.3 Main topics of research and development under each NAS component. . . 12

2.4 Representation of the NAS-Bench-201 and TransNAS-Bench-201 cell structure (a), and the pre-defined outer-skeleton (b). A cell is composed of 4 nodes and 6 edges, where edges perform operations from an input node and add them to a posterior node. There are 5 possible operations to be used as edges in NAS-Bench-201 and 4 in TransNAS-Bench-101. A cell is used as the building block of the outer-skeleton. 26

2.5 Normalized histogram of all architectures, based on their validation accuracy (%) in NAS-Bench-101. 27

2.6 Normalized histogram of all architectures, based on their validation accuracy (%), for all the data sets in NAS-Bench-201. 28

2.7 Normalized histogram of all architectures, based on their performance on all the data sets in TransNAS-Bench-101. Metrics were scaled to positive values to match accuracy ranges. 28

2.8 Mean accuracy (%) of all the architectures that have at least one operation of a given type in NAS-Bench-101. 29

2.9 Mean accuracy (%) of all the architectures that have at least one operation of a given type in NAS-Bench-201. Architectures with at least one convolutional layer show higher mean validation accuracies (%) in all data sets. 30

2.10 Mean performance of all the architectures with at least one operation of a given type in TransNAS-Bench-101. Architectures with at least one convolutional layer show higher mean performance in all data sets except in Autoencoding, where SC has the best mean performance. 30

2.11 Boxplot analysis of the mean accuracy (%) of all architectures based on the number of occurrences of the different operations on a cell in NAS-Bench-101. 31

2.12 Boxplot analysis of the mean accuracy (%) of all architectures based on the number of occurrences of the different operations on a cell in NAS-Bench-201. The top figure presents results on CIFAR10, the middle one on C100 and the bottom one on ImageNet16-120. 31

2.13 Boxplot analysis of the mean performance of all architectures based on the number of occurrences of the different operations on a cell in TransNAS-Bench-101. Tasks from top to bottom: object classification, scene classification, autoencoding, surface normal, semantic segmentation, room layout, and jigsaw. 33

2.14 Evaluation of NAS-Bench-201 architectures regarding their inter-data set ranking and the correlation between data sets. On (a) we show the ranking of the top-50 architectures from CIFAR-10 when transferred and evaluated to CIFAR-100 and ImageNet16-120; and (b) Kendall's Tau correlation between data sets. 43

2.15 Evaluation of TransNAS-Bench-101 architectures regarding their inter-task ranking and task correlation. On (a) we show the ranking of the top-50 architectures from object classification when transferred and evaluated to other tasks; and (b) Kendall's Tau correlation between tasks. 43

3.1 Visual representation of the difference between a CNN and the Auto-Classifier method. A CNN is composed of two components: Feature Extraction and Classification. In the Auto-Classifier, the classification component has been replaced by another one, represented by a gradient boosting machine. 53

3.2 Proposed multimodal architecture. The first container represents the preprocessing component that removes noise and non-important information. The second container shows both classification components, where the image and the text are individually classified using CNNs. The third container represents the fusion method that receives the concatenation of the individual classifications and performs a final classification using the model searched – represented by a gradient boosting machine. 54

3.3 Examples of defects in each DAGM2007 data set. 55

3.4 Examples of images and the correspondent texts of the three different classes in B-T4SA. (a) shows a negative example; (b) a neutral one; and (c) a positive example. 56

4.1 Example of scoring two different architectures using the same input. Generated architectures in each generation are ranked based on a score that correlates with their final performance, which determines which architecture is selected to be part of the population. 66
4.2 Example of mutating one operation of a cell using the NAS-Bench-201 search space: operation Identity becomes Conv. 3×3. 70
4.3 Mean accuracy and standard deviation over 25 runs of the proposed search method, GEA, and direct comparison with REA for different cycles (c) across CIFAR-10, CIFAR-100, and ImageNet16-120 data sets. 71
4.4 Mean validation accuracy (%) throughout the evolution for different parent sampling, s, schemes using NAS-Bench-201 CIFAR-10 for 5 runs. 75
4.5 Mean validation accuracy (%) throughout the evolution for different population sizes, p, using NAS-Bench-201 CIFAR-10 for 5 runs. 76
4.6 Mean validation accuracy (%) throughout the evolution for different regularization schemes using NAS-Bench-201 CIFAR-10 for 5 runs. 76
4.7 Comparison of the time, in seconds, required to score a single architecture using our proposed method (in blue) against NAS-WOT (in orange) for different image sizes (x-axis). The image size represents the image's width and height, with 3 channels (RGB). 78

5.1 Illustration of the proposed search space, where pre-trained models serve as feature extractors and a *middleware* architecture is searched to learn downstream tasks. 85

6.1 MANAS with single cell. Between each pair of nodes, an agent $\mathcal{A}_i$ selects action $a^{(i)}$ according to $\pi^{(i)}$. Feedback from the validation loss is used to update the policy. 96
6.2 Comparing MANAS, RS, RS with weight-sharing [1] and DARTS [2] on 8 and 14 cells. Average results of 8 runs. Note that DARTS was only optimized for 8 cells due to memory constraints. 105

7.1 An illustration of one iteration of LCMNAS, showing the process of sequentially designing and evaluating architectures. It begins by sampling one layer and associated parameters for an architecture. The sampling is repeated several times until an architecture is fully designed. Then, the architecture is evaluated using a mixed-performance strategy and added to the population. The process of designing and evaluating architectures is repeated multiple times in a generation, and the search continues until the evolution reaches a final generation. 110
7.2 WDG representation of DenseNet121. Nodes represent layers and edges represent layer-transition probabilities between two nodes. 112
7.3 Architecture of the best models found in each data set used to perform macro-search. 119

7.4 Test accuracy (%) obtain in NAS-Bench-201 by LCMNAS with different number of generations, g, and population sizes p. 127

List of Tables

2.1 Overview of NAS benchmarks and categorization based on their properties. 24
2.2 Mean validation accuracy (%) and standard deviation of all architectures that contain an operation on a specific edge in NAS-Bench-101. 35
2.3 Mean validation accuracy (%) and standard deviation of all architectures that contain an operation on a specific edge in NAS-Bench-201. 36
2.4 Mean performance and standard deviation of all architectures that contain an operation on a specific edge in TransNAS-Bench-101. 37
2.5 Mean validation accuracy (%) and standard deviation of all architectures, evaluated based on the combination of 2 operations in NAS-Bench-101. . . 37
2.6 Mean validation accuracy (%) and standard deviation of all architectures, evaluated based on the combination of 2 operations in NAS-Bench-201. . . 38
2.7 Mean performance metric and standard deviation of all architectures, evaluated based on combination of 2 operations in TransNAS-Bench-101. . . . 39
2.8 Top 10 cells on NAS-Bench-101 based on the validation accuracy (%). . . . 40
2.9 Top-10 cells for each one of the data sets on NAS-Bench-201. Validation accuracy (%) is given only for the top-10 cells in each data set, while the ranking is also presented for the architectures outside the 10 best, allowing inter-data set comparison. 41
2.10 Top cells for each task on NAS-Bench-201. Performance is given only for the top-10 cells in each task while ranking is also presented for architectures outside the top 10, promoting inter-task comparison. 42

3.1 Results of different state-of-the-art CNNs architectures and the two proposed methods in the task of defect detection using the DAGM2007 data sets. Accuracy and AUC values are shown in percentages. 57
3.2 Comparison of different methods in the task of defect detection in DAGM2007 problems regarding the True Positive Rate, True Negative Rate, and Average Accuracy. 59
3.3 Mean accuracy and standard deviation on B-T4SA data set. The first block shows the results for lexicon-based methods. The second block shows the result of the implemented deep learning methods. In the third block, we show the results of the best method from the second block (RCNN), pre-training on SST and fine-tuning on B-T4SA. Each model was evaluated three times under the same conditions. 60

3.4 Train and validation accuracy (%) of several neural networks in sentiment classification. 61
3.5 Accuracy (%) in the test set for the proposed method and a baseline using SVM. 61
3.6 Comparison of the proposed methods with the state-of-the-art. TM stands for substituting the text classifier from [3] for the one selected in our experiments. 62

4.1 Mean test accuracy (%) and standard deviation across 50 runs in NAS-Bench-101 CIFAR-10 data set. Experiments with REA and Guided Evolutionary Algorithm (GEA) were performed with $p/s/c = 10/5/200$. 70
4.2 Comparison of manually designed architectures and several NAS methods using the NAS-Bench-201 benchmark. Performance is shown in terms of accuracy (%) with mean and standard deviation, on CIFAR-10, CIFAR-100, and ImageNet-16-120. Search times are the mean time required to search for cells in CIFAR-10. Search time includes the time taken to train architectures as part of the process where applicable. 71
4.3 Performance comparison of different NAS methods on TransNAS-Bench-101. The first block shows the results for directly searching on each task. The second block shows the transferred versions of different methods, which are pretrained on the least time-consuming task, i.e., Jigsaw. The final row shows the possible best result in each task. 73
4.4 Comparison of different performance estimation methods on NAS-Bench-201 benchmark. Performance is shown in accuracy with mean and standard deviation, on CIFAR-10, CIFAR-100, and ImageNet-16-120. Search times are the mean time required to search for cells in CIFAR-10, using a single 1080Ti Graphics Processing Unit (GPU). Search time includes the time taken to train architectures as part of the process where applicable. For each sample size, the optimal architecture is also shown. 74
4.5 Ablation studies for the number of parent candidates, s, the population size, p, and the regularization mechanism to remove individuals from the population. Results are shown in mean validation accuracy (%) and standard deviation from 5 runs in NAS-Bench-201 CIFAR-10 data set. 74

5.1 Search Performance from NAS-Bench-201. "Optimal" indicates the best test accuracy achievable in the space. 88
5.2 Search Performance using DARTS space on CIFAR-10. The last 4 blocks shows direct comparisons of NTKInner against other zero-cost proxy methods with different search methods. 89
5.3 Comparison of different NAS methods and zero-proxy estimator's in the proposed search space that leverages large feature extractors. 90
5.4 Performance in test error obtained by the three evaluated methods in the proposed search space using the CIFAR-100 data set. 90

5.5 Comparison of different NAS methods and zero-proxy estimator's in CIFAR-100. 91

6.1 Comparison with state-of-the-art image classifiers on CIFAR-10. The four blocks represent: human-designed, NAS, Multi-Agent Neural Architecture Search (MANAS) search with DARTS training protocol and best searched MANAS retrained with an extended protocol (AutoAugment (AA) + 1500 Epochs + 50 Channels). Unless specified, all MANAS architectures use 20 cells. 103

6.2 Comparison with state-of-the-art image classifiers on ImageNet (mobile setting). The four blocks represent: human-designed, NAS, MANAS search with DARTS training protocol and best searched MANAS retrained with an extended protocol (AutoAugment (AA) + 600 Epochs + 60 Channels). . . . 104

6.3 Results of MANAS with complexity constraints using different penalty (ρ) values on CIFAR-10. 106

7.1 Comparison of the proposed method against ResNet, the best architecture in NAS-Bench-201 benchmark, and 3 types of RS. The proposed method is evaluated using different λ values. When applicable, the comparison is measured by the test error (%), the inference time (Milliseconds (ms)), the search cost in GPU days, and the number of parameters of the generated architecture in millions. 115

7.2 Comparison of different methods on CIFAR-10. The first block presents a state-of-the-art human-designed CNN. The second block presents the results of proposals that perform macro-search, and the third block presents the proposed method. For each method, the test error in percentages, the search cost in terms of GPU days, and the size of the architecture in millions of parameters are shown. The search cost is the total GPU computation used in days, based on the number of GPUs and the execution time. 117

7.3 Test error (%) obtained by transferring the best architectures found in one data set to other data sets without any modifications. Architectures were trained on the new data set from scratch. 118

7.4 Test error (%) of the best architectures found on CIFAR-10 using different search epochs e and fixed $\lambda = 0.75$. 120

7.5 Test error (%) in CIFAR-10 using $\lambda = 1$ for different number of generations, g, and population sizes, p. 121

7.6 Ensemble test error (%) for CIFAR-10, CIFAR-100 and ImageNet-16-120 with different ensemble sizes k. Architectures used to perform the ensemble are the final ones found by searching with different $\lambda \in \{0, 0.25, 0.5, 0.75, 1\}$. 121

7.7 Search Performance on NAS-Bench-101. Test regret is the difference between the optimal architecture in NAS-Bench-101 and the test accuracy obtained by a NAS method. Table results adapted from [4]. 122

7.8 Comparison of different methods on the NAS-Bench-201 benchmark. The first block shows manually design architectures, the second block shows weight-sharing NAS methods and the third block shows the results for non-weight sharing NAS methods. The results are shown in terms of accuracy (%) with mean and standard deviation for CIFAR-10, CIFAR-100 and ImageNet-16-120. Search times are the mean time required to search for cells in CIFAR-10 and include the time required to train architectures where applicable. 123
7.9 Comparison of different methods on the TransNAS-Bench-101 benchmark. The first block shows the results for NAS methods that performed direct search on each task, while the second block shows the results for NAS methods that searched on jigsaw and transferred the resulting architectures to other tasks. The final row shows the global best result. 125
7.10 Kendall's Tau correlation (τ) across each of NAS-Bench-201 data sets for different number of train epochs (e) and Lambda (λ) values. The results show that on all settings, the combination of both components of the performance estimation strategy leads to a higher correlation with respect to the final validation accuracy of the generated architectures. 125

Acronyms

AI	Artificial Intelligence
ASR	Automatic Speech Recognition
AUC	Area Under The Roc Curve
AutoML	Automated Machine Learning
BO	Bayesian Optimization
CNN	Convolutional Neural Network
CV	Computer Vision
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
EA	Evolutionary Algorithm
GB	Gradient Boosting
GEA	Guided Evolutionary Algorithm
GPU	Graphics Processing Unit
IC	Image Classification
LCMNAS	Towards Less Constrained Macro-Neural Architecture Search
LSTM	Long-Short Term Memory
MANAS	Multi-Agent Neural Architecture Search
MA	Multi-Agent
ML	Machine Learning
NAS	Neural Architecture Search
NLP	Natural Language Processing
NTK	Neural Tangent Kernel
OS	One-shot
RCNN	Recurrent Convolutional Neural Network
RL	Reinforcement Learning

RNN	Recurrent Neural Network
RS	Random Search
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
SMBO	Sequential Model-Based Optimization
SVM	Support Vector Machine
WDG	Weighted Directed Graph
ms	Milliseconds

Chapter 1

Introduction

1.1 Problem Definition and Research Objectives

The ever-growing success of Artificial Intelligence (AI) and Machine Learning (ML) applications created a demand for intelligent systems that can be used off the shelf without care for their inner components. This success is mostly attributed to deep learning algorithms, especially neural networks, which have been used with great success [5, 6]. Advances in deep learning and especially Convolutional Neural Networks (CNNs), can be largely attributed to the ingenuity and engineering efforts of human experts who have designed and optimized powerful neural network architectures that have been extensively applied with great success to different problems, obtaining unprecedented results [5, 6, 7, 8]. However, applying a ML algorithm to a problem for which it has not been explicitly tailor-made usually leads to sub-optimal results, which in extreme cases can even lead to poor performances, thus hindering the sustainability of a system and the wide-spread application of ML by non-experts. Designing tailor-made ML algorithms for specific problems is a difficult task, as many design choices are dependent on one another. This is especially true when designing CNNs. The design and engineering of CNNs is a grueling endeavor that heavily relies on human expertise and results from years of expertise and extensive architecture engineering. Design choices, such as selecting the appropriate types of layers, defining their connections, setting hyper-parameters, choosing activation functions, deciding when and where to normalize feature maps, and more, all contribute to the complexity of the task.

Given the complexity of designing efficient ML systems and recent advances in hardware, it became logical to (re-)explore automated approaches for this process [9, 10].

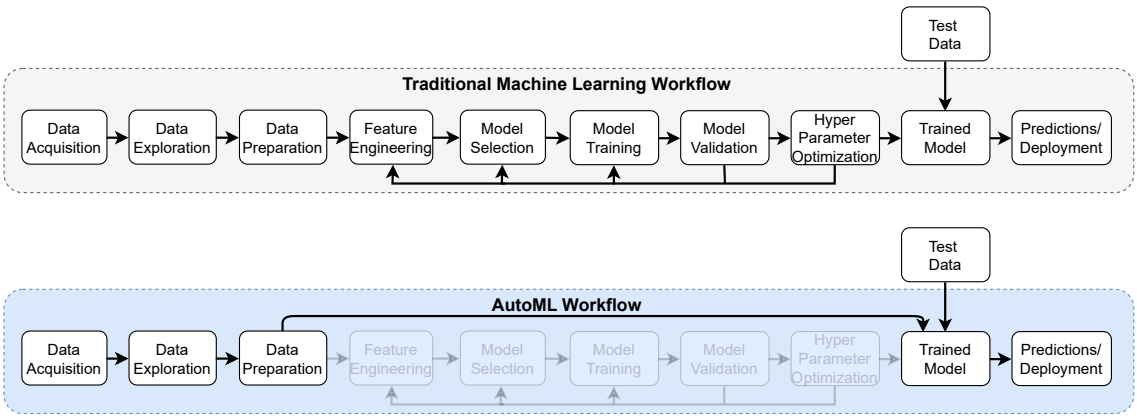


Figure 1.1: Comparison between traditional ML workflow and the workflow of an AutoML method.

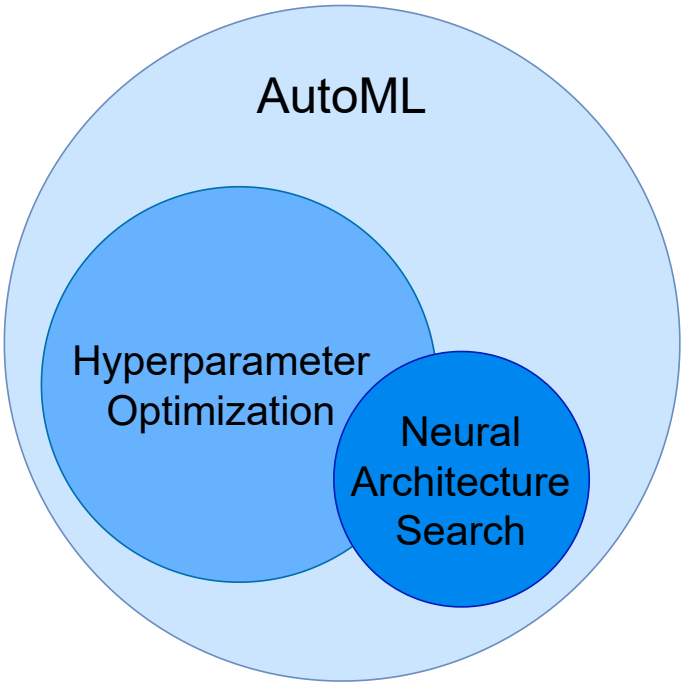


Figure 1.2: Main fields of research within AutoML.

The field of AutoML is a field of research whose aim is to develop methods and tools to provide efficient and automated mechanisms that can be used to design tailor-made ML algorithms for a user's problems [10]. AutoML systems typically automate the process of data processing and feature preprocessing steps for a given data set, as well as algorithm search, which might include the design of the model's hyper-parameters. In massified ML systems, these steps are typically performed by the users (except for feature extraction when using deep neural networks) [11]. A generalist comparison of the difference between traditional ML and AutoML workflows is depicted in Figure 1.1.

Within AutoML, there are two other significant and more focused fields of research (Figure 1.2). The first one is hyper-parameter optimization, which focuses on choosing a set of optimal hyper-parameters for a learning algorithm that ultimately leads to the right combination of values that allow a ML method to perform well on the given data. The second one, NAS, intends to automate architecture engineering and design for neural networks, which are amongst the most complex ML methods [12, 10]. This topic shares some similarities with hyper-parameter optimization whenever NAS methods optimize not only the architecture but also the layer's weights and associated hyper-parameters [13, 14]. The topic of NAS is relatively recent, as it has been overlooked due to its complexity and hardware inefficiencies [13]. Then, in 2016, Barret Zoph and Quoc Viet Le (re-)ignited the topic by formulating NAS as a Reinforcement Learning (RL) problem, where a controller is trained over time to sample efficient architectures [15]. Several NAS methods have since been proposed and applied to a vast panoply of problems, including Image Classification (IC) [16, 2, 17, 18], semantic segmentation [19, 20], object detection [21], image generation [22, 23], biometrics [24, 25, 26], natural language processing [27, 28], speech

recognition [29, 30], and many others [31, 32, 33, 34, 35]. In general, NAS methods can be broken down into three distinct components: i) the search space, which defines the pool of possible operations and, thereby, the types of architectures that can be designed; ii) the search strategy, which is the approach used to explore the search space and generate architectures; and iii) the performance estimation strategy, which is how the generated architectures are evaluated during the search process (we refer the reader to chapter 2 for an extensive description). The search strategy can be further classified into two main categories: micro-search and macro-search. Micro or cell-based search NAS focuses on creating cells or blocks that are replicated multiple times to form an architecture, while in macro-search, NAS methods aim to either design architectural connections or, in more unconstrained scenarios, entire architectures.

Formally, NAS can be considered an optimization problem where the goal is to maximize an objective function [36]. To define NAS optimization problem, we first define a deep learning algorithm, $\mathbb{L}$, as a mapping from the space of data sets, D and architectures, $\mathcal{A}$, to the space models $M, \mathbb{L} : D \times \mathcal{A} \rightarrow M$. For any data set $d \in D$, and architecture $a \in \mathcal{A}$, the mapping returns the solution to the problem, which consists of minimizing a loss function, $\mathcal{L}$, using a regularization mechanism, R, with respect to the model, m, parameters θ, architecture a, and the given data d:

$$\mathbb{L}(a, d) = \underset{m^{(a,\theta)} \in M^{(a)}}{\arg\min} \mathcal{L}(m^{(a,\theta)}, d^{(train)}) + R(\theta) \quad (1.1)$$

Thus, NAS can be defined as a nested optimization problem that given a data set, d, and a search space, $\mathcal{A}$, the optimization problem goal is to find the optimal architecture, $a^* \in \mathcal{A}$, that maximizes the objective function $\mathcal{Q}$, on the validation set:

$$a^* = \underset{a \in \mathcal{A}}{\arg\max}\, \mathcal{Q}(\mathbb{L}(a, d^{(train)}), d^{(valid)}) \quad (1.2)$$

In IC problems, $\mathcal{Q}(\cdot)$ usually takes the form of classification accuracy. However, as NAS goal is to design optimal neural networks for a given problem, NAS methods should allow the design of architectures for problems rather than IC, such as segmentation and regression, broadening the automation design to different problems.

Even though NAS research is very active and state-of-the-art methods perform well in designing CNNs, there are still significant limitations and drawbacks that disallow its use in a wider context:

- Most prominent NAS methods require substantial computational power, which translates to dozens or even hundreds of GPUs [12, 37].
- The time required to search for an architecture is considerable, where in some cases can ascend to thousands of GPU days [38, 15];

- Designed architectures can have a high inference latency (time to process an input) due to the architecture's structure growing in both breadth and depth dimensions [39].
- Search spaces depend heavily on human definitions and are usually small with forced operations [40, 41].
- The search is primarily performed in a cell-based manner, where NAS methods search for small cells that are later replicated in a human-defined outer-skeleton [42, 2].
- Architecture's parameters, such as the number of layers, inner-layer parameters (e.g., the kernel size, output channels), the final architecture skeleton, fixed operations, and head and tail of the final architectures are usually defined by the authors [40, 1, 41].
- Few methods can perform macro-search, and those that do still have considerable search costs, precluding wide-spread application [12, 43].
- Reproducing results from the literature is challenging due to the complexity of the algorithms and possible lack of detailed description of all the optimization and tools used [40, 1, 44].

More, pre-defining rules and carefully designing search spaces and architecture's structure undoubtedly induces human biases in the loop, which was found to often impact the final result of the architectures more than the search strategy itself, as it pushes NAS to constrained search spaces with very narrow accuracy ranges [40, 45]. Pre-defining rules also jeopardizes the generalization of NAS methods, even to simpler settings [42, 1]. The overall idea of heading to NAS to avoid needing deep knowledge regarding architecture design (Figure 1.1) ends up being frustrated since human involvement is required to design the search space, the outer-skeleton scheme, the pool of operations, and all the fixed hyper-parameters to ensure that a cell-based space is capable of designing efficient CNNs. So, the general objective of this thesis is to study and develop novel methods to improve NAS state-of-the-art, and ultimately AutoML by proposing a set of novel and reproducible methods to mitigate the aforementioned problems. This research has the following objectives as pillars:

- Evaluation and study of current state-of-the-art NAS methods and their inner components. More, study how impactful are benchmarks and their search spaces in the evaluation of NAS methods.
- Proposal of a new method that leverages NAS and AutoML to augment existing neural networks in useful time. For this, the goal is not only to design new architectures but to extend the use of already existing human-defined ones to understand how they work and how these can be further improved.

- Design and develop new mechanisms to evaluate generated architectures efficiently and quickly, thus reducing the time taken by the most timely and expensive component – the performance estimation.
- Development of several methods that reduce the time taken to perform the search while also remaining competitive in terms of the performance of the generated architectures.
- Improvement of existing NAS optimization strategies by allowing the search space to be progressively analyzed, possibly in a distributed way, therefore allowing NAS methods to efficiently search over very large search spaces, potentially unbounded.
- Development and validation of a novel NAS method that leverages information of human-designed networks without limiting or enforcing the search space or exploration. This should extend the field of NAS by heavily reducing human intervention and the need for user-specified rules or heuristics.
- Extend the use of NAS from cell-based search to macro-search, thus providing a mechanism for NAS methods to search for entire architectures from scratch without pre-defined rules or outer-skeletons.

1.2 Main Contributions

In the following paragraphs, a brief description and summary of the main contributions achieved throughout the development of this thesis and associated research initiatives to advance the state-of-the-art in NAS are presented.

The first contribution of this thesis is a review of the state-of-the-art in NAS. This review includes a study of NAS methods and 3 components that define a method: search strategy, performance estimation mechanism, and search space. The resulting study is shown in chapter 2.

The second contribution is a comprehensive review of NAS benchmarks and a study and evaluation of the impact that the different operations (layers), their combination, and their occurrences have in the final performance. More, in this study, an evaluation of the analyzed search spaces in terms of their architecture's performance distribution is also provided. This study resulted in a research paper submitted to a journal and is currently in the second review stage [41]. This study is included in chapter 2.

The third contribution of this thesis is a novel defect detection method based on improving CNNs by automatically searching for a new classification component. For this, a set of image classifiers (CNNs) were implemented and studied. Based on the results, an ensemble method and a novel method that searches for a new classifier to be concatenated to the best individual CNN were proposed. This work has been published in the

International Conference on Neural Information Processing (ICONIP) 2020 [46]. This contribution paved the way for the fourth one, where a deeper analysis of different image and text classifiers was performed for sentiment analysis. Based on the results, a new multimodal sentiment analysis method was proposed by leveraging AutoML and NAS to design a fusion component that combines the individual classifications of the text and image classifiers. This work was published in the International Joint Conference on Neural Networks (IJCNN) 2021 [47]. These contributions are included in chapter 3.

The fifth contribution of this work introduced a zero-proxy estimation mechanism that allows the evaluation of neural networks at initialization stage. With this proposal, NAS methods can be significantly optimized, as this proposal scores architectures without requiring any training, thus speeding up the search stage. This proposal was published at the International Conference on Artificial Neural Networks (ICANN) 2021 and was awarded the best paper award ("1st Springer & ENNS Best Paper Award @ ICANN21") [48]. This work is presented in chapter 4.

The sixth contribution regards the proposal of an efficient evolutionary algorithm that evolves architectures through mutation and guides the search based on a quick evaluation of the generated architectures. This work was initially accepted and presented at a NeurIPS 2021 workshop [49] and later improved and published in the Genetic and Evolutionary Computation Conference (GECCO) 2022 [50]. The proposed method was improved by using a zero-proxy estimator to guide the search efficiently, and a full version with extended experiments and justifications on multiple data sets and benchmarks is currently under review at a journal [51]. These are discussed in chapter 4.

The seventh contribution introduces a new way to leverage large CNNs trained on extensive data sets by combining their feature maps and designing a *middleware* architecture that receives those feature maps and learns to solve a downstream task. Coupled with this, a new zero-proxy estimator that evaluates architectures at initialization stage by looking at the alignment between the eigenvectors of the architecture's Neural Tangent Kernel (NTK) is proposed. The resulting work is in the submission stage. This contribution is the basis for chapter 5.

The eighth contribution of this work is framing NAS as a Multi-Agent (MA) problem and allowing multiple agents to collaborate in order to move the solution toward the global minima (generating efficient architectures). In this, the search space can be efficiently distributed by coupling each decision with an agent, where each agent is only responsible for sampling one operation (layer) on a cell-based search problem. This resulted in memory and speed gains, which allows direct search on large data sets, including searching for multiple cells at once. This was accepted for publication in Springer Machine Learning [52] and is part of chapter 6.

Improving Neural Architecture Search

The ninth and final contribution of this work is a NAS method capable of performing macro-search, i.e., searching for entire architectures from scratch and without predefined rules, generating search spaces from existing CNNs without requiring human intervention, and quickly and efficiently evaluate architectures by combining a zero-proxy estimator with a low-fidelity estimate. This work intends to push NAS boundaries to less constrained spaces, where human-expertize for the design of inner-architecture parameters and search spaces is reduced while at the same time generating architectures in a very efficient way. Therefore allowing a step towards wide-spread use of NAS for different problems and data sets. This has been submitted to a journal and is in the second review stage [43].

It is worth noting that the contributions of this research have been publicly disclosed, and the code is available on GitHub to promote advances in the field. During this work, a library to work with different NAS benchmarks has also been put publicly available, which allows researchers to quickly and effortlessly work with different benchmarks without needing to worry with the inner settings[1] [2] [3] [4] [5] [6].

1.3 Thesis Outline

The remainder of this thesis document is organized as follows: Chapter 2 introduces the background of NAS, presents a literature review, and a study of NAS benchmarks. Chapter 3 describes the proposed methods to augment CNNs by searching for new classification components and fusion mechanisms through NAS and AutoML. Chapter 4 presents the proposed zero-proxy estimator that calculates statistics over an untrained architecture to infer its performance and presents an evolutionary strategy that efficiently guides the search by generating multiple architectures through mutation and quickly evaluating their possible performance with the zero-proxy estimator. Chapter 5 extends the use of zero-proxy estimators by proposing a mechanism for evaluating architectures based on an architecture's NTK. More, this chapter also describes the search space proposed to leverage large CNNs in downstream tasks by generating a *middleware* architecture that takes their feature maps as inputs. Chapter 6 discusses and presents a mechanism that uses a MA framework to distribute the search space over different agents, thus providing a way to quickly and efficiently perform cell-based search over a high number of cells. Chapter 7 describes the contributions of this thesis that show how NAS can be extended to a macro-search setting and the associated proposed methods to reduce the need for human expertise when designing search spaces while remaining search efficient with excellent results. Finally, chapter 8 summarizes the work done, provides conclusions, and presents possible directions for future work.

[1] https://github.com/VascoLopes/AutoClassifier
[2] https://github.com/VascoLopes/Text-Classification
[3] https://github.com/VascoLopes/EPE-NAS
[4] https://github.com/VascoLopes/GEA
[5] https://github.com/VascoLopes/NAS-Benchmark-Evaluation
[6] https://github.com/VascoLopes/LCMNAS

Chapter 2

Related Work

2.1 Introduction

Advances in the field of deep learning obtained impressive breakthroughs and remarkable results in various problems. These advances can be largely attributed to the ingenuity and engineering efforts of human experts who have designed and optimized powerful neural network architectures. CNNs revolutionized the field of Computer Vision (CV) by obtaining remarkable success in various tasks and became the backbone of many state-of-the-art algorithms and applications in image understanding [7, 53, 54, 5, 6, 7, 8]. The excellent results and popularity of CNNs can be greatly attributed to their ability to automatically learn hierarchical representations from raw input data, capturing both low-level and high-level visual features without requiring human-engineered features.

CNN is a type of deep learning model inspired by the organization of the visual cortex in the human brain [55]. It consists of multiple layers of interconnected nodes, called neurons, that try to mimic the behavior of neurons in the human brain. The key concept behind CNNs is the use of convolutional layers, which perform local receptive field operations to detect patterns and features in the input data, allowing the recognition of patterns that are shifted (translation invariant), tilted or slightly warped within images. These layers are typically followed by pooling layers to reduce spatial dimensions and increase computational efficiency. The final layers of a CNN are usually fully connected layers that map the learned features to specific output classes or predictions.

CNNs consist of different types of layers, which in the context of NAS are commonly referred to as operations. Each operation serves a specific purpose in the learning process. The most common layers found in a CNN include convolutional layers, pooling layers, and fully connected layers. Most learnable layers, such as convolutional an pooling layers, applies a set of filters, also referred to as kernels, to the input data. In the case of convolutional layers, they perform local receptive field operations by convolving learnable filters, with the input data, thus extracting spatial features and preserving the spatial relationship between the input and the learned features. During the forward pass, the kernel slides across the height and width dimensions of the input, producing a high-dimensional representation of the input, usually referred to as an activation or feature map.

By combining different layers and components in varied configurations, CNN can effectively capture and extract meaningful features from input data. The design choices and configurations of these layers have a significant impact on the network's learning capacity

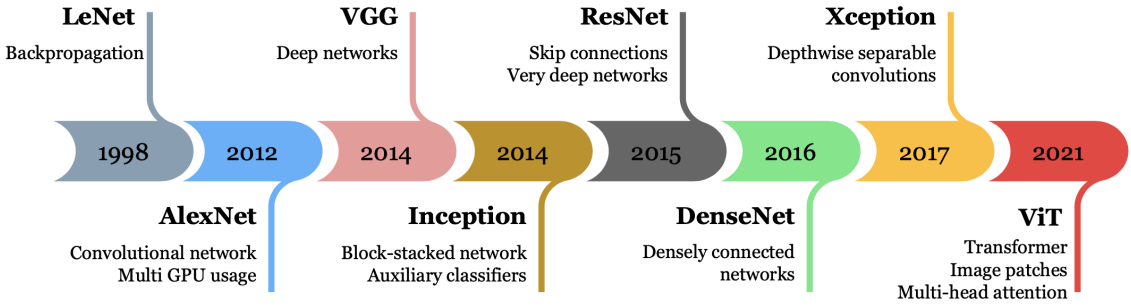


Figure 2.1: Examples of some of the most influential neural networks for CV applications and associated main contributions.

and performance. Different CNN architectures have gradually been proposed, incrementally showing that CNNs can be improved, by revising the architecture itself, adding additional components such as residual connections, reducing the number of parameters, the size or inference time, as well as the proposal of new and more efficient operations [56, 57, 58, 59, 60, 61]. Some examples of the most influential neural networks for CV applications are shown in Figure 2.1, where the associated contributions are depicted.

However, designing CNNs is a grueling endeavor that heavily relies on human expertise. Design choices, such as selecting the appropriate types of layers, defining their connections, setting hyper-parameters, choosing activation functions, deciding when and where to normalize feature maps, and more, all contribute to the complexity of the task. Designing efficient CNNs results from years of expertise and extensive architecture engineering. Thus, given the complexity of designing efficient CNNs, it has become logical to explore automated approaches to this process [10]. In the next sections, we dive deeper into the field of NAS and its inner components.

2.2 Neural Architecture Search

Deep learning, especially CNNs, automated the feature extraction and learning task, thus removing a laborious step from the ML pipeline, leaving the architecture design problem to human experts. However, given the complexity of designing efficient CNNs, as it requires years of expertise and extensive architecture engineering, it has become logical to explore automated approaches to this process. NAS is a topic of research that aims to automate architecture engineering and design by autonomously designing high-performance architectures for a given problem [12]. The origins of NAS and, more broadly, an automated design of neural networks can be traced back to neuroevolution, particularly with the development of NEAT by Stanley and Miikkulainen in 2002 [62]. NEAT combined evolutionary algorithms with neural networks, allowing an automatic generation and evolution of neural architectures. NEAT demonstrated promising results in various domains, and follow-up work improved upon NEAT by proposing different evolutionary strategies, optimization techniques, and extending it for new problems [63, 9, 64, 65, 66]. However, the application of neuroevolution was limited by the computational complexity and

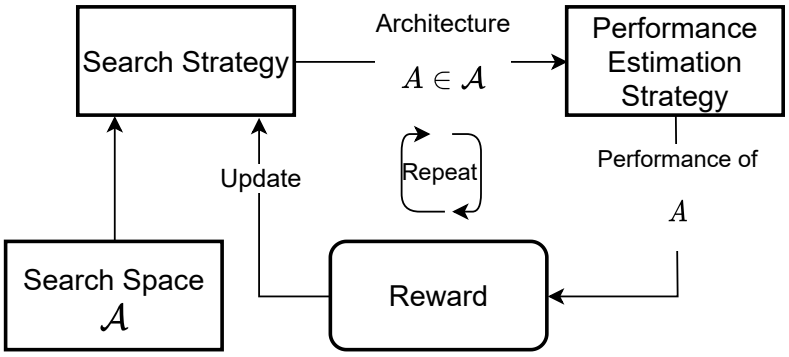


Figure 2.2: General flow of a NAS approach, where a controller generates an architecture A from a predefined space of possible architectures, $\mathcal{A}$. The generated architecture is then evaluated, and its performance is used as a reward to update the controller. In this thesis, we propose several methods for all three NAS components: search space, search strategy and the performance estimation strategy.

scalability challenges associated with evolving complex neural network architectures. For an in-depth analysis of neuroevolution, we refer the reader to the comprehensive overview presented in [67].

In 2016, B. Zoph and Q.V. Le re-ignited the topic of automating the design of neural networks by formulating NAS as a RL problem [15]. Here, a controller was trained based on how well the generated architectures performed on IC tasks. This work demonstrated remarkable success by achieving state-of-the-art performance on various challenging tasks, including IC and language modeling, highlighting the potential of using automated methods to discover architectures that surpass the performance of manually designed ones. Albeit yielding excellent results, it required more than 21900 days of GPU computation to search for an architecture. So, as a follow-up work, the authors tackle this problem by performing a cell-based search in a novel search space containing 13 operations [38]. Different NAS strategies have been proposed to improve upon initial results, including novel RL strategies, evolutionary strategy mechanisms, gradient-based methods, one-shot strategies and more [68, 69, 70, 71, 72, 73, 74, 75]. Follow-up works propose different search spaces, optimization mechanisms and applications of NAS to different problems, such as IC [16, 2, 17, 18], semantic segmentation [19, 20], object detection [21], image generation [22, 23], biometrics [24, 25, 26], natural language processing [27, 28], speech recognition [29, 30], and many other tasks [31, 32, 33, 34, 35], while consistently achieving state-of-the-art results.

In general, NAS methods can be broken down into three distinct components: i) the search space, which defines the pool of possible operations and, thereby, the types of networks that can be designed; ii) the search strategy, which is the approach used to explore the search space and generate architectures; and iii) the performance estimation strategy, which is how the generated architectures are evaluated during the search process. A diagram outlining the general interaction between these NAS components is depicted in Figure 2.2. The search strategy can be broadly classified into two main categories: micro-search and macro-search. Micro-search NAS focuses on creating cells or blocks that can

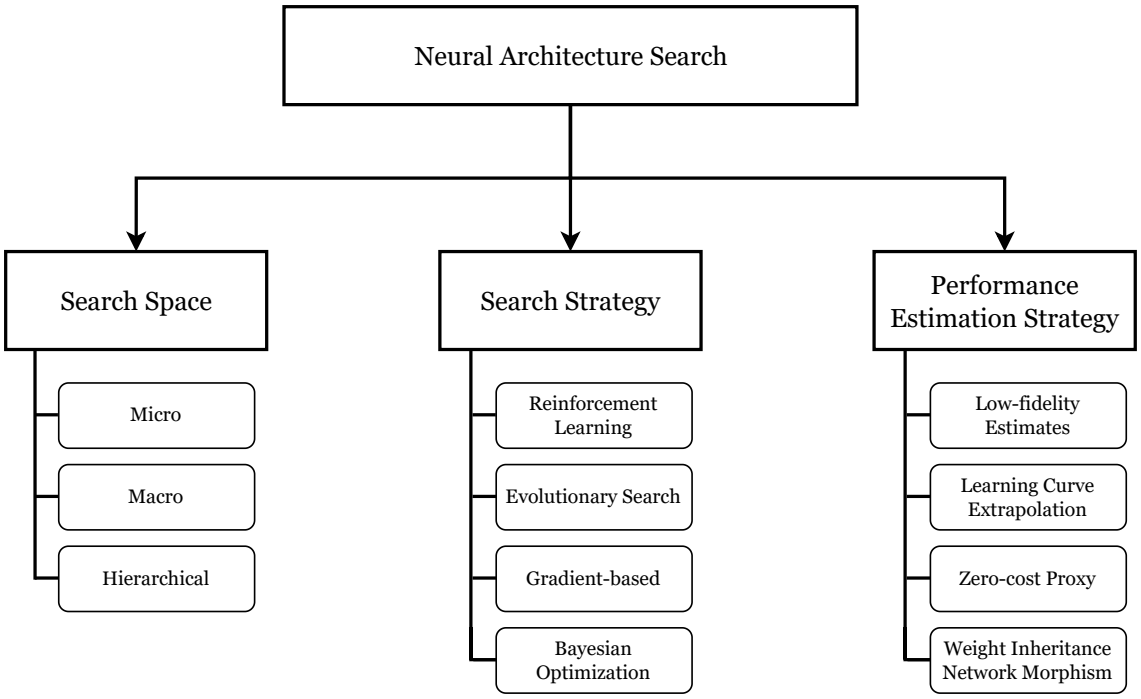


Figure 2.3: Main topics of research and development under each NAS component.

be replicated multiple times to form a complete architecture, while in macro-search, NAS methods aim to design entire network architectures. With an increasing number of NAS methods being proposed, the focus has been on optimizing the search process to reduce the required computations while simultaneously improving the performance of the generated architectures, mainly on cell-based constrained settings [12, 76, 74, 77, 78].

In the following sections, we dive deeper into each component and present an overview of several NAS proposals and their contributions. The structure of the following sections follows the one depicted in Figure 2.3, where for each NAS component, we present several topics of research.

2.2.1 Search Space

The search space refers to the set of all possible architectures that can be explored during the search process. It defines the pool of possible operations (layers), the space of architectural configurations, and, thereby, the types of architectures that can be designed.

A search space can be represented as a set of candidate architectures denoted by $\mathcal{A}$. Depending on the search space definition, each architecture $a \in \mathcal{A}$ represents a specific configuration of architectural decisions or choices, such as the number of layers, the type of operations, connectivity patterns, or hyperparameters. Typically, a generated architecture is represented as a vector, adjacency matrix, sequence of actions, or as a string, depending on the search strategy used. The search space $\mathcal{A}$ is usually discrete, allowing a graph representation, commonly defined as a Directed Acyclic Graph (DAG). The excep-

tion is some gradient-based methods that perform a relaxation of optimization problem during the search process.

The choice of search space is crucial for NAS algorithms, as it directly impacts the exploration and efficiency of the search process. Different search space definitions can lead to completely different architectures. The search space $\mathcal{A}$ is typically defined with constraints and assumptions about the architecture structure. These constraints include limitations on the number of layers, the total number of parameters, or specific architectural patterns. These definitions undoubtedly introduce human-biases in the search space's design. By forcing specific rules, sizes, and operations, search spaces are often so restricted that NAS algorithms easily find good architectures [1, 40, 45].

In the next sections, we detail three types of search spaces: micro or cell-based, macro, and hierarchical.

Micro Search Space

The most popular type of search space in NAS is the cell-based search space (also referred to as micro search space). In cell-based search spaces, the architecture consists of a fixed outer-skeleton, and a set of searchable cells make up the micro-structure. Instead of searching for the entire network architecture from scratch, cell-based search spaces propose searching over smaller, modular cells and stacking them in a pre-defined outer-skeleton to form the overall architecture. In this approach, the search algorithm focuses only on finding optimal cell architectures, as these cells are repeated throughout the network. Each cell, sometimes referred to as a motif, represents a set of inter-connected operations usually represented as DAG, with operations as either the edges or nodes, depending on the search space design, and feature maps as the result of those operations.

Cell-based search spaces are built upon the observation that effective handcrafted architectures are often made with repetitions of fixed structures [58, 59]. This allows formulating the NAS as a narrower and smaller problem, thus reducing the search complexity. The down-side with cell-based search spaces is that they require considerable human involvement, as the design choices such as the outer-skeleton, number of layers, operation pool, cell structure, and more are pre-defined by human experts. By smoothing the search problem with pre-defined settings and rules, there are undoubtedly human biases introduced into the optimization problem, and even though the impact of forcing these rules remains unclear, many works have shown that cell-based search spaces often have a very narrow performance distribution and redundancy, which restricts the search of NAS algorithms to easily find good architectures and hides the true contributions of NAS methods [1, 40, 45].

In [38], Zoph *et al.* introduced the NASNet search space as one of the first cell-based search spaces. This search space is comprised of two types of cells: normal cells and reduction cells. These cells share the same structure, but reduction cells handle tensor di-

mensionality reduction by employing initial operations with a stride of two to reduce the spatial resolution. The operation pool of NASNet has 13 different operations, including several convolutional layers, pooling operations, and identity.

Roughly at the same time, Zhong *et al.* proposed a cell-based search space, where instead of using reduction cells, max-pooling layers are used to handle feature dimensionality reduction [69]. And since these initial search spaces, several other cell-based search spaces have been proposed that follow a similar structure to the NASNet search space. The main differences are the design of the outer-skeleton, the cell structure, and the operation pool. In [79], the authors propose a variant of the NASNet, but instead of using two types of cells, here a search strategy only searches for one type of cell, and within the fixed outer-skeleton, some cells are pre-defined with a stride of two to reduce the feature map dimensionality. Also, in this search space, the operation pool was reduced to 8 by removing the lesser-used operations. Differently, DPP-Net proposed the use of a densely connected cell-based search space, where the operation pool has 3 normalization layers and 6 convolution layers optimized for inference, such as group and depth-wise convolutions [80].

DARTS is the most used cell-based search [2]. In this, operations are placed in the DAG's edges and the goal is to search for both normal and reduction cells, each with 8 searchable edges. The operation pool in DARTS is composed of 8 different operations: 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max pooling, 3×3 average pooling, identity, and zero. NAS-Bench-101 and NAS-Bench-201 are other two common search spaces. NAS-Bench-201 follows a similar definition as DARTS but in a more constrained and simpler setting. Here, the operation pool has 5 different operations, and the goal is to search for a cell with 6 edges. Differently, NAS-Bench-101 defines the nodes of a DAG as operations, with the cell structure having 7 nodes with an operation pool of size 3.

Cell-based search spaces have also been used in different applications other than IC. TransNAS-Bench-101 proposes the use of the NAS-Bench-201 cell structure for 7 different CV tasks. In this, the operation pool was reduced to only 4 operations. [81] extends a DARTS-alike search space for human parsing and pose estimation by designing encoder-decoder architectures. NAS-Bench-ASR looks into cell-based design for Automatic Speech Recognition (ASR) [30], AutoGAN for image generation [22], and several proposals have looked into using cell-based search spaces for Natural Language Processing (NLP) [27, 2, 70].

Cell-based search spaces are popular due to their ease of use and simplicity. By allowing the design of architectures based on repeated patterns, cell-based search spaces heavily reduce the complexity of the NAS optimization problem while allowing the design of state-of-the-art architectures in various tasks [82, 76, 83, 45]. However, given the aforemen-

tioned problems, such as human-induced biases and forced rules and restrictions, cell-based search spaces restrict the search in such a manner that it is impossible for a NAS method to design novel architectures that disrupt the design of current architectures.

Macro Search Space

Macro-search spaces in NAS refer to the exploration of architectures at a higher level of granularity, where the focus is on macro-level structure. In the literature, there are two types of macro search spaces: spaces that focus on the design of hyper-parameters and block connections, or search spaces that encode an entire architecture and allow a NAS method to search for all necessary components of an architecture. The latter allows NAS methods to operate at a coarser level, making architectural decisions that encompass multiple cells or blocks and, at the same time, force a granular search by requiring decisions about the type of operations to be sampled. This approach allows for a more diverse set of architectures with different macro-structures. In [15, 84, 85], the authors propose the use of macro search space in the sense that the search strategy decides the operations and the topology of a DAG. Differently, some proposals focus on forcing the architecture structure and allowing the design of the hyper-parameters of the layers [86, 87]. In Chapter 7, we show how a novel macro search space allows for an unbounded and less constrained search.

Several NAS methods also proposed using macro search spaces by designing architectures as a chain-structure. Chain-structured search spaces simplify the problem by reducing the complexity of designing and implementing macro-search search spaces where all decisions are open. In this, each sampled operation (layer) receives as input only the last layer [38], and an architecture is generated until a certain number of layers are sampled [15, 84]. Follow-up work proposed an extension on chain-structure architectures by incorporating the design of branches by allowing multi-layer connections, such as residual connections [38, 72, 12]. This type of macro search space has also been proposed with the use of existing architectures as initialization, where a neural network, such as ResNet or DenseNet, is used as the initial configuration, and the goal is to modify the architecture by adding or pruning operations, connections and hyper-parameters [16, 88, 89]. This initialization of the search space is leveraged by network morphism NAS methods [90].

Macro-search spaces differ from cell-based search spaces in the sense that they operate at a coarser level, making architectural decisions that go beyond cell design to also the design of the outer-skeleton. If not heavily restricted by human-defined rules, macro search spaces offer advantages regarding architecture diversity and the ability to explore a broader range of architecture designs. They allow for more complex and adaptive architectures that can provide solutions for the intricacies of different tasks and datasets. However, the search process in macro-search spaces can be more challenging and computationally expensive. Given that the search space has higher complexity and is broader, it is more challenging to find high-performing architectures.

Hierarchical Search Space

Hierarchical search spaces provide a structured framework for optimizing both macro and micro architectures. Instead of considering architectures as a single entity, hierarchical search spaces break down architectures into multiple levels or stages, each representing a different level of architectural complexity. These search spaces consist of multiple levels of motifs, where the motifs represent architectural patterns or building blocks. The first level often includes single operations, followed by motifs that connect these primitive operations, similar to cells, and if available, higher-level motifs encode connections between second-level motifs [73, 19, 91].

This hierarchical approach allows for a systematic exploration of architectural designs while providing control over the complexity and composition of the architecture. In [19], the authors propose the use of a two-level hierarchical search space, where cells are first designed and then a network level architecture search is performed. A three-level hierarchical search space was proposed in [92], whereas [73] proposes a multi-level search space, where the first level refers to single operations and higher-level motifs are designed by combining lower-level motifs (single operations if designing a second level motif) through the design of a DAG.

Hierarchical search spaces offer a structured and modular approach to NAS, allowing for a systematic and potentially progressive exploration of architectures through reusable motifs and incremental complexity. They provide benefits in terms of modularity, composition, incremental search complexity, and the ability to optimize architectures at multiple levels. However, designing hierarchical search spaces and determining the number of levels and connections between motifs require complex engineering and domain knowledge. Both micro and macro search spaces can be seen as single-level hierarchical search spaces, where decisions are made entirely on a single level and are not combined to form higher-level motifs.

2.2.2 Search Strategy

The search strategy refers to the optimization performed by the NAS algorithm to find optimal architecture within the defined search space. Broadly, the search strategy refers to the methodology employed to explore the search space and generate architectures, encompassing the algorithms and techniques used to search the space of possible architectures and identify promising candidates. Ultimately, the search strategy determines how architectures are generated and how the search progresses. Deciding on which optimization strategy to use depends on the task, resources available and the search space, as some optimization mechanisms are bound to specific settings. This section discusses various search strategy approaches in NAS, including RL, Evolutionary Algorithm (EA), Gradient Boosting (GB) methods, and Bayesian Optimization (BO).

Reinforcement Learning

RL-based NAS approaches treat architecture search as a sequential decision-making problem. A controller, often a Recurrent Neural Network (RNN), generates architectures by iteratively sampling decisions from a probability distribution to form a final action state. The set of decisions can include operations, connections, or hyper-parameters. The controller is trained using policy gradient methods, with rewards based on the performance of the generated architectures [93, 12].

NAS was formulated as a RL problem in 2016 by B. Zoph and Q.V. Le [15], in which a controller was trained based on how well the generated architectures performed on image classification tasks. Albeit yielding excellent results, it required more than 21900 days of GPU computation to search for an architecture. Follow-up work [38] reduced the search cost by proposing the use of a cell-based search space. In this, forming entire architectures, generated cells were stacked according to pre-defined rules. Even thought it still required more than 2000 days of GPU computation, it prompted many follow-up RL proposals to focus on cell-based designs.

Early NAS works focused on improving RL mechanisms. In [68], the authors use Q-learning to train a sampler agent. Similarly, [69] performs NAS by sampling blocks of operations instead of cells/architectures, which can then be replicated to form networks. More recently, ENAS [70], using a controller to discover architectures by searching for an optimal subgraph within a large computational graph, showed that it was possible to use RL in useful time, requiring only a few computational days by leveraging weight sharing. RL methods also differ in the way the controller is trained. Most proposals use REINFORCE [94] to update the policy [15, 16, 95], while some explore the use of proximal policy optimization [38, 96]. InstaNAS optimizes the search by sampling multiple architectures (actions) in each iteration to speed up the search while maintaining multi-objectives constraints, such that the generated architectures are both efficient in terms of performance and fast in terms of inference speed [97]. Several proposals have looked into different ways of sampling architectures, improving the search cost or mechanisms for rewarding the controller agent [98, 71, 52].

Evolutionary Search

Evolution-based search is one of the most used NAS optimization strategies to explore the space of architectures. EA draw inspiration from natural evolution processes, employing mechanisms such as mutation, crossover, selection, and reproduction to iteratively improve the population of architectures based on their performance [74].

Evolving neural networks can be traced back to neuroevolution, particularly with the development of NEAT [62]. NEAT showed that an automatic generation and evolution of neural architectures was possible, obtaining promising results. Follow-up neuroevolution

methods extended the applicability to new problems and focused on optimizing the search stage [63, 9, 64, 65, 66]. However, neuroevolution was limited by the computational complexity and scalability challenges associated with evolving complex neural network architectures. In 2017, Esteban Real *et al.* proposed an EA that evolves deep CNNs through mutations [84]. The problem with this proposal is that it requires a tremendous amount of computation to generate and evaluate architectures. REA, a follow-up work, improved the evolution process by evolving architectures through operation mutation and by employing a regularization mechanism for the population [72]. Even though REA required more than 2750 GPU days of computation, it showed capable of generating competitive architectures with smaller sizes than RL-based methods, thus prompting researchers to use evolutionary strategies, as they are flexible, easy to use and through different evolutionary processes can effectively search through a large search space [74, 12].

Several EA have been proposed to incrementally improve the performance of the method by designing novel heuristics to perform the evolution [73, 74, 75], by designing novel regularization mechanisms that favor different parameters, by employing rules for selecting parents to generate offsprings by ranking [99, 100, 101], niching [102, 103] and many others [84, 73, 104, 105, 106], as well as performing multi-objective search, where the goal is to find the architectures that satisfy performance and other objectives, such as size and inference speed [107, 108, 109, 110, 111, 112].

EA-based NAS search is popular, as it is versatile, flexible, and can handle complex constraints and incorporate domain-specific knowledge into the search process. EA can effectively explore and optimize non-differentiable and discrete spaces while easily adapting to different search space definitions.

Gradient-based

GB methods utilize gradient information to guide the search process. These methods typically involve formulating NAS optimization problem as a differentiable optimization, allowing architectures to be optimized via gradient descent. DARTS, one of the most prominent gradient-based NAS methods, relaxes the discrete decisions in architecture design to continuous variables, enabling efficient gradient-based optimization [2].

DARTS was the first gradient-based method capable of performing NAS in useful time by performing a continuous relaxation of the parameters and performing a bi-level gradient optimization [2]. This work was then improved using regularization mechanisms [17] and served as the basis for many others [98, 113, 16, 114]. NAO [115], instead of representing architectures as strings, as in DARTS, it uses a variational auto-encoder to learn a latent representation of candidate architectures and a performance predictor that uses the latent representation (vector) to predict the performance of an architecture. To reduce the computational cost of evaluating architectures during the search process, gradient-based NAS methods often leverage weight-sharing techniques. Often, gradient-based methods

work with an hypernetwork [116, 117] or a supernetwork to speed-up the search [2, 118]. By training the hypernetwork or the supernetwork, methods perform a One-shot (OS) approach to train all architectures in the search space. The use of hypernetworks allows methods to generate weights for candidate architectures, while supernetworks are large architectures that encompass all possible architectures in the search space [83].

GB methods are amongst the most popular and powerful approaches in NAS. By employing gradient computation, optimization algorithms, and differentiation strategies, these methods efficiently explore the search space and guide the search. When compared with non-optimized RL and EA approaches, GB NAS methods show improved performance.

Bayesian Optimization

BO can be categorized as a sequential design strategy for global optimization of black-box functions [119]. This definition broadly encompasses most NAS methods, especially the ones previously mentioned – EA, GB and RL. As these were described in detail in previous sections, here we look into BO methods that leverage probabilistic models to construct surrogate models to approximate the performance of generated architectures [83].

BO is a powerful NAS search strategy that combines probabilistic modeling and optimization techniques to efficiently explore and exploit the search space and optimize expensive functions. It aims to find an optimal architecture for a given problem by iteratively generating new candidates architectures based on their potential performance, training it, and updating the surrogate model to guide the subsequent iteration [120, 12, 83, 85, 121, 122]. The selection and sampling of candidate architectures is guided by an acquisition function, which measures the utility or potential of evaluating a candidate based on the current surrogate model and plays a pivotal role in balancing exploration and exploitation during the search process. Common approaches look into selecting candidate architectures based on evolutionary approaches and random search [123, 124, 125, 126, 50]

At its core, BO maintains a surrogate model that approximates the objective function that maps architectures to a performance score. This surrogate model is updated as new evaluations are performed, continuously improving its performance. Initially, researchers proposed the use of Gaussian processes to evaluate candidate architectures [127, 85, 128]. More recently, new proposals looked into the problem of evaluating architectures by using neural networks [127, 123], graph neural networks to leverage the inherent graph structure of architectures [129, 130, 125], tree-based approaches, such as random-forests [13, 131], and zero-proxy estimators [132].

2.2.3 Performance Estimation Strategy

Performance estimation strategy is a crucial component of NAS methods. It aims at estimating the performance of generated architectures, playing a crucial role in guiding the search process by determining the promising architectures to explore further. The choice of the performance estimation strategy or method has a tremendous impact on the search cost, as it is often the most computationally intense NAS component.

Albeit training and evaluating each architecture in the search space for a large number of epochs is the most reliable mechanism to rank it, this is extremely inefficient, being computationally expensive and time-consuming [15, 38, 79]. Therefore, researchers focused on proposing efficient and effective strategies to estimate the performance of architectures based on limited resources. In this section, we present the most used optimization techniques for estimating the performance of generated architectures – low-fidelity estimates, learning curve extrapolation, zero-cost proxies, and weight inheritance. Note that OS methods usually combine the search strategy and the performance estimation into one single component. Meta-learning is another approach to speed up the search, as it intends to use past knowledge to learn a new task [133]. However, meta-learning was out of the scope for this thesis work.

Low-fidelity Estimates

Low-fidelity estimates are approximations of the performance of a given architecture if trained until convergence and are typically used to perform a ranking of architectures. These estimates are used to guide the search process and make it computationally cheaper by reducing the resources required for evaluating architectures. There are several techniques for obtaining low-fidelity estimates. One common approach is to train architectures on a subset of the data [68, 38, 134, 135]. By using a smaller portion of the training data, the computational cost is significantly reduced. Another approach to low-fidelity estimates is training architectures for a smaller number of epochs [69, 136, 1, 137, 131], where instead of training an architecture until convergence, it is trained for a fraction of the total training. TSE creates a correlation between the training speed of an architecture with its final test performance, thus accelerating the evaluation of generated architectures by reducing the training required [138]. Differently, using downscaled architectures (such as fewer cells) [38, 72, 2] provides low-fidelity estimates that researchers leverage to speed up the search and architecture's evaluation cost.

The main challenge with low-fidelity estimates is balancing computational efficiency and accurate performance estimation. If the difference between the low-fidelity estimate and the true performance of an architecture is too significant, it can lead to unstable rankings and potentially misleading results [109, 139, 140, 141]. More, low-fidelity estimates undoubtedly introduce biases to the search process as the performance estimate is based on a limited amount of information, which may not accurately represent the performance

of architectures if trained, as different architectures behave differently for different data sets and parameters.

Learning Curve Extrapolation

Learning curve extrapolation is a promising approach to estimating the performance of architectures without fully training them. The idea here is to make predictions about the performance of an architecture based on its initial learning progress [142, 12, 83]. The process of learning curve extrapolation involves training architectures for a fraction of the total training until convergence and then extrapolating their performance based on the observed learning progress, thus making predictions about the architecture's potential.

In [143, 144], the authors leverage a continuous halving of the population by ignoring architectures that present bad learning progress. Even though no surrogate model is built, by continuously evaluating architectures and prioritizing the ones that present faster trainability, it extrapolates a future learning curve. However, a more common variation of learning curve extrapolation involves training a surrogate model. Instead of extrapolating the learning curves of individual architectures, a surrogate model is trained to predict the performance of novel architectures based on an architecture encoding and, if present, an initial training [120, 145, 146, 123, 147, 130, 148, 149].

Learning curve extrapolation, especially surrogate models, are often combined different search strategies and performance estimation mechanisms [123, 150], presenting a way to accelerate the performance estimation process in NAS if training the surrogate model does not hinder the process.

Zero-cost Proxies

Zero-cost proxy estimators estimate the performance of candidate architectures with minimal computational cost by scoring architectures at initialization stage. These estimators provide a way to approximate the performance of architectures using inexpensive computations or heuristics. Instead of training architectures until convergence, zero-cost proxy estimators analyze characteristics or properties of the architectures, such as the design or modeling capabilities [151, 152].

NAS-WOT was one of the first zero-cost proxies for NAS, showing remarkable success when scoring untrained architectures by looking at the separability of the input data into different linear regions [153]. Follow-up work tried to improve zero-cost proxies by removing the need for data [154, 155], by looking at the conditioning and spectrum of an architecture's NTK [156, 4], by pruning-at-initialization techniques [154, 157, 158], and by looking at evaluating regions of the search space [159] based on the evaluation of operations independently [160]. EZNAS looks into automating the process of designing zero-proxy estimators by directly evolving policies focused on interpretability and generability properties [161].

Krishnakumar *et al.* showed that zero-cost proxies can efficiently score architectures with good correlations with the final performance [151], and White *et al.* conducted a comprehensive evaluation of 31 different performance estimation strategies, many of which are zero-cost-proxies, showing that the combination of zero-cost proxies with other strategies yields the best results [162].

Weight Inheritance

Weight inheritance is an approach to speed up the performance estimation process by leveraging the knowledge extracted from previously trained architectures. Instead of training each new architecture from scratch, weight inheritance initializes the weights of an architecture based on the weights of architectures that have been trained before [90].

Network morphisms allow modifying an architecture while maintaining an identical output for the same input [163]. This means that the modified architecture can inherit the weights of the original architecture without sacrificing its performance. By progressively applying network morphisms, architectures can be successively expanded and retain high performance without the need for training from scratch [163, 12, 164, 165, 112, 127, 166, 84]. RandGrow performs morphisms based on random-search [167], and a follow-up work, Petridish, used gradient boost to further improve upon RandGrow [168].

One advantage of weight inheritance approaches, such as network morphisms, is that they enable the search space to have flexible architecture without constraining the maximum size of an architecture, thus promoting a search strategy closer to macro-search. However, it's important to note that most network morphism methods can only make architectures larger, which may lead to overly complex architectures. In [12], this is addressed by allowing architectures to be shrunk.

Weight inheritance techniques can significantly reduce the computational resources required for NAS. Initializing architectures with pre-trained weights can accelerate the training process and allow architectures to converge faster. However, network morphism requires architectures to share components with transferable weights, thus forcing the search to small increments upon base architectures and restricting the search of novel architectures.

2.3 Neural Architecture Search Benchmarks

Evaluating and comparing NAS methods is challenging, as different search spaces and training protocols difficult a fair comparison and hinder a true evaluation of how well a NAS method behaves. Researchers have continuously analyzed different NAS baselines to propose proper comparison metrics [44, 169, 170]. Li and Talwaker extensively studied Random Search (RS) in NAS and found that it presents a strong baseline for comparison, outperforming several NAS proposals [1]. Then, Yang *et. al.* conducted an extensive study

to evaluate the impact of the training protocols on the results of a generated architecture, showing that well-engineered training protocols usually have a more significant impact on the final performance of a NAS method than the search strategy, thus suggesting that NAS methods should provide results using the same protocols to ensure fair comparisons [40]. In [45], the authors analyzed popular cell-based search spaces. They found that existing search spaces contain a high degree of redundancy and generated architectures from distinct NAS methods have similar patterns. Studies to evaluate NAS performance predictors in terms of their correlation to the final architecture's performance, inference, and initialization time have also been conducted [171, 162, 152]. However, evaluating different NAS methods is still a challenge due to the human-defined search spaces and training protocols.

To address the aforementioned issues, NAS benchmarks have been proposed to facilitate the design and evaluation of NAS methods. These benchmarks allow NAS methods to access information about a unified search space and force fairer comparisons between methods by fixing hyper-parameters and training protocols [44, 172]. Nonetheless, little attention has been devoted to analyzing the benchmarks to ensure that these provide fair and competitive settings that can and should be used as common ground for NAS methods' comparisons. In the following sections, we provide a detailed overview of NAS benchmarks and a comprehensive study on how the operation pool of each benchmark out of the three most used ones influences the performance of generated architectures.

2.3.1 Overview

There are two types of benchmarks: tabular and surrogate. Tabular benchmarks provide pre-computed information on all possible architectures trained in one or more data sets, while surrogate benchmarks provide a model capable of predicting the performance of an architecture, along with pre-computed information about some of the architectures. By providing access to information through a tabular setting or a surrogate model, benchmarks can reduce the need for computational resources, enabling quick and efficient evaluations.

Table 2.1 provides an overview of the existing NAS benchmarks. NAS-Bench-101 [173] was the first tabular benchmark proposed and provides information about the validation and test accuracy of the $423,624$ cell-based architectures that make up the search space. Each architecture was trained on CIFAR-10 with different initialization random seeds. NAS-Bench-1Shot1 [175] leverages the information in NAS-Bench-101 by defining subsets of the search space with a fixed number of nodes, enabling one-shot NAS methods. The NDS benchmark [174], on the other hand, focuses on evaluating different randomly sampled architectures from five search spaces.

Another tabular benchmark, NAS-Bench-201 [42], proposes a smaller search space but with more data sets. This benchmark includes $15,625$ trained architectures on CIFAR-

Table 2.1: Overview of NAS benchmarks and categorization based on their properties.

Benchmark	Size	Search Scheme	Type		#Ops	Data sets	Task	License	Year	Venue
			Tab.	Surr.						
NAS-Bench-101 [173]	423k	cell	✓		3	1	IC	A2.0	2019	ICML
NDS [174]	139M	cell	✓†		5-8	2	IC	MIT	2019	ICCV
NAS-Bench-1Shot1 [175]	364k	cell	✓		3	1	IC	A2.0	2020	ICLR
NAS-Bench-201 [42] NATS-Bench [86]	15k 32k	cell macro	✓		5	3	IC	MIT	2020 2021	ICML TPAMI
LatBench [176]	15k	cell	✓		5	3	IC	A2.0	2020	NeurIPS
TransNAS-Bench-101 [87]	4k 3k	cell macro	✓		4	7	CV	MIT	2021	CVPR
NAS-Bench-Macro [177]	6k	macro	✓		3	1	IC	A2.0	2021	CVPR
NAS-Bench-ASR [30]	8k	cell	✓		8	1	ASR	A2.0	2021	ICLR
NAS-Bench-111 [178]	423k	cell		✓	3	1	IC	A2.0	2021	NeurIPS
NAS-Bench-311 [178]	10^{18}	cell		✓	8	1	IC	A2.0	2021	NeurIPS
NAS-Bench-NLP11 [178]	10^{53}	cell		✓	7	1	NLP	A2.0	2021	NeurIPS
HW-NAS-Bench-201 [179]	15k	cell	✓		3	3	IC	MIT	2021	ICLR
HW-NAS-Bench-FBNet [179]	10^{21}	chain	✓		9	1	IC	MIT	2021	ICLR
BLOX [180]	91k	macro	✓		5	1	IC	CC	2022	NeurIPS
JAHS-Bench-201 [181]	270k	cell		✓	5	3	IC	MIT	2022	NeurIPS
NAS-Bench-NLP [27]	10^{53}	cell	✓†		7	1	NLP	A2.0	2022	IEEE Access
NAS-Bench-301 [182] (Surr-NAS-Bench-DARTS)	10^{18}	cell		✓	8	1	IC	A2.0	2022	ICLR
Surr-NAS-Bench-FBNet [182]	10^{21}	chain		✓	9	1	IC	A2.0	2022	ICLR
NAS-Bench-MR [183]	10^{23}	cell		✓	-	9	CV	BSD	2022	ICLR

† only a small subset of the entire search space was fully trained and provided as a tabular benchmark.

10, CIFAR-100, and ImageNet16-120. LatBench augments NAS-Bench-201 by providing latency information about all the architectures in six different devices. Similarly, HW-NAS-Bench [179] proposes two benchmarks: HW-NAS-Bench-201, an extension of NAS-Bench-201, and HW-NAS-Bench-FBNet, which uses the FBNet search space. These benchmarks focus on measuring and estimating the latency and energy consumption of all architectures in the initial search spaces in six different devices. The authors of NAS-Bench-201 also proposed NATS-Bench [86], which provides a macro search space with tabular information for $32,768$ architectures. More, JAHS-Bench-201 [181] also extends the search space proposed by NAS-Bench-201 to a joint optimization of the architecture design and hyper-parameters. The authors designed a combination of the $15,625$ architectures with different hyper-parameters, totaling $270,000$ configurations for which 20 performance metrics in 3 data sets were evaluated, thus providing more than 161 million surrogate data points.

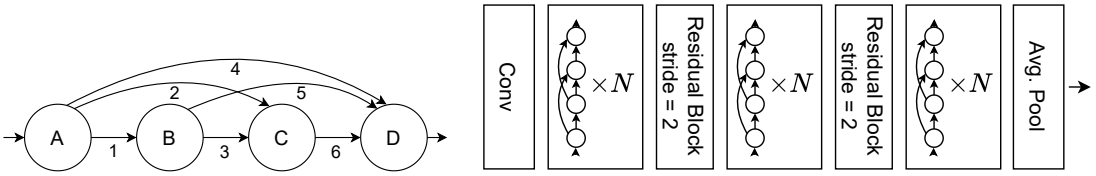
Benchmarks that provide a macro search space enable the evaluation of NAS methods beyond traditional operation sampling, extending to decisions about node connections,

operation parameters, or the architecture skeleton. One such benchmark is NAS-Bench-Macro [177], which proposes a search space with $6,561$ architectures. The goal when using this benchmark is to search for 8 layers with a pool of 3 possible blocks. Another macro-based benchmark is Blox [180], which provides tabular information about $91,125$ unique architectures. These architectures are designed by fixing an outer-skeleton and searching for 3 blocks with diverse connectivities.

First NAS benchmarks focused on evaluating architectures in IC tasks. However, more recent benchmarks have expanded to allow the evaluation of NAS methods in terms of generability and transferability to different tasks. One example is TransNAS-Bench-101, which offers both a cell-based and macro-based search space and evaluates architectures over seven different CV problems, with the aim of providing a framework for evaluating NAS transferability across tasks [87]. Another example is NAS-Bench-MR, which offers a surrogate benchmark for four different CV tasks, including segmentation [183]. To further expand the scope of NAS applications, NAS-Bench-ASR proposes a cell-based search space with $8,242$ trained architectures for ASR [30], while NAS-Bench-NLP proposed a benchmark for NLP, providing tabular information for a subset of the entire search space [27].

Surrogate benchmarks have been proposed to enable the design and evaluation of larger search spaces by training a set of architectures and then fitting a surrogate model capable of inferring the performance of newly generated architectures. Two such benchmarks are NAS-Bench-301 and Surr-NAS-Bench-FBNet, which fully train a set of architectures and fit surrogate models to estimate the final performance of the remaining architectures [182]. However, surrogate benchmarks often provide only the final surrogate performance estimation, which does not support the development of multi-fidelity NAS methods. To address this issue, the authors of [178] propose NAS-Bench-x11, which provides a surrogate method for predicting the full learning curve of an architecture. By using this method, the authors extend three existing benchmarks (NAS-Bench-101, NAS-Bench-301, and NAS-Bench-NLP) to include the full learning curve, enabling the development and evaluation of NAS methods that rely on the training curve.

NAS benchmark suites aim to unify various benchmarks by providing a common interface to query them, allowing for easy transferability and evaluation of NAS methods. The first suite, NAS-Bench-360 [184], introduced the use of 2D and 1D data sets in a unified way. NAS-Bench-Suite [172] is a more comprehensive suite, consisting of 28 different tasks. To further improve NAS-Bench-Suite, NAS-Bench-Suite-Zero [151] provides precomputed information about several zero-cost proxy estimators. These suite benchmarks make it easier for researchers to evaluate and compare NAS methods across different tasks and data sets, ultimately aiding in the development of more effective and generalizable NAS methods.



(a) Cell Structure (b) Outer-skeleton Structure

Figure 2.4: Representation of the NAS-Bench-201 and TransNAS-Bench-201 cell structure (a), and the pre-defined outer-skeleton (b). A cell is composed of 4 nodes and 6 edges, where edges perform operations from an input node and add them to a posterior node. There are 5 possible operations to be used as edges in NAS-Bench-201 and 4 in TransNAS-Bench-101. A cell is used as the building block of the outer-skeleton.

As many NAS methods use benchmarks to evaluate their performance, it is crucial to be certain that NAS benchmarks are well designed and that comparisons between methods when using NAS benchmarks are fair and well justified. The benchmark design plays a crucial role in determining the performance of the architecture pool and thus indirectly influences the success of NAS methods that learn the underlying operation preference. To evaluate this, we extensively study the importance of different operations in a benchmark search space and if it contains common patterns that jeopardize fair comparisons. In detail, we assess the impact that the operations, their combination, and their occurrences have in the final validation accuracy of an architecture in NAS-Bench-101, NAS-Bench-201, and TransNAS-Bench-101, as well as evaluating the search space distribution in terms of the architecture's accuracy and their correlation between data sets. The analysis shown in the following sections extends prior works that looked into proposing best practices for NAS methods and that evaluated search spaces in terms of their redundancy as well as the impact that the training protocols had in the final performance of a generated architecture [44, 40, 45, 172].

2.3.2 Studied Benchmark's Design

NAS-Bench-101

NAS-Bench-101 is a tabular benchmark used for evaluating the performance of different NAS methods. It consists of a cell-based search space comprising $423,624$ neural networks trained on CIFAR-10 for 108 epochs with 3 different weight initializations. All the architectures in this benchmark share a common outer-skeleton, and their differences lies in the cells placed in the outer-skeleton. The cells in NAS-Bench-101 are defined as DAG with a maximum of 7 nodes and 9 edges and are encoded as a 7×7 upper-triangular matrix. The operation pool consists of three possible layers: convolution 1×1 ($C_{1\times1}$), convolution 3×3 ($C_{3\times3}$), and 3×3 max pooling ($MP_{3\times3}$).

NAS-Bench-201

NAS-Bench-201 is a tabular cell-based search space with $15,625$ possible architectures, each composed of a fixed cell-based design with 5 possible operations: none, skip connection (SC), $C_{1\times1}$, $C_{3\times3}$, and average pooling 3×3 ($AP_{3\times3}$). The structure of a cell is

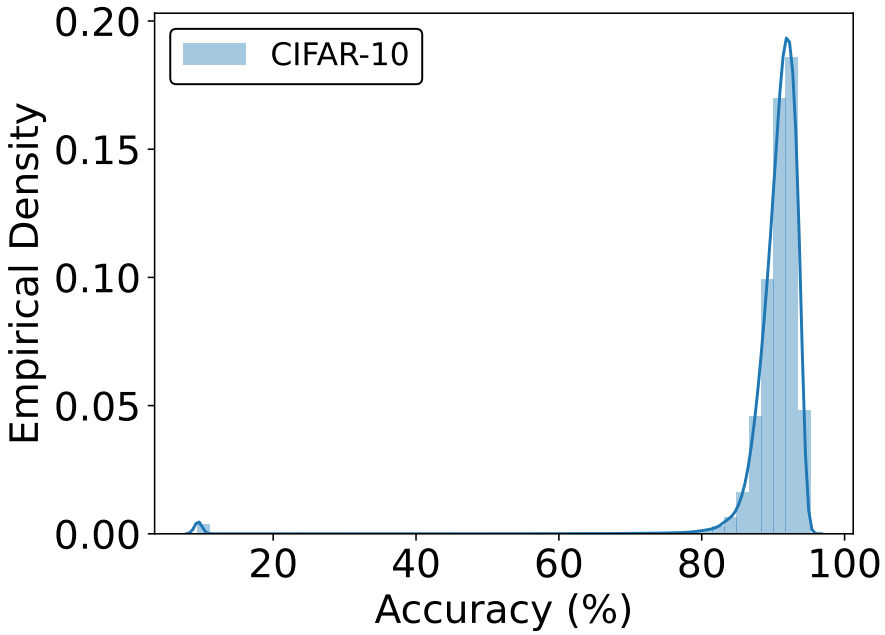


Figure 2.5: Normalized histogram of all architectures, based on their validation accuracy (%) in NAS-Bench-101.

represented as a DAG with 6 edges and 4 nodes, where edges represent operations. All architectures share a common outer-skeleton and are designed by interleaving searched cells with residual down-sampling blocks. The benchmark provides information about the learning curve and final performance of the architectures on CIFAR-10, CIFAR-100, and ImageNet16-120, recorded and provided as a tabular benchmark. Figure 2.4 depicts the representation of the cell structure and the outer-skeleton.

TransNAS-Bench-101

TransNAS-Bench-101 benchmark provides the performance of architectures across seven CV tasks, including classification, regression, pixel-level prediction, and self-supervised tasks. By having multiple tasks that can be evaluated using the same input, TransNAS-Bench-101 provides an easy setup to evaluate the generality and transferability of NAS across different tasks. This benchmark has two types of search spaces: a cell-based search space containing 4096 possible cells and a macro skeleton search space based on residual blocks containing 3256 architectures. For both search spaces, the pool of operations has four possible candidates: zeroize, SC, $C_{1\times1}$, and $C_{3\times3}$. TransNAS-Bench-101 provides tabular information about the training and performance of all architectures in the search space using the same training protocols and hyper-parameters within each task. In this study, we focus on the cell search space. The representation of the cell structure is depicted in Figure 2.4.

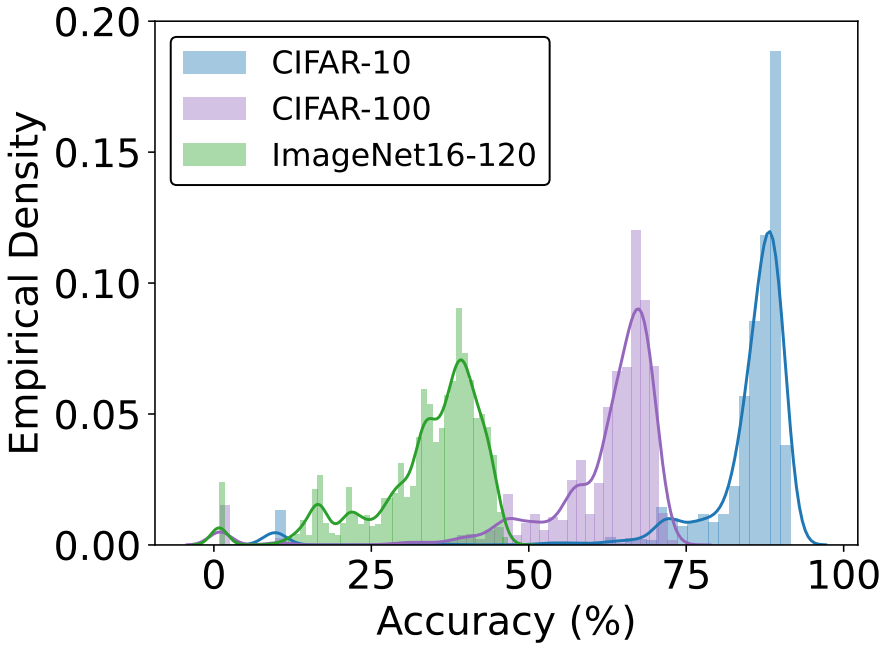


Figure 2.6: Normalized histogram of all architectures, based on their validation accuracy (%), for all the data sets in NAS-Bench-201.

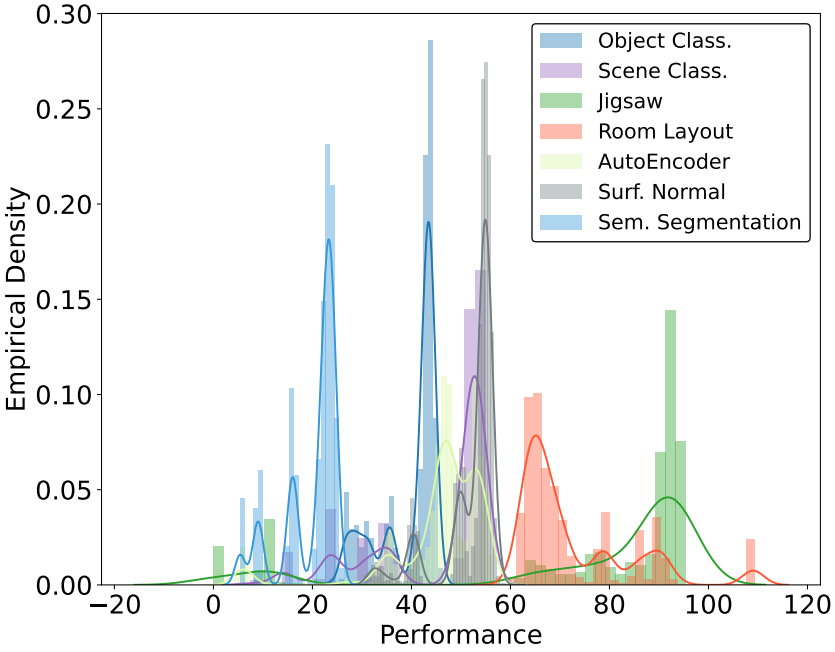


Figure 2.7: Normalized histogram of all architectures, based on their performance on all the data sets in TransNAS-Bench-101. Metrics were scaled to positive values to match accuracy ranges.

2.3.3 Evaluation

To assess the significance of the different operations in a search space across various benchmarks, we conduct several experiments: i) performance distribution analysis, where we look at the shape of the distribution of all architectures based on their final performance; ii) operation presence, where we evaluate architectures based on having at least once a given operation; iii) operation occurrence, where we evaluate how architectures

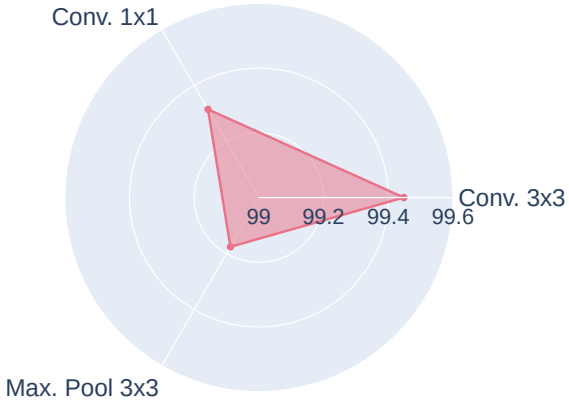
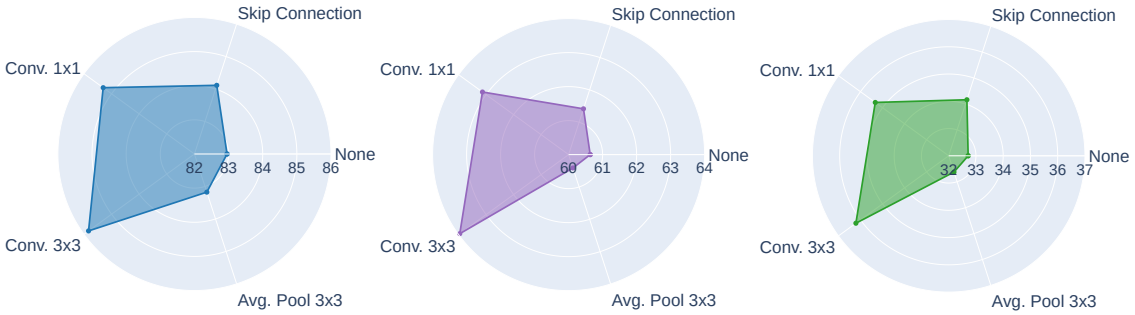


Figure 2.8: Mean accuracy (%) of all the architectures that have at least one operation of a given type in NAS-Bench-101.

perform based on having different occurrences of the same operation ($1, ..., N$; where N is the number of edges in a DAG); iv) operation positioning, where we evaluate an architecture based on having specific operations in specific positions of the DAG; v) operation combination, where architectures are evaluated based on combining operations 2 by 2; and the last experiment is evaluating the ranking of the architectures based on their performance on each data set and the inter-data set Kendall's tau correlation. The following sections present each evaluation in detail.

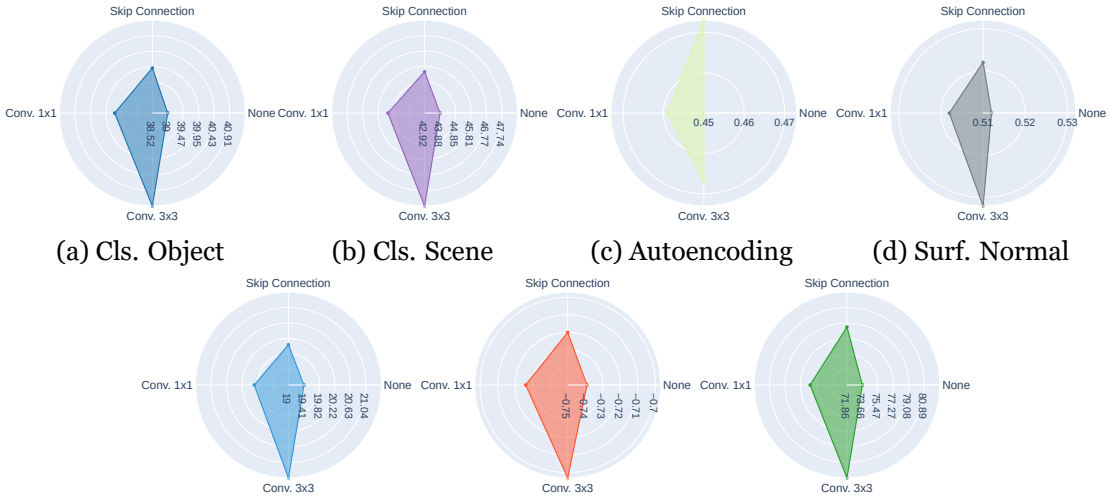
Performance Distribution

By providing tabular information about trained architectures, NAS benchmarks allow direct querying of an architecture performance while searching. Given the information on all architectures, we evaluated the distribution of the architecture's performance in NAS-Bench-101, NAS-Bench-201, and TransNAS-Bench-201. For NAS-Bench-101, Figure 2.5 shows a negatively skewed distribution of architecture performances, with a higher percentage (density) of architectures closer to the upper bound of validation accuracy. This indicates that there are many good architectures with slight differences, making it challenging to compare different NAS methods. Similarly, NAS-Bench-201 also shows a negatively skewed distribution, as seen in Figure 2.6. However, the skewed distributions are more evident on CIFAR data sets, suggesting that ImageNet16-120 is a more competitive data set with more likelihood of showing more differences between different NAS methods. Regarding TransNAS-Bench-201, shown in Figure 2.7, a large number of architectures are closer to the optimal performance in all tasks, with semantic segmentation and room layout being the tasks with the sparsest distribution. This observation suggests that it is important for NAS methods to indicate the results of RS when comparing on a specific benchmark to demonstrate how well a RS method can sample the search space. Additionally, when comparing gains in performance, it is important to compare with the most optimal architecture in each task to ensure that small increments in performance are significant if closer to the upper-bound.



(a) CIFAR-10 (b) CIFAR-100 (c) ImageNet16-120

Figure 2.9: Mean accuracy (%) of all the architectures that have at least one operation of a given type in NAS-Bench-201. Architectures with at least one convolutional layer show higher mean validation accuracies (%) in all data sets.



(a) Cls. Object (b) Cls. Scene (c) Autoencoding (d) Surf. Normal

(e) Sem. Segment. (f) Room Layout (g) Jigsaw

Figure 2.10: Mean performance of all the architectures with at least one operation of a given type in TransNAS-Bench-101. Architectures with at least one convolutional layer show higher mean performance in all data sets except in Autoencoding, where SC has the best mean performance.

Operation Presence

We now focus on evaluating the impact of each operation on the final architecture accuracy. First, we look at the mean performance of all architectures that contain at least one operation of a given type. This means that we evaluated all architectures that contain one or more $C_{3\times3}$ independently of the rest, and so on and so forth for all operations. What we found is that on all benchmarks and all data sets, convolutional layers yield the best mean performances, with the exception of the TransNAS-Bench-101 Autoencoding task, where SC has the best mean results. Radar graphs are depicted in Figures 2.8, 2.9 and 2.10 for NAS-Bench-101, NAS-Bench-201, and TransNas-Bench-101, respectively, which show the mean performance of all architectures with at least one operation of a given type. We justify the importance of convolutional layers due to the fact that cells were carefully designed and have fixed operations that perform pooling and residual connections, which benefit from receiving rich feature maps that can be obtained by performing convolutional operations. On the opposite spectrum, architectures with the operation $None$ have the worst

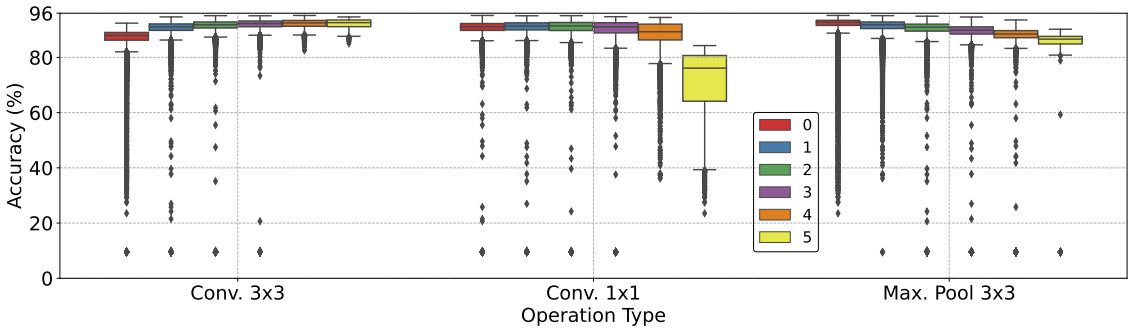


Figure 2.11: Boxplot analysis of the mean accuracy (%) of all architectures based on the number of occurrences of the different operations on a cell in NAS-Bench-101.

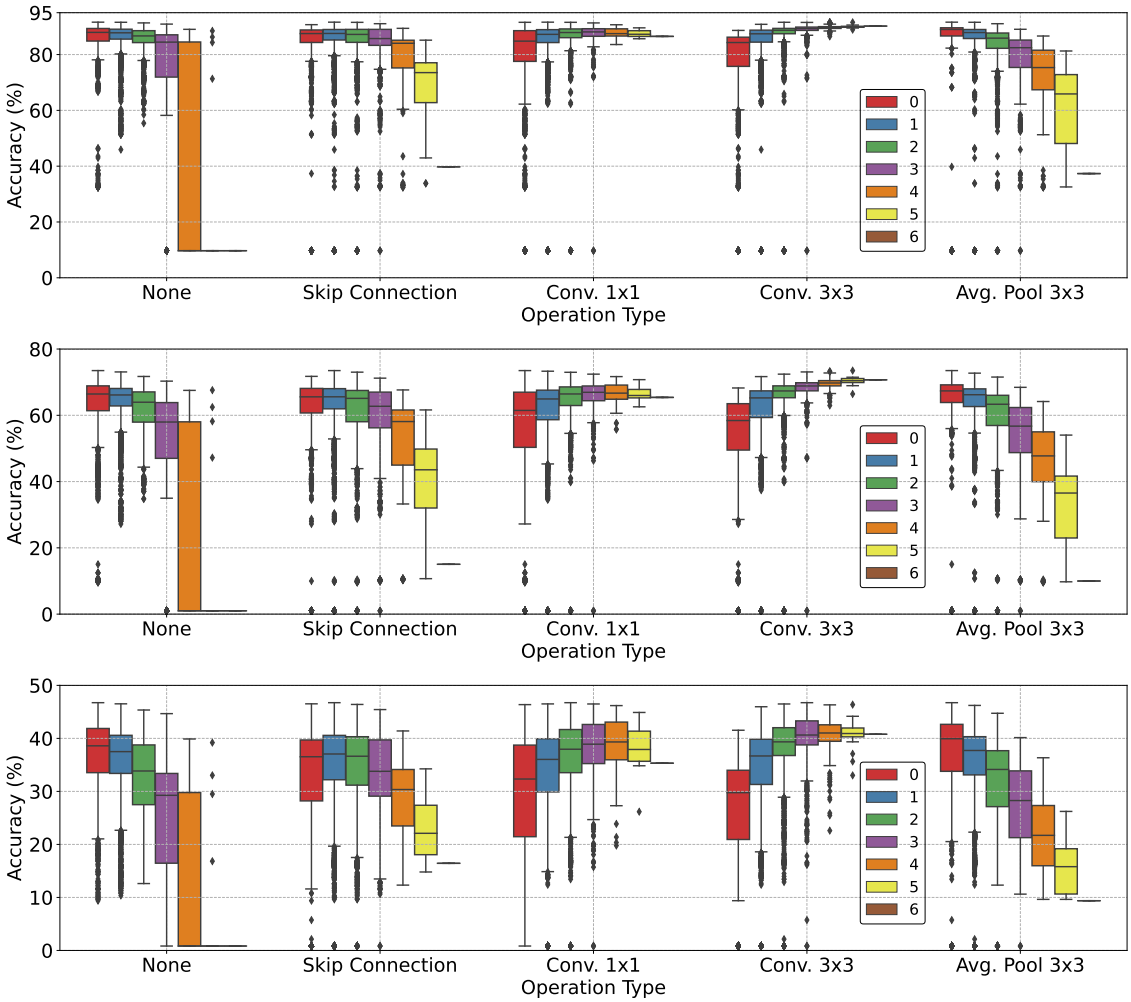
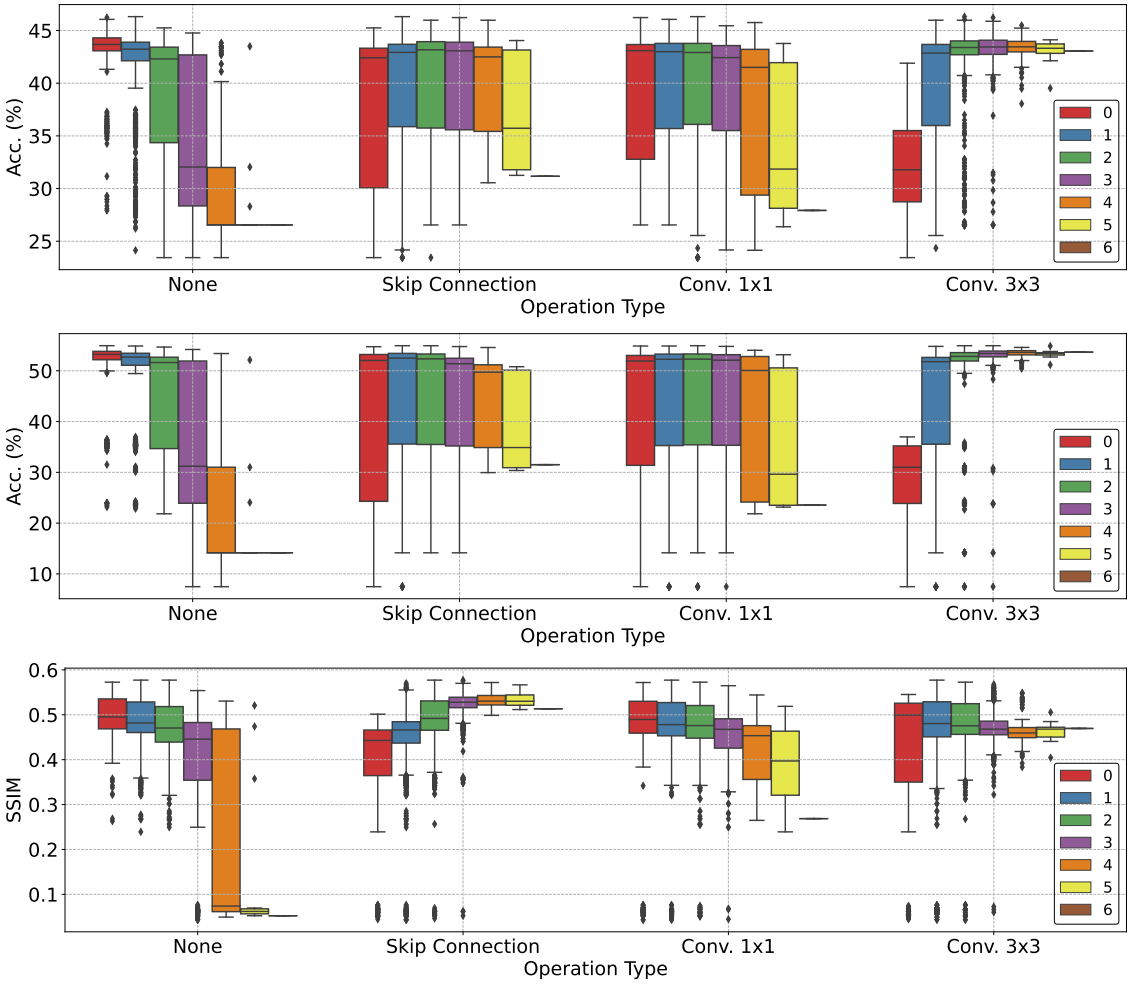


Figure 2.12: Boxplot analysis of the mean accuracy (%) of all architectures based on the number of occurrences of the different operations on a cell in NAS-Bench-201. The top figure presents results on CIFAR10, the middle one on C100 and the bottom one on ImageNet16-120.

results, which indicates that all edges and nodes on a small cell are important and benefit from having an operation that computes features.

Operation Occurrence

We also evaluated how architectures behave based on different operation occurrences. In NAS-Bench-101, we evaluated architectures based on a cell having 0 to 5 occurrences of a given operation and from 0 to 6 on NAS-Bench-201 and TransNas-Bench-101. Note that a cell with n occurrences of a given operation means that n operations in the cell are of the same type (e.g., $C_{3\times3}$), while the rest (until the cell is completed) are other operations (e.g., C_1). Boxplot graphs shown in Figures 2.11, 2.12 and 2.13 depict the minimum, first quartile, median, third quartile, maximum, and outlier values for all possible occurrences of all possible operations for NAS-Bench-101, NAS-Bench-201, and TransNAS-Bench-101 respectively. The results corroborate the hypothesis that convolutional layers yield better cells, as increasing the number of occurrences of such operations consistently increases mean performances. In all benchmarks, mean performances show that $C_{3\times3}$ is the optimal layer, followed by $C_{1\times1}$, and when present, SC. More, increasing occurrences of $None$ or pooling operations have a negative impact on the architecture's performance. This negative impact on the performance further corroborates the hypothesis that the structure of the cells is dependent on operations that are capable of creating rich features and that the fixed outer-skeleton and pre-defined operations perform enough data normalization for the data sets analyzed. These results indicate that NAS methods that exhaustively search



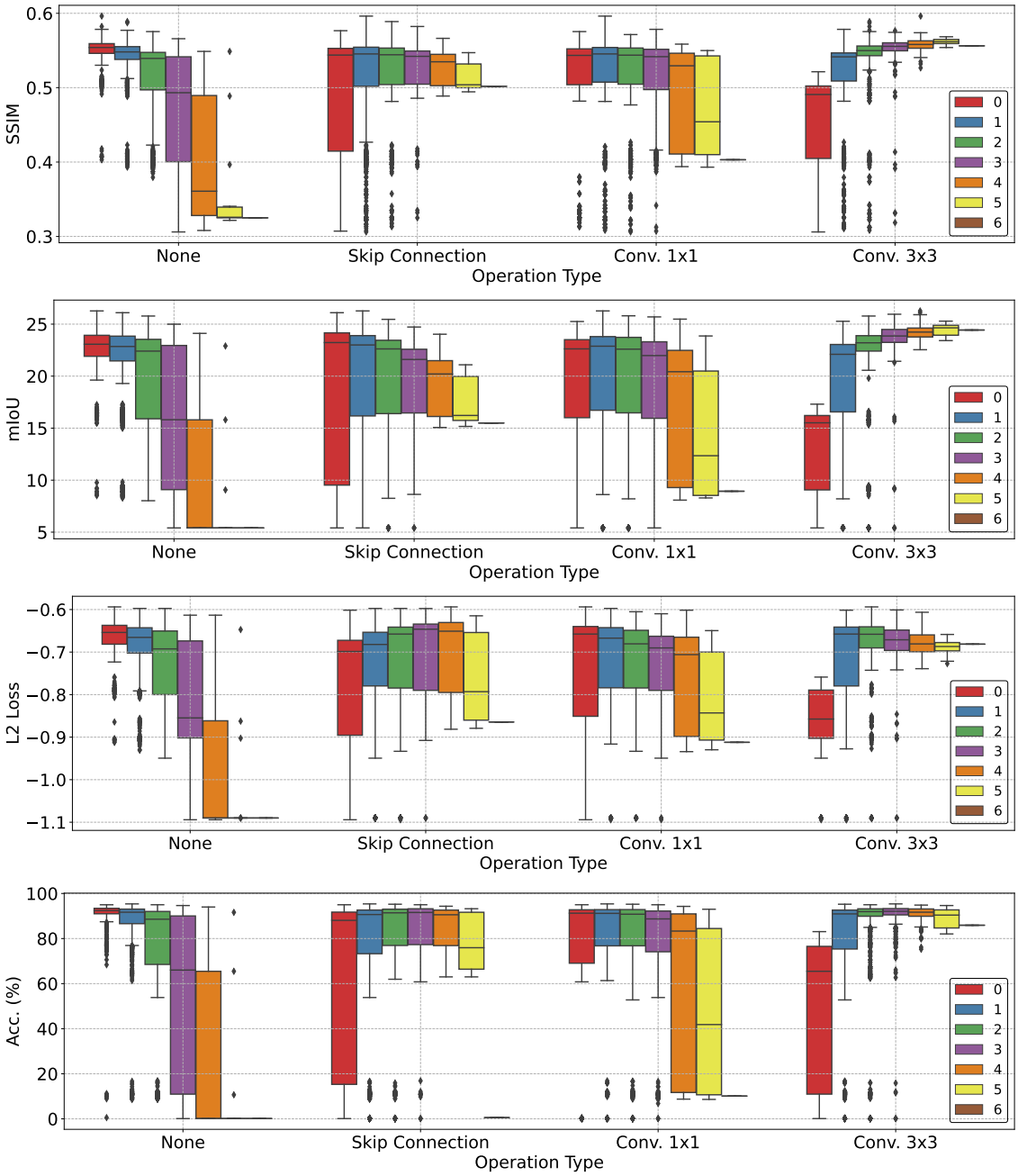


Figure 2.13: Boxplot analysis of the mean performance of all architectures based on the number of occurrences of the different operations on a cell in TransNAS-Bench-101. Tasks from top to bottom: object classification, scene classification, autoencoding, surface normal, semantic segmentation, room layout, and jigsaw.

only for convolutional layers will yield satisfactory results. Therefore, researchers should promote the indication of the performance of NAS methods over time and, if possible, representations of the designed cells, therefore promoting discussion if the method is indeed capable of searching or quickly falls into looking only for convolutional layers.

Operation Positioning

Given that convolutional layers, especially the ones with larger kernel sizes, lead on average to better results, we then focused on evaluating the performance of the architectures based on having an operation on specific edge positions. For this, we analyzed the mean performance and standard deviation of all architectures that contain an operation on a given edge position i, for $i = 1, ..., I$, where I is the maximum number of edges in a given benchmark. The rationale behind this evaluation is to further corroborate that the predefined structure of the cells and the outer-skeletons significantly impact the decisions that a NAS method makes while searching. NAS-Bench-101 allows operations in 5 out of the 7 positions, where the remaining 2 are reserved for input and output operations. The results in Table 2.2 show that architectures with $C_{3\times3}$ in any position yield the highest mean validation accuracies, followed by $C_{1\times1}$ with an exception for the second position, where $MP_{3\times3}$ has a higher mean validation accuracy, which is justified by the fact that the second position is the first to receive the input values from previous cells, thus operations that reduce the complexity of the input feature maps can have a direct impact in the final performance. Results for NAS-Bench-201, shown in Table 2.3, further show that convolutional layers are preferable, yielding the best results in all possible positions for all data sets. Results also indicate that SC layers show improved importance in positions that connect earlier nodes to later nodes in the cell, which benefit from residual connections. Also, $None$ operations tend to be better in the middle of the cell, and $AP_{3\times3}$ stays consistent throughout. These findings suggest that convolutions are preferable throughout the cell structure, but other operations, such as SC, can be important if placed strategically in specific cell positions. Similarly, in all data sets of TransNAS-Bench-101 except for Autoencoding, $C_{3\times3}$ yields the best results, whereas in Autoencoding, SC is the best operation to use in all positions (Table 2.4). SC and $C_{1\times1}$ behave similarly in TransNAS-Bench-101, achieving better results earlier in the cell, whereas $C_{3\times3}$ consistently performs throughout all possible positions in the cell. $None$ operation has the worst mean performances in all data sets, further promoting the idea that all cell operations have an impact on the final architecture's performance. Results in all data sets show that convolutional layers have the most positive impact on an architecture's performance and that, in some specific cases, a skip connection interleaved with convolutional layers can be fruitful.

Operation Combination

We further extended the evaluation of the operation's importance by looking at two consecutive combinations of operations as subsequent operations in the cell's structure. This analysis evaluates not only the importance of a single operation but how they behave if combined. For this, we looked at all possible combinations of two operations and calculated the mean performance and standard deviation of all architectures that contain such combinations. Results for NAS-Bench-101, shown in Table 2.5, further promote that $C_{3\times3}$ operations yield the best results, even when combined with any other operation. Furthermore, $C_{1\times1}$ follows the same pattern by achieving the next higher mean performances when combined with any convolutional operation, followed by $MP_{3\times3}$. In NAS-Bench-

Table 2.2: Mean validation accuracy (%) and standard deviation of all architectures that contain an operation on a specific edge in NAS-Bench-101.

Operation	Position				
	2	**3**	**4**	**5**	**6**
			CIFAR-10		
$C_{1\times1}$	89.64 ± 7.67	89.99 ± 6.36	90.13 ± 5.71	90.30 ± 5.00	90.58 ± 4.43
$C_{3\times3}$	$\mathbf{90.92} \pm 7.92$	$\mathbf{91.19} \pm 5.63$	$\mathbf{91.27} \pm 4.82$	$\mathbf{91.45} \pm 3.89$	$\mathbf{91.70} \pm 3.13$
$MP_{3\times3}$	90.18 ± 4.96	89.56 ± 8.62	89.34 ± 9.48	89.01 ± 10.22	88.22 ± 10.66

201, the same behavior is seen. Results in Table 2.6 show that combining convolutional layers results, on average, in the best performant architectures. From the table, it is also possible to see that combining operations with $None$ and $AP_{3\times3}$ leads to worse performant architectures and that a SC combined with convolutional layers, achieves results close to the best combination, further suggesting that in some specific cases, SC layers associated with convolutional layers result in good cell designs. Finally, the evaluation of the operation combinations on TransNAS-Bench-101 is shown in Table 2.7, where in all tasks, the combination of two $C_{3\times3}$ achieves the highest mean performances, with the exception for Autoencoding, where combining two SC has the best results. From the results, it is possible to see that for all tasks, except for Autoencoding, combining an operation with $None$ always yields the worst results, whereas combining any operation with $C_{3\times3}$ consistently achieves better results when compared with combining with other operation. Furthermore, the results show that in TransNAS-Bench-201, $C_{1\times1}$ and SC operations have similar results when combined with other operations, further suggesting that operations that create rich features are preferred, but when not present, operations that help reducing the complexity of the feature maps are good trade-offs, as they achieve good results by having a significant impact on the flatness of the loss landscape, as shown in previous works [185]. Results across all benchmarks corroborate the hypothesis that the operation pool and the structure of the cells (search space) have a high influence on the quality of the final architectures, which is shown by the consistent results that convolutional and skip connection layers have in all three benchmarks.

Cell's Ranking and Inter-data set Correlation

Lastly, we evaluate cells based on their ranking by sorting them using the performance on the validation set. The intuition behind looking at the ranking of the top cells is that it allows a direct analysis of the top cells based on their operation diversity and on benchmarks with more than one data set or task it allows an evaluation of how the different data sets correlate with each other. First, we look at the top 10 performant architectures in NAS-Bench-101. The cells are shown in Table 2.8, where on the left, the operations in each position are shown, and on the right, the associated rank and validation accuracy are depicted. The table shows that convolutional layers are the preferred operation on all positions, with the optimal cell entirely designed with $C_{3\times3}$ operations. Overall, convolutional

Table 2.3: Mean validation accuracy (%) and standard deviation of all architectures that contain an operation on a specific edge in NAS-Bench-201.

Operation	Position					
	1	2	3	4	5	6
CIFAR-10						
$None$	79.02 ± 20.17	81.02 ± 17.68	83.40 ± 13.34	75.04 ± 24.02	81.18 ± 17.68	78.99 ± 20.17
$C_{1\times1}$	86.62 ± 7.93	85.59 ± 9.93	84.51 ± 11.87	87.33 ± 2.41	85.37 ± 10.12	86.34 ± 8.24
$C_{3\times3}$	**87.61** ± 7.98	**86.31** ± 10.00	**84.93** ± 11.93	**88.31** ± 1.99	**85.99** ± 10.09	**87.15** ± 8.13
$AP_{3\times3}$	81.32 ± 11.08	81.91 ± 12.15	82.58 ± 13.38	81.46 ± 8.74	82.32 ± 12.09	82.11 ± 11.08
SC	83.81 ± 10.59	83.54 ± 11.85	82.95 ± 13.26	85.33 ± 7.38	83.50 ± 11.85	83.77 ± 10.62
CIFAR-100						
$None$	56.83 ± 17.16	58.90 ± 15.43	61.05 ± 12.23	54.47 ± 20.13	59.03 ± 15.41	56.80 ± 17.13
$C_{1\times1}$	64.95 ± 7.53	63.51 ± 9.47	62.37 ± 11.11	65.27 ± 4.30	63.34 ± 9.74	64.56 ± 8.00
$C_{3\times3}$	**66.35** ± 7.43	**64.58** ± 9.48	**63.03** ± 11.11	**66.87** ± 3.54	**64.35** ± 9.57	**65.85** ± 7.67
$AP_{3\times3}$	57.81 ± 11.64	58.82 ± 12.23	59.76 ± 12.97	57.96 ± 10.38	59.17 ± 12.16	58.61 ± 11.63
SC	60.47 ± 11.13	60.60 ± 11.91	60.20 ± 12.84	61.84 ± 9.40	60.52 ± 11.89	60.58 ± 11.24
ImageNet16-120						
$None$	29.77 ± 11.43	31.47 ± 10.86	33.22 ± 9.43	27.42 ± 12.38	31.74 ± 10.81	29.75 ± 11.38
$C_{1\times1}$	37.32 ± 6.65	35.75 ± 7.70	34.59 ± 8.87	37.83 ± 4.79	35.46 ± 8.18	36.54 ± 7.43
$C_{3\times3}$	**38.29** ± 6.68	**36.77** ± 7.75	**35.38** ± 8.98	**39.19** ± 2.98	**36.36** ± 8.13	**37.76** ± 7.31
$AP_{3\times3}$	30.10 ± 8.97	31.26 ± 9.33	32.62 ± 9.37	29.78 ± 8.93	31.82 ± 9.05	31.17 ± 8.57
SC	33.77 ± 8.17	33.70 ± 8.84	33.15 ± 9.25	34.74 ± 7.69	33.59 ± 8.82	33.75 ± 8.30

layers represent 84% of the operations present in the top-10 cells in NAS-Bench-101 (not considering the fixed input and output operations), thus suggesting that operations other than convolutional ones have a small positive impact on the final performance of the cell. This indicates that a NAS method that searches solely for convolutional layers will achieve high results without requiring exploring the search space for other operations. By looking at NAS-Bench-201 in Table 2.9, we see the same behavior, where convolutional layers represent 77% of the top cells considering all three data sets, having a representation of 75% in CIFAR-10, 80% in CIFAR-100 and 77% in ImageNet16-120. This further indicates that NAS-Bench-201 is heavily dependant on convolutional layers in top-scoring architectures. More, in all three data sets, it is possible to see patterns of combination between the possible two convolutional layers ($C_{1\times1}$ and $C_{3\times3}$), as well as skip connections on the fourth position of the cell structure. These skip connections allow residual connections from the input node to the output node, which, combined with different position permutations of the convolutional layers, yield the best results. Most of the best architectures for all three data sets contain convolutional layers as the first and fifth operations and an SC as the fourth. Hence, one could envision a NAS procedure where these three operations would be fixed, and only the remaining ones would be the target of the search, thus achieving excellent results across the three data sets, considering only a subspace related to half of the original number of operations. Furthermore, the cell's ranking between data sets suggests that top-scoring architectures on one data set will have good results on other data sets if transferred. This is more explicit between CIFAR-10 and CIFAR-100, as these two data sets have similarities at the level of the problem itself. To further study this, we evaluated the inter-data set ranking and cross-correlation between data sets in Figure 2.14. To

Table 2.4: Mean performance and standard deviation of all architectures that contain an operation on a specific edge in TransNAS-Bench-101.

Operation	Position					
	1	2	3	4	5	6
Cls. Object (Acc. (%) ↑)						
None	36.45 ± 6.89	37.95 ± 6.67	39.07 ± 6.24	37.10 ± 6.88	38.05 ± 6.66	36.50 ± 6.91
SC	40.36 ± 4.87	39.95 ± 5.33	39.57 ± 5.93	39.87 ± 5.36	39.67 ± 5.90	40.10 ± 5.60
$C_{1\times1}$	39.47 ± 5.99	39.17 ± 6.15	39.66 ± 5.90	38.36 ± 6.38	39.66 ± 5.85	40.06 ± 5.56
$C_{3\times3}$	**42.57** ± 3.73	**41.77** ± 4.68	**40.54** ± 5.51	**43.52** ± 0.64	**41.46** ± 4.60	**42.19** ± 3.69
Cls. Scene (Acc. (%) ↑)						
None	38.52 ± 14.57	41.67 ± 14.12	43.84 ± 12.68	40.30 ± 15.49	41.97 ± 13.81	38.69 ± 14.36
SC	46.27 ± 9.93	45.43 ± 10.86	44.92 ± 12.08	44.89 ± 9.79	44.74 ± 11.75	45.47 ± 11.16
$C_{1\times1}$	44.86 ± 11.85	44.33 ± 12.28	45.05 ± 12.16	42.79 ± 12.83	45.06 ± 11.90	45.97 ± 11.23
$C_{3\times3}$	**51.26** ± 7.54	**49.49** ± 9.54	**47.11** ± 11.43	**52.93** ± 0.84	**49.14** ± 9.71	**50.78** ± 7.73
Autoencoding (SSIM ↑)						
None	0.42 ± 0.14	0.43 ± 0.13	0.46 ± 0.11	0.38 ± 0.15	0.43 ± 0.13	0.42 ± 0.14
SC	**0.49** ± 0.07	**0.48** ± 0.09	0.46 ± 0.10	**0.53** ± 0.01	**0.48** ± 0.09	**0.48** ± 0.08
$C_{1\times1}$	0.46 ± 0.09	0.45 ± 0.09	0.46 ± 0.10	0.44 ± 0.06	0.46 ± 0.09	0.46 ± 0.08
$C_{3\times3}$	0.47 ± 0.07	0.47 ± 0.08	0.46 ± 0.10	0.48 ± 0.03	0.47 ± 0.08	0.47 ± 0.07
Surf. Normal (SSIM ↑)						
None	0.49 ± 0.07	0.50 ± 0.07	0.51 ± 0.06	0.49 ± 0.08	0.50 ± 0.07	0.49 ± 0.08
SC	0.53 ± 0.04	0.52 ± 0.05	0.52 ± 0.06	0.53 ± 0.03	0.52 ± 0.06	0.52 ± 0.05
$C_{1\times1}$	0.52 ± 0.06	0.51 ± 0.06	0.52 ± 0.06	0.51 ± 0.06	0.52 ± 0.06	0.52 ± 0.05
$C_{3\times3}$	**0.55** ± 0.04	**0.54** ± 0.05	**0.53** ± 0.06	**0.55** ± 0.01	**0.54** ± 0.05	**0.54** ± 0.04
Sem. Segment. (mIoU ↑)						
None	17.10 ± 6.68	18.48 ± 6.44	19.36 ± 5.78	17.81 ± 7.25	18.55 ± 6.38	17.12 ± 6.69
SC	20.32 ± 4.08	20.00 ± 4.64	19.81 ± 5.48	20.00 ± 3.31	19.67 ± 5.24	20.01 ± 4.91
$C_{1\times1}$	19.45 ± 5.52	19.36 ± 5.73	19.88 ± 5.53	18.44 ± 6.16	19.90 ± 5.44	20.24 ± 5.14
$C_{3\times3}$	**22.92** ± 3.57	**21.95** ± 4.42	**20.75** ± 5.18	**23.54** ± 0.88	**21.67** ± 4.38	**22.42** ± 3.52
Room Layout (L2 Loss ↓)						
None	0.79 ± 0.15	0.76 ± 0.14	0.74 ± 0.12	0.79 ± 0.16	0.76 ± 0.14	0.79 ± 0.15
SC	0.71 ± 0.10	0.72 ± 0.11	0.73 ± 0.12	0.72 ± 0.10	0.72 ± 0.11	0.71 ± 0.11
$C_{1\times1}$	0.73 ± 0.11	0.73 ± 0.11	0.73 ± 0.11	0.75 ± 0.10	0.73 ± 0.11	0.72 ± 0.10
$C_{3\times3}$	**0.68** ± 0.08	**0.69** ± 0.10	**0.71** ± 0.11	**0.66** ± 0.02	**0.70** ± 0.09	**0.69** ± 0.08
Jigsaw (Acc. (%) ↑)						
None	62.08 ± 35.94	68.13 ± 33.58	74.14 ± 30.10	62.40 ± 36.16	69.08 ± 33.38	62.27 ± 35.98
SC	83.16 ± 18.51	80.60 ± 22.42	76.46 ± 28.29	84.10 ± 11.54	77.31 ± 27.62	79.41 ± 25.60
$C_{1\times1}$	73.84 ± 29.95	73.30 ± 30.57	76.42 ± 28.18	67.65 ± 34.49	77.18 ± 27.43	79.12 ± 25.49
$C_{3\times3}$	**87.21** ± 17.53	**84.26** ± 22.14	**79.27** ± 26.43	**92.14** ± 1.77	**82.71** ± 22.09	**85.50** ± 17.72

Table 2.5: Mean validation accuracy (%) and standard deviation of all architectures, evaluated based on the combination of 2 operations in NAS-Bench-101.

Operation	*Input*	$C_{1\times1}$	$C_{3\times3}$	$MP_{3\times3}$	*Output*
Input	–	89.65 ± 7.60	90.92 ± 7.90	90.19 ± 4.83	83.16 ± 0.00
$C_{1\times1}$	–	89.24 ± 6.59	91.46 ± 3.77	89.29 ± 7.58	90.64 ± 4.40
$C_{3\times3}$	–	91.48 ± 3.98	91.66 ± 4.10	90.32 ± 8.36	**91.83** ± 3.00
$MP_{3\times3}$	–	90.00 ± 5.19	91.06 ± 5.47	87.67 ± 12.22	88.36 ± 10.46
Output	–	–	–	–	–

Table 2.6: Mean validation accuracy (%) and standard deviation of all architectures, evaluated based on the combination of 2 operations in NAS-Bench-201.

Operation	$None$	$C_{1\times1}$	$C_{3\times3}$	$AP_{3\times3}$	SC
			CIFAR-10		
$None$	70.78 ± 28.43	83.63 ± 14.53	84.65 ± 14.65	79.38 ± 15.28	82.10 ± 15.05
$C_{1\times1}$	83.73 ± 14.52	87.00 ± 6.60	87.80 ± 6.59	84.77 ± 7.58	86.12 ± 7.05
$C_{3\times3}$	84.80 ± 14.66	87.84 ± 6.59	$\mathbf{88.10} \pm 6.56$	85.55 ± 7.65	86.87 ± 7.11
$AP_{3\times3}$	79.10 ± 15.22	84.57 ± 7.69	85.36 ± 7.67	79.31 ± 12.09	81.25 ± 11.96
SC	82.11 ± 15.01	86.10 ± 7.14	86.78 ± 7.19	81.38 ± 11.93	82.76 ± 11.50
			CIFAR-100		
$None$	49.70 ± 23.23	61.65 ± 12.68	63.24 ± 12.75	56.44 ± 13.81	59.26 ± 13.51
$C_{1\times1}$	61.81 ± 12.63	65.42 ± 6.45	66.62 ± 6.33	62.04 ± 8.37	63.55 ± 7.75
$C_{3\times3}$	63.42 ± 12.77	66.65 ± 6.35	$\mathbf{67.12} \pm 6.25$	63.25 ± 8.29	64.73 ± 7.66
$AP_{3\times3}$	56.17 ± 13.75	61.83 ± 8.49	63.03 ± 8.35	55.20 ± 12.89	57.30 ± 12.87
SC	59.15 ± 13.42	63.51 ± 7.84	64.66 ± 7.72	57.41 ± 12.82	58.90 ± 12.50
			ImageNet16-120		
$None$	24.91 ± 13.43	33.68 ± 9.34	34.94 ± 9.60	28.17 ± 10.10	31.91 ± 9.86
$C_{1\times1}$	33.89 ± 9.23	37.67 ± 6.10	$\mathbf{39.11} \pm 5.85$	34.09 ± 7.46	35.90 ± 6.78
$C_{3\times3}$	35.38 ± 9.44	39.02 ± 5.93	38.91 ± 5.78	35.25 ± 7.36	37.44 ± 6.66
$AP_{3\times3}$	27.74 ± 10.10	33.82 ± 7.69	35.04 ± 7.51	28.10 ± 8.96	30.89 ± 9.02
SC	31.69 ± 9.70	35.98 ± 6.95	37.46 ± 6.71	31.04 ± 8.88	32.80 ± 8.44

analyze the correlation between data sets, we used Kendall's τ_b correlation as:

$$\tau_b = \frac{(P - Q)}{\sqrt{(P + Q + X) \times (P + Q + Y)}} \quad (2.1)$$

where P is the number of concordant pairs, Q the number of discordant pairs, X the number of ties only on the x-variable, and Y the number of ties on the y-variable [186].

In Figure 2.14-(a), we present the top-50 architectures on CIFAR-10 and their rankings on CIFAR-100 and ImageNet16-120, and on the right (b), the correlation between data sets in terms of the performance of all architectures. The results show a high positive correlation between all data sets, particularly between CIFAR data sets. These results suggest that NAS-Bench-201 should not be used alone when evaluating the generability of a NAS method in different data sets. More, when using NAS-Bench-201, it is paramount to show results by directly searching in each data set, and since ImageNet16-120 has the lowest correlation with other data sets, it allows a better understanding of how good a NAS method is. As for the position of the operations in the top cells of TransNAS-Bench-101, we see in Table 2.10 that convolutional layers have a smaller representation across all tasks, with 58% of the possible operations being convolutional layers, 31% SC and 11% $None$. Looking at specific tasks, semantic segmentation has the highest percentage of convolutional layers, with 80% and 66% respectively, while autoencoding, room layout, and

Table 2.7: Mean performance metric and standard deviation of all architectures, evaluated based on combination of 2 operations in TransNAS-Bench-101.

Operation	$None$	SC	$C_{1\times1}$	$C_{3\times3}$
		Cls. Object (Acc. (%) ↑)		
$None$	34.43 ± 7.07	38.11 ± 6.23	37.42 ± 6.69	40.94 ± 5.11
SC	38.05 ± 6.12	39.69 ± 5.29	39.60 ± 5.51	42.20 ± 4.03
$C_{1\times1}$	37.29 ± 6.71	39.34 ± 5.80	38.72 ± 6.21	41.71 ± 4.51
$C_{3\times3}$	41.18 ± 5.09	42.20 ± 4.20	41.79 ± 4.49	$\mathbf{42.74} \pm 3.05$
		Cls. Scene (Acc. (%) ↑)		
$None$	34.30 ± 15.69	41.74 ± 12.63	40.86 ± 13.75	48.15 ± 11.15
SC	41.79 ± 12.44	44.58 ± 10.25	44.60 ± 10.98	50.03 ± 7.90
$C_{1\times1}$	40.56 ± 13.72	44.12 ± 11.43	43.40 ± 12.44	49.59 ± 9.11
$C_{3\times3}$	48.52 ± 10.89	49.93 ± 8.19	49.70 ± 9.15	$\mathbf{51.79} \pm 6.29$
		Autoencoding (SSIM ↑)		
$None$	0.36 ± 0.17	0.47 ± 0.11	0.43 ± 0.11	0.45 ± 0.10
SC	0.47 ± 0.12	$\mathbf{0.51} \pm 0.06$	0.49 ± 0.07	0.50 ± 0.06
$C_{1\times1}$	0.42 ± 0.11	0.48 ± 0.08	0.45 ± 0.08	0.46 ± 0.07
$C_{3\times3}$	0.45 ± 0.10	0.50 ± 0.07	0.46 ± 0.07	0.47 ± 0.06
		Surf. Normal (SSIM ↑)		
$None$	0.46 ± 0.09	0.51 ± 0.06	0.50 ± 0.07	0.53 ± 0.06
SC	0.51 ± 0.06	0.53 ± 0.04	0.52 ± 0.05	0.54 ± 0.03
$C_{1\times1}$	0.49 ± 0.07	0.53 ± 0.05	0.51 ± 0.06	0.54 ± 0.05
$C_{3\times3}$	0.53 ± 0.06	0.54 ± 0.04	0.54 ± 0.04	$\mathbf{0.55} \pm 0.03$
		Sem. Segment. (mIoU ↑)		
$None$	15.18 ± 7.36	18.55 ± 5.55	17.91 ± 6.48	21.41 ± 5.09
SC	18.63 ± 5.39	19.71 ± 4.03	19.58 ± 4.79	21.93 ± 3.38
$C_{1\times1}$	17.66 ± 6.50	19.36 ± 5.06	18.81 ± 5.87	21.80 ± 4.33
$C_{3\times3}$	21.60 ± 5.09	21.99 ± 3.49	21.95 ± 4.36	$\mathbf{23.12} \pm 2.95$
		Room Layout (L2 Loss ↓)		
$None$	0.84 ± 0.17	0.75 ± 0.13	0.77 ± 0.13	0.71 ± 0.11
SC	0.75 ± 0.13	0.72 ± 0.10	0.72 ± 0.10	0.68 ± 0.08
$C_{1\times1}$	0.77 ± 0.13	0.73 ± 0.10	0.74 ± 0.11	0.69 ± 0.08
$C_{3\times3}$	0.71 ± 0.11	0.68 ± 0.08	0.69 ± 0.08	$\mathbf{0.68} \pm 0.06$
		Jigsaw (Acc. (%) ↑)		
$None$	50.57 ± 38.75	72.82 ± 29.02	65.32 ± 34.60	79.96 ± 25.76
SC	73.50 ± 27.72	81.67 ± 19.14	78.76 ± 24.37	87.38 ± 15.53
$C_{1\times1}$	63.77 ± 35.21	76.62 ± 26.85	70.90 ± 31.92	83.43 ± 22.59
$C_{3\times3}$	80.98 ± 25.54	87.20 ± 16.32	83.96 ± 22.50	$\mathbf{88.34} \pm 14.39$

object classification have the lowest, with 42%, 50% and 51% respectively. More, in those in which the percentage of convolutional layers is lower, SC represents a larger part, with approximately 42% in all three. As TransNAS-Bench-101 uses the same cell structure as NAS-Bench-201, we see the same pattern in terms of operation position, where SC is frequently present in the fourth position to create residual connections between the input

Table 2.8: Top 10 cells on NAS-Bench-101 based on the validation accuracy (%).

Operations							CIFAR-10	
1	**2**	**3**	**4**	**5**	**6**	**7**	**Rank**	**Val. Acc (%)**
$Input$	$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$Output$	-	**1**	95.18%
$Input$	$C_{1\times1}$	$MP_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$Output$	**2**	95.11%
$Input$	$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$Output$	-	**3**	95.11%
$Input$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$Output$	-	**4**	95.07%
$Input$	$MP_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$C_{1\times1}$	$Output$	**5**	95.06%
$Input$	$C_{3\times3}$	$MP_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$Output$	**6**	95.04%
$Input$	$C_{3\times3}$	$MP_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$Output$	-	**7**	95.03%
$Input$	$C_{3\times3}$	$MP_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$Output$	-	**8**	94.99%
$Input$	$MP_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$Output$	-	**9**	94.95%
$Input$	$MP_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$Output$	-	**10**	94.94 %

and output nodes, and convolutional layers are the preferred operation in the first positions. However, the evaluated tasks allow for a more diverse use of operations, requiring NAS methods to be more general and capable of learning which patterns are essential for each task, which is further justified by the inter-task ranking, where we see that a good architecture in a specific task will not guarantee a good result when transferred to other tasks. This is further studied in Figure 2.15, where on the left, the top-50 architectures on object classification appear as top architectures in other tasks. On the right figure, Kendall's Tau correlation suggests that the tasks are not as correlated as NAS-Bench-201, and Autoencoding is the task with the lowest correlation with any other. By having a lower correlation between tasks, TransNAS-Bench-101 becomes an interesting benchmark to evaluate the generability of NAS methods, as transferring architectures searched on a single task to others does not directly translate to good results. This is particularly important, as it allows evaluating if a NAS method is capable of generating suitable architectures for different data sets or if it only generates good architectures for data sets that are very similar.

2.3.4 Best Practice Suggestions

The presented results shed insights on the importance of the different operations in a NAS search space, how they are influenced by their positioning, their combinations, and how this influences the performance of an architecture. Based on those, we consider that they have a two-fold application: 1) on how NAS methods should be evaluated, and 2) what researchers should take into consideration when designing and creating new NAS benchmarks. The following suggestions extend those proposed by NAS researchers [44, 40, 45, 172], and we do believe that they contribute to fairer comparisons and better NAS designs.

Table 2.9: Top-10 cells for each one of the data sets on NAS-Bench-201. Validation accuracy (%) is given only for the top-10 cells in each data set, while the ranking is also presented for the architectures outside the 10 best, allowing inter-data set comparison.

Operations						**Rank / Val. Acc.** (%)					
1	**2**	**3**	**4**	**5**	**6**	**C10**		**C100**		**IN16-120**	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{1\times1}$	**1**	91.61%	11		74	
$AP_{3\times3}$	$C_{3\times3}$	$None$	SC	$C_{1\times1}$	$C_{3\times3}$	**2**	91.57%	202		1083	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{3\times3}$	**3**	91.55%	**1**	73.49%	12	
$C_{3\times3}$	$None$	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{1\times1}$	**4**	91.55%	52		87	
$C_{3\times3}$	$C_{1\times1}$	SC	SC	$C_{3\times3}$	$C_{1\times1}$	**5**	91.54%	219		112	
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{3\times3}$	**6**	91.53%	**3**	73.13%	14	
$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	SC	$C_{3\times3}$	$C_{1\times1}$	**7**	91.52%	31		20	
$C_{3\times3}$	$C_{3\times3}$	$None$	SC	$C_{3\times3}$	$C_{1\times1}$	**8**	91.51%	38		188	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{3\times3}$	**9**	91.50%	**2**	73.31%	66	
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{3\times3}$	**10**	91.48%	21		16	
$C_{3\times3}$	$C_{3\times3}$	$None$	SC	$C_{1\times1}$	$C_{3\times3}$	20		**4**	73.09%	94	
$C_{3\times3}$	$C_{3\times3}$	SC	SC	$C_{3\times3}$	$C_{1\times1}$	128		**5**	73.02%	253	
$C_{1\times1}$	$C_{3\times3}$	$C_{1\times1}$	SC	$C_{3\times3}$	$C_{3\times3}$	27		**6**	72.98%	134	
$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	SC	$C_{1\times1}$	$C_{3\times3}$	31		**7**	72.96%	142	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{1\times1}$	12		**8**	72.95%	18	
$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{3\times3}$	18		**9**	72.86%	85	
$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{3\times3}$	23		**10**	72.77%	77	
$C_{3\times3}$	$C_{1\times1}$	$C_{1\times1}$	SC	$C_{3\times3}$	$C_{3\times3}$	32		19		**1**	46.73%
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{1\times1}$	11		12		**2**	46.56%
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$C_{1\times1}$	$None$	$C_{3\times3}$	414		270		**3**	46.52%
$C_{3\times3}$	$None$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{1\times1}$	48		40		**4**	46.50%
$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{1\times1}$	15		26		**5**	46.50%
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{1\times1}$	29		75		**6**	46.49%
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$C_{1\times1}$	SC	$C_{1\times1}$	753		728		**7**	46.47%
$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{1\times1}$	280		55		**8**	46.45%
$C_{3\times3}$	SC	$C_{3\times3}$	SC	$C_{1\times1}$	$C_{3\times3}$	39		53		**9**	46.40%
$C_{1\times1}$	$C_{1\times1}$	$C_{3\times3}$	SC	$C_{3\times3}$	$C_{3\times3}$	83		22		**10**	46.38%

Evaluating NAS Methods

Our findings suggest that specific operations, such as $C_{3\times3}$, have a considerable impact on the final performance of an architecture. To promote a thorough evaluation of NAS methods, we recommend that a NAS evaluation method consider:

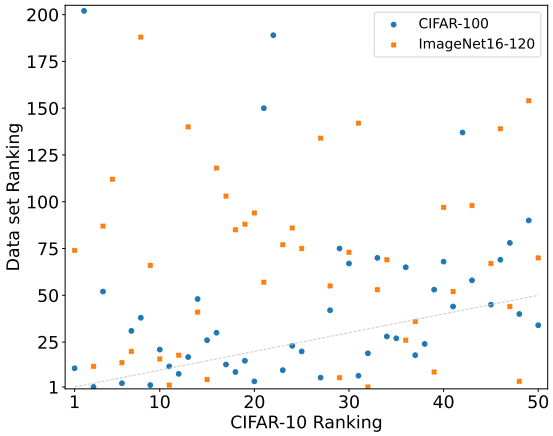
- Evaluating NAS methods over time, reporting performance as an average and standard deviation at regular intervals to allow the analysis of the performance and trainability of the method.
- Analyzing the cell's design to determine if a NAS method focuses only on a small subset of the operation pool, which can hinder optimal results. If this is done over time, it can provide information about how well a NAS method generalizes and learns.
- Evaluating NAS methods on multiple benchmarks, not just a single one, especially if the analyzed task allows most architectures to attain high performance, such as the CIFAR-10 data set. This corroborates past suggestions and findings [44, 40, 45, 172],

Table 2.10: Top cells for each task on NAS-Bench-201. Performance is given only for the top-10 cells in each task while ranking is also presented for architectures outside the top 10, promoting inter-task comparison.

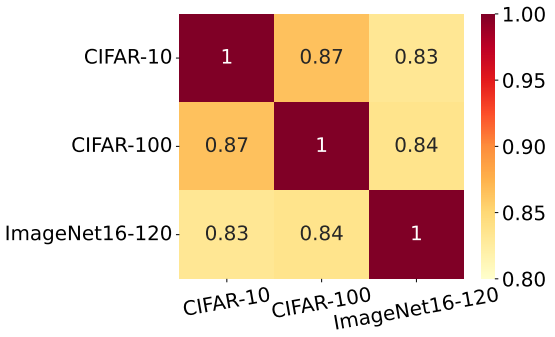
Operations						Rank / Performance											
						Cls. Object		Cls. Scene		Autoencoding		Surf. Normal		Sem. Segment.		Room Layout		Jigsaw	
1	2	3	4	5	6														
$C_{3\times3}$	*None*	$C_{3\times3}$	*SC*	$C_{1\times1}$	$C_{1\times1}$	**1**	46.32	534		791		172		582		2072		126	
$C_{3\times3}$	*SC*	$C_{3\times3}$	*SC*	*SC*	$C_{3\times3}$	**2**	46.23	469		1124		495		1645		1057		1460	
$C_{3\times3}$	*SC*	*SC*	$C_{1\times1}$	$C_{3\times3}$	*SC*	**3**	46.06	920		116		310		1191		736		111	
$C_{3\times3}$	*SC*	$C_{1\times1}$	$C_{1\times1}$	*None*	*SC*	**4**	45.98	369		1367		1331		2153		316		477	
$C_{3\times3}$	*SC*	*SC*	*SC*	$C_{3\times3}$	*SC*	**5**	45.98	292		390		701		2259		**1**	0.59	146	
SC	$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	*SC*	$C_{3\times3}$	**6**	45.88	115		768		332		519		752		1567	
$C_{3\times3}$	*SC*	$C_{3\times3}$	*SC*	*None*	*SC*	**7**	45.86	543		49		234		2504		74		150	
$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	*SC*	$C_{3\times3}$	*SC*	**8**	45.86	331		85		437		553		748		534	
SC	$C_{3\times3}$	*None*	$C_{1\times1}$	*SC*	$C_{3\times3}$	**9**	45.86	178		344		192		1442		1595		240	
$C_{3\times3}$	$C_{3\times3}$	*SC*	*SC*	$C_{3\times3}$	*SC*	**10**	45.81	297		1206		902		1536		115		778	
$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	*SC*	$C_{3\times3}$	$C_{1\times1}$	61		**1**	54.94	218		52		800		1647		423	
$C_{3\times3}$	*SC*	$C_{3\times3}$	*SC*	$C_{1\times1}$	$C_{1\times1}$	148		**2**	54.93	958		1728		2269		13		974	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	*SC*	*None*	$C_{1\times1}$	213		**3**	54.87	565		307		1871		1836		344	
$C_{3\times3}$	$C_{3\times3}$	*SC*	*SC*	$C_{3\times3}$	$C_{3\times3}$	572		**4**	54.87	1411		266		1274		2191		32	
$C_{3\times3}$	*None*	*SC*	$C_{1\times1}$	$C_{3\times3}$	*SC*	1129		**5**	54.86	3019		187		117		2214		790	
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	*None*	*SC*	*SC*	1132		**6**	54.86	3020		188		118		2215		791	
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$C_{1\times1}$	*SC*	*SC*	1490		**7**	54.85	2724		547		796		1772		153	
$C_{3\times3}$	*SC*	$C_{1\times1}$	$C_{1\times1}$	*SC*	$C_{1\times1}$	93		**8**	54.80	2296		1834		229		1298		197	
$C_{3\times3}$	$C_{3\times3}$	*None*	$C_{1\times1}$	*SC*	$C_{3\times3}$	928		**9**	54.79	1906		339		23		1058		1323	
$C_{3\times3}$	*SC*	$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	*SC*	381		**10**	54.77	127		361		1287		310		123	
SC	$C_{1\times1}$	*None*	$C_{3\times3}$	*SC*	*None*	529		2545		**1**	0.58	1355		2367		995		1839	
SC	$C_{1\times1}$	*SC*	$C_{3\times3}$	*SC*	*None*	530		2546		**2**	0.58	1356		2368		996		1840	
SC	$C_{3\times3}$	$C_{1\times1}$	*SC*	*None*	*SC*	1704		2619		**3**	0.57	1192		2454		380		502	
$C_{1\times1}$	*SC*	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	*SC*	390		1208		**4**	0.57	607		2598		11		1119	
$C_{3\times3}$	*SC*	*SC*	*SC*	*None*	*SC*	1042		2754		**5**	0.57	2432		2535		365		2312	
$C_{1\times1}$	*SC*	$C_{3\times3}$	*SC*	$C_{3\times3}$	*None*	2314		2159		**6**	0.57	2205		1612		1161		836	
SC	*SC*	$C_{3\times3}$	*SC*	*None*	*SC*	1445		2821		**7**	0.57	2549		2789		234		953	
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	*SC*	$C_{1\times1}$	*None*	313		348		**8**	0.57	906		2042		1241		30	
SC	$C_{1\times1}$	$C_{3\times3}$	*SC*	*SC*	$C_{3\times3}$	145		790		**9**	0.57	1894		2647		21		431	
SC	$C_{3\times3}$	$C_{3\times3}$	*None*	*SC*	$C_{1\times1}$	468		404		**10**	0.57	1431		1781		1406		827	
$C_{3\times3}$	$C_{3\times3}$	*SC*	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	1011		203		2933		**1**	0.60	97		2643		1089	
$C_{3\times3}$	$C_{3\times3}$	*None*	*SC*	*SC*	$C_{1\times1}$	173		206		384		**2**	0.59	685		242		503	
SC	$C_{3\times3}$	*None*	*SC*	$C_{1\times1}$	$C_{3\times3}$	96		96		1102		**3**	0.59	1847		38		1099	
$C_{3\times3}$	*SC*	$C_{3\times3}$	$C_{1\times1}$	*SC*	*SC*	20		76		246		**4**	0.58	1539		364		871	
$C_{3\times3}$	$C_{1\times1}$	*SC*	*SC*	$C_{1\times1}$	$C_{1\times1}$	598		766		141		**5**	0.58	1937		1069		625	
$C_{3\times3}$	*SC*	$C_{3\times3}$	*SC*	*None*	$C_{1\times1}$	672		1203		1352		**6**	0.58	1385		1221		275	
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{1\times1}$	2488		51		2304		**7**	0.58	12		2368		1471	
SC	$C_{3\times3}$	*None*	*SC*	*None*	$C_{3\times3}$	663		15		364		**8**	0.58	864		240		**9**	94.90
SC	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	*SC*	*SC*	614		2518		825		**9**	0.57	629		610		1129	
$C_{3\times3}$	*None*	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	*SC*	953		785		2451		**10**	0.57	414		2456		2023	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	*SC*	$C_{3\times3}$	845		24		1782		101		**1**	26.27	2250		781	
$C_{1\times1}$	*None*	$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	1448		1324		2708		1187		**2**	26.10	2622		1910	
$C_{3\times3}$	*None*	$C_{3\times3}$	*SC*	$C_{1\times1}$	$C_{3\times3}$	30		224		912		342		**3**	25.95	2184		175	
$C_{3\times3}$	*SC*	$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	658		210		3139		44		**4**	25.91	1464		136	
$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$C_{3\times3}$	*None*	*SC*	362		540		1928		206		**5**	25.83	1693		1957	
$C_{3\times3}$	*None*	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	2081		788		2778		197		**6**	25.82	1923		2194	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	$C_{1\times1}$	1802		620		2358		36		**7**	25.80	2713		2172	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{3\times3}$	*SC*	703		507		2552		367		**8**	25.79	2617		1598	
$C_{3\times3}$	*None*	$C_{3\times3}$	*None*	$C_{1\times1}$	$C_{1\times1}$	2848		2103		3500		1208		**9**	25.78	2697		2822	
$C_{3\times3}$	$C_{3\times3}$	$C_{3\times3}$	*None*	$C_{1\times1}$	$C_{1\times1}$	2707		1878		3260		971		**10**	25.76	2573		2821	
$C_{3\times3}$	$C_{1\times1}$	*None*	*SC*	$C_{3\times3}$	*None*	344		59		510		252		1391		**2**	0.60	1263	
$C_{3\times3}$	$C_{1\times1}$	*SC*	*SC*	$C_{3\times3}$	*None*	345		60		511		253		1392		**3**	0.60	1264	
SC	$C_{3\times3}$	$C_{3\times3}$	*SC*	*SC*	$C_{1\times1}$	241		1246		103		1927		2559		**4**	0.60	938	
SC	$C_{3\times3}$	$C_{3\times3}$	*SC*	$C_{1\times1}$	*SC*	2058		2406		552		291		1383		**5**	0.60	292	
$C_{3\times3}$	*SC*	*SC*	*SC*	$C_{3\times3}$	$C_{3\times3}$	461		70		1118		15		2081		**6**	0.60	684	
$C_{3\times3}$	$C_{1\times1}$	*None*	$C_{1\times1}$	$C_{1\times1}$	$C_{1\times1}$	862		1389		2117		1659		370		**7**	0.60	1148	
SC	$C_{3\times3}$	$C_{3\times3}$	*SC*	*SC*	$C_{3\times3}$	15		529		1306		512		1805		**8**	0.60	599	
$C_{3\times3}$	*SC*	*None*	*SC*	$C_{3\times3}$	*SC*	125		384		25		53		2323		**9**	0.60	346	
SC	$C_{3\times3}$	*SC*	*SC*	$C_{3\times3}$	$C_{3\times3}$	307		807		1072		54		2595		**10**	0.60	1709	
$C_{3\times3}$	$C_{3\times3}$	*None*	*SC*	$C_{3\times3}$	$C_{1\times1}$	428		95		1241		186		613		887		**1**	95.37
$C_{3\times3}$	$C_{1\times1}$	*SC*	*SC*	*None*	$C_{1\times1}$	1431		1059		402		1357		2361		144		**2**	95.22
$C_{3\times3}$	$C_{3\times3}$	*None*	*SC*	*SC*	*None*	1405		2652		1309		1761		2619		213		**3**	95.00
$C_{3\times3}$	$C_{3\times3}$	*SC*	*SC*	*SC*	*None*	1407		2653		1310		1762		2620		214		**4**	95.00
None	$C_{3\times3}$	$C_{1\times1}$	$C_{1\times1}$	$C_{1\times1}$	$C_{3\times3}$	2671		321		2855		352		1529		2302		**5**	95.00
$C_{3\times3}$	$C_{1\times1}$	$C_{1\times1}$	$C_{3\times3}$	*SC*	*SC*	1275		361		1404		311		958		1245		**6**	94.99
$C_{3\times3}$	*None*	*None*	*SC*	$C_{1\times1}$	$C_{1\times1}$	328		1158		1172		1681		2442		649		**7**	94.96
$C_{3\times3}$	$C_{3\times3}$	$C_{1\times1}$	$C_{1\times1}$	*SC*	$C_{1\times1}$	2146		493		1975		169		209		2041		**8**	94.91
$C_{1\times1}$	*SC*	$C_{1\times1}$	$C_{3\times3}$	*SC*	*SC*	922		2121		196		1249		2145		421		**10**	94.88

and would allow a proper evaluation of the method's generability, as it could be easily overfitting a setting from a small search space on a given benchmark.

- Evaluating NAS methods by directly searching on all benchmark data sets. It is common that NAS methods search on a small data set (e.g., CIFAR-10) and transfer the generated architecture to other data sets based on human-defined heuristics (e.g., ImageNet) [2, 40]. However, NAS methods should be capable of generating archi-

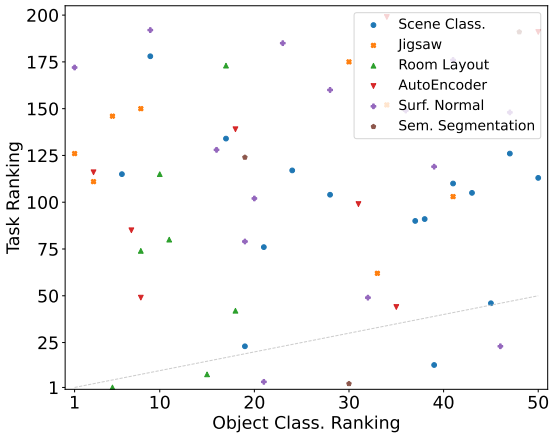


(a) Top-50 Architectures inter-rank.

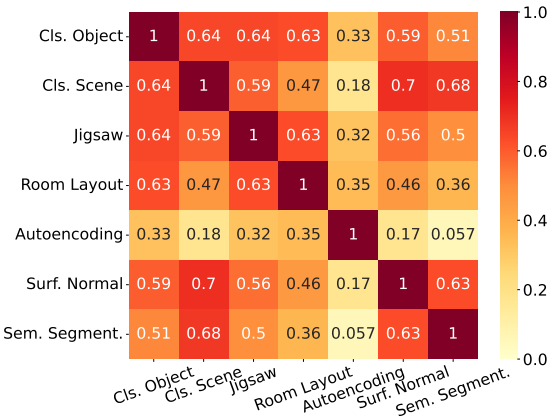


(b) Data set Kendall's tau correlation.

Figure 2.14: Evaluation of NAS-Bench-201 architectures regarding their inter-data set ranking and the correlation between data sets. On (a) we show the ranking of the top-50 architectures from CIFAR-10 when transferred and evaluated to CIFAR-100 and ImageNet16-120; and (b) Kendall's Tau correlation between data sets.



(a) Top-50 Architectures inter-rank.



(b) Task Kendall's tau correlation.

Figure 2.15: Evaluation of TransNAS-Bench-101 architectures regarding their inter-task ranking and task correlation. On (a) we show the ranking of the top-50 architectures from object classification when transferred and evaluated to other tasks; and (b) Kendall's Tau correlation between tasks.

tectures in different scenarios and constraints. Thus it is important that the authors report the performance of directly searching different data sets.

Designing NAS benchmarks

Based on the findings presented in this section, we propose several suggestions for researchers interested in designing and creating new NAS benchmarks. Firstly, it is important to note that within the search space of all three benchmarks evaluated, there are operations that have the most importance – convolutional layers and SC. This suggests that a subset of the operations pool is sufficient to generate the best architectures in the studied benchmarks. Additionally, our results show a high correlation between data sets within a benchmark, hindering the evaluation of NAS methods in terms of generability and transferability. To address these, researchers should consider the following when

designing NAS benchmarks:

- The cell structure greatly impacts the final architecture performance. This is seen in NAS-Bench-201 and TransNAS-Bench-101 when SC layers are often placed in the fourth position. Therefore, researchers should consider this when designing their benchmarks by progressively analyzing the architectures as the benchmark is developed. This can be done by looking at a small set of architectures and partially training them. In this, looking at the learning curve and zero-proxy estimators can help obtain quicker estimates of how the operations and the cell structure impact the final performance.
- Small search spaces could lead to operation overfitting, suggesting that, if possible, authors should create an operation space that contains more operations than existing ones. This could be manageable by reducing the cell structure, fixing specific operations, and increasing the operation pool to include more operations, such as activation layers and a wider range of pooling operations.
- New benchmarks should strive to increase the number of architectures. While a small number of architectures is required due to the extreme computational costs of training thousands of architectures, benchmarks could be designed with a larger search space and similar computational costs by training the architectures with a smaller number of epochs or simplifying the architecture's outer-skeleton to reduce the number of parameters. In the studied benchmarks, NAS-Bench-101 is the only one with a large set of possible architectures, with over $423k$ candidates. In contrast, NAS-Bench-201 only has $15k$ architectures, and TransNAS-Bench-101 cell space has approximately $4k$ possible architectures.
- NAS benchmarks should include different types of constraints and information regarding the architectures, such as latency, power consumption, and model size, among others. This will allow the evaluation of NAS methods that search architectures that meet different practical constraints.
- Benchmarks should provide a comprehensive evaluation metric that considers both accuracy and practical constraints to evaluate the overall performance of the searched architectures. This will ensure that the searched architectures are accurate, practical, and feasible for real-world applications.
- NAS benchmarks would benefit from going beyond training architectures with different seeds to evaluating architectures with different training protocols. With this, a NAS method could be evaluated on the performance of the generated architectures with a specific training protocol (e.g., different loss criteria). This would further allow NAS methods to go beyond architecture design to also select the proper training protocol.

Furthermore, as architectures become less dependent on convolutional layers, their training time will be reduced. Convolutional layers are known for their excellent feature

extraction capabilities but come with the cost of more parameters. To reduce inference time and the number of parameters, researchers can use convolutional layers that reduce computational costs, such as depth-wise separable convolutions [187]. These layers have been used extensively in the DARTS search space [2].

2.4 Conclusions

In this chapter, we presented a comprehensive review of the concepts that make up a NAS method and how NAS benchmarks can be a step forward into fair evaluations. First, we introduced the problem of designing CNNs and how NAS automates the process by having 3 components: the search space, the search strategy, and the performance estimation strategy. For each, we reviewed different methods, their advantages, and how these proposals impact a NAS method. With this, we showed that NAS methods still depend on many design choices, greatly affecting the final performance and computation required to perform the search. Performing fair comparisons is extremely hard, as different methods have different training protocols and search spaces, so we also presented an extensive overview of NAS benchmarks and analyzed three popular NAS benchmarks regarding their operations. We found that convolutional layers are essential to have architectures with high performance and also found that the distribution of the accuracy ranges is skewed, suggesting that finding architectures with accuracies closer to the upper bound is not hard, as several operation patterns, such as their positioning in a cell and their combinations, tend to yield top-scoring architectures.

Based on the comprehensive review presented in this chapter, in the next chapter, we focused on evaluating different neural networks for two different problems and proposed improvements on their architectures based on NAS-searched classifiers.

Chapter 3

Improving Convolutional Neural Networks Through Method Fusion

3.1 Introduction

In this chapter, we focus on understanding different neural network architectures and study how NAS and AutoML can be used to improve their structure without the need to search for an entirely new architecture. For this, we focused on evaluating different CNNs and RNNs in two different problems: defect detection and sentiment analysis. Then, based on the evaluation of several state-of-the-art neural networks, we propose three different methods: i) CNN-fusion, an ensemble method that combines all CNNs and outputs a final prediction; ii) Auto-Classifier, which uses the best CNN and further improves it using NAS and AutoML by automatically searching for a new classifier component; and iii) Auto-fusion, which combines two individual classifiers into a final decision by leveraging a searched fusion model. The problems used to evaluate the proposed methods were chosen since they provide completely different settings: industrial defect detection and social media information, as well as because existing solutions still flounder to solve them efficiently [3, 188, 189, 190, 191, 192]. This makes the chosen problems suitable for the analysis of different neural networks in the tasks of single classification and multimodal analysis. With this, we use NAS and AutoML to improve upon the classifiers in a controlled way, where the goal is not to search for new architectures but to improve upon existing ones with new searched methods. The use of NAS and AutoML also allows us to have a deep understanding of how it can be used to create new neural networks, which was crucial for the rest of this thesis work.

The rest of this chapter is organized as follows. Section 3.2 presents and discusses the proposed methods for defect detection and sentiment analysis. Section 3.3 showcases the conducted experiments and discusses the results, while section 3.4 provides the conclusions of this chapter.

3.2 Proposed Methods

In this section, we present two CNN-based methods for classifying texture defects in 2D images and one for multimodal sentiment analysis. For this, we evaluate the performance of multiple state-of-the-art architectures in IC problems: VGG{11, 16, 19} [57], ResNet{18, 34, 50, 101, 152} [59], and DenseNet{121, 161} [60]. In text sentiment analysis, we evaluated the performance of Vader [193], TextBlob [194], Long-Short Term Memory (LSTM)

[195], LSTM with Attention [196], Bi-directional LSTM [197], RNN [198], Recurrent Convolutional Neural Network (RCNN) [199], TextCNN [200] and VDCNN{9, 17, 29, 49} [201] (Section 3.2.2). Then, based on the evaluation of all neural networks, we propose three methods: i) CNN-fusion, an ensemble method that combines all CNNs and outputs a final prediction (Section 3.2.2); ii) Auto-Classifier, which uses the best CNN, based on the initial performance evaluation, and then improves it by using NAS and AutoML to automatically search for a new classifier component (Section 3.2.3); and iii) Auto-fusion, which combines two individual classifications: one from the text classifier and one from the image classifier, into a single one by leveraging a searched fusion model.

3.2.1 Deep Neural Networks

In this section, we sought to evaluate the performance of different state-of-the-art Deep Neural Networks (DNNs) that are known to do well in a variety of classification problems. By looking closely at different neural networks on textual and visual problems and evaluating those in defect detection and sentiment analysis problems, we obtained information about their performance, but more important to this thesis, we were forced to implement and deeply understand how their inner components are combined to form entire architectures.

In the context of defect detection, we looked at CNNs for visual classification, and for sentiment analysis, we used both CNNs for visual classification and several different types of neural networks for text analysis. In the rest of this section, we detail the neural networks that were implemented and evaluated.

Vision-based Neural Networks

For IC, we focused on using VGG{11, 16, 19} [57], ResNet{18, 34, 50, 101, 152} [59], and DenseNet{121, 161} [60]. These 3 families of CNNs allow a deep understanding of fundamental topics of deep learning: designing deep models, understanding vanishing and exploding gradients, and residual and dense connections. There are other image classifiers such as transformers-based models [202] and Xception-based architectures [187] that work efficiently, but for simple IC purposes, the 3 CNNs families examined show excellent results, often outperforming all others.

VGG was one of the first proposed CNN architectures with a deep structure [57]. It was introduced by researchers at Oxford University, and their main proposal was to use a simple and unified architecture for performing classification tasks. The main advantage of VGG is its simplicity and uniformity. VGG only uses 3×3 convolutional layers with a fixed number of filters in each layer, followed by max pooling layers to downsample the feature maps. The architecture is simple to implement and can be easily modified to fit different image sizes and data sets. When first proposed, VGG achieved state-of-the-art performance on the ImageNet data set.

Improving Neural Architecture Search

ResNet (short for Residual Network) is a deep CNN [59] that introduced the notion of residual connections. Its main proposal is to overcome the vanishing gradient problem that can occur in very deep neural networks. This problem arises because the gradients that are propagated through the layers can become very small, making it difficult for the network to learn effectively. The main advantage of ResNet is that it allows designing very deep neural networks (hundreds of layers) that can still be trained effectively. This is achieved by proposing the use of residual connections, which allow the network to learn a residual mapping in addition to the mapping that is learned by the convolutional layers. By doing so, the gradient can be directly propagated to earlier layers during training, making it easier to train deep architectures. ResNets achieved state-of-the-art performance on a wide range of CV tasks, including IC, object detection, and segmentation. Its success has led to the development of many other deep residual networks, such as ResNeXt [203] and Wide ResNet [204].

DenseNet (short for Dense Convolutional Network) is a deep convolutional neural network architecture that further improves upon the idea of learning residual connections by proposing the use of densely connected architectures [60]. The main advantage of DenseNet is that it promotes feature reuse by connecting each layer to every other layer in a feed-forward fashion. This is achieved through dense connections, where the output of each layer is concatenated with the output of all previous layers. By doing so, each layer can have access to all of the feature maps that were produced by previous layers, which helps to mitigate the vanishing gradient problem. DenseNets have achieved state-of-the-art performance on a wide range of CV tasks [54].

Text-based Neural Networks

For text classification, we evaluated lexicon and rule-based methods and different DNNs. The implemented DNNs can be categorized into two types: CNNs, which are based on convolutional layers, and RNNs, which promote temporal information throughout the architecture. Specifically, we looked into: Vader [193], TextBlob [194], LSTM [195], LSTM with Attention [196], Bi-directional LSTM [197], RNN [198], RCNN [199], TextCNN [200] and VDCNN{9, 17, 29, 49} [201]. The next paragraphs introduce each in more detail.

VADER and TextBlob are tools for sentiment analysis in natural language processing. Both VADER and TextBlob are lexicon and rule-based sentiment analysis tools designed to handle social media texts with non-standard language, sarcasm, and other nuances. These methods provide a good baseline for comparison with deep learning methods.

FastText is a shallow network architecture that can be quickly trained [205]. The idea behind FastText is that instead of representing each word as a single vector, FastText represents each word as a bag of character n-grams, which allows it to capture morphological information and handle unseen words [206]. In this work, we implemented a two-layer FastText architecture with a hidden size of 256.

RNNs were designed to allow neural networks to have temporal information, which simple neural networks didn't have [198]. Basically, RNNs form a chain structure in which each node receives as input the output from the predecessor node and one part of the input sequence (e.g., a word or a vector). Each node outputs a value both to the successor node and to the next layer. So, what an RNN layer does is, for each input element, it computes:

$$h_t^i = \tanh(W_{hh}h_{t-1} + b_{ih} + \sum_{j<i,x} W_{x^j h}x_t^j + b_{ih}) \quad (3.1)$$

where h_t is the hidden state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time 0, W_{hh} is the weight of the recurrent neuron, $W_{x^i h}$ is the weight of input neuron x from layer i, and b_{hh} are the bias weights. The architecture implemented is an Embedding layer followed by a multi-layer Elman RNN with 2 recurrent layers and a final linear layer.

LSTM is a type of RNN architecture designed to address the vanishing gradient problem that can occur in traditional RNNs while also being able to remember or forget previous inputs selectively. Its main advantage is the ability to handle long-term dependencies in sequential data. LSTM accomplishes this by using a memory cell, which allows the network to retain or discard information from previous timesteps selectively. Formally, an LSTM layer computes for each element in the input sequence:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \quad (3.2)$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \quad (3.3)$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \quad (3.4)$$

$$c_t = f_t \circ c_{(t-1)} + i_t \circ g_t \quad (3.5)$$

$$h_t = o_t \circ \tanh(c_t) \quad (3.6)$$

where h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $\circ$ is the element-wise product. The implemented architecture has 1 embedding layer, 1 LSTM layer with 256 hidden states, and 1 linear layer.

LSTM with attention (LSTM-Attn) is a variant of the LSTM architecture that uses an attention mechanism to improve the handling of long-term dependencies in sequential data [196, 207]. The main advantage of LSTM with attention is its ability to selectively focus on different parts of the input sequence, allowing it to better capture important information and handle long-term dependencies. The attention mechanism calculates a weight for each input timestep, indicating how much attention the network should give to each input component. This allows the network to selectively focus on the most relevant parts of the input sequence while ignoring irrelevant or redundant information. In our implementation, the attention layer is a global attention mechanism that computes

the soft alignment score between the output of the LSTM and its final hidden state. The implemented architecture is similar to the aforementioned LSTM, with the addition of an attention layer before the linear layer.

Bi-directional LSTMs (Bi-LSTM) are an extension of traditional LSTMs that can better capture the context and dependencies in sequential data by processing it in both forward and backward directions [197]. The main advantage of Bi-LSTMs is their ability to capture dependencies between past and future contexts of a sequence, allowing a better understanding of the meaning and context of each word. Bi-LSTM accomplishes this by using two separate hidden layers for each timestep of the input sequence, one for the forward direction and one for the backward direction. The outputs of these hidden layers are then concatenated to produce the final output at each timestep. In our implementation, we followed the same architecture scheme as for LSTM but with a bi-directional LSTM and two linear layers with max and average pooling in between.

RCNN is a type of neural network architecture that combines the strengths of RNNs and CNNs to process sequential data. Its main advantage is the ability to capture both local and global context in sequential data, allowing it to better infer the meaning and context of each word. RCNN accomplishes this by using a sliding window of fixed size to extract features from the input sequence, and then passing these features through a bi-directional RNN to capture the context and dependencies between words. The implemented RCNN architecture is: an embedding layer, a bi-directional LSTM followed by a dropout. Then, a linear layer processes the embeddings from the bi-LSTM and finally, a max pooling layer and a linear layer complement the architecture.

TextCNN is a type of neural network architecture that uses CNNs to process textual data [200]. The main advantage of TextCNN is its ability to capture local context in sequential data, allowing it to extract information about the meaning and context of each word. TextCNN accomplishes this by using convolutional layers to extract features from the input sequence and then using max-pooling to reduce the dimensionality of the features. In our implementation, the architecture is composed of an embedding layer followed by five convolutional blocks and a final linear layer. Based on TextCNN, we defined a second CNN-based network to perform text classification, which we named sCNN. The architecture is similar, but instead of having 5 blocks of Convolutions-ReLU-Max Pooling, it has only 3 with kernel sizes of 1, 3, and 5 respectively.

Similar to TextCNN, Very Deep CNNs is a neural network that uses convolutional neural networks to process text data. The main advantage of VDCNN is its ability to capture both local and global context in sequential data, allowing it to infer the meaning and context of each word. VDCNN accomplishes this by using convolutional layers with increasing filter sizes to extract features from the input sequence and then using max-pooling to reduce the dimensionality of the features. The resulting features are then passed through a series of

fully connected layers to make the final prediction. We implemented 4 architectures with different depths: 9, 17, 29, and 49. Every architecture starts with an embedding layer, followed by a 1D convolutional layer and a pre-defined number of convolutional blocks (depth size of the architecture). The architecture's head is then composed of a max pooling layer and three linear layers.

3.2.2 Convolutional Neural Networks Fusion

By leveraging the implemented CNNs (Section 3.2.1) and training those on a visual task of detecting defects, we created a pool of architectures that are specifically trained for the problem at hand and that can be combined to form an ensemble model. For this, we harness all individual networks and propose CNN-Fusion, an ensemble method created with all CNNs in the pool. CNN-Fusion allows a combination of all the predictions of the individually trained CNNs into a final and unique classification. The proposed CNN-Fusion works by fusing all the individual predictions into a final, weighted, prediction by making a weighted sum of each class, where each CNN votes using a normalized weight based on the Area Under The Roc Curve (AUC) obtained in the validation set during training. The weights are obtained with the following expression:

$$w_i = \frac{V_i}{\sum_{j=1}^{n} V_j}, i \in 1, ..., n \quad (3.7)$$

where n is the number of CNNs, and V is a vector with the AUC values for all CNNs.

Hence, the final classification using the normalized contribution of the individual for a given input can be calculated with:

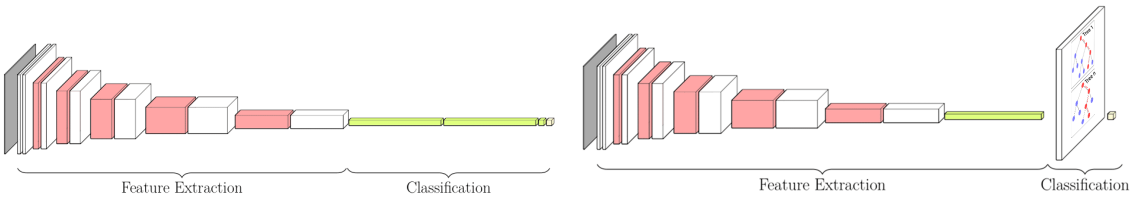
$$argmax_i(P_{ij} \cdot w_j), \begin{array}{l} i \in 1, ..., c, \\ j \in 1, ..., n \end{array} \quad (3.8)$$

where c is the number of classes, n the number of CNNs, and P_{ij} represents the output classification of network j for class i.

The idea behind CNN-Fusion is that by balancing the importance of each CNN through the process of normalization, where networks that have higher AUC scores in the validation set have higher confidence, we can perform weighted voting that improves upon classifying the existence of defects using individual CNNs.

3.2.3 Auto-Classifier

CNNs are usually composed of two components: the feature extraction and the classification component. The idea is: CNNs start by extracting simple representations of the input as features maps, which gradually increase in complexity at deeper layers of the network, then, these feature maps are fed into fully connected layers that provide the output of the network (activation patterns), normally in the form of a classification map.



(a) Individual CNN. (b) Auto-Classifier.

Figure 3.1: Visual representation of the difference between a CNN and the Auto-Classifier method. A CNN is composed of two components: Feature Extraction and Classification. In the Auto-Classifier, the classification component has been replaced by another one, represented by a gradient boosting machine.

To further improve the use of CNNs, we propose Auto-Classifier that uses NAS and AutoML to increment upon a CNN instead of searching entirely for a new one. Auto-Classifier improves upon the best individual CNN, by replacing its classification component with a new one. For this, we leverage the feature extraction capabilities of a CNN and remove its classification component. Then, a RS is performed over a pool of different methods, composed of: random forest, gradient boosting machines, neural network, XGBoost gradient boosting machine, extremely randomized forest, random grid of XGBoost, random grid of gradient boosting machines, and a random grid of DNNs [208, 209]. After the search, a post-processing step in which the best models found are stacked is also conducted. This process is efficient, as the sampled methods are evaluated only on the final feature map created by the CNN and a performance metric is drawn by evaluating a small number of epochs.

We hypothesize that training a CNN from scratch and then partially or entirely removing its classification component and replacing it with other types of classification methods will increase the CNN capabilities and outperform the initial architecture. This is because other classifiers, such as random forests, have shown to be very good in different classification problems. The problem with many classifiers is that to be able to process and classify images correctly, they require extensive processing power, translating into huge models. By processing the image with the feature extraction of a CNN, complex and rich feature maps are created, which can then be used by a classifier without the need to perform any feature extraction or feature processing. So, the proposed method, Auto-Classifier, works by initially using an individual CNN, from which the classification component is partially removed. Then, we use RS to generate a new classifier based on the output features of the modified CNN. In our experiments, we use the feature extraction capabilities of a CNN and allow the first layer of the CNN's classification component to be kept for dimensionality reduction purposes. This ultimately enables more types of classifiers to work with that data.

In short, the proposed Auto-Classifier is composed of two parts: i) the best individual CNN without the classification component, leaving a trained CNN that outputs a rich representation map of the input; ii) an automated search for a new classification component that receives as input the representations generated in the previous step. Then, the final

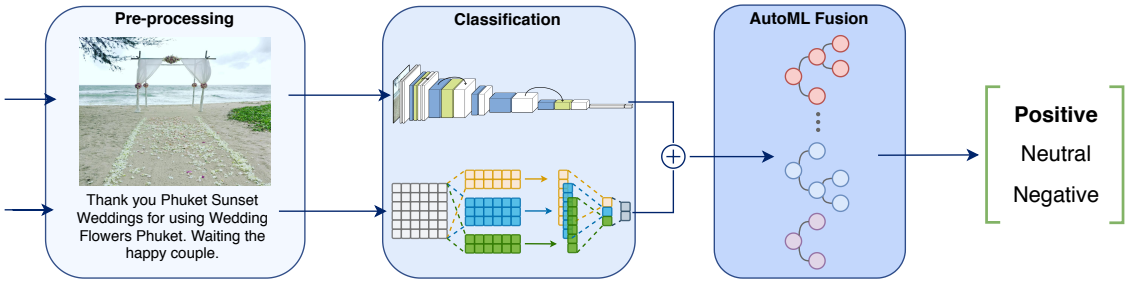


Figure 3.2: Proposed multimodal architecture. The first container represents the pre-processing component that removes noise and non-important information. The second container shows both classification components, where the image and the text are individually classified using CNNs. The third container represents the fusion method that receives the concatenation of the individual classifications and performs a final classification using the model searched – represented by a gradient boosting machine.

model is composed by the partial CNN, sequentially followed by the new searched classifier. In Figure 3.1, we present the difference between a CNN and the Auto-Classifier method. In this, the Auto-Classifier method (b), based on the CNN presented in (a), has a new classification component.

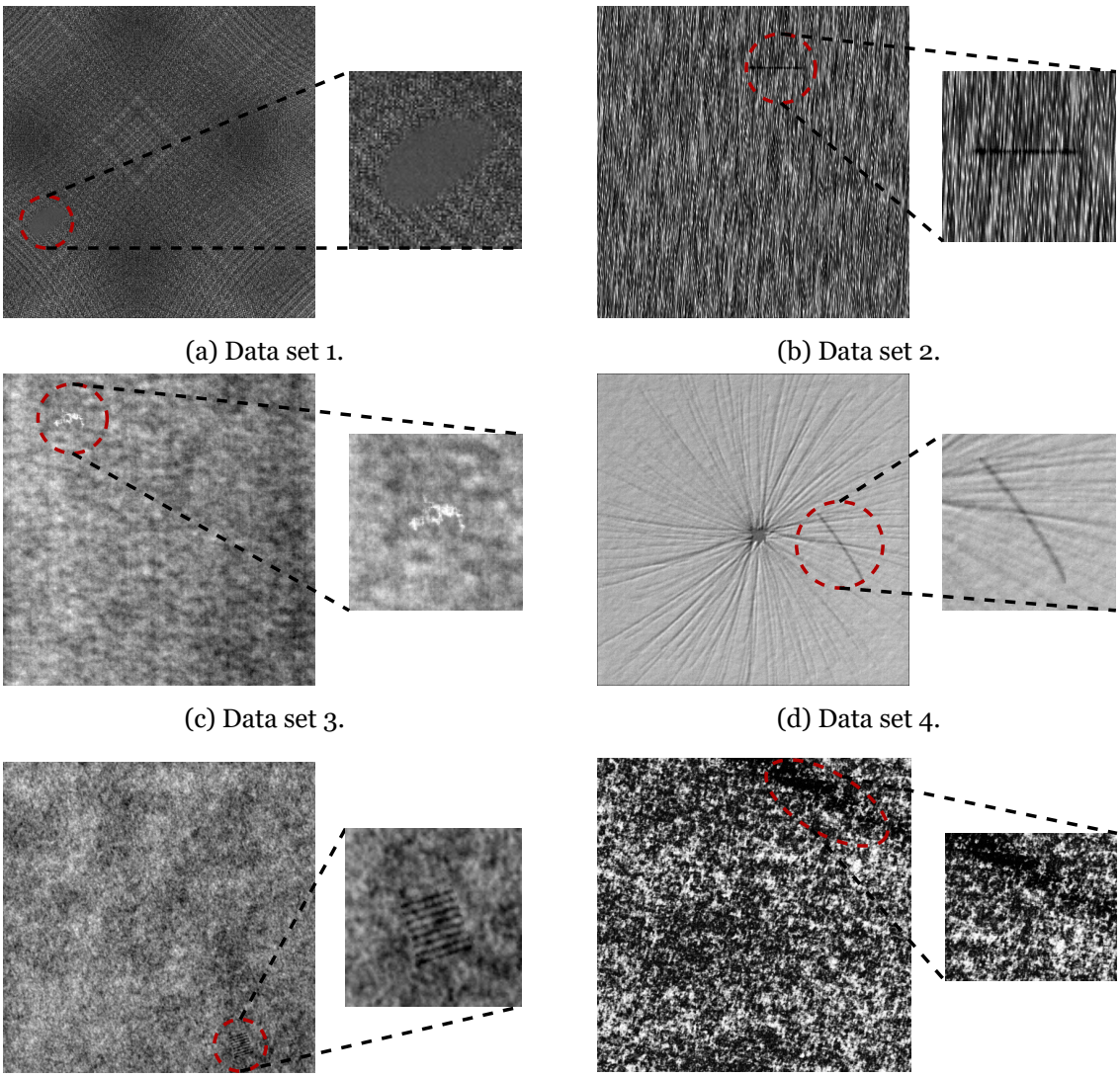
3.2.4 Auto-Fusion

In this section, we propose Auto-Fusion that, similarly to Auto-Classifier, leverages NAS and AutoML and designs a fusion mechanism. For this, we improve upon CNN-Fusion by combining different classifiers. In the case of sentiment analysis, this means: one text and one image classifier are combined to perform a final classification. With this, we can perform a multimodal classification by leveraging individual data-specialized models.

To create Auto-Fusion, we used a similar approach to the one proposed in the previous section (Section 3.2.3), where the goal is to create an optimal model to classify a given data set without requiring extensive human modeling. However, in this case, we focused on creating a fusion model that receives two separate inputs from different neural networks. For this, the first step is to get the final X based on Y_{img} and Y_{text}, where Y represents the classification vector and img and $text$ represent the respective classifiers, and fuse them into a unique feature map: $X = Y_{img} \bigoplus Y_{text}$, where X will be the input for the optimization problem (final classifier). Here, the objective function $\mathcal{O}$ is defined as accuracy in the task of 3-class sentiment classification.

Given X, we search for the optimal machine learning model using the RS strategy over the search space discussed in Section 3.2.3. After the search, the model with the best performance on the validation set is the one selected to be the fusing method of Auto-Fusion. Figure 3.2 shows Auto-Fusion's architecture. Initially, the pre-processing model is used to normalize the images, resizing the images to $224 \times 224 \times 3$, tokenizing the textual data as well as removing the stop words.

3.3 Experiments



(a) Data set 1. (b) Data set 2.

(c) Data set 3. (d) Data set 4.

(e) Data set 5. (f) Data set 6.

Figure 3.3: Examples of defects in each DAGM2007 data set.

To evaluate the proposed methods and compare them with baseline approaches, we conduct experiments on several data sets. CNN-Fusion and Auto-classifier are evaluated on a defect detection problem, and Auto-Fusion on a multimodal sentiment analysis task. All the experiments were conducted using a computer with a single GeForceGTX 1080 Ti, 16Gb of RAM, and an AMD Ryzen 7 2700 processor.

In the following sections, we detail the data sets used, present the results obtained and provide a discussion for each task independently.

(a) **Negative**: "His eyes speak of the horror of war that no one should go through. War doesn't help, it only kills. Pls Stop."

(b) **Neutral**: "And they say it's grim up north..."

(c) **Positive**: "Thank you Phuket Sunset Weddings for using Wedding Flowers Phuket. Waiting the happy couple."

Figure 3.4: Examples of images and the correspondent texts of the three different classes in B-T4SA. (a) shows a negative example; (b) a neutral one; and (c) a positive example.

3.3.1 Data Sets

For the task of classifying the presence of defects, we used the DAGM2007 set of problems. It consists of six different data sets, each with 1150 images. For which, 1000 images are of background textures without defects and 150 with a type of defect. For each data set, we performed a stratified split into 3 sets: 70% for the train set, 15% for the validation set, and 15% for the test set. The train and validation sets were used to train the algorithms, and the test set to evaluate the final performance. The test set is never used for training purposes, and for the proposed defect detection methods, the best individual CNN was selected based on its validation AUC. Figure 3.3 shows an example of the type of defect present in each data set of the DAGM2007.

As for the multimodel problem, we used the B-T4SA data set, which is a data set with Twitter information for sentiment analysis. Each data point of the 470 thousand has both text and image information [210]. In Figure 3.4, we show an example of an image and the corresponding text for each class (negative, neutral, positive). In B-T4SA all classes are balanced, and the splits are stratified. The train set has approximately 80% of the data set, while both the validation and test sets have 10% each. To conduct transfer learning, we also leveraged the Stanford Sentiment Treebank [211] for the text classification and Flickr and Instagram Data set [212] for the IC component. The first has 5 classes and 215 thousand phrases, while the latter has 8 classes and approximately 23 thousand images.

3.3.2 Results and Discussion on Detecting Defects

To validate the proposed methods on the problem of defect detection, we conducted three experiments: i) evaluate the performance of multiple state-of-the-art CNN architectures; ii) evaluate the performance of the CNN-fusion method by fusing all individual CNNs; and iii) evaluate the performance of the Auto-Classifier method.

To evaluate the performance of state-of-the-art CNNs, we fixed the same settings for all of them: Stochastic Gradient Descent (SGD) with a batch size of 10, a learning rate of $1e^{-3}$, and momentum of 0.9 for 100 epochs of training. As for loss function, we used cross-entropy loss. For testing purposes, the CNN's weights are the ones that yielded the highest validation AUC while training. Note that this testing step is only used to compare the different individual CNNs under the same conditions and is not used in any situation nor to select the best CNNs to be used in the proposed methods. The results presented in the first 9 columns of Table 3.1 show that amongst all the individual CNNs tested, VGG16 was the one that achieved the best results – 99.9% mean accuracy and 99.7% mean AUC. The use of AUC as a metric for evaluating performance is extremely important because the accuracy metric is not, by its own, a good representative of a good classifier when using unbalanced data sets, whilst AUC is sensitive to class imbalance.

Table 3.1: Results of different state-of-the-art CNNs architectures and the two proposed methods in the task of defect detection using the DAGM2007 data sets. Accuracy and AUC values are shown in percentages.

Problem	VGG11		VGG16		VGG19		ResNet18		ResNet34		ResNet50		ResNet101		Densenet121		**CNN-Fusion**		**Auto-Classifier**	
	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC	Acc.	AUC
1	100	100	100	100	85.0	50.0	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	85.0	50.00	100	100	100	100	100	100	100	100	100	100	99.4	98.1	100	100	100	100	100	100
3	100	100	100	100	100	100	86.1	53.9	97.1	90.4	99.4	98.1	100	100	99.4	98.1	100	100	100	100
4	100	100	99.4	98.1	99.4	98.1	98.8	97.7	100	100	99.4	98.1	99.4	98.1	100	100	100	100	100	100
5	98.8	96.2	100	100	100	100	98.8	96.2	98.8	96.2	99.4	98.1	99.4	98.1	99.4	98.1	99.4	98.1	100	100
6	100	100	100	100	100	100	100	100	100	100	98.8	96.2	99.4	100	98.3	94.2	100	100	100	100
μ	97.3	91.0	**99.9**	**99.7**	97.4	91.4	97.3	91.3	99.3	97.8	99.5	98.4	99.6	99.1	99.5	98.4	**99.9**	**99.7**	**100**	**100**

We believe the reason that VGG16 achieves the best results in almost all the six data sets and the best overall mean performances when compared to other individual CNNs is because, even if ResNets and DenseNets are more powerful, their larger number of layers is a drawback when using small data sets. In DAGM2007, we only have 1150 images per data set, which makes it a challenging data set for large models. Even though residual connects and short circuits in the mentioned architectures can mitigate problems such as the vanishing gradient, their bigger complexity is a factor that will undermine their performance in problems where data sets are small.

By having the validation AUC for each CNN, we can complete the process of CNN-Fusion by combining the individual classifications into a final one by first normalizing the validation AUC values, and then performing weighted voting using the CNN's classifications. The results for the CNN-Fusion are shown in 10th column of Table 3.1, achieving a mean accuracy of 99.9% and mean AUC of 99.7%. CNN-Fusion achieved the same mean values as VGG16. However, the difference is that CNN-Fusion perfectly identified defects in problems one to four and six, having some miss-classifications in problem five, while VGG16 had some miss-classifications in problem four and perfectly solved all the other problems. Moreover, CNN-Fusion was capable of achieving an overall high performance, but in problem five, as many individual CNNs had classification errors, the CNN-Fusion was not capable of having 100% AUC. This can be justified due to the fact that all CNNs have good individual performances, meaning that the differences between

the normalized weights for the CNN-Fusion are minimal. A possible improvement would be to perform a non-linear normalization, where better models have a larger difference from their neighbors. The problem with using CNN-Fusion in a fast-paced environment, e.g., quality control in industrial production lines, is that it takes more time to have a final classification, as it requires all CNNs to perform their classification. If done in parallel, the time taken to perform a classification will be the maximum time, t, from the set of times, T, that contain the time taken for each CNN to perform a classification for a given input, plus the time taken, t_{fusion}, to perform the final classification: $m_{time} = max\{T(x) : x = 1, .., n\} + t_{fusion}$, where n represents the number of individual CNNs. The problem is that in most systems, conducting a forward pass in all the individual CNNs in a parallel manner is extremely challenging. When done sequentially, the m_{time} will be increasingly higher: $m_{time} = (\sum_{x=1}^{n} T(x)) + t_{fusion}$.

The Auto-Classifier solves the problem of having an inference time that is dependent on all individual CNNs by using the feature extraction capabilities of only the overall best CNN on the validation set (VGG16 in our experiments) and improving its classification component. We partially removed the classification component of VGG16, by removing the last two fully connected layers, leaving only the first one to serve the purpose of dimensionality reduction. To generate a new classification component we ran RS for 2 hours for each problem, and at the end, we selected the best candidate on the validation set to be coupled to the modified VGG16 to create a final model – Auto-Classifier. The best classifier from the search step was a gradient-boosting machine model. The results of Auto-Classifier on the test sets are shown in the last column of Table 3.1, where it is possible to see that it not only improved upon the individual CNNs, but it also correctly classified each data point in all the six data sets from DAGM2007.

When using ML methods in industrial systems, it is of utmost importance to take into consideration the time required to train the models. Models that have a quick training step allow quick changes and experimentation. Moreover, the inference time is also critical, as real-time inference is often required. The overall mean time and standard deviation to train each CNN, was: 50.7 ± 0.18 minutes for VGG11, 95.2 ± 0.33 minutes for VGG16, 115.6 ± 0.71 minutes for VGG19, 17.0 ± 0.02 minutes for ResNet18, 28.8 ± 0.02 minutes for ResNet34, 44.8 ± 0.04 minutes for ResNet50, 71.1 ± 0.09 minutes for ResNet101, and 47.63 ± 0.02 minutes for Densenet121. From this, we can infer that the time required to train any of the CNNs studied here is feasible in an environment with rapid changes since the CNN that took the longest time to train was VGG19, requiring less than 2 hours. Regarding CNN-Fusion, it requires no further training, as it is the combination of training all individual CNNs, and for the Auto-Classifier, the search cost was limited to 2 hours. By adding it to the time required by VGG16 to train, it required 235.6 minutes on average to create the complete model. As for the inference times, all CNNs were capable of executing in real-time, with the fastest one being ResNet18 with an inference time of 0.057 ± 0.016 seconds, and the slowest one VGG19, with an inference time of 0.269 ± 0.002 seconds. As

for VGG16, the one selected to be the feature extraction component of Auto-Classifier, it obtained an inference time of 0.225 ± 0.002 seconds, which is enough for detecting surface defects in real-time (approximately 5 frames per second). CNN-fusion, in a serial manner, has an m_{time} of 1.152 seconds, which is not suitable for real-time defect detection. As for the Auto-Classifier, which consists of the partial VGG16 and the new classification component, we found that it is extremely fast, requiring only 0.100 ± 0.004 seconds to classify an image, which is justified by having an efficient new classifier that uses less, and faster operations than the removed layers.

Table 3.2: Comparison of different methods in the task of defect detection in DAGM2007 problems regarding the True Positive Rate, True Negative Rate, and Average Accuracy.

Problem	**Auto-Classifier (Ours)**	**CNN-Fusion (Ours)**	**VGG16 (Ours)**	Deep CNN [213]	Statistical features [190]	SIFT and ANN [191]	Weibull [214]
True Positive Rate (%)							
1	100	100	100	100	99.4	98.9	87.0
2	100	100	100	100	94.3	95.7	-
3	100	100	100	95.5	99.5	98.5	99.8
4	100	100	99.3	100	92.5	-	-
5	100	99.3	100	98.8	96.9	98.2	97.2
6	100	100	100	100	100	99.8	94.9
True Negative Rate (%)							
1	100	100	100	100	99.7	100	98.0
2	100	100	100	97.3	80.0	91.3	-
3	100	100	100	100	100	100	100
4	100	100	100	98.7	96.1	-	-
5	100	100	100	100	96.1	100	100
6	100	100	100	99.5	96.1	100	100
Average Accuracy (%)							
	100.0	99.9	99.9	99.2	95.9	98.2	97.1

Finally, we compare our two proposed methods and the best individual CNN with the methods proposed in the literature that achieve the best results in the DAGM2007 defect classification in Table 3.2. In this, it is possible to see that both the proposed methods in this section achieved the highest average accuracy, which is calculated by summing the true positive rate and true negative rate means, and divide it by two: $(TPR+TNR)/2$. It is also worth noting that our proposed method, Auto-Classifier, not only achieved a perfect classification on all DAGM2007 problems, but outperformed all other methods in this set of data sets.

3.3.3 Results and Discussion on Multimodal Sentiment Analysis

First, we evaluate all text classifiers to choose the best text sentiment analysis model to use in Auto-Fusion. For this, we evaluated each model three times in classifying sentiments using only textual data in the B-T4SA data set, using the Adam optimizer [215], and cross-entropy loss.

In Table 3.3, the results for the evaluation of the training and validation set are shown. In this, the first block represents the results for two lexicon-based models (VADER and TextBlob), the second block shows the results for deep learning models, and the third

Table 3.3: Mean accuracy and standard deviation on B-T4SA data set. The first block shows the results for lexicon-based methods. The second block shows the result of the implemented deep learning methods. In the third block, we show the results of the best method from the second block (RCNN), pre-training on SST and fine-tuning on B-T4SA. Each model was evaluated three times under the same conditions.

	Mean Accuracy (%)	
Method	Train	Validation
VADER	41.04 ± 0	41.02 ± 0
VADER-PP	56.84 ± 0	56.82 ± 0
Textblob	64.22 ± 0	64.27 ± 0
Textblob-PP	64.88 ± 0	64.78 ± 0
FastText	42.86 ± 0.03	42.76 ± 0.05
LSTM	33.33 ± 0.04	33.13 ± 0.00
LSTM-Attn	97.36 ± 0.03	93.48 ± 0.57
BI-LSTM	96.56 ± 0.97	94.35 ± 0.07
RNN	90.72 ± 0.78	91.24 ± 0.48
RCNN	98.12 ± 1.10	$\mathbf{94.61 \pm 0.03}$
TextCNN	95.47 ± 0.86	93.73 ± 0.00
sCNN	90.69 ± 0.10	92.69 ± 0.02
VDCNN9	94.18 ± 0.81	93.33 ± 0.23
VDCNN17	88.29 ± 2.28	92.05 ± 0.52
VDCNN29	93.90 ± 0.65	93.19 ± 0.18
VDCNN49	92.61 ± 0.33	92.81 ± 0.11
RCNN-sst *ft* B-T4SA FC	86.73 ± 0.69	86.51 ± 0.67
RCNN-sst *ft* B-T4SA	$\mathbf{98.60 \pm 1.29}$	94.60 ± 0.03

block represents the best model of the previous block with fine-tuning. In the first block, we show two results for both methods without the pre-processing step and with ("-PP"). Both methods achieved better results with the pre-processing step, showing that cleaning textual data to remove noise allows the methods to yield better results. In the second and third blocks, all experiments were conducted using pre-processed data. Here, it is possible to see that, except for FastText, which achieved approximately 42% and LSTM that was incapable of learning how to solve the task (even with different learning rates, hidden features, and optimizers), all models achieved a mean accuracy of over 90%. This is well above the plug-and-play methods and the state-of-the-art [3], which used TextBlob methods to achieve 64.27% accuracy. The best method was RCNN, which achieved a mean accuracy of 94.61%, outperforming all others. This can be justified due to the strong capability of RCNNs to evaluate a word based on its embeddings coupled with the right and left context, which are extracted by the recurrent structures. This combination allows features to be extracted more accurately when working on problems that require context, on which sentiment analysis is highly dependent. On the third block of the table, the results of fine-tuning the RCNN model are presented. In this case, the fine-tuning was performed by initially training the model on the SST-5 data set and then replacing its final classification layer to output three values instead of five. In detail, "RCNN-sst *ft* B-T4SA FC" was initially trained on SST-5 and then only on the linear layers were fine-tuned using B-T4SA, whereas "RCNN-sst *ft* B-T4SA" is fine-tuned entirely on B-T4SA. By conducting such experiments, it is possible to see that fine-tuning the entire model yields better results when

compared to only transfer learning and fine-tuning the last classification layers. However, these results do not improve upon the original model.

From the evaluation of all text classifiers, we selected RCNN as the text classifier for the proposed method since it presented the best performance in the validation set.

Table 3.4: Train and validation accuracy (%) of several neural networks in sentiment classification.

Method	**RGB**		**RGB+LBP**		**RGB**	
	Train	Val	Train	Val	Train	Val
Pre-trained	×		×		✓	
ResNet18	**47.4**	47.7	47.4	47.9	46.6	49.7
ResNet34	47.2	**49.8**	47.3	**48.0**	45.6	**49.8**
ResNet50	46.3	46.4	47.2	47.4	**48.5**	48.7
ResNet101	44.9	45.1	47.1	47.1	47.6	47.7
ResNet152	44.5	44.5	45.9	45.9	47.1	47.5
DenseNet161	46.9	47.1	**47.5**	47.5	47.2	47.3

Regarding the selection of the model to perform the IC, we first evaluated the pool of neural networks in the task of image sentiment analysis. Table 3.4 shows the results for all neural networks. Note that every network was evaluated using the same learning rate ($1e^{-3}$), Adam optimizer, and cross-entropy loss. The second row of the table indicates if the experiment was done using transfer learning, where models were initially trained on the Flicker and Instagram data set. We also evaluated how the different models behave with RBG images, and RBG with local binary patterns [216]. In the latter, the models receive four inputs channels, where the last is the local binary patterns. The goal of adding the fourth channel is to promote texture analysis, which can contribute for the sentiment polarity. The results show that classifying the sentiment of an image is challenging, mainly due to the subjectivity of the image and inter-class similarities, where images that belong to different classes are visually similar. Neither the addition of the local binary patterns nor pre-training the models improved the results when compared to using only RGB with randomly initialized weights. Furthermore, all models performed similarly, but ResNet34 was the best one, achieving 49.8% accuracy on the validation set.

Table 3.5: Accuracy (%) in the test set for the proposed method and a baseline using SVM.

Method	**Test Accuracy (%)**
SVM	95.16
AutoML-based Fusion **(ours)**	**95.19**

Based on the text classifier, RCNN, and the image classifier, ResNet34, we evaluated the performance of Auto-Fusion as a whole. For this, we searched for a fusion model using the method described in 3.2.4 with a maximum budget of two hours. The resulting fusion

method was a gradient boosting machine, which was the model that obtained the highest validation performance. Evaluating Auto-Fusion on the B-T4SA test set obtained an accuracy of 95.19% (Table 3.5). To validate that the performance of the proposed method is competitive, we evaluate the model with the same settings but with a different fusion classifier. For this, we used a Support Vector Machine (SVM), which achieved a final performance of 95.16%. The difference between the searched fusion method and the SVM classifier is small, 0.03%, but the automatically searched method has several advantages: the time required to train is substantially less, requiring only two hours while SVM required several hours. More, SVMs tend not to scale well. The more features are added to the training data, the slower they can fit the data. In the case of Auto-Fusion, the proposed methodology to search for a classifier is extremely robust to the number of features in the training set, as the search space contains methods that can handle a large number of features without becoming untenable. The SVM baseline consolidates the proposed method by showing that the proposed architecture can obtain excellent performance.

Table 3.6: Comparison of the proposed methods with the state-of-the-art. TM stands for substituting the text classifier from [3] for the one selected in our experiments.

Method	**B-T4SA Test Accuracy (%)**
Random Classifier	33.33
Hybrid-T4SA FT-F [210]	49.90
Hybrid-T4SA FT-A [210]	49.10
VGG-T4SA FT-F [210]	50.60
VGG-T4SA FT-A [210]	51.30
Information Fusion [3]	60.42
Information Fusion [3] (TM)	76.35
SVM-fusion (**ours**)	95.16
AutoML-based Fusion **(ours)**	**95.19**

To further validate the results obtained by Auto-Fusion we compare it with state-of-the-art methods in Table 3.6. In this, it is possible to see that Auto-Fusion outperforms existing approaches by more than 35%. More, we also improved upon the proposed method in [3] by replacing its text classifier with our RCNN, resulting in a 15.9% accuracy improvement (represented in the table by Information Fusion (TM)). However, this is still 18.8% below Auto-Fusion, showing that the proposed method of fusing the individual classifications and then performing a random search to find the optimal fusion classifier is an efficient method.

3.4 Conclusions

This chapter was initially devoted to studying how different neural networks work and behave in the task of detecting surface defects and multimodal sentiment analysis. Then, based on the results obtained by evaluating several neural networks, we propose two methods for defect detection: CNN-Fusion, which fuses the different CNN classifications into

a final one, and Auto-Classifier, which leverages the feature extraction capabilities of a state-of-the-art CNNs and complements them by performing an automated search for a new classifier component; and one method for multimodal sentiment analysis that performs individual text and ICs and fuses them with a searched ML that performs a final classification.

The results obtained by the three proposed methods show that improving existing state-of-the-art CNNs with AutoML and NAS can further improve their classification capabilities. With this, we are motivated and convinced that NAS can be used to generate architectures (or components) to solve different CV problems, if the search can be executed in useful time.

In the next chapter, we focus on extending NAS field by proposing two methods: a zero-proxy estimator that is capable of scoring architectures at initialization stage as a metric to their performance if trained, and a EA that efficiently guides the search by quickly evaluating architectures to extract knowledge of the search space.

Chapter 4

Guided Neural Architecture Search Through Efficient Performance Estimation

4.1 Introduction

Despite excellent results obtained by architectures designed with prominent NAS methods, the computational cost of most NAS approaches is high, which in some cases can be in the order of months of GPU computation [15, 217, 38]. To mitigate this, approaches focus on a cell-based design, where NAS methods design small cells that are replicated through an outer-skeleton, thus alleviating the complexity of the search space [218, 15, 38, 219]. Furthermore, several performance estimation strategies have been proposed to reduce the time constraint of NAS methods, by mainly conducting low-fidelity estimates, learning curve extrapolations, statistical approaches [162, 12] or by proposing one-shot methods, where the weights of the generated models are inherited from a super-network [2, 70, 114]. However, searching through high-dimensional search spaces is highly complex, even when there is some prior knowledge about the space. The most reliable approach to obtain information about the search space while searching is to fully train generated architectures and optimize the search based on the most performant ones. However, this is extremely costly, and results are highly dependent on the training schemes and initialization setups [40].

To mitigate the aforementioned problems, in this chapter we propose a zero-proxy estimation mechanism and an evolutionary-based NAS search method that uses the former to efficiently guide the search. By evaluating how the gradients of an architecture behave with respect to the input, we show that it is possible to score untrained architectures, eliminating the need to train generated architectures to update parameters. Coupling this scoring with an evolutionary strategy where operations are mutated and younger architectures are preferred, the proposed search method can efficiently search a complex search space while forcing exploitation of the most performant architectures, and exploration of the search space by performing mutations. By doing so, the proposed method is capable of continuously extracting knowledge about the search space without compromising the search, resulting in state-of-the-art results in NAS-Bench-101, NAS-Bench-201, and TransNAS-Bench-101 search spaces. Figure 4.1 shows the process of evaluating generated architectures.

The contributions of the methods proposed in this chapter can be summarized as:

- We propose a novel performance estimation strategy that can score untrained architectures to infer their final performance if trained, which can be easily integrated

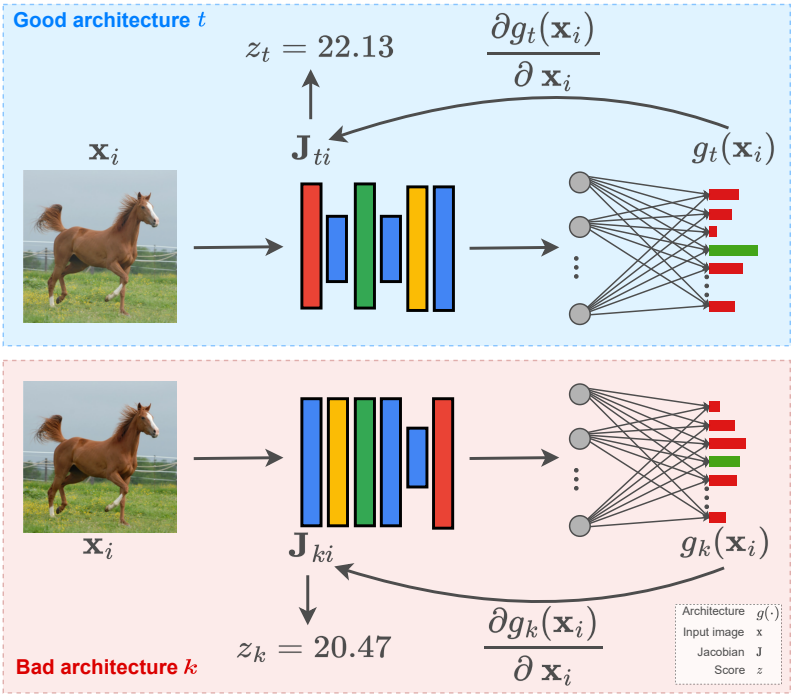


Figure 4.1: Example of scoring two different architectures using the same input. Generated architectures in each generation are ranked based on a score that correlates with their final performance, which determines which architecture is selected to be part of the population.

into almost any NAS method.

- We propose a guided NAS method based on evolutionary strategies and zero-proxy estimation to generate image classifier architectures (CNNs).

- We empirically show that guided mechanisms can be used without compromising time efficiency or the generated model's performance. Also, we detail the algorithm, emphasizing the accessible transferability of the guiding mechanism.

- We achieve state-of-the-art results on all data sets of NAS-Bench-101, NAS-Bench-201, and TransNAS-Bench-101 benchmarks, thus showing the generability of the proposed search method.

- We perform extensive ablation studies and show the importance of different parameters and regularization, thus shedding insights for the design of NAS evolutionary models.

The remainder of this chapter is organized as follows. Section 4.2 describes the proposed zero-cost proxy performance estimator and the proposed guided evolutionary algorithm to perform the search. Section 4.3, presents the experiments performed, the results, and a discussion. Finally, a conclusion is drawn in Section 4.4.

4.2 Proposed Method

4.2.1 Zero-cost Proxy Performance Estimation

In this section, we propose a zero-cost proxy performance estimation strategy, whose goal is to estimate the performance of generated architectures without requiring any training, neither for the generated architectures nor for the performance estimator. To do this, we score untrained architectures as an indicator of their accuracy when trained.

To evaluate the generated architectures, we look at the behavior of local linear operators using different data points at initialization stage. The local linear operators are obtained by multiplying the linear maps at each layer interspersed with the binary rectification units. To do this, one can define a linear mapping, $w_i = g(\mathbf{x}_i)$, which maps the input $\mathbf{x}_i \in \mathbb{R}^D$, through the architecture, $g(\mathbf{x}_i)$, where $\mathbf{x}_i$ represents an image that belongs to a batch $\mathbf{X}$, and D is the input dimension. Then, the Jacobian of the linear map can be computed using:

$$\mathbf{J}_i = \frac{\partial g(\mathbf{x}_i)}{\partial \mathbf{x}_i} \quad (4.1)$$

This allows us to evaluate the architecture's behavior for different images by calculating $\mathbf{J}_i$ for different data points, $g(\mathbf{x}_i)$, of a single batch $\mathbf{X}$, $i \in 1, \cdots, N$:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial g(\mathbf{x}_1)}{\partial \mathbf{x}_1} & \frac{\partial g(\mathbf{x}_2)}{\partial \mathbf{x}_2} & \cdots & \frac{\partial g(\mathbf{x}_N)}{\partial \mathbf{x}_N} \end{pmatrix}^\top \quad (4.2)$$

$\mathbf{J}$ contains information about the architecture's output with respect to the input for several data points (images). Based on those, we can evaluate how points belonging to the same class correlate with each other, allowing the evaluation of how an untrained architecture is capable of modeling complex functions. Explicitly, a flexible architecture should simultaneously be able to distinguish local linear operators for each data point but also have similar results for similar data points, which in a supervised approach means that the data points belong to the same or very similar classes. In a perfect scenario, an untrained architecture would have a low correlation between data points from different classes, and a higher correlation between data points from the same class, meaning that the architecture would easily learn to distinguish the two data points during training. To measure this behavior, we evaluate the correlation of $\mathbf{J}$ values with respect to their class, by splitting $\mathbf{J}$ into several sets, where each set, $\mathbf{M}_k$, contains all $\mathbf{J}_i$ that belong to the same class k. Then, we can calculate a per-class correlation matrix, $\mathbf{\Sigma}_{\mathbf{M}_k}$, using the obtained sets, $\mathbf{M}_k$, where $k = 1, ...K$.

Individual correlation matrices allow the analysis of how an untrained architecture behaves for images from each class, which is an indication of the ability of the local linear operators to perceive differences between classes. However, different correlation matrices

might yield different sizes, as the number of images per class differs. To be able to compare different correlation matrices, they are individually evaluated:

$$\mathbf{E}_k = \begin{cases} \sum_{i=1}^{N}\sum_{j=1}^{N} \log(|(\mathbf{\Sigma}_{\mathbf{M}k})_{i,j}| + t), & \text{if } K \leq \tau \\ \\ \frac{\sum_{i=1}^{N}\sum_{j=1}^{N} \log(|(\mathbf{\Sigma}_{\mathbf{M}k})_{i,j}| + t)}{\sqrt{\#\mathbf{\Sigma}_{\mathbf{M}k}}}, & \text{otherwise} \end{cases} \tag{4.3}$$

where t is a small-constant with the value of 1×10^{-5}, K is the number of classes in batch $\mathbf{X}$, and $\#$ represents the number of elements.

Finally, an architecture is scored based on the individual evaluations of the correlation matrices by:

$$z = \begin{cases} \sum_{w=1}^{K} |\mathbf{e}_w|, & \text{if } K \leq \tau \\ \\ \frac{\sum_{i=1}^{K}\sum_{j=i+1}^{K} |\mathbf{e}_i - \mathbf{e}_j|}{\#\mathbf{e}}, & \text{otherwise} \end{cases} \tag{4.4}$$

where $\mathbf{e}$ contains all the correlation matrices' scores. The final score is dependent on the number of classes present in $\mathbf{X}$, as data sets with a higher number of classes commonly have more noise, which is mitigated by conducting a normalized pair-wise difference. For the conducted experiments, we empirically defined $\tau = 100$, based on the search space and data sets used.

We can then use z to rank the generated architectures, providing an efficient mechanism of differentiating between good and bad architectures, thus allowing the search to be guided towards better settings without compromising the search cost.

4.2.2 Guided Evolution

In this section, we propose GEA, a NAS search method that leverages the proposed zero-cost proxy estimation mechanism to score untrained architectures and efficiently guide the search through evolution. GEA is summarised in Algorithm 1. In detail, GEA starts by randomly generating c architectures from the search space of possible architectures, $\mathcal{A}$. The architectures that belong to the search space have equal probabilities of being randomly sampled. Sampled architectures are then evaluated using the proposed zero-proxy estimator to score the architectures at initialization stage, without requiring any training (the zero-proxy estimation mechanism is detailed in section 4.2.1). Then, from the c scored architectures, only the top p scoring ones are added to the population and trained to extract their fitness, f. The fitness, f, is the validation accuracy after a partial train (small number of epochs). By scoring c architectures at initialization stage, GEA acquires knowledge of the search space, which is then exploited by selecting the top performant architecture, therefore guiding the upcoming search by weeding out bad architectures.

Algorithm 1 Guided Evolution

```
population ← empty queue                                   ▷ Population.
history ← ∅                                                ▷ Models history.
while #population < c do                                   ▷ Initialize population.
    model.arch ← RandomArchitecture()
    model.accuracy ← ZeroProxy(model.arch)
    add model to right of population                       ▷ Force model age
end while
drop the c − p worst individuals from population
for model ∈ population do
    model.accuracy ← TrainAndEval(model.arch)
    add model to history
end for
while #history < c do
    sample ← s random candidates from the population (with replacement)
    parent ← highest-accuracy model in sample
    generation ← ∅
    while #generation < p do
        child.arch ← Mutate(parent.arch)
        child.accuracy ← ZeroProxy(child.arch)
        add child to generation
    end while
    top_child ← highest-performant model in generation
    top_child.accuracy ← TrainAndEval(top_child.arch)
    add top_child to right of population
    add top_child to history
    remove dead from left of population                    ▷ Oldest model.
end while
return highest-accuracy model in history                   ▷ Most performant.
```

Once the initial population is defined, the evolution takes place for c cycles. In each generation, the first step is to perform a tournament selection. For this, s architectures are randomly and uniformly sampled from the population. Then, the architecture with the highest fitness score, f, from the pool of s architectures is selected to be the parent of the next generation (cycle). To generate new architectures, GEA performs a mutation over the parent architecture. The mutation works by randomly mutating one operation of the architecture for another one from the pool of operations. An example of a mutation using the NAS-Bench-201 search space is visually represented in Figure 4.2. p new architectures are generated in each generation by performing operation mutations over the selected parent, which are then scored using the zero-proxy estimator. The highest-scoring architecture is kept and added to the population after evaluating its fitness. Generating and evaluating p architectures strengthens the search method to find the best direction for the parent's evolution across the search space. This allows the method to be guided through a complex space without jeopardizing the time required to perform the evolution or the search method's complexity. When a new architecture is added to the population, a regularization mechanism (survivor selection) takes place, where the oldest architecture is removed and discarded, thus forcing exploration of the search space by favoring younger

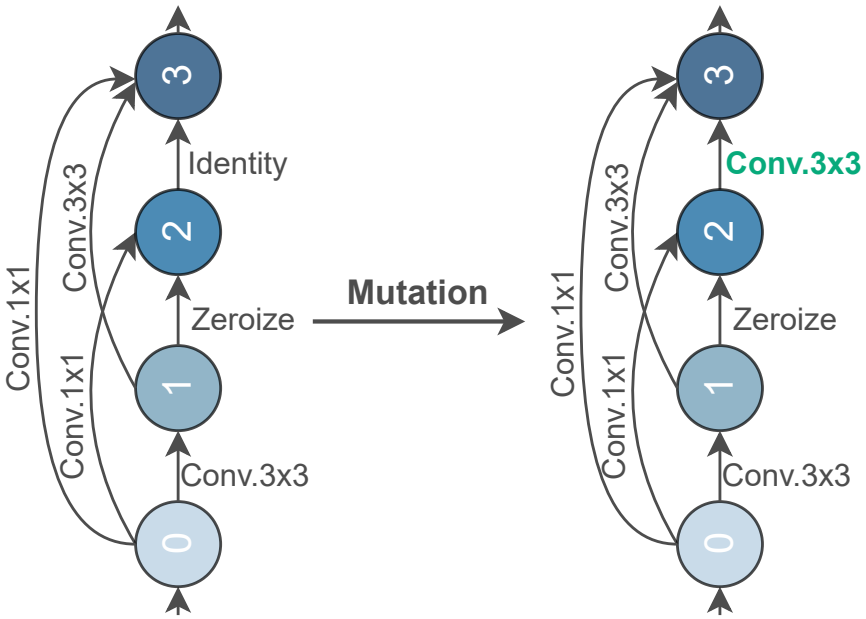


Figure 4.2: Example of mutating one operation of a cell using the NAS-Bench-201 search space: operation Identity becomes Conv. 3×3.

architectures that represent new settings evolved by prior acquired knowledge.

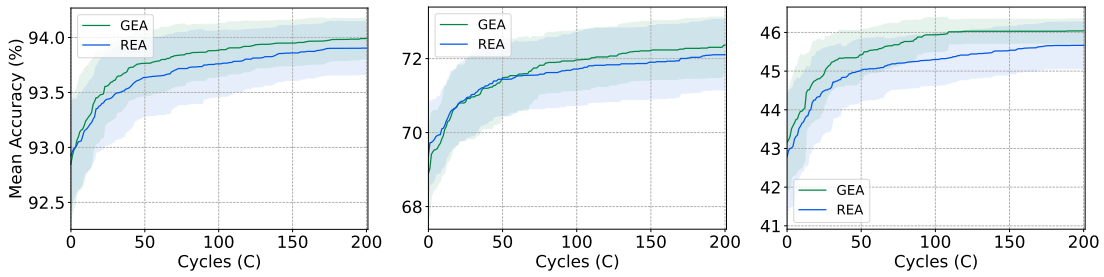
Inherently, higher p values represent a higher degree of exploration of the search space, while higher s values represent higher exploitation by increasing the probability of the best architectures in the population being selected as parents for the next generation.

4.3 Experiments

To evaluate the effectiveness of the proposed NAS method, we utilize three different search spaces: NAS-Bench-101 [173], NAS-Bench-201 [42] and TransNAS-Bench-101 [87] benchmarks. As benchmarks were designed to have tractable NAS search spaces with metadata for the training of thousands of architectures within those search spaces, it allows evaluating the proposed method in a common setting and fairly comparing it with previous proposals (detailed description of NAS benchmarks in Section 2.3).

Table 4.1: Mean test accuracy (%) and standard deviation across 50 runs in NAS-Bench-101 CIFAR-10 data set. Experiments with REA and GEA were performed with $p/s/c = 10/5/200$.

Method	Search Time (s) $\downarrow$	Mean Test Accuracy (%) $\uparrow$
RS [220]	N/A	90.38 ± 5.51
REA [72]	26676.49	93.12 ± 0.48
GEA (ours)	30128.32	$\mathbf{93.99 \pm 0.25}$



(a) CIFAR-10 (b) CIFAR-100 (c) ImageNet16-120

Figure 4.3: Mean accuracy and standard deviation over 25 runs of the proposed search method, GEA, and direct comparison with REA for different cycles (c) across CIFAR-10, CIFAR-100, and ImageNet16-120 data sets.

Table 4.2: Comparison of manually designed architectures and several NAS methods using the NAS-Bench-201 benchmark. Performance is shown in terms of accuracy (%) with mean and standard deviation, on CIFAR-10, CIFAR-100, and ImageNet-16-120. Search times are the mean time required to search for cells in CIFAR-10. Search time includes the time taken to train architectures as part of the process where applicable.

Method	Search Time (s)↓	CIFAR-10		CIFAR-100		ImageNet-16-120	
		Val. Acc (%)↑	Test Acc. (%)↑	Val. Acc (%)↑	Test Acc. (%)↑	Val. Acc (%)↑	Test Acc. (%)↑
Manually designed							
ResNet [59]	-	90.83	93.97	70.42	70.86	44.53	43.63
Weight sharing							
RSPS [1]	7587	84.16 ± 1.69	87.66 ± 1.69	59.00 ± 4.60	58.33 ± 4.34	31.56 ± 3.28	31.14 ± 3.88
DARTS, 1st order [2]	10890	39.77 ± 0.00	54.30 ± 0.00	15.03 ± 0.00	15.61 ± 0.00	16.43 ± 0.00	16.32 ± 0.00
DARTS, 2nd order [2]	29902	39.77 ± 0.00	54.30 ± 0.00	15.03 ± 0.00	15.61 ± 0.00	16.43 ± 0.00	16.32 ± 0.00
GDAS [156]	28926	90.00 ± 0.21	93.51 ± 0.13	71.14 ± 0.27	70.61 ± 0.26	41.70 ± 1.26	41.84 ± 0.90
SETN [221]	31010	82.25 ± 5.17	86.19 ± 4.63	56.86 ± 7.59	56.87 ± 7.77	32.54 ± 3.63	31.90 ± 4.07
ENAS [70]	13315	39.77 ± 0.00	54.30 ± 0.00	15.03 ± 0.00	15.61 ± 0.00	16.43 ± 0.00	16.32 ± 0.00
Non-weight sharing							
RS [220]	12000	90.93 ± 0.36	93.70 ± 0.36	70.93 ± 1.09	71.04 ± 1.07	44.45 ± 1.10	44.57 ± 1.25
REINFORCE [94]	12000	91.09 ± 0.37	93.85 ± 0.37	71.61 ± 1.12	71.71 ± 1.09	45.05 ± 1.02	45.24 ± 1.18
BOHB [137]	12000	90.82 ± 0.53	93.61 ± 0.52	70.74 ± 1.29	70.85 ± 1.28	44.26 ± 1.36	44.42 ± 1.49
REA [72]†	26070	91.22 ± 0.25	93.97 ± 0.31	72.36 ± 1.07	72.14 ± 0.86	45.09 ± 0.92	45.55 ± 1.02
GEA (ours)†	26911	$\mathbf{91.26 \pm 0.20}$	$\mathbf{93.99 \pm 0.23}$	$\mathbf{72.62 \pm 0.77}$	$\mathbf{72.36 \pm 0.66}$	$\mathbf{45.97 \pm 0.72}$	$\mathbf{46.04 \pm 0.67}$

† Results of 25 runs using the same settings: $p/s/c = 10/5/200$, using a single 1080Ti GPU.

4.3.1 Results and Discussion

First, we evaluate the proposed method on NAS-Bench-101. For this, we fixed $p/s/c = 10/5/200$, following standard settings used and assessed by prior works [42, 72], and directly compare it against RS and REA [72]. In Table 4.1 we present this comparison in terms of search cost, in seconds, and mean test accuracy and standard deviation, calculated from running GEA and REA 50 times. From the results, it is clear that GEA outperforms REA and heavily improves when compared to RS. GEA is highly efficient, requiring only 0.35 GPU days to complete each run. The results show that the guiding mechanism can improve the search, promoting regions that yield better architectures in terms of accuracy.

Then, we evaluated GEA using the NAS-Bench-201 search space. The first experiment in this search space was to directly compare GEA with REA for a different number of

generations/cycles, c. This also allows evaluating the importance of c, which is the main parameter that inherently defines the time required for the search procedure. Higher c values will take longer to finish. More, c establishes the number of architectures that are evaluated: $c \times p$ architectures (p per cycle) are generated and evaluated using the zero-proxy estimation method to provide information about the search space, from which c architectures (1 per cycle) are selected and trained. The results presented in Figure 4.3 are expressed as mean test accuracy and standard deviation as colored areas, obtained by the best architecture found by each method for different c values over 25 different runs. In this experiment, the p/s used to allow a fair comparison was set to $p/s = 10/5$, following typical settings used by prior works [42, 72]. The results show that across all data sets, GEA consistently outperforms REA, and is capable of converging to better results even for small numbers of c. These results demonstrate that the search method converges more quickly to regions of the search space that contain better architectures by leveraging the guided mechanism. Also, on ImageNet16-120, the noisier data set on NAS-Bench-201, the result from the T-test analysis was $\rho = 0.033$, thus showing a statistical significance between the results obtained by GEA when compared to REA. Note that for our proposed method, GEA, p value means that at any given time of the search, the population is equal to 10 architectures and that those, a pool of parents is sampled, where 5 architectures are sampled with replacement. From the pool of possible parents, one is selected. The sampled parent then generates p architectures through mutation per cycle, which are evaluated using the zero-proxy estimator wherein only the top-scoring architecture is selected to integrate the population. By selecting $s >= 1$ architectures to have the opportunity of being a parent, we are leveraging the intrinsic exploitation characteristics of the evolutionary strategy, whereas by generating p architectures, we are forcing exploitation that guides the search more effectively.

In Table 4.2 we further compare GEA, using $p/s/c = 10/5/200$, against several state-of-the-art methods on the NAS-Bench-201 search space, using as evaluation metrics the mean accuracy, standard deviation, and search time in seconds. Across all 3 data sets, GEA consistently outperforms both weight sharing and non-weight sharing NAS methods, achieving state-of-the-art results. Moreover, GEA is extremely efficient in terms of search time, requiring only 0.3 GPU days to complete the search. Even though GEA evaluates $c \times p$ architectures with the zero-proxy estimator and further evaluates c architectures by partially training them, it requires a similar search time as REA under the same settings and considerably less than most weight-sharing methods. Lower standard deviation further indicates that GEA is precise and capable of generating high-performant architectures, which is especially valid in ImageNet16-120, a data set with low-resolution images and high levels of noise, in which GEA considerably outperforms existing NAS methods.

Finally, we evaluate GEA on all 7 tasks from TransNAS-bench-101. Evaluating GEA on several tasks contributes to validating its generability and transferability across different problems, which is where NAS methods commonly fail [40, 42, 45]. For this, we conduc-

Table 4.3: Performance comparison of different NAS methods on TransNAS-Bench-101. The first block shows the results for directly searching on each task. The second block shows the transferred versions of different methods, which are pretrained on the least time-consuming task, i.e., Jigsaw. The final row shows the possible best result in each task.

	Tasks	Cls. Object	Cls. Scene	Autoencoding	Surf. Normal	Sem. Segment.	Room Layout	Jigsaw
	Metric	*Acc. (%)* ↑	*Acc. (%)* ↑	*SSIM* ↑	*SSIM* ↑	*mIoU* ↑	*L2 loss* ↓	*Acc. (%)* ↑
Direct Search	RS [220]	45.16 ± 0.4	54.41 ± 0.3	55.94 ± 0.8	56.85 ± 0.6	25.21 ± 0.4	61.48 ± 0.8	94.47 ± 0.3
	REA [72]	45.39 ± 0.2	54.62 ± 0.2	56.96 ± 0.1	57.22 ± 0.3	25.52 ± 0.3	61.75 ± 0.8	94.62 ± 0.3
	PPO [96]	45.19 ± 0.3	54.37 ± 0.2	55.83 ± 0.7	56.90 ± 0.6	25.24 ± 0.3	61.38 ± 0.7	94.46 ± 0.3
	DT	42.03 ± 5.0	49.80 ± 8.6	51.20 ± 3.3	55.03 ± 2.7	22.45 ± 3.2	66.98 ± 2.3	88.95 ± 9.1
	BONAS [222]†	45.50	54.56	56.73	57.46	25.32	61.10	94.81
	weakNAS [223]†	45.66	54.72	56.77	57.21	25.90	60.31	94.63
	Arch-Graph-single [224]†	45.48	54.70	56.52	57.53	25.71	61.05	94.66
	GEA (Ours)	45.98 ± 0.2	54.85 ± 0.1	57.11 ± 0.3	58.33 ± 1.0	25.95 ± 0.2	59.93 ± 0.5	94.96 ± 0.2
	GEA-Best (Ours)†	**46.32**	**54.94**	**57.72**	**59.62**	**26.27**	**59.38**	**95.37**
Transfer Search	REA-t [72]	45.51 ± 0.3	54.61 ± 0.2	56.52 ± 0.6	57.20 ± 0.7	25.46 ± 0.4	61.04 ± 1.0	-
	PPO-t [96]	44.81 ± 0.6	54.15 ± 0.5	55.70 ± 1.5	56.60 ± 0.7	24.89 ± 0.5	62.01 ± 1.0	-
	CATCH [225]	45.27 ± 0.5	54.38 ± 0.2	56.13 ± 0.7	56.99 ± 0.6	25.38 ± 0.4	60.70 ± 0.7	-
	BONAS-t [222]†	45.38	54.57	56.18	57.24	25.24	60.93	-
	weakNAS-t [223]†	45.29	54.78	56.90	57.19	25.41	60.70	-
	Arch-Graph-zero [224]†	45.64	54.80	56.61	57.90	25.73	60.21	-
	Arch-Graph [224]†	45.81	54.90	56.58	58.27	25.69	60.08	-
	GEA-t (Ours)	46.03 ± 0.3	54.86 ± 0.1	56.99 ± 0.3	58.21 ± 0.9	25.88 ± 0.3	59.85 ± 0.5	-
	GEA-t-Best (Ours)†	**46.32**	**54.94**	**57.72**	**59.62**	**26.27**	**59.38**	-
	Global Best	46.32	54.94	57.72	59.62	26.27	59.38	95.37

† Results provided for the best run only.

ted two different experiments: i) directly searching on each task independently; and ii) performing transfer search. For the latter, we followed common procedures [87], where we first search on jigsaw and use the final population as initialization for the evolution when searching on the other tasks. The results for both experiments are shown in Table 4.3. From the results, it is possible to see that: first, directly searching on each task is an effective approach, where GEA is capable of achieving a higher mean performance obtained from 25 runs, which, in all tasks, than any other NAS method. Also, when looking only at the best result, GEA is capable of achieving the best possible results in TransNAS-Bench-101, meaning that GEA is capable of generating the most optimal architecture for each task. The same behaviors are present when transferring the knowledge from jigsaw to other tasks, where GEA-t achieves state-of-the-art results on all tasks. When compared to directly searching on each task, GEA-t has a slight improvement only on classification tasks, meaning that GEA does not require prior information to achieve state-of-the-art performances when compared to other NAS methods.

The obtained results in all 3 benchmarks, which contain 11 different data sets, show that an evolutionary strategy, coupled with a mechanism to quickly evaluate architectures to guide the search, can achieve state-of-the-art results while still having competitive search times. Despite the complexity of search spaces and severe difficulty in obtaining their global information, the results show that guiding mechanisms powered by scoring architectures at initialization stages have the advantage of acquiring preliminary information regarding in which direction the search should evolve. Therefore, GEA can quickly converge to better results by avoiding local minima, while still being efficient in terms of the

required time.

Table 4.4: Comparison of different performance estimation methods on NAS-Bench-201 benchmark. Performance is shown in accuracy with mean and standard deviation, on CIFAR-10, CIFAR-100, and ImageNet-16-120. Search times are the mean time required to search for cells in CIFAR-10, using a single 1080Ti GPU. Search time includes the time taken to train architectures as part of the process where applicable. For each sample size, the optimal architecture is also shown.

Method	Search Time (s)	CIFAR-10		CIFAR-100		ImageNet-16-120	
		validation	test	validation	test	validation	test
Training-free							
NAS-WOT ($N = 10$) [153] †	3.1	89.56 ± 0.56	92.47 ± 0.04	69.36 ± 1.55	69.20 ± 1.05	42.08 ± 1.61	$\mathbf{42.20 \pm 1.37}$
Ours+RS ($N = 10$)	**2.3**	89.90 ± 0.21	$\mathbf{92.63 \pm 0.32}$	69.78 ± 2.44	$\mathbf{70.10 \pm 1.71}$	41.73 ± 3.60	41.92 ± 4.25
NAS-WOT ($N = 100$) [153] †	25.7	89.91 ± 0.80	91.41 ± 2.24	67.13 ± 4.03	67.18 ± 4.14	41.39 ± 1.13	$\mathbf{41.42 \pm 1.53}$
Ours+RS ($N = 100$)	**20.5**	88.74 ± 3.16	$\mathbf{91.59 \pm 0.87}$	67.28 ± 3.68	$\mathbf{67.19 \pm 3.82}$	38.66 ± 4.75	38.80 ± 5.41
NAS-WOT ($N = 500$) [153] †	126.8	88.73 ± 0.81	91.71 ± 1.37	67.62 ± 1.61	67.54 ± 2.23	39.37 ± 3.01	39.84 ± 3.68
Ours+RS ($N = 500$)	**105.8**	88.17 ± 1.35	$\mathbf{92.27 \pm 1.75}$	69.23 ± 0.62	$\mathbf{69.33 \pm 0.66}$	41.93 ± 3.19	$\mathbf{42.05 \pm 3.09}$
NAS-WOT ($N = 1000$) [153] †	252.6	89.60 ± 0.90	91.20 ± 2.04	68.57 ± 0.41	68.95 ± 0.72	38.01 ± 1.66	38.08 ± 1.58
Ours+RS ($N = 1000$)	**206.2**	87.87 ± 0.85	$\mathbf{91.31 \pm 1.69}$	69.44 ± 0.83	$\mathbf{69.58 \pm 0.83}$	41.86 ± 2.33	$\mathbf{41.84 \pm 2.06}$
Optimal ($N = 10$)	N/A	90.00 ± 0.95	93.41 ± 0.45	70.11 ± 1.70	70.11 ± 1.70	44.67 ± 1.87	44.67 ± 1.87
Optimal ($N = 100$)	N/A	91.12 ± 0.11	94.12 ± 0.21	72.73 ± 0.78	72.73 ± 0.78	46.31 ± 0.47	46.31 ± 0.47
Optimal ($N = 500$)	N/A	91.15 ± 0.12	94.13 ± 0.22	72.83 ± 0.64	72.83 ± 0.64	46.06 ± 0.66	46.06 ± 0.66
Optimal ($N = 1000$)	N/A	91.24 ± 0.21	94.19 ± 0.15	72.92 ± 0.53	72.92 ± 0.53	46.57 ± 0.59	46.57 ± 0.59
Surrogate Estimator with Training							
SVM ($N = 10$)	359426.3‡	89.74 ± 1.10	92.80 ± 0.97	65.21 ± 6.48	65.46 ± 6.37	37.50 ± 8.56	37.31 ± 8.66
SVM ($N = 100$)	359449.4‡	87.03 ± 2.33	92.68 ± 1.47	62.82 ± 5.75	63.25 ± 5.70	41.57 ± 3.55	41.73 ± 3.55
SVM ($N = 500$)	359547.7‡	87.37 ± 2.63	93.05 ± 0.71	66.83 ± 4.34	67.36 ± 4.28	41.84 ± 1.38	41.49 ± 1.39
SVM ($N = 1000$)	359666.2‡	87.06 ± 3.14	91.24 ± 2.28	68.40 ± 0.48	69.02 ± 0.84	41.32 ± 1.31	41.19 ± 1.29

† Results obtained by running the author's publicly available code 3 times with the same settings as the proposed method.
‡ Includes the time required to train, which was done using information about the performance of 100 fully trained architectures, which collectively required 4.16 training days.

Table 4.5: Ablation studies for the number of parent candidates, s, the population size, p, and the regularization mechanism to remove individuals from the population. Results are shown in mean validation accuracy (%) and standard deviation from 5 runs in NAS-Bench-201 CIFAR-10 data set.

Parameter	Value	Mean Validation Accuracy (%)
s	1	91.09 ± 0.45
	3	91.45 ± 0.20
	5	91.41 ± 0.24
	7	91.47 ± 0.16
	10	$\mathbf{91.56 \pm 0.05}$
	Highest	91.50 ± 0.19
	Lowest	89.93 ± 0.55
p	1	91.19 ± 0.10
	3	91.45 ± 0.09
	5	$\mathbf{91.58 \pm 0.02}$
	7	91.55 ± 0.06
	10	91.41 ± 0.24
Regularization	Oldest	$\mathbf{91.56 \pm 0.05}$
	Highest	90.59 ± 0.46
	Lowest	91.30 ± 0.23

4.3.2 Ablation Studies

This section extends the study about the importance of different parameters used by the proposed method. First, we look into the importance of the parameter s by i) incrementally increasing its value from 1 to 10, and ii) randomly sampling s architectures from the pool of candidates by simply selecting the architectures from the population pool with the highest and lowest fitnesses. For these experiments, p and c were fixed to 10 and 200 respectively. In Table 4.5, a clear pattern can be seen, where the best results are obtained when s is higher. Logically, sampling the lowest-scoring individual to be the parent of the next generation yields the worst results, as this forces the evolution to follow the worst-known settings. $p = 10$ achieved a better mean validation accuracy than sampling the highest-scoring individual. We justify that this is due to the fact that by having a high p value, it allows that most of the time, one of the best architectures is chosen to be parent, while at the same time, promoting exploration of the search space by not using the best-known setting every time. A visual representation of the evolution of the best architecture for the different parameter values, over 5 different runs, can be seen in Figure 4.4.

We also evaluate the importance of the population size, p. Similarly to s, we incrementally increase p from 1 to 10. From Table 4.5, it is possible to see that higher values of p achieve better results than lower values and the best results are obtained with $p = 5$. This is due to the fact that a smaller population size promotes exploitation, as the candidates sampled to be parents are more often among the best individuals, thus leading the search to better regions of the search space. In Figure 4.5 it is possible to see the evolution of the best architecture found by GEA for each p value evaluated over 5 different runs.

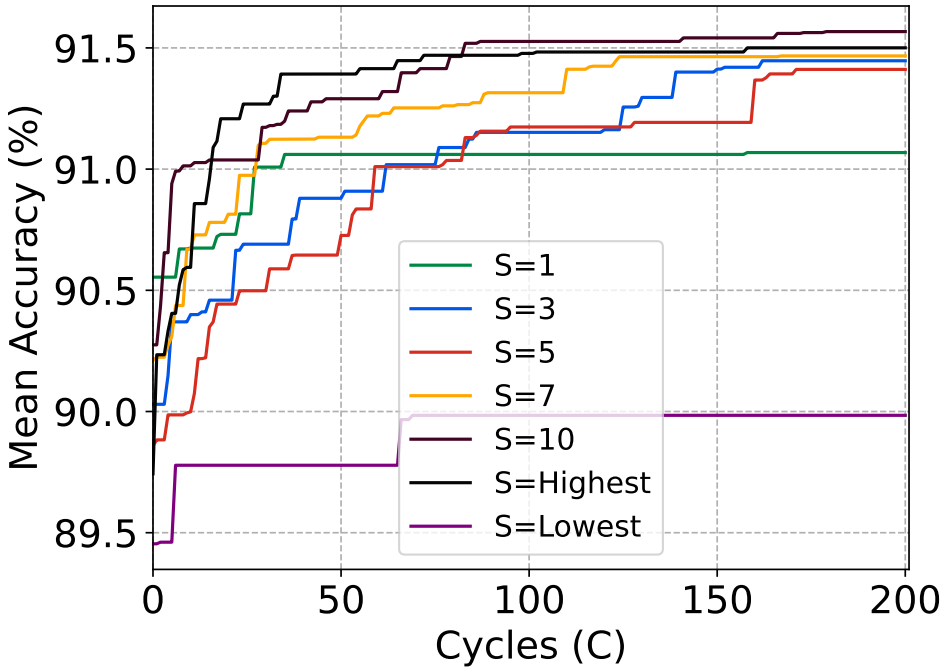


Figure 4.4: Mean validation accuracy (%) throughout the evolution for different parent sampling, s, schemes using NAS-Bench-201 CIFAR-10 for 5 runs.

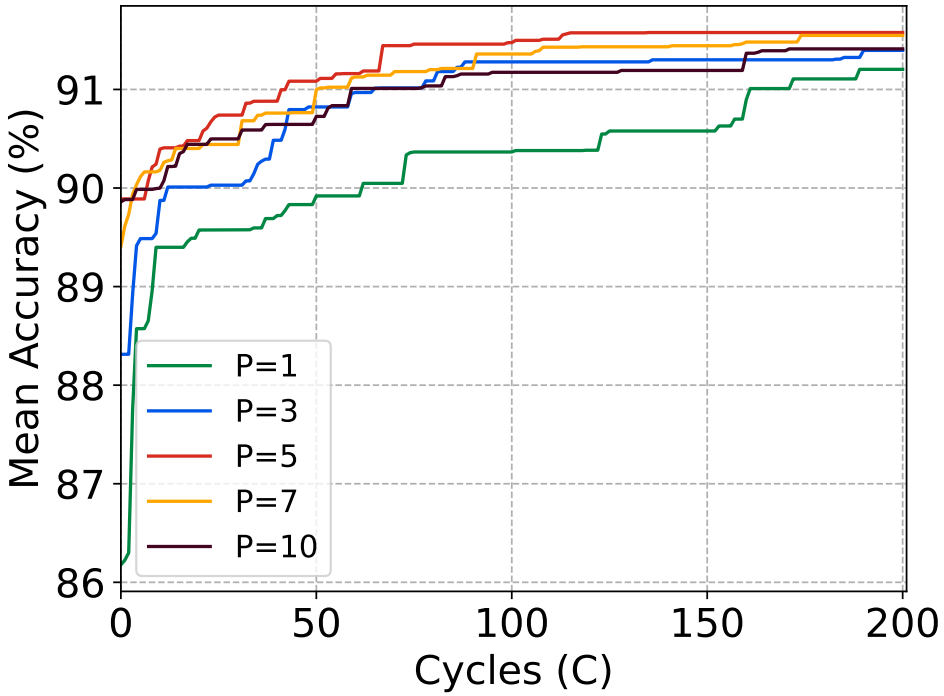


Figure 4.5: Mean validation accuracy (%) throughout the evolution for different population sizes, p, using NAS-Bench-201 CIFAR-10 for 5 runs.

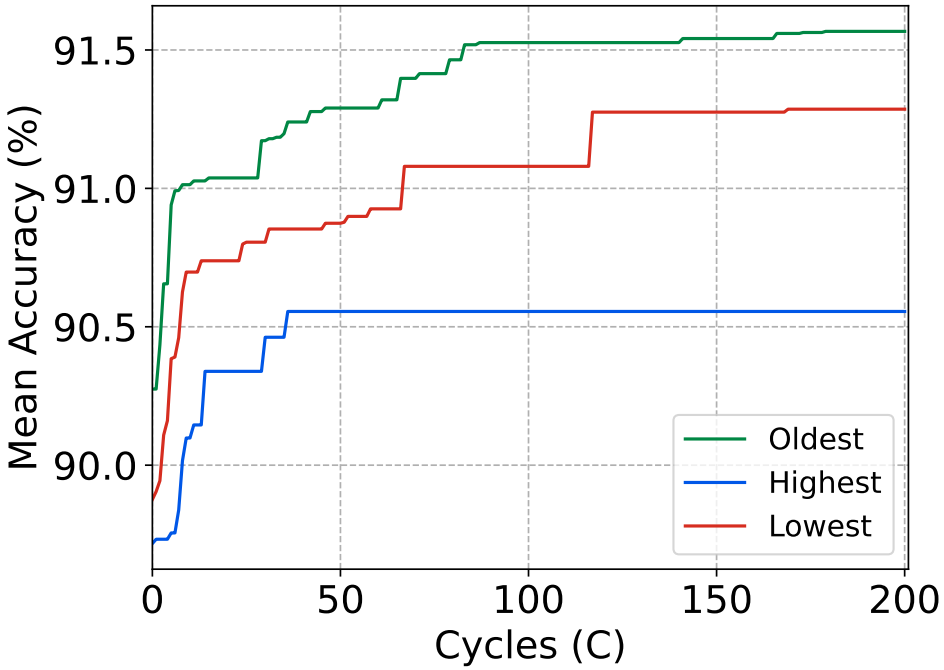


Figure 4.6: Mean validation accuracy (%) throughout the evolution for different regularization schemes using NAS-Bench-201 CIFAR-10 for 5 runs.

More, we evaluated the importance of the regularization mechanism that removes individuals from the population. For this, we assessed the already discussed elimination by age by removing the oldest individual in the population. Also, we evaluated the results if the best (highest fitness) and the worst (lowest fitness) individuals were removed

instead. From Table 4.5, it is clear that removing the best individual is the worst possible strategy, as it forces the search to ignore the best-known settings so far. Then, comparing removing the worst and the oldest, the best results are yielded when the oldest individual is removed, as it promotes further exploration of the search space. The results shown in Figure 4.6 for the evolution of the different regularization mechanisms clearly show that removing the oldest individuals yields the best results.

Finally, we evaluated the proposed estimation performance mechanism outside an evolutionary setting. For this, we combined the proposed performance estimation mechanism with a RS strategy. In this setting, an architecture is randomly sampled from the search space, and instead of training it, we infer its performance by scoring it at initialization stage. Table 4.4 shows the results for scoring architectures sampled using RS for different sample sizes n, where n represents the number of architectures evaluated. We also evaluate using an SVM as a performance predictor to further showcase how good the proposed zero-cost proxy estimator performance method is. The SVM was trained with information about 100 trained architectures. The input received by the SVM totaled 4.16 GPU days of computation to train all 100 architectures. When the SVM is trained, there is no need to further train any architecture, as the SVM infers the performance of untrained architectures. From the results presented in Table 4.4, it is possible to see that the proposed performance estimation method is extremely efficient in evaluating an architecture (search cost), and when compared with other NAS methods (previously presented in Table 4.2), it requires orders of magnitude less time to search for efficient architectures. When directly compared with NAS-WOT and the surrogate SVM, it is faster and capable of selecting high-performant architectures without losing precision when increasing n, which is of extreme importance, as it is improbable that the optimal architectures are present in small sample sizes.

To further evaluate the gains in search cost when compared to NAS-WOT, we explored how both methods behave in scoring an architecture with images of increasing sizes, which can be seen in Figure 4.7. The proposed performance estimation method can evaluate images with sizes $256 * 256 * 3$ in approximately 5 seconds, whereas NAS-WOT requires 23% more time, approximately 6.5 seconds. As for images with a size of $512 * 512 * 3$, it can evaluate an architecture in under 23 seconds, whereas NAS-WOT requires 34 seconds, approximately 34% more. This shows that the proposed method can also improve current NAS methods that solely search for architectures in CIFAR-10 due to the reduced image size. With our method, NAS methods that were incapable of searching on larger data sets due to time complexity can now use the proposed method to directly search architectures in larger data sets.

The reason why the proposed method is capable of outperforming NAS-WOT in terms of time is directly linked with the time complexity of creating a correlation matrix, which is highly dependent on the number of data points and features. By evaluating individual

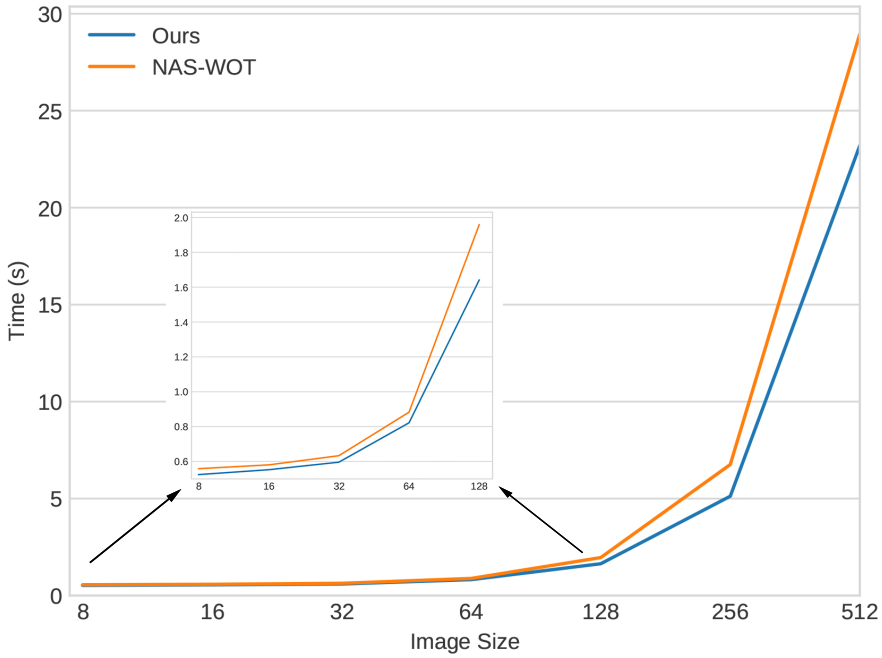


Figure 4.7: Comparison of the time, in seconds, required to score a single architecture using our proposed method (in blue) against NAS-WOT (in orange) for different image sizes (x-axis). The image size represents the image's width and height, with 3 channels (RGB).

correlation matrices, one per class, we reduce each correlation matrix's size, allowing for faster computations.

Considering the mean time required to evaluate 1000 architectures by our method (Table 4.4), the proposed performance estimation method also allows exhaustive exploration of a search space, as it is capable of evaluating over 1 million architectures in just two days of GPU computing when using $32 * 32 * 3$ image sizes. Therefore, this method could be used to evaluate a search space's behavior, providing prior information to the search method, which is a significant benefit when considering large, possibly unbounded, search spaces where information about their shape is limited. More, the proposed zero-cost proxy can easily be incorporated in almost any NAS method either as the sole method that evaluates architectures or as a complementary method to perform mixed training, where the reward to update the controller parameters is a combination of complementary evaluations (e.g., combining it with the inference/latency of the architecture in a mobile setting [226, 227]).

4.4 Conclusions

In this chapter, we presented two methods: EPE-NAS, a performance estimation strategy that scores untrained architectures with a high correlation to their trained performance, and a guided EA that leverages the zero-proxy estimator to obtain information about the search space and efficiently guide the search.

First, we showed that by leveraging information about the gradients of the output of an architecture with regards to its input, a scoring method can accurately infer if the generated architecture is good in less than one second, being capable of evaluating thousands of architectures in a matter of minutes. Then we presented the evolutionary strategy that forces exploitation of the most performant architectures by descendant generation and an exploration of the search space by performing mutations. The proposed EA guides the evolution by generating several architectures in each generation and evaluating them at the initialization stage using a zero-proxy estimator, where only the highest-scoring architecture is trained and kept for the next generation. The generation of multiple architectures from an existing one in the population at each generation allows constant extraction of knowledge about the search space without compromising the search itself.

The results for both methods in all data sets of NAS-Bench-101, NAS-Bench-201, and TransNAS-Bench-101 benchmarks show that they can obtain state-of-the-art results while being extremely efficient regarding search cost. More, in this chapter, we showed that using a simple RS coupled with the proposed estimation strategy, it is possible to sample high-performant architectures in seconds that can outperform many current NAS methods.

In short, the results obtained provide evidence that the proposals described in this chapter contribute to the state-of-the-art in NAS, not only by allowing efficient and fast evaluation of architectures but also by providing a mechanism to guide the search without compromising performance. The proposed EA approach can be extended to multiple strategies, where the search method can be further improved by incorporating new regularisation and mutation mechanisms. Also, the components of the guiding mechanism can easily be transferred to other evolutionary algorithms, allowing existing NAS evolutionary methods to be further improved.

In the next chapter, we extend the use of zero-proxy estimators by looking at evaluating architectures based on their NTK and by proposing a search space that allows combining large vision classifiers by searching for a *middleware* architecture that learns to solve downstream tasks.

Chapter 5

Designing Architectures with Neural Tangent Kernel and Large Feature Extractors

5.1 Introduction

In the previous chapter, we proposed a zero-proxy estimator based on the evaluation of the Jacobian J of an architecture. With this, a new evolutionary search method that efficiently guides the search by quickly evaluating architectures based on the zero-proxy estimator was proposed. However, the proposed zero-proxy estimator still has some shortcomings: i) for optimal results, τ in Eq. 4.4 has to be empirically defined for different data sets; and ii) results on NAS-Bench-201 using RS on noisy data sets (ImageNet16-120) still have a considerable gap to the optimal architectures (Table 4.4). To further improve the evaluation of NAS methods and their applicability, in this chapter, we propose a novel approach that combines two key components to advance the state-of-the-art in NAS: an enhanced search space and a zero-proxy estimator based on NTK.

The first component of our approach focuses on proposing a novel search space. We build upon the popular cell-based search spaces and introduce a novel extension by incorporating large vision classifiers as feature extractors. Traditionally, large models have been primarily used for processing input data independently, where they are fine-tuned to learn new tasks. However, in our approach, we exploit the feature maps generated by these large models as inputs to a *middleware* architecture. The *middleware* architecture, which is the target of the search, is responsible for learning to leverage these extracted features effectively. This design allows us to search for compact architectures while simultaneously capitalizing on the representation power of large models in downstream tasks. By leveraging large models as feature extractors, we introduce a hierarchical approach that mitigates the limitations of conventional cell-based search spaces and enhances the overall performance of the discovered architectures.

The second component of our approach introduces a zero-proxy estimator based on NTK – NTKInner. Evaluating the performance of candidate architectures during the search process is a computationally expensive task, as seen in the last chapter. To address this challenge, we draw inspiration from using NTK in evaluating the behavior of DNN during training and the fact that the alignment between eigenvectors serves as a strong indicator of a network's generability [228, 229, 228]. Leveraging this concept, we propose a novel performance estimator that efficiently evaluates the trainability of generated architectures. By utilizing the information captured by NTK, we show that our estimator

offers a computationally efficient alternative to traditional evaluation methods, enabling faster and effective performance estimation.

These two components, the enhanced search space and the zero-proxy estimator, improve the effectiveness and efficiency of NAS and extends the use NAS by leveraging of existing knowledge in the form of large models. By leveraging those as feature extractors, our search space allows the design of small architectures to exploit the learned representations from the large models to learn downstream tasks. Moreover, the zero-proxy estimator based on NTK provides a computationally efficient way of evaluating the trainability of generated architectures, enabling faster exploration of the search space and easy integration on several existing NAS methods. By combining these two components, our approach offers significant improvements over the state-of-the-art NAS methods, leading to the discovery of highly performant architectures. To evaluate the proposed search space, we conducted experiments using different NAS methods, including different zero-proxy estimators (section 5.3.4). As for the evaluation of NTKInner, experiments were conducted on the proposed search space, the DARTS search space, and NAS-Bench-201 (section 5.3.3). In short, the contributions of this chapter can be summarized as:

- We propose extending the conventional cell-based search spaces by incorporating large vision classifiers as feature extractors. This approach allows searching for small architectures that leverage the representation power of large models to solve downstream tasks. Without re-training those large models.
- We propose NTKInner, a performance estimator that efficiently evaluates the trainability of generated architectures by leveraging NTK and the alignment between eigenvectors.
- We conduct exhaustive experiments to evaluate the performance and effectiveness of the proposed NTKInner, and the behavior of the introduced search space. Results show that the proposed NTKInner achieves state-of-the-art results and that the proposed search space helps generate competitive architectures, especially in more challenging data sets (like CIFAR-100).

The remainder of this chapter is organized as follows. Section 5.2 describes the proposed search space and zero-proxy estimator. Section 5.3, presents the experiments and the results, and finally, section 5.4 draws a conclusion.

5.2 Proposed method

5.2.1 Performance Estimation Mechanism

NTK is an important concept for understanding a neural network training via gradient descent [230]. At its core, NTK allows explaining how updating the model parameters on one data sample affects the predictions for other samples. In [230, 231, 232], it was found

that the NTK of a neural network converges during training and is independent of the initialization of the random parameters. These are crucial for evaluating architectures, as it allows defining a scoring to estimate how *trainable* an architecture is based on its NTK. In this section, we present NTKInner, an improved zero-proxy estimator based on the analysis of the NTK inspired by the findings of [4].

To capture the training dynamics of neural networks, the finite width NTK has been defined as [230, 231, 233, 4]:

$$\hat{\Theta}(\boldsymbol{x}, \boldsymbol{x}') = \mathbf{J}(\boldsymbol{x})\mathbf{J}(\boldsymbol{x}')^T, \quad (5.1)$$

where $\mathbf{J}(\boldsymbol{x})$ is the Jacobian evaluated at a point $\boldsymbol{x}$. [231] provided additional evidence that wide neural networks evolve as linear models through gradient descent. This study demonstrated that the training dynamics of these networks can be characterized by ordinary differential equations:

$$\mu_t(\boldsymbol{x}_{\text{train}}) = (\mathbf{I} - e^{-\eta\hat{\Theta}(\boldsymbol{x}_{\text{train}}, \boldsymbol{x}_{\text{train}})t})\boldsymbol{y}_{\text{train}} \quad (5.2)$$

where $\mu_t(\boldsymbol{x})$ is the output of the network on the training set, η is the learning rate, $\boldsymbol{x}_{\text{train}}$ and $\boldsymbol{y}_{\text{train}}$ are obtained from the training data, and t defines the training time step. Then based on the findings of [234, 4] Eq. 5.2 can be estimated based on the spectrum of $\hat{\Theta}$:

$$\mu_t(\boldsymbol{x}_{\text{train}})_i \approx (\mathbf{I} - e^{-\eta\lambda_i t})\boldsymbol{y}_{\text{train},i}. \quad (5.3)$$

where λ_i are the eigenvalues of $\hat{\Theta}(\boldsymbol{x}_{\text{train}}, \boldsymbol{x}_{\text{train}})$.

In [234], Eq. 5.3 is used to measure the trainability of a network. Also, in [4] the authors stated that Eq. 5.3 indicates that each eigenmode requires a different amount of time, as expressed by $(\mathbf{I} - e^{-\eta\lambda_i t})\boldsymbol{y}_{\text{train},i}$. Thus, the diversity in learning speeds among eigenmodes affects the network's optimization difficulty. A greater diversity implies a more challenging optimization task. Based on these findings and inspired by the fact that the alignment between eigenvectors has been shown as a good metric to evaluate the generalization of a neural network [228, 229, 228], we evaluate the trainability of an untrained neural network by looking at the NTK:

$$\kappa = \mathop{\mathbb{E}}_{\substack{\boldsymbol{x}_{\text{train}} \sim D_{\text{train}} \\ \theta \sim \mathcal{N}(0, \frac{2}{N_l})}} V_{\max}(\hat{\Theta}(\boldsymbol{x}_{\text{train}}, \boldsymbol{x}_{\text{train}})) \cdot V_{\min}(\hat{\Theta}(\boldsymbol{x}_{\text{train}}, \boldsymbol{x}_{\text{train}})) \quad (5.4)$$

where parameters θ are drawn from the He initialization: $\mathcal{N}(0, \frac{2}{N_l})$, N_l is the width at layer l [235], and V represents the eigenvectors. Since the parameters of the architecture are

initialized randomly, κ is evaluated at initialization phase when the architecture is not yet trained. In this way, κ can be used as an evaluation measure for comparing different untrained architectures, where the lower κ is, the better the trainability of the architecture.

5.2.2 Leveraging Large Feature Extractors Search Space

NAS methods tend to focus on designing cells that are then replicated to generate new architectures for a given problem [2, 42]. The exception is network morphism, where methods use CNNs as starting point and change its structure based on morphism [90]. However, these methods overlook the potential benefits of utilizing existing large DNNs as effective feature extractors in solving downstream tasks. In this section, we propose a novel NAS search space that integrates large vision models as feature extractors and focuses the search on a *middleware* architecture. By doing so, we harness the power of large models trained on extensive data sets without the need for re-training or fine-tuning them for smaller data sets.

The search space builds upon previous works [38, 72, 2], as it considers the design of a computation cell as the fundamental building block of the final architecture. The cell is represented by a DAG with V nodes and N edges, where each vertex $\boldsymbol{x}^{(i)}$ denotes a latent representation for $i \in 1, \ldots, V$. The directed edges (i, j) with $i < j$ correspond to operations $o^{(i,j)}$ that connects $\boldsymbol{x}^{(i)}$ to $\boldsymbol{x}^{(j)}$, using the operation $o \in \mathcal{O}$, where $\mathcal{O}$ defines the pool of operations. The intermediate node values are computed by summing the transformations applied to their preceding nodes, resulting in $\boldsymbol{x}^{(j)} = \sum_{i<j} o^{(i,j)}(\boldsymbol{x}^{(i)})$.

In the proposed search space, there are two types of cells: normal cells, which produce feature maps of the same dimension, and reduction cells, which generate scaled-down feature maps with reduced height and width by a factor of two. All cells consist of $N = 7$ nodes, and the output of a cell is defined as the concatenation of all preceding nodes except for the input nodes. We follow the operation pool $\mathcal{O}$ from DARTS [2], which includes eight possible operations: 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max pooling, 3×3 average pooling, identity, and zero.
The outer-skeleton of the architecture (Figure 5.1) consists of interleaved cells, where reduction cells are placed at $1/3$ and $2/3$ of the skeleton. Additionally, the architecture functions as a middleware, receiving the concatenation of feature maps from different pre-trained large models as the initial input. This setup forces the architecture to learn to solve the downstream task based on the representations created by the large models. At $1/3$ of the architecture, the feature maps are concatenated with new inputs from the large models, enabling the generated architecture to receive deeper feature maps from the feature extractors. We justify this architecture because large CNNs when trained on extensive data sets (e.g., ImageNet21k) produce high-quality feature representations of the input, and searching for a specialized architecture for the downstream task is cheaper than re-training large models. More, trained CNNs tend to follow a rule of generating generic features in the early stages of the convolutional layers, becoming more task-specific

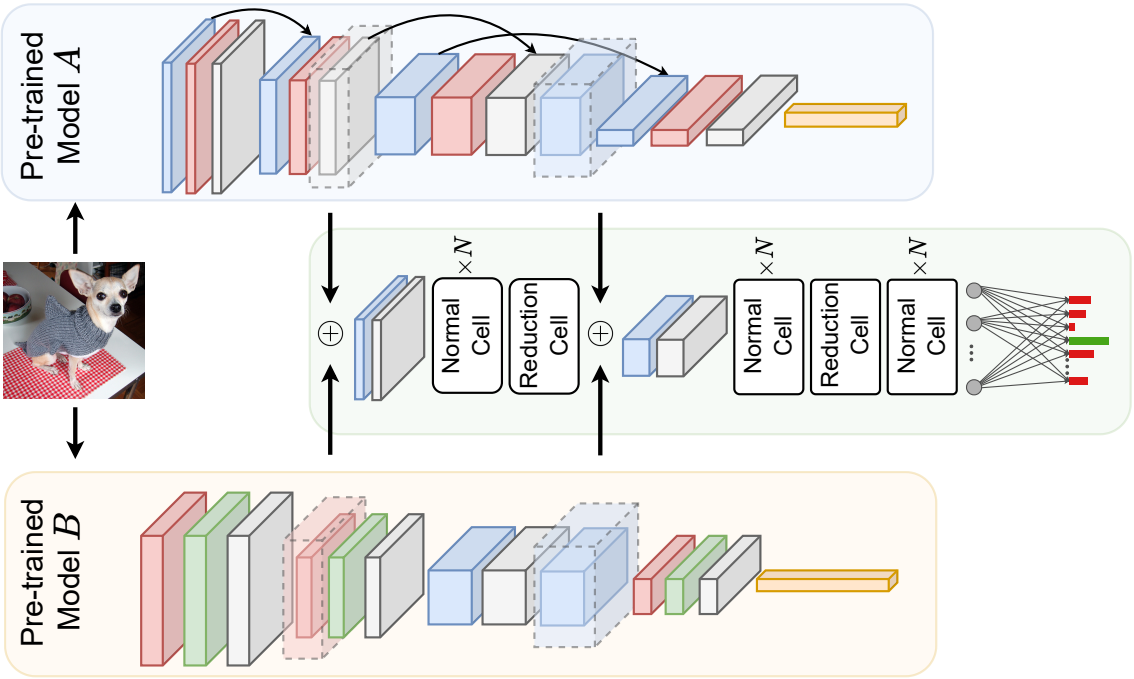


Figure 5.1: Illustration of the proposed search space, where pre-trained models serve as feature extractors and a *middleware* architecture is searched to learn downstream tasks.

the deeper in the network [236]. Since large CNNs possess a higher capacity to represent inputs in a multidimensional space, the early and mid-stage convolutional layers yield more informative feature maps than smaller architectures. Therefore, combining these feature maps in a compact *middleware* architecture allows us to leverage the representational power of large vision models while facilitating an efficient search process.

5.3 Experiments

To evaluate the performance of the proposed methods, the experiments conducted consist of two stages: architecture search in different search spaces with multiple NAS methods to evaluate the generability of NTKInner (section 5.3.3), and architecture search in the proposed search space, including the use of the proposed NTKInner (section 5.3.4).

5.3.1 Search Spaces

To allow comparison with the state-of-the-art, we follow NAS literature [172, 4, 153] and evaluate the effectiveness of NTKInner in the problem of cell-based NAS in three different search spaces: the proposed one that leverages pre-trained feature extractors; DARTS [2], in which cells have 7 nodes and the operation pool is composed of 8 different operations; and NAS-Bench-201 [42], which is a cell-based with 6 possible layers per cell and an operation pool of 5 operations. For more details on the search spaces, we reference the reader to Chapter 2.

5.3.2 Implementation Details

In terms of studying the performance gains of different NAS methods when combined with NTKInner and in the proposed search space, we focused on studying: RS-approach [14, 1], an EA [51, 72], a RL-based method [94, 4], DARTS [2] and TENAS [156]. Following, we dive deeper into each one.

Random Search

As a baseline, we implement a RS strategy, where 256 architectures are randomly sampled, and the best-scoring one is selected as the final architecture. By using RS as a baseline, it allows evaluating the complexity of the search space, but also how good a zero-proxy estimator is, as this shows the true gains of the scoring method, as no complex NAS sampling method is involved.

Evolutionary Algorithm

For the evolutionary algorithm, we followed the previously implementation on Chapter 4 and the proposed methods in [72, 4] to promote a fair comparison. The evolution starts by randomly sampling 256 individuals. In each generation, 64 individuals are sampled from the population, and the top-scoring one is selected to generate a new individual. The new individual is generated based on operation mutation and added to the population. At the same time, a regularization mechanism removes the oldest individual to promote exploration. At the end of 1000 generations, the best scoring architecture is selected and trained from scratch.

Reinforce

The implemented RL-based method solves the NAS optimization problem as sequential decision-making where a policy agent chooses a sequence of actions (operations) and learns by using the performance metric as a reward. For this we followed the implementations of [94, 4]. In this, the method maintains an internal state, denoted as $\boldsymbol{\theta^A}$, which serves as a representation of the architecture search space. During training, the RL agent samples one architecture per iteration, denoted as a_t, from the distribution $\boldsymbol{\mathcal{A}}$, and it learns by updating the representation of the architecture search space, $\boldsymbol{\theta^A}$, through policy gradients [4]:

$$\boldsymbol{\theta}^{\boldsymbol{A}}_{t+1} = \boldsymbol{\theta}^{\boldsymbol{A}}_{t} - \eta \cdot \nabla_{\boldsymbol{\theta^A}} f(\boldsymbol{\theta}^{\boldsymbol{A}}_{t}) \quad t = 1, \cdots, T \quad (5.5)$$

where $f(\boldsymbol{\theta}^{\boldsymbol{A}}_{t})$ represents the objective function, defined as: $f(\boldsymbol{\theta}^{\boldsymbol{A}}_{t}) = -\log(\sigma(\boldsymbol{\theta}^{\boldsymbol{A}}_{t})) \cdot (r_t - b_t)$, r_t denotes the reward obtained at iteration t, while b_t is an exponential moving average of the rewards, denoted by: $b_t = \gamma b_{t-1} + (1-\gamma) r_t$, starting at $b_0 = 0$ and with a smoothing operator $\gamma = 0.9$. During training, the reward is obtained using a zero-proxy estimator that scores an untrained architecture to try to infer its final performance if trained.

For setting the parameters of the RL agent, we follow [4] to ensure fair comparisons with the state-of-the-art. The RL agent is trained for 500 steps with a learning rate of $\eta = 0.04$ on NAS-Bench-201 and $\eta = 0.07$ on DARTS and the proposed search space.

DARTS

We analyse how DARTS behaves in the proposed search space. For this, we follow the proposed DARTS method [2], where the goal is to solve a bi-level optimization problem on the weights and the architecture:

$$\boldsymbol{\alpha}^{\star} = \min_{\boldsymbol{\alpha}} \mathcal{L}^{(\text{val})}(\boldsymbol{\alpha}, \boldsymbol{w}^{\star}(\boldsymbol{\alpha})) \quad \text{s.t.} \quad \boldsymbol{w}^{\star}(\boldsymbol{\alpha}) = \arg\min_{\boldsymbol{w}} \mathcal{L}^{(\text{train})}(\boldsymbol{\alpha}, \boldsymbol{w}). \quad (5.6)$$

where α represents an architecture, w the associated weights, and $\mathcal{L}$ the loss function. DARTS solves this problem by relaxing the discrete set of candidate operations and with gradient descent. For our experiments, we follow the settings initially proposed in [2].

TENAS

The last NAS method used to perform the search was TENAS [156]. This method works by pruning a super-network until a single-path network is found. In [156], the authors perform this pruning by evaluating the trainability and expressitivy of an architecture to measure the importance of an operator. By doing this evaluation iteratively for the super-network without a single operator, the method can quickly evaluate which operator is the the least important. In our implementation, when using NTKInner to calculate the trainability of an architecture, we update algorithm 1 in [156] to calculate $\mathcal{K}$ using the proposed NTKInner.

5.3.3 Results and Discussion On Performance Estimation

We first focused on evaluating the proposed performance estimation mechanism – NTKInner. For this, we looked into coupling it with different methods and evaluated it on NAS-Bench-201 and DARTS search space. Results are shown in Table 5.1. The first block of the table depicts human-engineered architectures, the second one depicts state-of-the-art NAS methods, and the third shows the results for three different NAS methods with and without the proposed NTKInner. From the results is possible to see that NTKInner is capable of improving the final performance of EA (Evolution), RL (REINFORCE), and pruning-based NAS methods (TE-NAS), consistently achieving state-of-the-art results while still being extremely efficient in terms of search cost. For REINFORCE and Evolution NAS methods we directly compare NTKInner with the zero-cost proxy TEGNAS [4] and the results show an improved performance when using the proposed estimation mechanism. When compared to NASWOT [153], NTKInner augmented NAS methods achieve much higher performances, especially on ImageNet16-120, a noisier data set. The overall results shown in Table 5.1 bespeak the performance of NTKInner and its capability of

Table 5.1: Search Performance from NAS-Bench-201. "Optimal" indicates the best test accuracy achievable in the space.

Architecture	CIFAR-10	CIFAR-100	ImageNet-16-120	Search Cost (GPU sec.)	Search Method
ResNet [59]	93.97	70.86	43.63	-	-
RSPS [1]	87.66 ± 1.69	58.33 ± 4.34	31.14 ± 3.88	8007.1	RS
ENAS [70]	54.30 ± 0.00	15.61 ± 0.00	16.32 ± 0.00	13314.5	RL
DARTS, 1st order [2]	54.30 ± 0.00	15.61 ± 0.00	16.32 ± 0.00	10889.9	GB
DARTS, 2nd order [2]	54.30 ± 0.00	15.61 ± 0.00	16.32 ± 0.00	29901.7	GB
GDAS [156]	93.61 ± 0.09	70.70 ± 0.30	41.84 ± 0.90	28925.9	GB
DrNAS [237]	94.36 ± 0.00	73.51 ± 0.00	46.34 ± 0.00	-	GB
RLNAS [238]	93.45	70.71	43.70	-	GB
GEA [51]	93.98 ± 0.18	72.12 ± 0.35	45.94 ± 0.71	18567	GB
β-DARTS [239]	94.36 ± 0.00	73.51 ± 0.00	46.34 ± 0.00	11520	GB
Single-DARTS [240]	94.36 ± 0.00	73.51 ± 0.00	46.34 ± 0.00	-	GB
NASWOT ($N = 1000$) [153]	91.20 ± 2.04	68.95 ± 0.72	38.08 ± 1.58	252.6	training-free
TE-NAS	92.99	70.04	42.27	452	training-free
TE-NAS + ours	93.99	71.40	45.20	395	training-free
REINFORCE + TEGNAS	90.48	69.06	44.90	1407	training-free
REINFORCE + ours	90.86	70.99	45.75	1427	training-free
Evolution + TEGNAS	90.04	69.46	44.90	3457	training-free
Evolution + ours	90.97	70.17	45.19	4182	training-free
Optimal	94.37	73.51	47.31	-	-

being implemented with different search strategies while improving their performances.

Then, we evaluated the proposed estimation method on DARTS, a larger cell-based search space. For this, we focused on improving upon RS, EA (Evolution), RL (REINFORCE), and pruning-based NAS methods (TE-NAS) by coupling them with NTKInner. The results shown on Table 5.2 show that when directly comparing the results obtained baseline NAS with the results when augmented with NTKInner, its clear that NTK greatly improves the final performance while still designing architectures that are competitive in terms of the number of parameters. RS is a good NAS baseline, as it was found to be competitive in most search spaces [1, 45] and in our experiments, by random sampling 256 architectures, we obtained a test error of 3.29%. When using NTKInner, the top-scoring architecture from the 256 is selected as the final architecture, obtaining a test error of 2.63%, which is on par with state-of-the-art NAS methods that use complex search strategies and represents a 0.66% improvement upon the baseline RS with only 0.06 GPU days of computation. When comparing EA and RL-based NAS methods with TEGNAS and the proposed NTKInner we further see a consistent improvement in terms of accuracy, 0.34% in RL and 0.27% in EA, without added computational costs. The results obtained show the efficiency of the proposed NTKInner and validate that the proposed method is a fast and accurate zero-proxy estimator that can be coupled with different search strategies.

5.3.4 Results and Discussion On Combining Large Vision Classifiers

To evaluate the proposed search space, we first analyzed how different NAS methods perform when directly searching in it using CIFAR-10. Table 5.3 shows the results for using a ResNet20, which is a human-designed architecture, and 4 NAS methods: RS, EA,

Table 5.2: Search Performance using DARTS space on CIFAR-10. The last 4 blocks shows direct comparisons of NTKInner against other zero-cost proxy methods with different search methods.

Architecture	Test Error (%)	Params (M)	Search Cost (GPU days)	Search Method
AmoebaNet-A [72]	3.34	3.2	3150	EA
PNAS [217]	3.41	3.2	225	SMBO
ENAS [70]	2.89	4.6	0.5	RL
NASNet-A [38]	2.65	3.3	2000	RL
DARTS, 1st order [2]	3.00	3.3	1.5	GB
SNAS [98]	2.85	2.8	1.5	GB
GDAS [156]	2.82	2.5	0.2	GB
BayesNAS [241]	2.81	3.4	0.2	GB
NASP [242]	2.83 ± 0.09	3.3	0.1	GB
P-DARTS [113]	2.50	3.4	0.3	GB
PC-DARTS [114]	2.57	3.6	0.1	GB
R-DARTS (L2) [17]	2.95 ± 0.21	-	1.6	GB
SGAS [243]	2.66 ± 0.24	3.7	0.3	GB
SDARTS-ADV [244]	2.61	3.3	1.3	GB
DrNAS [237]	2.46 ± 0.03	4.1	0.6	GB
β-DARTS [239]	2.53 ± 0.08	3.75 ± 0.15	0.4	GB
Single-DARTS [240]	2.46	3.3	-	GB
TE-NAS [156]	3.83	2.8	0.04	training-free
TE-NAS + TEGNAS	3.71	3.8	0.04	training-free
TE-NAS + ours	3.50	3.4	0.04	training-free
Evolution + TEGNAS	2.98	3.0	0.4	training-free
Evolution + ours	2.71	3.3	0.4	training-free
RS [2]	3.29	3.2	–	RS
RS + TEGNAS	2.77	3.6	0.06	training-free
RS + Ours	2.63	3.5	0.06	training-free
REINFORCE + TEGNAS	3.09	4.2	0.1	training-free
REINFORCE + ours	2.75	3.2	0.1	training-free

prune-based NAS and DARTS. First, ResNet20 was used to evaluate if the accuracy when trained on CIFAR-10 would improve if the same pre-trained large feature extractors were used instead of receiving an image as input. The results show an 0.71% improvement in accuracy, thus suggesting that the use of feature maps from large pre-trained CNNs is beneficial. Then, we evaluated the use of RS with three different zero-cost proxies: TEGNAS [4], EPE-NAS [48] and the proposed NTKInner, showing that NTKInner is a superior method, obtaining 2.12% higher accuracy than when using TEGNAS to augment a RS. More, the results of using RS with NTKInner are competitive to the results of DARTS in the proposed search space, but with orders of magnitude less computation required. The results of using an EA strategy further show the superiority of the proposed NTKInner and further cement that the proposed search space is capable of yielding excellent architectures

Table 5.3: Comparison of different NAS methods and zero-proxy estimator's in the proposed search space that leverages large feature extractors.

Method	Test Error (%)	Search Cost (GPU Days)	Search Method
ResNet20	8.75	-	manual
ResNet20 †	8.04	-	manual
RS + TEGNAS †	6.82	0.006	training-free
RS + EPE-NAS †	5.14	0.005	training-free
RS + NTKInner †	4.70	0.006	training-free
Evolution + TEGNAS †	5.84	0.012	training-free
Evolution + EPE-NAS †	6.67	0.009	training-free
Evolution + NTKInner †	5.75	0.012	training-free
TENAS †	7.26	0.130	training-free
DARTS †	4.43	4	GB

† Using the proposed search space where large vision models serve as feature extractors.

without forcing all NAS methods to good results, which is noticeable with TENAS.

Finally, we evaluate the proposed search space in the task of IC with CIFAR-100. For this, we evaluated DARTS, RS with TEGNAS and EA with the proposed NTKInner. Results are depicted in Table 5.4, showing that DARTS fails to generalize, which further suggests that the proposed search space does not hold only good architectures, and the results obtained when using RS and EA are very competitive. A comparison with several other methods that used CIFAR-100 but the DARTS search space is shown in 5.5, where the evaluated EA with the zero-cost proxy obtained state-of-the-art results with an improvement of 2.59% when compared to the previous best result (NAT-M4), while still only requiring a fraction of the search cost.

Table 5.4: Performance in test error obtained by the three evaluated methods in the proposed search space using the CIFAR-100 data set.

Method	Top-1 Test Error (%)	Top-5 Test Error (%)	Search Cost (GPU Days)	Search Method
DARTS †	41.08	16.8	4	EA
RS + TEGNAS †	15.34	0.95	0.01	RS
Evolution + NTKInner †	9.11	0.2	0.006	EA

† Using the proposed search space where large vision models serve as feature extractors.

Table 5.5: Comparison of different NAS methods and zero-proxy estimator's in CIFAR-100.

Architecture	Test Error(%)		Params	Search Cost	Search
	top-1	top-5	(M)	(GPU days)	Method
MetaQNN [68]	27.14	-	11.2	80	RL
NasNet-A [38]	16.82	-	3.3	1800	RL
ENAS [70]	19.43	-	4.6	0.45	RL
AmoebaNet-A (REA) [72]	18.93	-	3.1	3150	EA
DARTS, 2nd order [2]	17.54	-	3.4	4	GB
DARTS †	41.08	16.8	3.3	4	GB
SNAS [98]	17.55	-	2.8	1.5	GB
GDAS [156]	18.13	-	2.5	0.2	GB
P-DARTS [113]	15.92	-	3.6	0.3	GB
R-DARTS (L2) [17]	18.01	-	3.4	-	GB
PC-DARTS [114]	16.90	-	3.6	0.1	GB
DOTS [245]	16.28	-	3.5	0.3	GB
DARTS- [246]	17.51	-	3.4	0.4	GB
DU-DARTS [247]	16.74	-	3.1	0.4	GB
β-DARTS [239]	16.24	-	3.8	0.4	GB
NAT-M4 [248]	11.70	-	9.0	6.3	EA
DARTS-PRIME [249]	17.44	-	3.16	0.5	GB
RS + NTKInner †	15.34	0.95	2.7	0.01	training-free
Evolution + NTKInner †	9.11	0.2	3.7	0.01	training-free

† Using the proposed search space where large vision models serve as feature extractors.

5.4 Conclusions

In this chapter, we presented two approaches to improve NAS methods. The first is a search space that leverages large vision models as feature extractors. This allows harnessing the power of pre-trained models and focuses the search on a *middleware* architecture that learns how to solve a downstream task. The second is a new zero-cost proxy estimator inspired by the use of NTK and the alignment between eigenvectors to evaluate architectures at initialization stage.

By incorporating large vision models as feature extractors, the proposed search space benefits from their rich representation capabilities, enabling the design of small architectures that learn downstream tasks based on these rich representations. Furthermore, the searched *middleware* architecture focuses on learning a downstream task using the representations created by the large models, eliminating the need for re-training or fine-tuning them on smaller data sets. As for the proposed zero-cost proxy estimator, by evaluating architectures based on their NTK and leveraging the fact that the alignment between eigenvectors has been shown to be a good indicator of an architecture generability, the proposed method provides a quick and efficient way to evaluate generated architectures without requiring any training.

The results of our experiments using different NAS methods validate the effectiveness of the proposed methods. Extensive evaluations in different search spaces, with different NAS searching methods and different performance estimation mechanisms, shows the improved performance of the generated architectures, where generated architectures obtain state-of-the-art results, improving upon baseline NAS methods.

In the next chapter, we focus on improving NAS in terms of memory and time efficiency by proposing a multi-agent framework that allows distributed search.

Chapter 6

Neural Architecture Search as a Multi-Agent Problem

6.1 Introduction

Researchers have used a wealth of techniques ranging from RL, where a controller network is trained to sample promising architectures [15, 38, 70, 250], to EA that evolve a population of networks for optimal architecture design [72, 73, 43], to optimization on random graphs [251]. Alas, most of these approaches are inefficient and can be extremely computationally and/or memory intensive as some require all tested architectures to be trained from scratch or that all possible operations on a super-network are loaded into memory. Gradient-based frameworks enabled efficient solutions by introducing a continuous relaxation of the search space. For example, DARTS [2] uses this relaxation to optimize architecture parameters using gradient descent in a bi-level optimization problem, while SNAS [98] updates architecture parameters and network weights under one generic loss. Still, due to memory constraints, DARTS has to perform the search on 8 cells, which are then stacked 20 times to form the final architecture. This approach has received criticisms in [17, 40, 45], which showed that the training protocols overshadow the impact of the NAS search and that RS usually outperforms NAS methods. More, these techniques cannot easily scale to large data sets, e.g., ImageNet, relying on human-defined heuristics for architecture transfer. In fact, in this chapter, we show that searching directly over 20 cells leads to a reduction in test error (8% relative to [2]).

Due to the large architecture parameter space, lack of efficiency is a key bottleneck preventing NAS from its practical use. Even in the current settings where flexibility is limited by expertly-designed search spaces, NAS problems are computationally very intensive with early methods requiring hundreds or thousands of GPU-days to discover state-of-the-art architectures [15, 84, 217, 73]. In the previous chapter, we looked into proposing a zero-proxy estimator for quick evaluation of cells at initialization stage. In this chapter, we focus on mitigating efficiency problems of gradient-based methods by framing NAS as a multi-agent problem where agents control a subset of the network and coordinate to reach optimal architectures, allowing direct design of 20 cells instead of 8.

To enable large-scale joint optimization of deep architectures, in this chapter, we propose MANAS, a multi-agent learning algorithm for NAS. MANAS combines the memory and computational efficiency of multi-agent systems, achieved through action coordination with the theoretical rigor of online machine learning, allowing a balancing between exploration versus exploitation optimally. MANAS leverages multi-agents, where each

agent is associated with one layer selection and a global network is optimized based on each agent decision (detailed in Section 6.2.3). The multi-agent framework is inspired by a multi-arm bandit setting [252]. Multi-arm bandit algorithms are a class of reinforcement learning algorithms used to balance exploration and exploitation in a decision-making process. This class of algorithms has been widely explored to solve a panoply of problems [253], such as designing recommendation systems [254]. By employing a multi-arm bandit, MANAS allows for a distributed space search through multiple agents with a global optimization goal.

Due to its distributed nature, MANAS enables large-scale optimization of deeper networks while learning different operations per cell. We show that our method achieves state-of-the-art accuracy results among methods using the same evaluation protocol but with significant reductions in memory (1/8th of [2]) and search time (70% of [2]). The MA framework is inherently scalable and allows us to tackle an optimization problem that would be extremely challenging to solve efficiently otherwise: the search space of a single cell is DARTS search space is 8^{14} and there is no fast way of learning the joint distribution, as needed by a single controller. More cells to learn exacerbates the problem, which is why a multi-agent framework is required, as for each agent, the size of the search space is always constant.

To validate the proposed method, we performed experiments on CIFAR-10 and ImageNet, conducted additional experiments on three alternative data sets, evaluated MANAS with complexity constraints, and two network configurations. Aware that RS with weight-sharing and RS are effective baselines, we evaluate these throughout. When compared with other NAS optimization methods that use the same search space, MANAS achieves better performance and requires less memory.

In short, the contributions of this chapter can be summarised as:

- Framing NAS as a multi-agent learning problem (MANAS) where each agent supervises a subset of the network; agents coordinate through a credit assignment technique which infers the quality of each operation in the network, without suffering from the combinatorial explosion of potential solutions.
- Proposal of two lightweight implementations of our framework that are theoretically grounded. The algorithms are computationally and memory efficient, and achieve state-of-the-art results on CIFAR-10 and ImageNet when compared with competing methods. Furthermore, MANAS allows searching *directly* on large data sets (e.g. ImageNet).
- Presenting 3 news data sets for NAS evaluation to minimize algorithmic overfitting; offering a fair comparison with the often ignored RS with weight-sharing [1] and RS [40, 170] baselines; as well as presenting a complexity constraint analysis of MANAS.

The remainder of this chapter is organized as follows. Section 6.2 provides an introduction to the problem, describes the MA setting, and describes the proposed method. Section 6.3, presents the experiments, the results, and presents a discussion. Finally, a conclusion is drawn in Section 6.4.

6.2 Proposed Method

6.2.1 Preliminary: Neural Architecture Search Cell Search

For MANAS, we consider NAS as formalized in DARTS [2]. At a higher level, the architecture is composed of a *computation cell* that is a building block to be learned and stacked to form the final architecture. The cell is represented by a DAG with V nodes and N edges in which edges connect all nodes i, j from i to j where $i < j$. Each vertex $\boldsymbol{x}^{(i)}$ is a latent representation for $i \in \{1, \ldots, V\}$. Each directed edge (i, j) (with $i < j$) is associated with an operation $o^{(i,j)}$ that transforms $\boldsymbol{x}^{(i)}$. Intermediate node values are computed based on all of the predecessors as $\boldsymbol{x}^{(j)} = \sum_{i<j} o^{(i,j)}(\boldsymbol{x}^{(i)})$. For each edge, a NAS method needs to select one operation (layer) $o^{(i,j)}$ from a finite set of K operations, $\mathcal{O} = \{o_k(\cdot)\}_{k=1}^{K}$, where operations represents some function to be applied to $\boldsymbol{x}^{(i)}$ to compute $\boldsymbol{x}^{(j)}$, e.g., convolutions or pooling layers. Each $o_k^{(i,j)}(\cdot)$ is associated with a set of operational weights $w_k^{(i,j)}$ that need to be learned (e.g. the weights of a convolution filter). Additionally, a parameter $\alpha_k^{(i,j)} \in \mathbb{R}$ characterises the importance of operation k within the pool $\mathcal{O}$ for edge (i, j). The sets of all the operational weights $\{w_k^{(i,j)}\}$ and architecture parameters (edge weights) $\{\alpha_k^{(i,j)}\}$ are denoted by $\boldsymbol{w}$ and $\boldsymbol{\alpha}$, respectively. DARTS defines the operation $\bar{o}^{(i,j)}(\boldsymbol{x})$ as:

$$\bar{o}^{(i,j)}(\boldsymbol{x}) = \sum_{k=1}^{K} \frac{e^{\alpha_k^{(i,j)}}}{\sum_{k'=1}^{K} e^{\alpha_{k'}^{(i,j)}}} \cdot o_k^{(i,j)}(\boldsymbol{x}) \quad (6.1)$$

in which $\boldsymbol{\alpha}$ encodes the architecture, and the optimal choice of architecture is defined by:

$$\boldsymbol{\alpha}^\star = \min_{\boldsymbol{\alpha}} \mathcal{L}^{(\text{val})}(\boldsymbol{\alpha}, \boldsymbol{w}^\star(\boldsymbol{\alpha})) \quad \text{s.t.} \quad \boldsymbol{w}^\star(\boldsymbol{\alpha}) = \arg\min_{\boldsymbol{w}} \mathcal{L}^{(\text{train})}(\boldsymbol{\alpha}, \boldsymbol{w}). \quad (6.2)$$

The final objective is to obtain a *sparse* architecture $\mathcal{Z}^\star = \{\mathcal{Z}^{(i,j)}\}, \forall i, j$ where $\mathcal{Z}^{(i,j)} = [z_1^{(i,j)}, \ldots, z_K^{(i,j)}]$ with $z_k^{(i,j)} = 1$ for k corresponding to the best operation and 0 otherwise. That is, for each pair (i, j) a *single operation* is selected.

6.2.2 Online Multi-agent Learning

NAS suffers from a combinatorial explosion in its search space. A recently proposed approach to tackle this problem is to approximate the discrete optimization variables (i.e., edges in our case) with continuous counterparts and then use gradient-based optimization methods. DARTS [2] introduced this method for NAS, though it suffers from two important drawbacks. First, the algorithm is computationally and memory intensive ($\mathcal{O}(NK)$) with K being the total number of operations between a pair of nodes and N the number of nodes) as it requires loading all operation parameters into GPU memory.

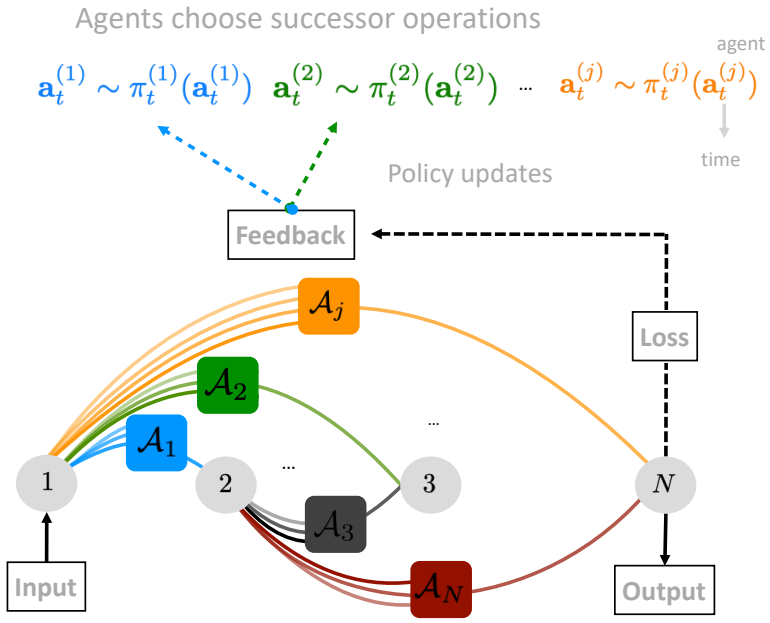


Figure 6.1: MANAS with single cell. Between each pair of nodes, an agent $\mathcal{A}_i$ selects action $a^{(i)}$ according to $\pi^{(i)}$. Feedback from the validation loss is used to update the policy.

As a result, DARTS only optimizes over a small subset of 8 repeating cells, which are then stacked together to form an architecture of 20. Naturally, such an approximation is bound to be sub-optimal. Second, evaluating an architecture on a validation set requires the optimal set of network parameters. Learning these, unfortunately, is highly demanding since for an architecture $\mathcal{Z}_t$, one would like to compute $\mathcal{L}_t^{(\text{val})}(\mathcal{Z}_t, \boldsymbol{w}_t^\star)$ where $\boldsymbol{w}_t^\star = \arg\min_{\boldsymbol{w}} \mathcal{L}_t^{(\text{train})}(\boldsymbol{w}, \mathcal{Z}_t)$. DARTS, uses a weight-sharing mechanism that updates $\boldsymbol{w}_t$ once per architecture, with the hope of tracking $\boldsymbol{w}_t^\star$ over learning rounds. Although this technique leads to a significant speed-up in computation in comparison with previous work, it is not clear how this approximation affects the validation loss function.

In the following paragraphs, a novel methodology based on a combination of multi-agent and online learning to tackle the above two problems is detailed (Figure 6.1). Notably, MA learning scales the proposed algorithm by reducing memory consumption by an order of magnitude from $\mathcal{O}(NK)$ to $\mathcal{O}(N)$; and online learning enables understanding of the effect of tracking $\boldsymbol{w}_t^\star$ over learning rounds.

NAS as a Multi-Agent Problem

To address the computational complexity we use the weight-sharing technique used in DARTS. However, here the effect of approximating $\mathcal{L}_t^{(\text{val})}(\mathcal{Z}_t, \boldsymbol{w}_t^\star)$ by $\mathcal{L}_t^{(\text{val})}(\mathcal{Z}_t, \boldsymbol{w}_t)$ is handled in a more theoretically grounded way. Indeed, such an approximation can lead to arbitrary bad solutions due to the uncontrollable weight component. To analyze the learning problem with no stochastic assumptions on the process generating $\nu = \{\mathcal{L}_1, \ldots, \mathcal{L}_T\}$ MANAS uses an adversarial online learning framework.

Algorithm 2 GENERAL FRAMEWORK: [steps with asterisks (*) are specified in section 6.2.3]

1: **Initialize:** π_1^i is uniform random over all $j \in \{1, \dots N\}$. And random $\boldsymbol{w}_1$ weights.
2: **For** $t = 1, \dots, T$
3: * Agent $\mathcal{A}_i$ samples $\boldsymbol{a}_t^i \sim \pi_t^i(\boldsymbol{a}_t^i)$ for all $i \in \{1, \dots, N\}$, forming architecture $\mathcal{Z}_t$.
4: Compute the training loss $\mathcal{L}_t^{(\text{train})}(\boldsymbol{a}_t) = \mathcal{L}_t^{(\text{train})}(\mathcal{Z}_t, \boldsymbol{w}_t)$
5: Update $\boldsymbol{w}_{t+1}$ for all operation $\boldsymbol{a}_t^i$ in $\mathcal{Z}_t$ from $\boldsymbol{w}_t$ using back-propagation.
6: Compute the validation loss $\mathcal{L}_t^{(\text{val})}(\boldsymbol{a}_t) = \mathcal{L}_t^{(\text{val})}(\mathcal{Z}_t, \boldsymbol{w}_{t+1})$
7: * Update π_{t+1}^i for all $i \in \{1, \dots N\}$ using $\mathcal{Z}_1, \dots, \mathcal{Z}_t$ and $\mathcal{L}_1^{(\text{val})}, \dots, \mathcal{L}_t^{(\text{val})}$.
8: **Recommend** $\mathcal{Z}_{T+1}$, after round T, where $\boldsymbol{a}_{T+1}^i \sim \pi_{T+1}^i(\boldsymbol{a}_{T+1}^i)$ for all $i \in \{1, \dots, N\}$.

Neural Architecture Search as Multi-Agent Combinatorial Online Learning

In Section 6.2.1, NAS cell-based search is defined as a problem where one out of K operations needs to be recommended ("sampled") for each pair of nodes (i, j) in a DAG. Here, each pair of nodes is associated with an agent in charge of exploring and quantifying the quality of these K operations to recommend one ultimately. The only feedback for each agent is the loss associated with a global architecture $\mathcal{Z}$, which depends on all agents' choices.

We introduce N decision makers, $\mathcal{A}_1, \dots, \mathcal{A}_N$ (see Figure 6.1 and Algorithm 2). At training round t, each agent chooses an operation (e.g., convolution or pooling layer) according to its local action-distribution (or policy) $\boldsymbol{a}_t^j \sim \pi_t^j$, for all $j \in \{1, \dots, N\}$ with $\boldsymbol{a}_t^j \in \{1, \dots, K\}$. These operations have corresponding operational weights $\boldsymbol{w}_t$ that are learned in parallel. Altogether, these choices $\boldsymbol{a}_t = \boldsymbol{a}_t^1, \dots, \boldsymbol{a}_t^N$ define a sparse graph/architecture $\mathcal{Z}_t \equiv \boldsymbol{a}_t$ for which a validation loss $\mathcal{L}_t^{(\text{val})}(\mathcal{Z}_t, \boldsymbol{w}_t)$ is computed and used by the agents to update their policies. After T rounds, an architecture is recommended by sampling $\boldsymbol{a}_{T+1}^j \sim \pi_{T+1}^j$, for all $j \in \{1, \dots, N\}$. These dynamics resemble bandit algorithms where the actions for an agent $\mathcal{A}_j$ are viewed as separate arms. Multi-arm bandit algorithms allow balancing exploration and exploitation in a decision-making process. By employing a multi-arm bandit, MANAS allows for a distributed space search through multiple agents with a global optimization goal, achieving both memory and time efficiency as a result. The employed MA framework leaves open the design of **1)** the sampling strategy π^j and **2)** how π^j is updated from the observed loss.

Minimization of Worst-Case Regret Under Any Loss

The following two notions of regret motivate the proposed NAS method. Given a policy π the *cumulative regret* $\mathcal{R}_{T,\pi}^\star$ and the *simple regret* $r_{T,\pi}^\star$ after T rounds and under the

worst possible environment ν, are:

$$\mathcal{R}^{\star}_{T,\pi} = \sup_{\nu} \mathbb{E} \sum_{t=1}^{T} \mathcal{L}_t(\boldsymbol{a}_t) - \min_{\boldsymbol{a}} \sum_{t=1}^{T} \mathcal{L}_t(\boldsymbol{a}), \quad (6.3)$$

$$r^{\star}_{T,\pi} = \sup_{\nu} \mathbb{E} \sum_{t=1}^{T} \mathcal{L}_t(\boldsymbol{a}_{T+1}) - \min_{\boldsymbol{a}} \sum_{t=1}^{T} \mathcal{L}_t(\boldsymbol{a}) \quad (6.4)$$

where the expectation is taken over both the losses and policy distributions, and $\vec{\boldsymbol{a}} = \{\boldsymbol{a}^{(\mathcal{A}_j)}\}_{j=1}^{N}$ denotes a joint action profile. The simple regret leads to minimizing the loss of the recommended architecture $\boldsymbol{a}_{T+1}$, while minimizing the cumulative regret adds the extra requirement of having to sample, at any time t, architectures with close-to-optimal losses. The proposed models and solutions in Section 6.2.3 are designed to be robust to arbitrary $\mathcal{L}_t(\boldsymbol{a}_t)$.

Because of the discrete nature of the NAS problem, during the search, the loss can take on large values or alternate between large and small values arbitrarily. Gradient-descent methods perform best under smooth loss functions, which is not the case in NAS. The worst-case regret minimization is a theoretically-grounded objective that we use in order to provide guarantees on the convergence of the algorithm when no assumptions are made on the process generating the losses.

Factorizing the Regret

Let us first formulate the multi-agent combinatorial online learning in a more formal way. At each round, agent $\mathcal{A}_i$ samples an action from a fixed discrete collection $\{\boldsymbol{a}_j^{(\mathcal{A}_i)}\}_{j=1}^{K}$. Therefore, after each agent chooses its action at round t, the resulting architecture $\mathcal{Z}_t$ is described by joint action profile $\vec{\boldsymbol{a}}_t = \{\boldsymbol{a}_{j_1}^{(\mathcal{A}_1),[t]}, \ldots, \boldsymbol{a}_{j_N}^{(\mathcal{A}_N),[t]}\}$ and thus, $\mathcal{Z}_t$ and $\vec{\boldsymbol{a}}_t$ are used interchangeably. Due to the discrete nature of the joint action space, the validation loss vector at round t is given by $\vec{\boldsymbol{\mathcal{L}}}_t^{(\text{val})} = \left(\mathcal{L}_t^{(\text{val})}\left(\mathcal{Z}_t^{(1)}\right), \ldots, \mathcal{L}_t^{(\text{val})}\left(\mathcal{Z}_t^{(K^N)}\right)\right)$ and for the environment one can write $\nu = \left(\vec{\boldsymbol{\mathcal{L}}}_1^{(\text{val})}, \ldots, \vec{\boldsymbol{\mathcal{L}}}_T^{(\text{val})}\right)$. The interconnection between joint policy $\boldsymbol{\pi}$ and an environment ν works in a sequential manner: at round t, the architecture $\mathcal{Z}_t \sim \boldsymbol{\pi}_t(\cdot|\mathcal{Z}_1, \mathcal{L}_1^{(\text{val})}, \ldots, \mathcal{Z}_{t-1}, \mathcal{L}_{t-1}^{(\text{val})})$ is sampled and the validation loss $\mathcal{L}_t^{(\text{val})} = \mathcal{L}_t^{(\text{val})}(\mathcal{Z}_t)$ is observed. As mentioned previously, assuming linear contribution of each individual action to the validating loss, one goal is to find a policy $\boldsymbol{\pi}$ that keeps the regret:

$$\mathcal{R}_T(\boldsymbol{\pi}, \nu) = \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\beta}_t^{\mathsf{T}} \boldsymbol{Z}_t - \min_{\boldsymbol{Z} \in \mathcal{F}} \left[\sum_{t=1}^{T} \boldsymbol{\beta}_t^{\mathsf{T}} \boldsymbol{Z}\right]\right] \quad (6.5)$$

small with respect to all possible forms of environment ν. We justify here with the cumulative regret the reasoning applies as well to the simple regret. Here, $\boldsymbol{\beta}_t \in \mathbb{R}_{+}^{KN}$ is a contribution vector of all actions, and $\boldsymbol{Z}_t$ is a binary representation of architecture $\mathcal{Z}_t$ and $\mathcal{F} \subset [0,1]^{KN}$ is set of all feasible architectures[1]. In other words, the quality of the policy

[1]Here, an architecture is feasible if and only if each agent chooses exactly one action.

is defined with respect to worst-case regret:

$$\mathcal{R}_T^* = \sup_{\nu} \mathcal{R}_T(\boldsymbol{\pi}, \nu) \quad (6.6)$$

The linear decomposition of the validation loss allows rewriting the total regret (Eq. 6.5) as a sum of agent-specific regret expressions $\mathcal{R}_T^{(\mathcal{A}_i)}\left(\boldsymbol{\pi}^{(\mathcal{A}_i)}, \nu^{(\mathcal{A}_i)}\right)$ for $i = 1, \ldots, N$:

$$\begin{aligned}
\mathcal{R}_T(\boldsymbol{\pi}, \nu) &= \mathbb{E}\left[\sum_{t=1}^{T}\left(\sum_{i=1}^{N} \boldsymbol{\beta}_t^{(\mathcal{A}_i),\mathsf{T}} \boldsymbol{Z}_t^{(\mathcal{A}_i)} - \sum_{i=1}^{N} \min_{\boldsymbol{Z}^{(\mathcal{A}_i)} \in \mathcal{B}^{(K)}_{||\cdot||_0,1}(\mathbf{0})} \left[\sum_{t=1}^{T} \boldsymbol{\beta}_t^{(\mathcal{A}_i),\mathsf{T}} \boldsymbol{Z}^{(\mathcal{A}_i)}\right]\right)\right] \\
&= \sum_{i=1}^{N} \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\beta}_t^{(\mathcal{A}_i),\mathsf{T}} \boldsymbol{Z}_t^{(\mathcal{A}_i)} - \min_{\boldsymbol{Z}^{(\mathcal{A}_i)} \in \mathcal{B}^{(K)}_{||\cdot||_0,1}(\mathbf{0})} \left[\sum_{t=1}^{T} \boldsymbol{\beta}_t^{(\mathcal{A}_i),\mathsf{T}} \boldsymbol{Z}^{(\mathcal{A}_i)}\right]\right] \\
&= \sum_{i=1}^{N} \mathcal{R}_T^{(\mathcal{A}_i)}\left(\boldsymbol{\pi}^{(\mathcal{A}_i)}, \nu^{(\mathcal{A}_i)}\right) \qquad (6.7)
\end{aligned}$$

where $\boldsymbol{\beta}_t = \left[\boldsymbol{\beta}_t^{\mathcal{A}_1,\mathsf{T}}, \ldots, \boldsymbol{\beta}_t^{\mathcal{A}_N,\mathsf{T}}\right]^{\mathsf{T}}$ and $\boldsymbol{Z}_t = \left[\boldsymbol{Z}_t^{(\mathcal{A}_1),\mathsf{T}}, \ldots, \boldsymbol{Z}_t^{(\mathcal{A}_N),\mathsf{T}}\right]^{\mathsf{T}}$, $\boldsymbol{Z} = \left[\boldsymbol{Z}^{(\mathcal{A}_1),\mathsf{T}}, \ldots, \boldsymbol{Z}^{(\mathcal{A}_N),\mathsf{T}}\right]^{\mathsf{T}}$ are decomposition of the corresponding vectors on agent-specific parts, joint policy $\boldsymbol{\pi}(\cdot) = \prod_{i=1}^{N} \boldsymbol{\pi}^{(\mathcal{A}_i)}(\cdot)$, and joint environment $\nu = \prod_{i=1}^{N} \nu^{(\mathcal{A}_i)}$, and $\mathcal{B}^{(K)}_{||\cdot||_0,1}(\mathbf{0})$ is unit ball with respect to $||\cdot||_0$ norm centered at $\mathbf{0}$ in $[0,1]^K$. Moreover, the worst-case regret (Eq. 6.6) also can be decomposed into agent-specific form:

$$\mathcal{R}_T^\star = \sup_{\nu} \mathcal{R}_T(\boldsymbol{\pi}, \nu) \quad \Longleftrightarrow \quad \sup_{\nu^{(\mathcal{A}_i)}} \mathcal{R}_T^{(\mathcal{A}_i)}\left(\boldsymbol{\pi}^{(\mathcal{A}_i)}, \nu^{(\mathcal{A}_i)}\right), \quad i = 1, \ldots, N. \quad (6.8)$$

This decomposition allows us to significantly reduce the search space and apply the two algorithms described in the next section for each agent $\mathcal{A}_i$ in a parallel fashion.

6.2.3 Adversarial Implementations

The following subsections describe the proposed approaches for NAS when considering adversarial losses. The two algorithms presented are MANAS and MANAS-LS, which implement two different credit assignment techniques specifying the update rule in line 7 of Algorithm 2. The first one approximates the validation loss as a linear combination of edge weights, while the second handles non-linear losses.

In this context, adversarial refers to the adversarial multi-arm bandit [252] framework: we model the fact that a weight-sharing super-network returns noisy rewards as having an adversary that explicitly tries to confuse the learner. Adversarial multi-arm bandit is the strongest generalization of the bandit problem, as it removes all assumptions on the distribution. The proposed MA formulation and algorithm explicitly account for this adversarial nature.

MANAS-LS

Linear Decomposition of the Loss. A simple credit assignment strategy is to approximate edge-importance (or edge-weight) by a vector $\boldsymbol{\beta}_s \in \mathbb{R}^{KN}$ representing the importance of all K operations for each of the N agents. $\boldsymbol{\beta}_s$ is an arbitrary, potentially adversarially-chosen vector and varies with time s to account for the fact that the operational weights $\boldsymbol{w}_s$ are learned online and to avoid any restrictive assumption on their convergence. The relation between the observed loss $\mathcal{L}_s^{(\text{val})}$ and the architecture selected at each sampling stage s is modeled through a linear combination of the architecture's edges (agents' actions) as:

$$\mathcal{L}_s^{(\text{val})} = \boldsymbol{\beta}_s^{\mathsf{T}} \boldsymbol{Z}_s \quad (6.9)$$

where $\boldsymbol{Z}_s \in \{0,1\}^{KN}$ is a vectorised one-hot encoding of the architecture $\mathcal{Z}_s$ (active edges are 1, otherwise 0). After evaluating S architectures, at round t, $\boldsymbol{\beta}$ can be estimated by solving the following via least-squares:

$$\text{Credit assignment:} \quad \widetilde{\boldsymbol{B}}_t = \arg\min_{\boldsymbol{\beta}} \sum_{s=1}^{S} \left(\mathcal{L}_s^{(\text{val})} - \boldsymbol{\beta}^{\mathsf{T}} \boldsymbol{Z}_s \right)^2. \quad (6.10)$$

The solution gives an efficient way for the agents to update their corresponding action-selection rules and leads to implicit coordination. In Section 6.2.2 it is demonstrated that the worst-case regret $\mathcal{R}_T^\star$ (Eq. 6.3) can actually be decomposed into an agent-specific form $\mathcal{R}_T^i\left(\boldsymbol{\pi}^i, \nu^i\right)$ defined as: $\mathcal{R}_T^\star = \sup_\nu \mathcal{R}_T(\boldsymbol{\pi}, \nu) \iff \sup_{\nu^i} \mathcal{R}_T^i\left(\boldsymbol{\pi}^i, \nu^i\right), \quad i = 1, \ldots, N$. This decomposition allows us to significantly reduce the search space complexity by letting each agent $\mathcal{A}_i$ determine the best operation for the corresponding graph edge.

Zipf Sampling for $r_{T,\pi}^\star$. $\mathcal{A}_i$ samples an operation k proportionally to the inverse of its estimated rank $\widetilde{\langle k \rangle}_t^i$, where $\widetilde{\langle k \rangle}_t^i$ is computed by sorting the operations of agent $\mathcal{A}_i$ w.r.t $\tilde{\boldsymbol{B}}_t^i[k]$, as

$$\text{Sampling policy:} \quad \boldsymbol{\pi}_{t+1}^i[k] = 1 \Big/ \widetilde{\langle k \rangle}_t^i \overline{\log} K \quad \text{where } \overline{\log} K = 1 + 1/2 + \ldots + 1/K. \quad (6.11)$$

Zipf explores efficiently, is parameter-free, minimizes optimally the simple regret in multi-armed bandits when the losses are adversarially designed, and adapts optimally to stationary losses [255].

MANAS

Coordinated Descent for Non-Linear Losses. In some cases, the linear approximation may be crude. An alternative is to make no assumptions on the loss function and have each agent directly associate the quality of their actions with the loss $\mathcal{L}_t^{(\text{val})}(\boldsymbol{a}_t)$. This results in all the agents performing a coordinated descent approach to the problem. For this, each agent updates for operation k its $\widetilde{\boldsymbol{B}}_t^i[k]$ as:

$$\text{Credit assignment: } \widetilde{\boldsymbol{B}}_t^i[k] = \widetilde{\boldsymbol{B}}_{t-1}^i[k] + \mathcal{L}_t^{(\text{val})} \cdot \mathbb{1}_{\boldsymbol{a}_t^i=k} / \boldsymbol{\pi}_t^i[k]. \quad (6.12)$$

Softmax Sampling for $\mathcal{R}_{T,\pi}^{\star}$. Following EXP3 [256], actions are sampled from a softmax distribution (with temperature η) w.r.t. $\tilde{\boldsymbol{B}}_t^i[k]$:

$$\text{Sampling policy: } \boldsymbol{\pi}_{t+1}^i[k] = \exp\left(\eta \tilde{\boldsymbol{B}}_t^i[k]\right) \Big/ \sum_{j=1}^{K} \exp\left(\eta \tilde{\boldsymbol{B}}_t^i[j]\right). \quad (6.13)$$

Using this sampling strategy, EXP3 [256] is run for each agent in parallel. If the regret of each agent is computed by considering the rest of the agent as fixed, then each agent has regret $\mathcal{O}\left(\sqrt{TK\log K}\right)$ which sums over agents to $\mathcal{O}\left(N\sqrt{TK\log K}\right)$. For this, a simplified notion of regret is considered, where each agent is considering the rest of the agents as part of the adversarial environment. Now, the goal is to minimize:

$$\sum_{i=1}^{N} \mathcal{R}_T^{\star,i}(\pi^{(\mathcal{A}_i)}) = \sum_{i=1}^{N} \sup_{\boldsymbol{a}_{-i},\nu} \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{L}_t^{(\text{val})}(\boldsymbol{a}_t^{(\mathcal{A}_i)}, \boldsymbol{a}_{-i}) - \min_{\boldsymbol{a}\in\{1,\dots,K\}} \left[\sum_{t=1}^{T} \mathcal{L}_t^{(\text{val})}(\boldsymbol{a}, \boldsymbol{a}_{-i})\right]\right], \quad (6.14)$$

where $\boldsymbol{a}_{-i}$ is a fixed set of actions played by all agents to the exception of agent $\mathcal{A}_i$ for the T rounds of the game and ν contains all the losses as $\nu = \{\mathcal{L}_t^{(\text{val})}(\boldsymbol{a})\}_{t\in\{1,\dots,T\},\boldsymbol{a}\in\{1,\dots,K^N\}}$.

The proposed MA formulation thus provides a gradient-free, credit assignment strategy. Gradient methods are more susceptible to bad initialization and can get trapped in local minima more easily than our approach, which, not only explores more widely the search space, but makes this search optimally according to multi-armed bandit derived regret minimization. Concretely, MANAS can escape from local minima as the reward is scaled by the probability of selecting an action (Eq. 6.12). Thus, the algorithm has a higher chance of revising its estimate of the quality of a solution based on new evidence. This is important as one-shot methods (such as MANAS and DARTS) change the super-network—and thus the environment—throughout the search process. Put differently, MANAS' optimal exploration-exploitation allows the algorithm to move away from 'good' solutions towards 'very good' solutions that do not live in the former's proximity. In contrast, gradient methods will tend to stay in the vicinity of a 'good' discovered solution.

6.3 Experiments

To evaluate the efficiency of MANAS, experiments on CIFAR-10 and ImageNet were performed, and the obtained results are compared with existing NAS methods. More, ablation studies were performed to further showcase MANAS efficiency: i) compared MANAS, DARTS, RS and RS with weight-sharing [1] on three new data sets for NAS (Sport-8, Caltech-101, MIT-67); and ii) evaluated MANAS with inference time as complexity constraint. The performance of two algorithms, MANAS and MANAS-LS is reported for all

experiments except the complexity constraint. More, with the exception of the results marked as *+AutoAugment*, all experiments were performed with the same final training protocol as DARTS [2], to allow a fair comparison. As for computational resources, experiments in ImageNet were conducted on multi-GPU machines with $8\times$ Nvidia Tesla V100 16GB GPUs (used in parallel), while the remaining experiments were performed on a single GeForce GTX 1080 GPU.

6.3.1 Search Space and Search Protocols

To evaluate MANAS, we use the same CNN search space proposed by DARTS [2]. This search space focuses on cell-search on a DAG, where the edges represent operations over the feature map of the input nodes. The operation pool of the search space has 8 possible operations: $\{3\times3, 5\times5\}$ separable convolutions; $\{3\times3, 5\times5\}$ depth separable convolutions; 3×3 max pooling; identity and zero.

Since MANAS is memory efficient, it can search for the final architecture without needing to stack *a posteriori* repeated cells. Therefore, all MANAS' searched cells are unique. For a fair comparison, we use 20 cells on CIFAR-10 and 14 on ImageNet. Experiments on Sport-8, Caltech-101 and MIT-67 use both 8 and 14 cell networks. In both MANAS variants, the number of agents is defined by the search space and is not tuned. Specifically, there exists one agent for each pair of nodes tasked with selecting the optimal operation. As there are 14 pairs (edges) in each cell, the total number of agents is $14 \times C$, with C being the number of cells ($8, 14$ or 20, depending on the experiment).

For data sets other than ImageNet, 500 epochs were used in the search phase for architectures with 20 cells, 400 epochs for 14 cells, and 50 epochs for 8 cells. All other hyperparameters are as in [2]. For ImageNet, we use 14 cells and 100 epochs during the search. In our experiments on the three new data sets, we rerun the DARTS code to optimize an 8 cell architecture, while for 14 cells the best cells were simply stacked for the appropriate number of times.

6.3.2 Data Sets

We directly search and evaluate the proposed method on five different data sets: CIFAR-10 [257], in which we followed standard data pre-processing and augmentation techniques [2, 70], i.e. subtracting the channel mean and dividing the channel standard deviation, centrally padding the training images to 40×40 and randomly cropping them back to 32×32; and randomly flipping them horizontally. ImageNet [258] with data pre-processing and augmentation techniques proposed in previous NAS methods [2, 70], i.e. subtracting the channel mean and dividing the channel standard deviation, cropping the training images to random size and aspect ratio, resizing them to 224×224, and randomly changing their brightness, contrast, and saturation, while resizing test images to 256×256 and cropping them at the center. And 3 data sets for which we used the same

pre-processing and data-augmentation techniques as for ImageNet: Sport-8, an action recognition data set containing 8 sport event categories with a total of 1579 images [259], Caltech-101, which contains 101 unbalanced categories [260], and MIT-67 that has 67 classes, totaling 15,620 images of different sizes. The small size of Sport-8 and MIT-67 allows stressing the generalization capabilities of a NAS method.

Table 6.1: Comparison with state-of-the-art image classifiers on CIFAR-10. The four blocks represent: human-designed, NAS, MANAS search with DARTS training protocol and best searched MANAS retrained with an extended protocol (AutoAugment (AA) + 1500 Epochs + 50 Channels). Unless specified, all MANAS architectures use 20 cells.

Method	**Test Error** (%) ↓	**Params** (M) ↓	**Search Cost** (GPU Days) ↓	**Search Method**
DenseNet-BC [60]	3.46	25.6	–	manual
NASNet-A [38]	2.65	3.3	1800	RL
AmoebaNet-B [72]	2.55	2.8	3150	EA
PNAS [217]	3.41	3.2	225	SMBO
ENAS [70]	2.89	4.6	0.5	RL
SNAS [98]	2.85	2.8	1.5	GB
DARTS, 1st order [2]	3.00	3.3	1.5†	GB
DARTS, 2nd order [2]	2.76	3.3	4†	GB
SDARTS-ADV [244]	2.61	3.3	4.3	GB
DARTS- [246]	2.63	3.5	4†	GB
GDAS [261]	3.75	3.4	0.2	GB
NPENAS-BO [130]	2.52	4.0	2.5	EA
EffPNet [262]	3.49	2.54	3	EA
BANANAS [123]	2.64	–	11.8	BO + predictor
RS + cutout [2]	3.29	3.2	–	–
RS + weight-sharing [1]	2.85	4.3	9.7	RS
MANAS (8 cells)	3.05	1.6	0.8†	MA
MANAS	2.63	3.4	2.8†	MA
MANAS–LS	**2.52**	3.4	4†	MA
MANAS + AA	1.97	3.4	–	MA
MANAS–LS + AA	1.85	3.4	–	MA

† Search cost is for 4 runs and the test error is for the best result (for a fair comparison with other methods).

6.3.3 Results and Discussion

The first experiment to evaluate the proposed methods was conducted using CIFAR-10, for this we follow DARTS's protocol [2]: MANAS is executed 4 times with different random seeds and pick the best architecture based on its validation performance. We then randomly reinitialize the weights and retrain for 600 epochs. During the search stage, half of the training set is used as validation. To fairly compare with more recent methods,

the best searched architecture is re-trained using AutoAugment and Extended Training [263].

Results depicted in Table 6.1 show that both MANAS implementations perform well on CIFAR-10 using DARTS search space. The proposed method was designed to perform comparably to [2] but with an order of magnitude less memory. However, MANAS is actually capable of achieving higher accuracy. The reason for this is that DARTS is forced to search for an 8 cell architecture and subsequently stack the same cells 20 times, while MANAS can directly search on the final number of cells, thus leading to better results. Results for MANAS searching for only 8 cells are also shown in Table 6.1. Even though the network is much smaller, it still performs competitively with 1st-order DARTS. In terms of memory usage with a batch size of 1, MANAS 8 cells required only 1GB of GPU memory, while DARTSv1 utilized more than 8.5GB and DARTSv2 required 9.6GB, making both versions of DARTS unpractical to work with data sets with larger image sizes. Lastly, when training the best MANAS/MANAS-LS architectures with an extended protocol (AutoAugment + 1500 Epochs + 50 Channels, in addition to the DARTS protocol) the results obtained clearly outperform state-of-the-art NAS methods, while also providing memory and time efficiency gains.

Table 6.2: Comparison with state-of-the-art image classifiers on ImageNet (mobile setting). The four blocks represent: human-designed, NAS, MANAS search with DARTS training protocol and best searched MANAS retrained with an extended protocol (AutoAugment (AA) + 600 Epochs + 60 Channels).

Method	**Test Error** (%) ↓	**Params** (M) ↓	**Search Cost** (GPU Days) ↓	**Search Method**
ShuffleNet 2x (v2) [264]	26.3	5	—	manual
NASNet-A [38]	26.0	5.3	1800	RL
AmoebatNet-C [72]	24.3	6.4	3150	EA
PNAS [217]	25.8	5.1	225	SMBO
SNAS [98] (search on C10)	27.3	4.3	1.5	GB
DARTS [2] (search on C10)	26.7	4.7	4	GB
GDAS [261] (search on C10)	27.5	4.4	0.17	GB
EffPNet [262] (search on C10)	27.1	–	3	EA
NASP [242] (search on C10)	26.3	9.5	0.2	proximal
RS	27.75	2.5	—	—
MANAS (search on C10)	26.47	2.6	2.8	MA
MANAS (search on IN)	26.15	2.6	110	MA
MANAS (search on C10) + AA	26.81	2.6	—	MA
MANAS (search on IN) + AA	25.26	2.6	—	MA

To evaluate MANAS on a larger data set, experiments on ImageNet were also conducted. For these, the generated architectures were trained for 250 epochs. As search and augmentation are very expensive we use only MANAS and not MANAS-LS, as the former is computationally cheaper and performs slightly better on average. In Table 6.2, results for

networks searched both on CIFAR-10 and directly on ImageNet, which is made possible by the computational efficiency of MANAS, are shown. When compared to SNAS, DARTS, GDAS and other methods, using the same search space, MANAS achieves state-of-the-art results both with architectures searched directly on ImageNet and also with architectures transferred from CIFAR-10. More, when training the best MANAS architecture searched directly on ImageNet with an extended training protocol (AutoAugment + 600 Epochs + 60 Channels, in addition to the DARTS protocol), the obtained performance improves by 0.89%, resulting in a final test error of 25.26%. The obtained results show that MANAS is an efficient approach, both when transferring architectures from smaller data sets, but also when directly searching on ImageNet, which other methods usually cannot.

From the results in both CIFAR-10 and ImageNet, it is clear that in specific settings, sampling architectures randomly performs very competitively. Since the search space is very large (between 8^{112} and 8^{280} architectures exist in the DARTS experiments), finding the global optimum (randomly) is practically impossible. Previous studies [170, 1] together with the results obtained here seem to indicate that the available operations and meta-structure have been carefully chosen and, as a consequence, most architectures in DARTS search space generate good results. This suggests that human effort has simply transitioned from finding a good architecture to finding a good search space—a problem that needs careful consideration. RS with weight-sharing [1], has also shown to perform competitively but it falls short when compared to the proposed MA framework (including in the ablation studies shown in the next section).

It is worth noting that all comparisons were performed using the same final training protocol. This is relevant as studies have shown that the final training protocols can influence the final performance of an architecture more than the search strategy itself [40]. As such, any improvement in the final accuracy obtained by the generated architectures is indicative of MANAS improvements and not the final training of the architectures.

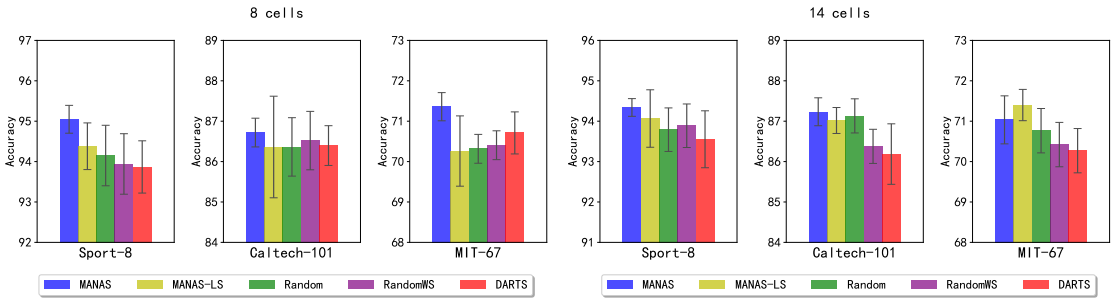


Figure 6.2: Comparing MANAS, RS, RS with weight-sharing [1] and DARTS [2] on 8 and 14 cells. Average results of 8 runs. Note that DARTS was only optimized for 8 cells due to memory constraints.

6.3.4 Ablation Studies

The idea behind NAS is to find the optimal (or close) architecture for a given problem. Limiting the evaluation of current methods to CIFAR-10 and ImageNet could potentially

lead to algorithmic overfitting. Recent studies suggest that the search space was engineered in a way that makes it very hard to find a bad architecture [1, 170, 40, 45]. To mitigate this, we tested different NAS methods on 3 data sets, which are composed of images larger than those of CIFAR-10 and were not used before in this setting, but have been used in the CV field: Sport-8, Caltech-101 and MIT-67. For this set of experiments, each algorithm was run 8 times, and mean accuracy and standard deviation is reported. This evaluation is performed for both 8 and 14 cells. As for the evaluation of DARTS in these data sets, due to memory constraints, it can only search for 8 cells. RS with weight-sharing and RS were used to serve as baselines. For the latter 8 architectures from the search space were uniformly sampled. To efficiently implement RS with weight-sharing, we follow [1]. Each proposed architecture is trained from scratch for 600 epochs as in the previous section. Results are depicted in Figure 6.2. From these, it is clear that MANAS manages to outperform the random baselines and significantly outperform DARTS, especially on 14 cells, clearly showing that the optimal cell architecture for 8 cells is not the optimal one for 14 cells. More, the obtained results further promote the generability of MANAS across different problems.

Table 6.3: Results of MANAS with complexity constraints using different penalty (ρ) values on CIFAR-10.

ρ	**Test Error** (%) $\downarrow$	**Inference Time** (ms) $\downarrow$
0	2.91	1.255 ± 0.012
0.1	3.12	1.196 ± 0.009
0.25	2.94	1.190 ± 0.008
0.5	3.04	1.183 ± 0.006
0.75	2.54	1.179 ± 0.005
1	2.69	1.164 ± 0.006

The last experiment conducted to evaluate MANAS is in a complexity constraint setting. For this, the inference time of the generated architectures was added as a complexity constraint to the training. To accommodate the added constrain, the training loss was updated to take into consideration the inference time that the generated architecture takes to classify an image: $\mathcal{L}_t^{(\text{train})}(\boldsymbol{a}_t) = \mathcal{L}_t^{(\text{train})}(\mathcal{Z}_t, \boldsymbol{w}_t) + \rho L_t(\mathcal{Z}_t, \boldsymbol{w}_t)$, where L_t is the inference time to classify one image, and ρ defines the importance given to L_t. Here, the ρ serves to vary the importance given to the inference time while searching. By increasing ρ, the inference time constraint has a higher importance in the loss calculation. This experiment allows evaluating MANAS in a simple multi-objective setting while ensuring that MANAS can perform in different settings. Table 6.3 shows the results of running MANAS with different ρ values using a single GPU. The depicted results show that by increasing the importance of the inference component, MANAS consistently generates architectures with a lower inference time with similar accuracies. These show that MANAS can be extended to a multi-objective search by modifying the training loss to accommodate different components.

6.4 Conclusions

In this chapter, we framed NAS as a MA problem and presented MANAS, a theoretically grounded multi-agent online learning framework for NAS. Then, the proposed two extremely lightweight implementations were described. The proposed implementations differ regarding the sampling and credit assignment during training. Based on this, experiments using the DARTS search space in both CIFAR-10 and ImageNet were conducted, showing that MANAS outperforms state-of-the-art while reducing memory consumption by an order of magnitude.

Furthermore, in this chapter, we propose the use of 3 data sets that have been exhaustively used in CV but not within NAS: Sport-8, Caltech-101, and MIT-67. In those, we empirically show the effectiveness of MANAS and evaluate how baselines (RS-based) perform. Finally, we confirm concerns raised in recent works [170, 1, 40] claiming that NAS algorithms often achieve minor gains over random architectures. We further demonstrate that MANAS produces competitive results with limited computational budgets, even in complexity-constrained scenarios.

Even though this chapter improves upon current NAS state-of-the-art, we still relied on a cell-based search in a human-defined search space. In the next chapter, we extend NAS for a macro-search and propose a set of methods to heavily reduce human involvement.

Chapter 7

Towards Less Constrained Macro Neural Architecture Search

7.1 Introduction

Even though NAS methods perform well on designing CNNs, they still encounter several drawbacks: i) search spaces are heavily dependent on human definitions, and are usually small with forced operations; ii) the search is mostly cell-based, where NAS methods search for small cells that are later replicated in a human-defined outer-skeleton; and iii) architecture's parameters, such as the number of layers, inner-layer parameters (e.g., the kernel size, output channels), the final architecture skeleton, fixed operations, and head and tail of the final architectures are usually defined by the authors; and iv) the time required to perform macro-search (search entire networks) is still considerable. By forcing rules and carefully designing search spaces, there are undoubtedly human biases introduced in the loop, as shown in the previous chapter. This often impacts more the final result of the architectures than the search strategy itself, as it pushes NAS to constrained search spaces with very narrow accuracy ranges [40, 45], thus jeopardizing the generalization of the NAS methods, even to simpler settings [42, 1]. Moreover, the idea of heading to NAS to avoid needing deep knowledge regarding architecture design ends up being frustrated since human involvement is required to design the search space, the outer-skeleton scheme, the pool of operations, and all the fixed hyper-parameters to ensure that a cell-based space allows designing efficient CNNs.

In this chapter, we propose Towards Less Constrained Macro-Neural Architecture Search (LCMNAS), a NAS method that is capable of: i) autonomously generating complex search spaces by creating Weighted Directed Graphs (WDGs) with hidden properties from existing architectures to leverage information that is the result of years of expertise, practice, and trial-and-error; ii) performs macro-architecture search without explicitly defined architecture's outer-skeletons, restrictions or heuristics; and iii) uses a mixed-performance strategy for estimating the efficiency of the generated architectures, that combines information about the architectures at initialization stage with information about their validation accuracy after a partial train on a partial data set. To validate the proposed method, we conduct extensive experiments in both macro and micro-based search settings to allow comparison with existing NAS methods. Extensive ablation studies show the importance of different NAS components in both micro and macro-search settings and discuss their usability and importance.

The main contributions of LCMNAS are summarized as follows:

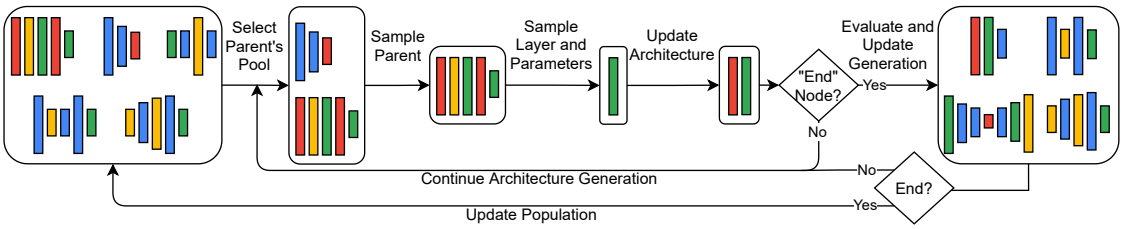


Figure 7.1: An illustration of one iteration of LCMNAS, showing the process of sequentially designing and evaluating architectures. It begins by sampling one layer and associated parameters for an architecture. The sampling is repeated several times until an architecture is fully designed. Then, the architecture is evaluated using a mixed-performance strategy and added to the population. The process of designing and evaluating architectures is repeated multiple times in a generation, and the search continues until the evolution reaches a final generation.

- A novel search space design method that leverages information about existing CNNs to autonomously design complex search spaces.

- An evolutionary search strategy that goes beyond micro-search to macro-search. where entire architectures are designed, without forced architecture shape or structure, or well-engineered protocols. More, the proposed search strategy goes beyond solely designing architectures to also determining the hyper-parameters associated with each layer.

- A mixed-performance estimation mechanism that correlates untrained architecture scores with their final performance, and combines it with partial training to infer the architecture's trainability with minimal training. This allows for remarkable computation efficiency improvements.

- Experiments demonstrate that the architectures found with LCMNAS achieve state-of-the-art results in both micro and macro-search settings, spanning across 14 different data sets.

- Extensive ablation studies show the importance of different NAS components, e.g., zero-cost proxy estimation in both micro and macro-search settings, and evaluation of standard practices, such as transferring searched architectures to new data sets, and architecture diversity via ensembling.

- Finally, we extract insights about architecture design based on the decisions made by LCMNAS that might inspire future research.

The remainder of this chapter is organized as follows. Section 7.2 describes the proposed NAS method. Section 4.3, presents the experiments performed in a macro-search setting, the results, and a discussion, while section 7.4 presents the experiments performed in a cell-based search and discusses the obtained results. Finally, a conclusion is drawn in Section 7.5.

7.2 Proposed Method

7.2.1 Search Space

As seen in chapters 5 and 6, search spaces are commonly defined as DAGs, where edges represent operations and nodes are tensors. The goal in those is usually to design cells that are repeated to form entire architectures, where the operations pool and the architectures' outer-skeletons are pre-defined by the users. However, this has been shown to force the search to very narrow accuracy ranges and to compromise the generalization beyond the evaluated data sets [40, 1, 45], and at the same time, requires deep expertise to apply to new problems.

Instead of explicitly defining a search space, we propose a method to automatically design search spaces without requiring human-specified settings by leveraging information about existing CNNs, which are the result of years of expertise, practice, and many trial-and-error experiments. For this, we extend the use of DAGs, and represent existing CNNs as WDGs with hidden properties: $\boldsymbol{G}(\boldsymbol{Q}, \boldsymbol{E}, \boldsymbol{H})$, where $\boldsymbol{Q}$ is the set of nodes, $\boldsymbol{E}$ is the set of edges and associated weights as layer-transition probabilities and $\boldsymbol{H}$ is the set of inner states of the nodes and associated probabilities.

To generate a WDG, $\boldsymbol{G}(\boldsymbol{Q}, \boldsymbol{E}, \boldsymbol{H})_i$, one can perform a feed-forward on an architecture with an input x, allowing an analysis of the architecture's layers, including those that are not defined on the architecture design but on the forward loop. With this, a summary of n_i can be generated and used to produce the sets $\boldsymbol{Q}$ and $\boldsymbol{E}$. To obtain the layer-transition probability between two layers, $P(l_i|l_j)$, we divide the frequency of the layer-transition $C(l_i, l_j)$ by the sum of frequencies of layer-transitions from l_i to any other possible layer: $C(l_i, l_k), k = 1, ..., K$, where K is the number of layers that appear after the layer l_i in the summary. Thus, the weight of an edge $e(l_i, l_j) \in \boldsymbol{E}$ is defined as:

$$e(l_i, l_j) = P(l_i|l_j) = \frac{C(l_i, l_j)}{\sum_{k=1}^{K} C(l_i, l_k)} \quad (7.1)$$

The inner-states $h_i \in \boldsymbol{H}$ of a layer l_i represent all possible parameters and associated values that appear in the CNN, e.g., for a convolutional layer, possible inner-states will be the output channels and kernel sizes and their associated probabilities. The inner-states of a layer $h \in \boldsymbol{H}$ are calculated similarly to the edges between nodes. For each possible component (e.g., kernel size), the probability associated with the value (e.g., kernel size 3×3) is calculated based on the frequency of that value divided by all possible values for that specific parameter. Note that a start and end node are added to all WDG to allow the search strategy to begin and stop the search.

At the end, the search space, $\mathcal{A}$, is composed by all CNNs in the form of their WDGs and associated fitness (see section 7.2.3) for the given problem: $(\boldsymbol{G}(\boldsymbol{Q}, \boldsymbol{E}, \boldsymbol{H}), f)$, such that

$\mathcal{A} = \{(\boldsymbol{G}(\boldsymbol{Q_1}, \boldsymbol{E_1}, \boldsymbol{H_1}), f_1), ..., (\boldsymbol{G}(\boldsymbol{Q_N}, \boldsymbol{E_N}, \boldsymbol{H_N}), f_N)\}$, where N is the number of CNNs used to create the initial search space.

By using WDGs to represent CNNs, the proposed method extracts information about the combination of layers and their associated parameters. This allows NAS methods to extract past information and move beyond from solely designing layers and using heuristics to select their parameters, to a broader search, where the focus is on designing entire architectures by finding both the layers and associated parameters. The method is, thus, inferring human expertise directly from the search space without requiring human intervention in the form of defined settings or parameters. Moreover, by combining the evaluation of CNNs for a specific problem in the form of the fitness f, into the WDGs, $\boldsymbol{G}(\boldsymbol{Q}, \boldsymbol{E}, \boldsymbol{H})$, the search strategy is capable of guiding the search more efficiently.

In Figure 7.2, the WDG of DenseNet121 is depicted, where layers are the nodes and connecting edges represent layer-transition probabilities. The generated WDGs differ from the usual definition of DAGs, as here, the nodes represent layers instead of tensors, and edges represent layer-transition probabilities instead of layers. The proposed definition of WDG further differs from DAGs by introducing inner-states that allow the definition of parameters associated with the layers (nodes).

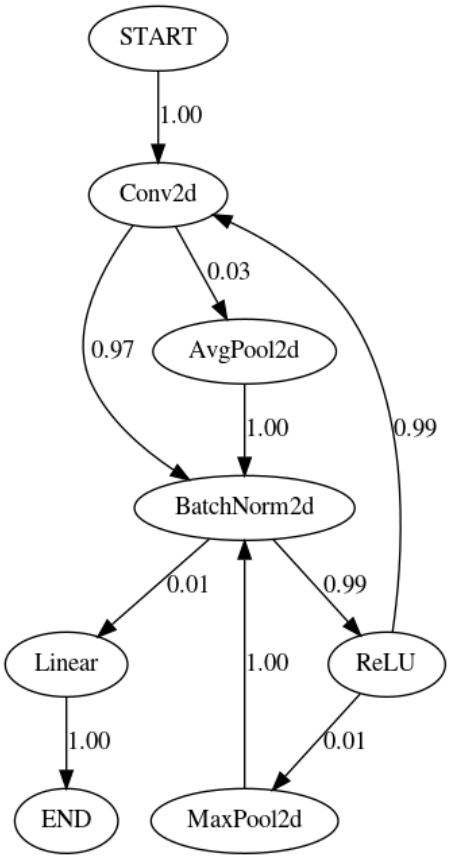


Figure 7.2: WDG representation of DenseNet121. Nodes represent layers and edges represent layer-transition probabilities between two nodes.

7.2.2 Search Strategy

We propose an evolutionary strategy that leverages the information in the WDGs to generate architectures from scratch, without requiring heuristics or pre-defined schemes for the layer's hyper-parameter and architecture structures. Architectures are generated by sequentially sampling layers and associated hyper-parameters from the individuals present in $\mathcal{A}$. To generate a layer, l_i, a pool of parents, $D \subseteq \mathcal{A}$, is sampled based on the presence of the last sampled layer l_{i-1} on all individuals from $\mathcal{A}$ (l_1 is the start node). An individual $j \in D$, is selected to be the parent of the layer l_i by performing a ranked roulette wheel selection using the fitness of the individuals in D. This ensures that individuals in D that have high fitness scores do not dominate the selection, thus promoting exploration of the search space. After an individual $j \in D$ is selected as parent, a layer l_i is sampled using fitness proportionate selection by having as weights the layer-transition probabilities in $\boldsymbol{G}(\boldsymbol{Q_j}, \boldsymbol{E_j}, \boldsymbol{H_j})$, denoted as $P(l_t|l_{i-1}),\ t = 1, ..., T$, where T is the number of the nodes that have an input layer-transition from l_{i-1}. Finally, the components of the sampled layer, e.g., output channels, are also sampled using fitness proportionate selection from the inner-state of the sampled layer l_i from the sampled parent $j \in D$. The iterative process of sampling layers from different parents promotes diversity in the generated architecture. Furthermore, following the insights obtained by evolutionary NAS methods like large-scale evolution [72] regarding the benefits of applying mutations, LCMNAS also employs a mutation process: when a convolutional layer is sampled, it has a fixed 50% probability of being mutated to a skip-connection.

The evolutionary strategy is performed for g generations, where at each generation, p individuals are generated, and the top 15% architectures from the parents population are passed to the next generation through elitism. Figure 7.1 illustrates the evolutionary process, where architectures are represented with varying widths and bar lengths to illustrate their diversity in terms of operations and architecture's scheme.

7.2.3 Performance Estimation Mechanism

The greatest NAS bottleneck is the evaluation of the generated architectures, as training each one until convergence is extremely expensive, requiring thousands of GPU days of computation [15, 38]. Accordingly, we propose a mixed-performance estimation approach that combines low-fidelity estimates with zero-cost proxies to speed up the evaluation, while at the same time ensuring that the resulting score is a reliable approximation of the fully trained architecture ranking.

First, the proposed method evaluates the trainability of a generated architecture in a partial data set for a small number of epochs, e. Let the objective function that calculates the accuracy of the architecture n_i on a small validation set, d^{valid} be denoted by $\mathcal{O}(n_i, d^{valid})$. The validation accuracy is used as an indication if the architecture can learn from the small number of examples shown, which ultimately can be used to distinguish

between architectures that can be trained efficiently from those that cannot. The proposed method then looks at the capability of the untrained architecture at initialization stage to model complex functions through Jacobian analysis [153, 48]. For this, we use the Jacobian $\boldsymbol{J}$ previously defined in Section 4.2.1. Equation 4.2 is then used to evaluate how an architecture behaves for different images.

Then, the correlation matrix, $\boldsymbol{\Sigma}_J$, allows the analysis of how the architecture models the target function at different data points, where ideally, for different images, the architecture would have different $\boldsymbol{J}$ values, thus resulting in low correlated mappings. Then, we can use KL divergence to score an architecture based on the eigenvalues of its correlation matrix. Let $\sigma_{J,1} \leq \cdots \leq \sigma_{J,V}$ be the V eigenvalues of $\boldsymbol{\Sigma}_J$. The untrained architecture can be scored as:

$$s = 1 \times 10^4 / \sum_{i=1}^{V} [\log(\sigma_{J,i} + k) + (\sigma_{J,i} + k)^{-1}], \quad (7.2)$$

where $k = 1 \times 10^{-5}$. The small constant values provide scaling factors and bound the logarithmic value, allowing better precision.

The proposed mixed-performance estimation approach calculates the fitness, f, by combining the evaluation of the architecture's trainability, using $\mathcal{O}(n_i, d^{valid})$, and their capability of modeling complex functions, using s by:

$$f_{n_i} = (1 - \lambda) \times \mathcal{O}(n_i, d^{valid}) + \lambda \times s \quad (7.3)$$

where λ serves the purpose of giving different weights to each component. When $\lambda = 0$, the fitness of an architecture is only based on the trainability of the network by looking at the validation accuracy using a partial train on a partial data set, $\mathcal{O}(n_i, d^{valid})$. When $\lambda = 1$, the architecture's capability of modeling complex functions at initialization stage is the only considered factor. We conducted several experiments to validate the importance of λ (see Sections 7.3 and 7.4).

7.3 Designing Entire Architectures: Experiments and Results Analysis

7.3.1 Data Sets

We directly search and evaluate the proposed method on three different data sets: CIFAR-10 [257], CIFAR-100 [257] and a reduced and noisy version from ImageNet: ImageNet16-120 [42]. CIFAR-10 and CIFAR-100 both contain $60K$ images, from which $50k$ are training images and $10k$ are test images. All images have 32×32 pixel sizes, and the data sets have 10 and 100 classes, respectively. ImageNet16-120 is a down-sampled variant of ImageNet, where all images have 16×16 pixels and only considers the first 120

Table 7.1: Comparison of the proposed method against ResNet, the best architecture in NAS-Bench-201 benchmark, and 3 types of RS. The proposed method is evaluated using different λ values. When applicable, the comparison is measured by the test error (%), the inference time (ms), the search cost in GPU days, and the number of parameters of the generated architecture in millions.

Architecture	CIFAR-10				CIFAR-100				ImageNet16-120			
	Test Error (%) ↓	Inference Time (ms) ↓	Search Cost (GPU days) ↓	Params (M) ↓	Test Error (%) ↓	Inference Time (ms) ↓	Search Cost (GPU days) ↓	Params (M) ↓	Test Error (%) ↓	Inference Time (ms) ↓	Search Cost (GPU days) ↓	Params (M) ↓
ResNet [59]	6.43	0.24±0.19	-	1.7	29.14	0.21±0.13	-	1.7	56.37	0.10±0.19	-	1.7
NAS-Bench-201 Top Architecture [42]	5.63	0.16±0.13	-	1.1	26.49	0.18±0.13	-	1.3	52.69	0.08±0.12	-	0.9
RS-L	10.73	1.06±0.10	0.27	34.4	31.95	0.28±0.09	0.01	0.7	82.08	0.01±0.01	0.26	0.5
RS-M	21.67	0.09±0.01	0.01	0.3	46.82	0.06±0.01	0.01	0.8	60.02	0.03±0.01	0.01	1.3
RS	7.96	0.03±0.01	0.01	0.6	47.33	0.01±0.01	0.01	0.4	64.33	0.02±0.01	0.01	0.7
Ours (λ=0)	6.05	2.08±0.04	1	43.9	29.44	1.67±0.02	0.67	49.5	58.45	0.75±0.01	1.29	224.7
Ours (λ=0.25)	5.49	0.97±0.02	1.23	32.5	28.27	0.92±0.35	1.9	60.4	57.33	3.43±0.04	3	171.1
Ours (λ=0.50)	4.64	0.26±0.04	1.36	13.9	25.45	1.81±0.02	2.7	51.5	49.12	0.19±0.01	5.01	12.8
Ours (λ=0.75)	4.23	0.33±0.01	1.03	6.7	25.99	0.39±0.18	2.8	32.3	45.78	0.90±0.01	4.79	42.7
Ours (λ=1)	4.81	0.15±0.01	0.1	2.5	26.52	0.16±0.01	0.01	2.2	51.17	0.76±0.01	0.12	31.5
Ours (best)+AA†	**2.96**	0.33±0.01	1.03	6.7	**20.94**	1.81±0.02	2.7	51.5	**43.35**	0.90±0.01	4.79	42.7

† Results obtained by training the best model found in each data set: $\lambda = 0.75$ for CIFAR-10 and ImageNet16-120 and $\lambda = 0.5$ for CIFAR-100 with AutoAugment for 1500 epochs.

classes of ImageNet, resulting in approximately $158K$ images. From which, $152k$ are for the training set, $3K$ for the validation and $3K$ for the test set. For searching purposes, we generated a partial data set for all the data sets. Here, 8% and 2% of the training set were randomly sampled to serve as partial training and partial validation sets. When using larger data sets, smaller subsets of the original data sets can be used to ensure that the search method remains efficient. The partial training requires only a small set of images from different classes to assess the trainability of a network. For reference, on CIFAR-100 we used 40 images per class for the training set, which was found sufficient to evaluate if an architecture can model the target function. More, the data set could be further reduced to an even smaller number of images or trimmed in terms of classes to accommodate more images per class. For the final training of the selected architecture, we use the original, entire splits.

7.3.2 Final Training

To evaluate the final architecture, we follow common training procedures [2, 70, 72]. However, we do not take advantage of drop path and other well-engineered training protocols that hide the contributions of the search strategy and the search space [40], or that require forcing architectures to have specific schemes, e.g., auxiliary towers [265], as suggested by NAS best practices in order to showcase the true contributions of the proposed method [266, 1, 40, 45].

The final architecture is trained for 600 epochs with a batch size of 96. We use SGD optimization, with an initial learning rate, $\eta = 0.025$ annealed down to zero following a cosine schedule without restart [267], momentum of 0.9 [268], weight decay of 3×10^{-4} and cutout [269]. The use of these settings follows common training settings from previous NAS methods, allowing a fairer comparison with the results obtained by other NAS methods [2, 270, 40].

7.3.3 Search Space

To create the search space, we use all the 34 models present in TorchVision version 0.8 [271]: AlexNet [272], DenseNet$\{121, 161, 169, 201\}$ [60], GoogLeNet [58], MNASNet$\{0.5, 0.75, 1, 1.3\}$ [226], MobileNetv2 [273], ResNet$\{18, 34, 50, 101, 152\}$ [59], ResNext$\{50, 101\}$ [203], ShuffleNetv2$\{0.5, 1.0, 1.5, 2.0\}$ [274], SqueezeNet$\{1.0, 2.0\}$ [275], VGG$\{11, 11BN, 13, 13BN, 16, 16BN, 19, 19BN\}$ [57] and Wide ResNet$\{50, 101\}$ [204]. For each model, a WDG is automatically generated and the associated fitness is calculated using the performance estimation method.

The resulting search space is extremely large, composed of 17 possible operations: $\{1\times1$, 3×3, 5×5, 7×7, $11 \times 11\}$ convolution; $\{2 \times 2, 3 \times 3\}$ max pooling; 2×2 average pooling; $\{1 \times 1, 6 \times 6, 7 \times 7\}$ adaptive average pooling, $\{4096, 1024, 1000\}$ linear, $\{0.2, 0, 5\}$ dropout and skip-connection. In the WDGs, $\boldsymbol{G}(\boldsymbol{Q}, \boldsymbol{E}, \boldsymbol{H})$, the inner-state choices for a given node, e.g., the output channels for a convolutional layer, are also present.

As we do not bound the search by defining the architecture's outer-skeletons or the number of layers, the search space is highly complex, meaning that a generated architecture can be of any length and scheme. More, in our experiments, we create a search space for each evaluated data set, as the CNNs yield different fitnesses for different problems. Also, by using information about the probabilities of layer-transition and the inner-states, LCMNAS leverages existing information to efficiently guide the search and design the architecture and the layer's parameters without requiring heuristics. Future works can use this search space without leveraging such information and solely focus on the possible operations to design cell-based architectures.

7.3.4 Results and Discussion

To evaluate the proposed method, we conducted extensive experiments using a single 1080Ti GPU. First, following the best practices proposed for NAS [266], we evaluate the performance of RS to infer the complexity of the search space [1]. For this, we evaluate RS by randomly sampling architectures from the search space, $\mathcal{A}$, and two RS strategies based on the proposed evolutionary strategy: RS-L, which randomly samples layer components, and RS-M, which randomly samples models to serve as parents. The results for these three methods are shown in the second block of Table 7.1. As the three RS methods attain high values for the test error, it shows that the search space is complex, and unlike other search spaces, RS is not sufficient to design competitive architectures [1, 40]. Then, we evaluated the proposed method by giving different importance (λ) to the two components of the mixed-performance estimation. Results are presented in the third block of Table 7.1. Following the best practices suggested in [40], we first present the test errors without any added training protocol. From these, we see that a combination of both evaluating an architecture based on its trainability using low-fidelity estimates and its modeling capabilities at initialization stage, yields the best results. Moreover, higher λ values serve

Table 7.2: Comparison of different methods on CIFAR-10. The first block presents a state-of-the-art human-designed CNN. The second block presents the results of proposals that perform macro-search, and the third block presents the proposed method. For each method, the test error in percentages, the search cost in terms of GPU days, and the size of the architecture in millions of parameters are shown. The search cost is the total GPU computation used in days, based on the number of GPUs and the execution time.

Method	**Test Error** (%) ↓	**GPUs** ↓	**Time** (Days) ↓	**Search Cost** (GPU Days) ↓	**Params** (M) ↓	**Search Method**
ResNet [59]	6.43	-	-	-	1.7	manual
Super Nets [276]	9.21	-	-	-	-	-
ConvFabrics [277]	7.43	-	-	-	21.2	-
MetaQNN [68]	6.92	10	10	100	11.2	RL
EAS [166]	5.70	5	2	10	19.7	RL
Large-scale Evolution [84]	5.40	250	11	2750	5.4	EA
EPNAS [278]	5.14	1	1.2	1.2	5.9	SMBO
NAS [15]	4.47	800	28	22400	7.1	RL
ENAS [70]	4.23	1	0.32	0.32	21.3	RL
SMASH [116]	4.03	1	1.5	1.5	16.0	OS
EPNAS + more channels [278]	4.01	1	1.2	1.2	38.8	SMBO
ENAS + more channels [70]	3.87	1	0.32	0.32	38.0	RL
NSGA-NET [110]	3.85	1	8	8	3.3	EA
NAS + more filters [15]	3.65	800	28	22400	37.4	RL
LEMONADE [90]	3.60	16	5	80	8.9	EA
RandGrow [167]	3.38	4	1.5	6	3.1	RS
RandGrow + DropPath [167]	2.93	4	1.5	6	3.1	RS
Petridish [168]	**2.83**	1	5	5	2.2	GB
Ours ($\lambda = 0.75$)+AA	2.96	1	1	1	6.7	EA

as regularization, reducing the number of parameters in the generated architectures. The search cost is lower when $\lambda = 1$, due to the fast evaluation of untrained networks. Differences in search cost between data sets are due to the unconstrained properties of the search. As for noisier data sets, larger models tend to be generated, thus taking more time to evaluate. The larger size of ImageNet16-120 also slows the evaluation. For the best architecture found in each data set ($\lambda = 0.75$ in CIFAR-10 and ImageNet16-120 and $\lambda = 0.5$ in CIFAR-100), we also show the test error by training the architecture for 1500 epochs with AutoAugment [263], attaining 2.96% test error in CIFAR-10, 20.94% in CIFAR-100 and 43.35% in ImageNet16-120. By comparing the proposed method with ResNet, and the best possible architecture in the NAS-Bench-201 (first block in Table 7.1), it is possible to see that the proposed method heavily outperforms existing state-of-the-art manually designed CNNs and all cell-based CNNs present in NAS-Bench-201. Moreover, comparing with RS indicates the effectiveness of the search, where lower test errors show that LCMNAS effectively searches through the complex space of architectures.

Table 7.2 compares the proposed method with different NAS methods that perform macro-search using CIFAR-10. It is important to note that most of these methods heavily rely on tuned search spaces and constraints, such as architecture skeletons, number of layers, and forced initial and final operations, which were shown to hide the real contributions of the search strategy [40]. From this table, it is possible to see that LCMNAS achieves high performances (low test error) while at the same time being orders of mag-

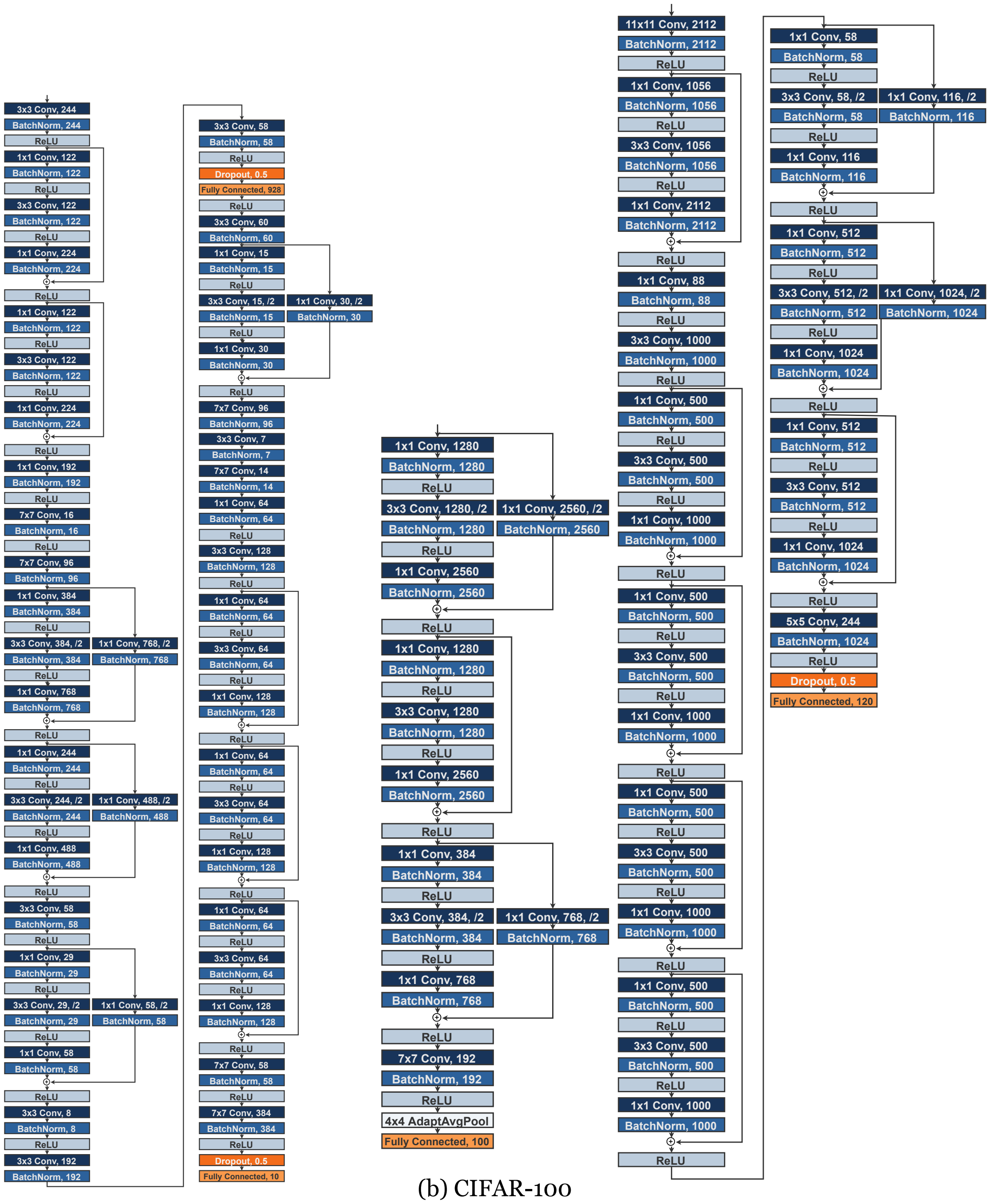
Table 7.3: Test error (%) obtained by transferring the best architectures found in one data set to other data sets without any modifications. Architectures were trained on the new data set from scratch.

Searched On	Transferred To		
	CIFAR-10	CIFAR-100	ImageNet16-120
CIFAR-10	**4.23**	**24.42**	53.45
CIFAR-100	5.09	25.45	55.42
ImageNet16-120	8.15	30.39	**45.78**

nitude faster (search cost) and only using a single 1080Ti GPU (800x lower than NAS [38]). LCMNAS closely relates with large-scale evolution [84] since both apply evolutionary strategies and heavily reduce the amount of human intervention in the decision process. By direct comparison, LCMNAS achieved a lower test error by 2.44% while being orders of magnitude faster (2750x faster) and requiring less computation (250x fewer GPU). By directly comparing with RandGrow and Petridish, the closest in terms of test error, we see that LCMNAS is 5x faster while achieving similar test errors. However, RandGrow [167], and Petridish [168] rely on DropPath, which in RandGrow's case, reduced its' base test error from 3.38% to 2.93%. Regarding CIFAR-100, few proposals directly search on the data set. Instead, they focus on transferring architectures from CIFAR-10. In comparison, Large-scale evolution [84] achieved a test error of 23.7%, which is 2.76% higher than the 20.94% obtained by LCMNAS.

Most NAS methods focus on designing architectures in one data set and transferring it to larger ones. We also study this using the best-generated architectures without AutoAugment. However, contrary to standard practices, we do not change the architecture's scheme when transferring them to emphasize the true contributions of the search strategy. We found that for similar data sets, i.e., CIFAR-10 and CIFAR-100, the generated architecture in the first and transferred to the second outperforms the architecture found directly searching in the second. We justify this due to the similar nature of the data sets, whereas CIFAR-100 is a more demanding data set, meaning that searching on it is harder. However, the same does not apply when transferring CIFAR-10 or CIFAR-100 to ImageNet16-120 and the other way around. The result of such transference falls heavily short when compared to directly searching in the desired data set. These results are shown in Table 7.3. From these, we conclude that directly searching on a data set is better than transferring the generated architectures if no similar and simpler data sets are available that might aid in the search process.

Regarding the architecture's design, a visualization of the best ones can be seen in Figure 7.3. LCMNAS found that placing batch normalization layers after convolutional layers was an important building block, usually followed by a Rectified Linear Unit (ReLU) activation function, meaning regularization layers are essential after a convolutional layer. In CIFAR-10, LCMNAS specifically designed a very deep network with a small number of



(a) CIFAR-10 ($\lambda = 0.75$). (b) CIFAR-100 ($\lambda = 0.50$). (c) ImageNet16-120 ($\lambda = 0.75$).

Figure 7.3: Architecture of the best models found in each data set used to perform macro-search.

output channels, thus controlling the number of parameters. In the middle of this architecture, a dropout layer and a linear layer were added. We hypothesize that this served the purpose of regularization by dropping connections and creating linear representations of the feature maps. In many architectures, a pattern of having convolutional layers with larger kernel sizes (7×7) interspersed with smaller ones (1×1 and 3×3) is seen. Also common to all architectures, LCMNAS tends to add dropout layers and reduction mechanisms, e.g., adaptive average pooling, at the end of the architecture, especially in ImageNet16-120, the noisier data set.

Table 7.4: Test error (%) of the best architectures found on CIFAR-10 using different search epochs e and fixed $\lambda = 0.75$.

Search Epochs e	CIFAR-10	
	Test Error (%) ↓	Params (M) ↓
$e = 1$	6.33	21.4
$e = 2$	4.92	170.1
$e = 3$	4.69	44.1
$e = 4$	4.23	6.7

7.3.5 Ablation Studies

We extend the studies of LCMNAS by evaluating the importance of the epochs used to partially train a generated architecture, using $\lambda = 0.75$ (results shown in Table 7.4). Naturally, reducing e implied an increase in the final test error and the number of parameters of the final architecture. This increase can be justified by the difficulty in measuring the trainability of the architectures with a small number of training epochs. Furthermore, in Table 7.5, we evaluated the impact of the number of generations, g, and population per generation, p, using $\lambda = 1$. The resulting test error shows that $g = 50$ and $p = 100$ yield the best results and that these parameters can influence the final result by a significant amount. In a perfect scenario, where computational costs are not considered, one could use a large e and evaluate more architectures by increasing g and p.

Also, we evaluate the diversity of the generated architectures by performing ensembling [279]. For this, we ensemble the architectures found, using different λ values (Table 7.6) as an indication of the architectures' diversity [280]. Ensemble sizes k, go from the top-1 architecture ($k = 1$) to the top-5 architecture found ($k = 5$). The resulting test error was obtained using weighted majority voting, with the accuracies attained while training used as weights. Results show that incrementally adding lower-performant architectures yields better results for all data sets evaluated than the top-1 architecture alone. This experiment validates the premise that less constrained macro-search has benefits due to its' inherent architecture generation diversity.

We've also studied the behavior of the best-generated architecture by introducing pruning and automatically adding batch normalization layers as post-processing. These experiments allow further evaluation of important components of the generated architectures and provide insights if the sampled layers and associated parameters were good selections. For the first experiment, pruning, we added a train restriction in which at the end of one epoch, all weights, w_i, under a threshold: $|w_i| <= 1e-2$, have their value changed to 0. The results showed that this experiment did not have an effect on the final performance of the networks. For the second experiment, we focused on adding a regularization mechanism in the form of batch normalization layers. To do this, we manually added the batch normalization layers in two ways: i) after all layers that are not activation layers (e.g.,

ReLU); and ii) after all layers. We found that these changes would result in a decrease in final test accuracy by approximately 5%, meaning that the search strategy was already giving the proper importance to regularization mechanisms. More, LCMNAS learned that batch normalization layers are mostly useful after convolutional layers, not adding those after other types of layers.

Table 7.5: Test error (%) in CIFAR-10 using $\lambda = 1$ for different number of generations, g, and population sizes, p.

Parameters		CIFAR-10	
Generations (g)	Population (p)	Test Error (%) ↓	Params (M) ↓
5	5	16.52	**0.3**
10	10	5.46	3.9
10	25	5.25	3.6
10	50	6.97	0.9
10	100	6.83	12.7
15	15	6.08	0.4
15	25	5.58	8.7
15	50	6.16	7.6
15	100	5.37	13.9
25	25	5.49	2
25	50	6.52	3.1
25	100	4.97	8.4
50	50	5.24	4.7
50	100	**4.81**	2.5
100	50	6.69	5.3
100	100	5.94	13.6
100	250	5.54	55.9
150	150	6.19	3.3
150	250	6.46	2.2
250	100	5.27	36.1
250	250	6.14	4.8

Table 7.6: Ensemble test error (%) for CIFAR-10, CIFAR-100 and ImageNet-16-120 with different ensemble sizes k. Architectures used to perform the ensemble are the final ones found by searching with different $\lambda \in \{0, 0.25, 0.5, 0.75, 1\}$.

Ensemble Size (k)	Test Error (%) ↓		
	CIFAR-10	CIFAR-100	ImageNet16-120
$k = 1$	4.23	25.45	45.78
$k = 2$	4.23	25.45	45.78
$k = 3$	3.67	22.64	**44.67**
$k = 4$	**3.51**	22.01	44.85
$k = 5$	3.59	**21.52**	44.72

7.4 Designing Cells: Experiments and Results Analysis

To further allow comparison with the state-of-the-art, we evaluate the effectiveness of LCMNAS in the problem of cell-based NAS. We utilize three different search spaces: NAS-Bench-101 [281], NAS-Bench-201 [42] and TransNAS-Bench-101 [87] benchmarks. These benchmarks were designed to have fixed NAS search spaces with metadata about the training of thousands of architectures within those search spaces. For a detailed description of the benchmarks, we reference the reader to chapter 2.3.

Table 7.7: Search Performance on NAS-Bench-101. Test regret is the difference between the optimal architecture in NAS-Bench-101 and the test accuracy obtained by a NAS method. Table results adapted from [4].

Method	GPU Hours ↓	Test Acc. (%) ↑	Test Regret (%) ↓
LaNAS [282]	107.3	93.90	0.42
BONAS [222]	107.3	94.09	0.23
NASBOWLr [283]	80.5	94.09	0.23
CATE [284]	80.5	94.10	0.22
RS [50]	-	90.38 ± 5.51	3.94
Synflow [285]	-	91.31 ± 0.02	3.01
WeakNAS [223]	80.5	94.10 ± 0.19	0.22
NASWOT [153]	0.01	91.77 ± 0.05	2.55
AREA [153]	3.33	93.91 ± 0.29	0.41
GenNAS-N [155]	5.75	93.92 ± 0.00	0.40
REA [72]	7.42	93.12 ± 0.48	1.12
Evolution + TEG [4]	0.78	92.52 ± 1.30	1.80
REINFORCE [4]	2.77	93.80 ± 0.12	0.52
REINFORCE + TEG [4]	0.24	94.11 ± 0.11	0.21
LCMNAS (Ours)	3.21	**94.19** ± 0.08	**0.13**
Optimal	-	94.32	0.00

7.4.1 Results and Discussion

For LCMNAS to perform micro-search in different search spaces, the search method was slightly modified to accommodate the forced rules that micro-search carries. First, the initial population is randomly sampled from the search space. Secondly, for NAS-Bench-101, the search method samples layers until reaching either $l = 5$ or the layer "Output", whereas for NAS-Bench-201 and TransNAS-Bench-201, the search method samples a specific number of layers $l = 6$. Finally, the search method focuses solely on sampling layers (operations) instead of sampling both layers and associated parameters.

First, we evaluate LCMNAS on NAS-Bench-101 using the evolution settings proposed in [4, 72] to allow a fair comparison with previous evolutionary NAS methods. As shown in Table 7.7, LCMNAS performs better than previous state-of-the-art NAS methods with a 94.19% mean test accuracy achieved by the top scoring architecture found in approximately 3 GPU hours. When compared to REA [72], LCMNAS achieves a 1.07% higher mean

Table 7.8: Comparison of different methods on the NAS-Bench-201 benchmark. The first block shows manually design architectures, the second block shows weight-sharing NAS methods and the third block shows the results for non-weight sharing NAS methods. The results are shown in terms of accuracy (%) with mean and standard deviation for CIFAR-10, CIFAR-100 and ImageNet-16-120. Search times are the mean time required to search for cells in CIFAR-10 and include the time required to train architectures where applicable.

Method	Search Time (s)↓	CIFAR-10		CIFAR-100		ImageNet-16-120	
		Val. Acc (%)↑	Test Acc. (%)↑	Val. Acc (%)↑	Test Acc. (%)↑	Val. Acc (%)↑	Test Acc. (%)↑
Manually designed							
ResNet [59]	-	90.83	93.97	70.42	70.86	44.53	43.63
Weight sharing							
RSPS [1]	7587	84.16±1.69	87.66±1.69	59.00±4.60	58.33±4.34	31.56±3.28	31.14±3.88
DARTS-V1 [2]	10890	39.77±0.00	54.30±0.00	15.03±0.00	15.61±0.00	16.43±0.00	16.32±0.00
DARTS-V2 [2]	29902	39.77±0.00	54.30±0.00	15.03±0.00	15.61±0.00	16.43±0.00	16.32±0.00
GDAS [261]	28926	90.00±0.21	93.51±0.13	71.14±0.27	70.61±0.26	41.70±1.26	41.84±0.90
SETN [221]	31010	82.25±5.17	86.19±4.63	56.86±7.59	56.87±7.77	32.54±3.63	31.90±4.07
ENAS [70]	13315	39.77±0.00	54.30±0.00	15.03±0.00	15.61±0.00	16.43±0.00	16.32±0.00
Non-weight sharing							
RS [220]	12000	90.93±0.36	93.70±0.36	70.93±1.09	71.04±1.07	44.45±1.10	44.57±1.25
RLNAS [238]	-	89.94	93.45	70.98	70.71	43.86	43.70
TE-NAS [156]	1558	-	93.90±0.47	-	71.24±0.56	-	42.38±0.46
REINFORCE [94]	12000	91.09±0.37	93.85±0.37	71.61±1.12	71.71±1.09	45.05±1.02	45.24±1.18
BOHB [137]	12000	90.82±0.53	93.61±0.52	70.74±1.29	70.85±1.28	44.26±1.36	44.42±1.49
REA [72]	12000	91.19±0.31	93.92±0.30	71.81±1.12	71.84±0.99	45.15±0.92	45.54±1.03
LCMNAS (ours)†	11521	**91.22±0.17**	**94.05±0.07**	**71.96±0.96**	**72.01±0.82**	**45.55±0.78**	**45.61±0.08**

† Results shown are of 10 runs using the same settings: $p/g = 10/35$.

test accuracy with $1/2$ the search cost.

To evaluate LCMNAS on NAS-Bench-201, we set $P = 10$ and $G = 35$ to share similar settings with other evolutionary methods [72] and allow a fair comparison with other non-weight sharing NAS methods in terms of search time. The results for searching on all NAS-Bench-201 data sets: CIFAR-10, CIFAR-100, and ImageNet16-120, are presented in Table 7.8. Notably, LCMAS achieved competitive results, outperforming current state-of-the-art cell-based methods [72, 2], while requiring less computation (search time) when compared to both non-weight sharing and weight sharing methods. Compared with GDAS [261], the current best weight sharing method in NAS-Bench-201 in terms of accuracy, LCMNAS outperforms GDAS by 0.54%, 1.4% and 3.77% in CIFAR-10, CIFAR-100 and ImageNet16-120 respectively. The gains in terms of accuracy are more notable in noisier data sets, and ImageNet16-120 is the noisiest, where images are small, with a size of 16×16 pixels, and a training set of 140 thousand images spanning across 120 classes. LCMNAS is also more efficient than GDAS in terms of computation, requiring less than 40% of the computation in terms of search time. When comparing non-weight sharing methods, LCMNAS outperforms REA [72] in all data sets while requiring less computation. Moreover, LCMNAS is highly precise in finding good architectures, shown by small standard deviations in all data sets, especially in ImageNet16-120.

To further assess LCMNAS on a micro-search setting, we also evaluate its performance on TransNas-Bench-101 in all seven tasks. This evaluation validates LCMNAS generability

and transferability across different problems, a problem where NAS methods commonly fail [40, 42, 45]. For this, we conducted two experiments: i) directly searching on each task independently, and ii) performing transfer search. For the latter, we followed standard procedures [87], where first the method searches on jigsaw and uses the final population as initialization for the evolution when searching on the other tasks. The results for both experiments are shown in Table 7.9. In the first block of the table, results are shown for directly searching in each data set independently. In this, RS [220] serves as a baseline to whether NAS methods generate good architectures and learn anything. The results obtained by LCMNAS show that it is not only learning but that it is capable of generating architectures that achieve the overall global best performance in all data sets. The results are also comparable with existing methods, where LCMNAS outperforms existing state-of-the-art results in all data sets, where improvements are significant, corresponding to an improvement of 3.6% in the best cases when compared to Arch-Graph-Single [224].

For the task of transferability between data sets, results are shown in the second block of Table 7.9. Results show that the cell-based search of LCMNAS is transferable between different data sets and achieves state-of-the-art results. By searching on Jigsaw and then transferring to other data sets, LCMNAS was capable of generating cells that achieve the global best result in TransNas-Bench-101. More, LCMNAS outperforms other NAS methods in all data sets, where in some cases, the performance gains when compared to the current state-of-the-art, Arch-Graph [224], are 2.3%. The overall results in TransNas-Bench-101 support LCMNAS in terms of performance, generalibity, and transferability, showing that it can efficiently be used to directly search and the created architectures can be transferred to new and different problems.

The results in all 11 data sets across the three benchmarks used to evaluate LCMNAS cell-based search show that it is efficient in generating neural networks. In terms of performance, LCMNAS generates state-of-the-art architectures in all data sets while requiring less computation (time and GPUs) to perform the search.

7.4.2 Ablation Studies

Since LCMNAS relies on a hybrid performance estimation approach to decide which architectures are worth keeping, we further evaluate its value on cell-based search. For this, we use the NAS-Bench-201 benchmark to evaluate Kendall's Tau correlation τ between the proposed mixed-performance estimation and the final validation accuracy for the first 1000 architectures in each data set using different λ values and partial training epochs e. The results presented in Table 7.10 show the effectiveness of the performance estimation strategy. Notably, when $\lambda = 0$, the performance estimation strategy relies solely on the partial training of architectures. In contrast, when $\lambda = 1$ it relies only on scoring the architectures at initialization stage, which is on par with the τ correlations obtained with only the partial training ($\lambda = 0$) with $e = 4$ in CIFAR-10 and $e = 7$ on the other two. On all data sets, it is clear that combining both components in the performance estimation

Table 7.9: Comparison of different methods on the TransNAS-Bench-101 benchmark. The first block shows the results for NAS methods that performed direct search on each task, while the second block shows the results for NAS methods that searched on jigsaw and transferred the resulting architectures to other tasks. The final row shows the global best result.

	Tasks	Cls. Object	Cls. Scene	Autoencoding	Surf. Normal	Sem. Segment.	Room Layout	Jigsaw
	Metric	*Acc. (%)* ↑	*Acc. (%)* ↑	*SSIM* ↑	*SSIM* ↑	*mIoU* ↑	*L2 loss* ↓	*Acc. (%)* ↑
Direct Search	RS [220]	45.16±0.4	54.41±0.3	55.94±0.8	56.85±0.6	25.21±0.4	61.48±0.8	94.47±0.3
	REA [72]	45.39±0.2	54.62±0.2	56.96±0.1	57.22±0.3	25.52±0.3	61.75±0.8	94.62±0.3
	PPO [96]	45.19±0.3	54.37±0.2	55.83±0.7	56.90±0.6	25.24±0.3	61.38±0.7	94.46±0.3
	DT	42.03±5.0	49.80±8.6	51.20±3.3	55.03±2.7	22.45±3.2	66.98±2.3	88.95±9.1
	BONAS [222]†	45.50	54.56	56.73	57.46	25.32	61.10	94.81
	weakNAS [223]†	45.66	54.72	56.77	57.21	25.90	60.31	94.63
	Arch-Graph-single [224]†	45.48	54.70	56.52	57.53	25.71	61.05	94.66
	LCMNAS (Ours)†	**46.32**	**54.94**	**57.72**	**59.62**	**26.27**	**59.38**	**95.37**
Transfer Search	REA-t [72]	45.51±0.3	54.61±0.2	56.52±0.6	57.20±0.7	25.46±0.4	61.04±1.0	-
	PPO-t [96]	44.81±0.6	54.15±0.5	55.70±1.5	56.60±0.7	24.89±0.5	62.01±1.0	-
	CATCH [225]	45.27±0.5	54.38±0.2	56.13±0.7	56.99±0.6	25.38±0.4	60.70±0.7	-
	BONAS-t [222]†	45.38	54.57	56.18	57.24	25.24	60.93	-
	weakNAS-t [223]†	45.29	54.78	56.90	57.19	25.41	60.70	-
	Arch-Graph-zero [224]†	45.64	54.80	56.61	57.90	25.73	60.21	-
	Arch-Graph [224]†	45.81	54.90	56.58	58.27	25.69	60.08	-
	LCMNAS-t (Ours)†	**46.32**	**54.94**	**57.72**	**59.62**	**26.27**	**59.38**	-
	Global Best	46.32	54.94	57.72	59.62	26.27	59.38	95.37

† Results provided for the best run only.

Table 7.10: Kendall's Tau correlation (τ) across each of NAS-Bench-201 data sets for different number of train epochs (e) and Lambda (λ) values. The results show that on all settings, the combination of both components of the performance estimation strategy leads to a higher correlation with respect to the final validation accuracy of the generated architectures.

Lambda (λ)	Train epochs (e)							
	0	1	2	3	4	5	6	7
CIFAR-10								
0	-	0.298	0.392	0.458	0.530	0.570	0.599	0.631
0.25	-	0.550	0.535	0.572	0.614	0.639	0.639	0.675
0.5	-	0.581	0.567	0.598	0.629	0.645	0.650	0.674
0.75	-	**0.586**	**0.579**	**0.599**	**0.630**	**0.647**	**0.653**	**0.676**
1	**0.578**	-	-	-	-	-	-	-
CIFAR-100								
0	-	0.045	0.254	0.326	0.395	0.435	0.497	0.507
0.25	-	0.524	0.547	0.553	0.561	0.572	0.589	0.591
0.5	-	0.544	0.560	0.565	0.571	0.583	**0.599**	**0.602**
0.75	-	**0.555**	**0.563**	**0.567**	**0.574**	**0.587**	**0.599**	**0.602**
1	**0.558**	-	-	-	-	-	-	-
ImageNet16-120								
0	-	0.221	0.286	0.371	0.389	0.425	0.453	0.492
0.25	-	0.526	0.540	0.552	0.559	0.570	0.576	**0.589**
0.5	-	0.536	0.551	0.558	0.565	0.576	0.579	0.588
0.75	-	**0.545**	**0.555**	**0.562**	**0.566**	**0.578**	**0.580**	**0.589**
1	**0.544**	-	-	-	-	-	-	-

strategy yields the best correlation, where $\lambda = 0.75$ tends to yield the best τ correlation. This means that more importance is given to scoring architectures at initialization stage

than to partial training. Expectedly, increasing the number of epochs e contributes significantly to the final τ correlation, as using only one epoch of partial training does not add valuable information to the scoring. However, going further than $e = 4$ shows a small increment in the τ correlations in all settings except when using only the partial training. The small increments, coupled with the time taken to train the architectures for more epochs, justify that $e = 4$ is a good threshold for the proposed mixed performance estimation mechanism.

The results in Table 7.10 show that the combination of both partial training and scoring architectures at initialization stage is a good indicator of their final performance, thus validating the importance of the performance estimation mechanism.

To further evaluate the importance of the different parameters, we performed direct search on NAS-Bench-201 with different p (population size) and g (number of generations) and reported the final test accuracy (%) for each experiment. Results depicted in Figure 7.4 show that when using $p/g = 30/100$, $p/g = 60/80$ and $p/g = 80/60$, LCMNAS was capable of achieving the best possible result in NAS-Bench-201 - 94.37% test accuracy. These results further justify that LCMNAS can generate efficient architectures. More, higher mean test accuracies than the ones presented in Table 7.2 can be attained by using higher p and g values with a higher search cost budget. In Figure 7.4, we see that increasing the number of individuals in each generation p, which promotes exploration of the search space, leads to better results. However, over $p = 60$, the differences come from increasing g, which encourages exploitation of known settings. The results of using different p and g show that LCMNAS consistently finds excellent architectures with a smaller compute budget than competing approaches (small p and g values).

7.5 Conclusions

This chapter proposes a NAS approach capable of performing unconstrained macro and cell-based search. For this, we designed three novel components for the NAS method. For the search space design, we propose a method that autonomously generates complex search spaces by creating WDGs with hidden properties from existing CNNs. The proposed search strategy can perform both micro and macro-architecture search through evolution without requiring human-defined restrictions, such as outer-skeleton, initial architecture schemes, or heuristics. To quickly evaluate generated architectures, we propose using a mixed-performance strategy that combines information about architectures at initialization stage with their validation accuracy after partial training on a partial data set. Our experiments show that LCMNAS generates state-of-the-art architectures in cell-based search, outperforming current methods in 11 different data sets, and in 3 data sets on macro-based search, where it can generate architectures from scratch with minimal GPU computation, achieving test errors of 2.96% in CIFAR-10, 20.94% in CIFAR-100 and 43.35% in ImageNet16-120. Furthermore, we also study the importance of different

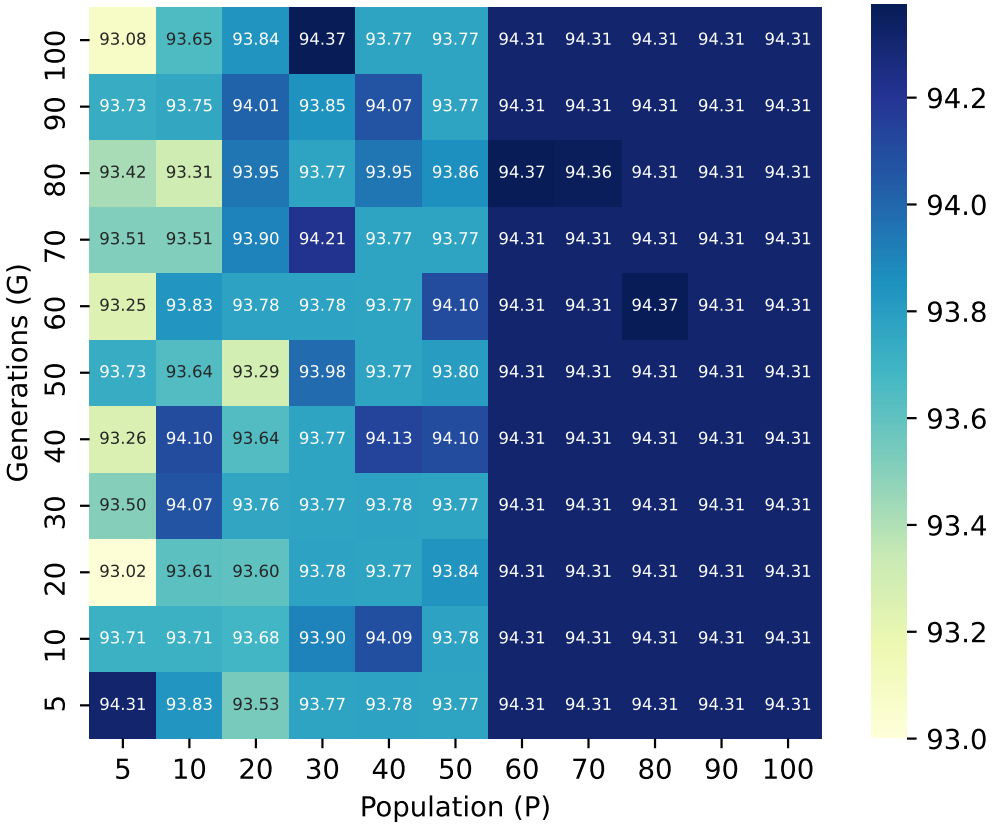


Figure 7.4: Test accuracy (%) obtain in NAS-Bench-201 by LCMNAS with different number of generations, g, and population sizes p.

NAS components and draw insights from architecture design choices.

The methods proposed in this chapter serve the purpose of pushing NAS boundaries to less constrained spaces, where human-expertize for the design of inner-architecture parameters and search spaces is reduced while at the same time generating architectures in a very efficient way, thus allowing a step towards wide-spread use of NAS for different problems and data sets.

The next chapter presents the general conclusions of this thesis and discusses possible future developments.

Chapter 8

Conclusion

8.1 Conclusion

Throughout this thesis, several contributions were made to improve NAS by improving the efficiency of evaluating architectures, by designing methods that automate the design of search spaces, and by improving existing CNNs by designing new components with novel NAS methods. With the obtained results, we pushed the field of NAS to less constrained settings. The following paragraphs present the main conclusions of this work.

The analysis of the NAS state-of-the-art allowed us to identify several problems: i) most prominent NAS methods require substantial computational power; ii) the required time to search for an architecture is considerable; iii) designed architectures can have a high inference latency (time to process an input); iv) search spaces are heavily dependent on human-definitions, and are usually small with fixed operations; v) the search is primarily performed in a cell-based manner, where NAS methods search for small cells that are later replicated in a human-defined outer-skeleton; vi) architecture's parameters, such as the number of layers, inner-layer parameters (e.g., the kernel size, output channels), the final architecture skeleton, fixed operations, and head and tail of the final architectures are usually defined by the authors; vii) very few methods are capable of performing macro-search. More, we conducted a comprehensive study of three popular benchmarks and found that a small subset of the operations were responsible for a generated architecture achieving optimal performance in those search spaces.

Based on the preliminary analysis, we proposed several contributions to mitigate the aforementioned problems. First, we studied different neural networks and proposed a set of methods that augment vanilla neural networks with NAS and AutoML. For this, we introduced novel methods that improve upon individual CNNs with searched components: one by automatically searching for a new classification head and the other by searching for a fusion model to perform multimodal classification. With this, we showed that CNNs can be improved by automatically searching for new components with a very tight search budget.

Then, we focused on improving the search cost of NAS methods by proposing a zero-cost proxy estimator that evaluates untrained architectures based on their Jacobian. By extracting statistics of an untrained architecture in a few seconds, the proposed method can distinguish between good and bad architectures. Based on the excellent results, we

proposed a novel EA that guides the search by evaluating several architectures at initialization stage and only keeping for further evaluation the top-scoring one. Results showed that it is possible to perform an efficient and quick evolutionary search, obtaining very competitive results in multiple problems.

To further improve upon the problem of evaluating architectures, we presented two approaches to further improve NAS evaluation and capabilities. The first is a search space that leverages large vision models as feature extractors. This allows harnessing the power of pre-trained models and focuses the search on a *middleware* architecture that learns how to solve a downstream task. The second is a new zero-cost proxy estimator inspired by the use of NTK and the alignment between eigenvectors to evaluate architectures at initialization stage. These contributions show excellent results and depict a new way of leveraging existing knowledge to further augment NAS.

Aware that most NAS methods only work in restricted cell-based search problems, with memory requirements that preclude the search on the entire super-network, we framed NAS as a MA problem and coupled each edge of a DAG with an independent agent. By doing so, multiple agents collaborate to solve a global optimization problem (generating efficient architectures). Here, the search space can be efficiently distributed by coupling each decision with an agent, where each agent is only responsible for sampling one operation (layer) on a cell-based search problem. This resulted in improvements in memory and speed, which allows direct search on large data sets, including searching for multiple cells at once.

Finally, to push NAS to less constrained search spaces (possibly unbounded), we proposed a NAS approach capable of performing unconstrained macro and cell-based search. For this, we designed three novel components. For the search space design, we proposed a method that autonomously generates complex search spaces by creating WDGs with hidden properties from existing CNNs. The proposed search strategy can perform both micro and macro-architecture search through evolution without requiring human-defined restrictions, such as outer-skeleton, initial architecture schemes, or heuristics. To quickly evaluate generated architectures, we propose using a mixed-performance strategy that combines information about architectures at initialization stage with their validation accuracy after partial training on a partial data set. The results of the extensive experiments show that it is possible to perform macro-search in useful time and obtain state-of-the-art results. This work intends to push NAS boundaries to less constrained spaces, where human-expertize for the design of inner-architecture parameters and search spaces is reduced while at the same time generating architectures in a very efficient way, thus allowing a step towards wide-spread use of NAS for different problems and data sets.

To conclude, we believe that the contributions presented in the thesis contribute to bridging the gap between NAS and widespread use. By proposing NAS methods that are

efficient, that can quickly generate architectures for a given problem, and that heavily reduce the need for human expertise, we believe that this allows for an easier application of NAS in problems where ML experts are not available. More, by publicly releasing all the code with usage instructions and easy-to-adapt performance estimators, this work not only improved the field of NAS but intended to push it towards less constrained scenarios and widespread application.

8.2 Future Directions

Building upon the advancements made during this thesis on NAS, there are several exciting avenues for future exploration and innovation. This work has contributed novel approaches to address key challenges in NAS, including the development of zero-proxy estimators, exploration of new search spaces, automated search space design methods, the introduction of a macro-NAS method for architecture search and design from scratch, as well as benchmark study and CNN augmentation through searching for new classification components. Looking ahead, the following directions hold immense potential for further advancements in NAS:

- Enhancing performance estimators: while this thesis proposes several performance estimation strategies, there is room for further refinement and improvement. Investigating alternative formulations and architectures for zero-proxy estimators could lead to even better performance and more efficient training dynamics. Exploring the application of zero-proxy estimators in different domains or adapting them for specific tasks could also be a fruitful avenue for research, as well as designing NAS-based methods to search for those performance estimation strategies automatically.
- Efficiency and scalability: improving the efficiency and scalability of NAS algorithms remains a crucial area of research. As ML research delves deeper into large models that use enormous data sets, developing methods that can effectively search over larger search spaces, handle diverse architectures, and utilize parallel computing techniques can further contribute to reducing the computational cost of NAS.
- Expanding search space diversity: although significant advances to automatically design new search spaces were made during this thesis, there is still a wide range of possibilities to be explored. Further work on novel architectural components, structural variations, and alternative connectivity patterns can expand the diversity of search spaces. Additionally, developing methods to dynamically adapt and evolve search spaces during the NAS process could lead to more flexible and adaptive architectures. Designing NAS methods that could add or create new operations during the search would be something interesting to pursue, as it would expand even more the search spaces proposed here.
- Automating search space design: expanding on this work, future research could focus on developing more sophisticated algorithms and techniques that can automat-

ically generate diverse and high-quality search spaces. Exploring the integration of domain-specific knowledge, leveraging meta-learning approaches, or incorporating multi-objective optimization principles could further enhance the automated search space design process.

- Joint combination of NAS and hyper-parameter optimization: on chapter 7 we developed a NAS method capable of designing architectures from scratch, including the layer's hyper-parameters. Future work could leverage this and go beyond by allowing the search to control all parameters, including the training protocols and data processing.
- Data processing search: design search algorithms that can optimize the data augmentation and processing policies directly within the search step. This approach would allow data policies to be jointly learned with the model parameters, enabling end-to-end optimization and avoiding the need for separate search procedures.
- Multi-objective macro-based search: in this work, we explored how macro-based search can be conducted and proposed an efficient way of doing so. However, we did not extend this for multi-objective NAS, where NAS methods incorporate multiple conflicting objectives, such as model accuracy, model size, energy efficiency, or inference latency. Developing optimization algorithms that can handle the trade-off between these objectives and generate a diverse set of Pareto-optimal architectures is a very important topic for future work.
- NAS benchmarks: based on the results obtained from studying prominent NAS benchmarks, there is a panoply of future directions. Benchmarks can be designed to include progressive search, as well as multi-objective optimization while considering the addition of new operations and data-aware metrics, where architectures are trained with different training protocols.
- Continual learning and lifelong NAS: NAS methods typically assume a fixed data set during the search process. However, for widespread use of NAS, it is important that NAS methods can quickly adapt to new settings or problems. For this, the scenario of continual learning or lifelong learning, where the model needs to adapt to new tasks or data sets over time, is an interesting direction. Lifelong NAS algorithms could dynamically update their inner setting to accommodate new tasks while leveraging knowledge from previous tasks. There are proposals in the NAS literature that try to solve this with meta-learning.
- Adversarial and interpretable NAS: this topic was out of the scope for this thesis, but is an interesting and important direction. Researchers could study the robustness and vulnerability of NAS algorithms to adversarial attacks. Investigate the impact of adversarial manipulations on the search process and architectural performance. More, the development of mechanisms that allow interpretability of NAS methods is crucial to understand their behavior and learning process. This can lead to finding patterns in the design of architectures that leads the field of CV to new directions.

As a concluding remark, we feel that the objectives of this thesis were achieved and that the contributions of this work allow solving, or at least mitigating, a wide range of NAS problems. Also, this work has contributed to a deeper understanding of NAS capabilities and proposed a set of different methods that push the boundaries of NAS to new domains.

Bibliography

[1] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *Uncertainty in Artificial Intelligence*. PMLR, 2020, pp. 367–377. xxv, 4, 13, 20, 22, 71, 86, 88, 94, 101, 103, 105, 106, 107, 109, 111, 115, 116, 123

[2] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *International Conference on Learning Representations (ICLR)*, 2019. xxv, 2, 4, 11, 14, 18, 19, 20, 42, 45, 65, 71, 84, 85, 86, 87, 88, 89, 91, 93, 94, 95, 102, 103, 104, 105, 115, 123

[3] A. Gaspar and L. A. Alexandre, "A multimodal approach to image sentiment analysis," in *Intelligent Data Engineering and Automated Learning (IDEAL)*, November 2019. xxviii, 47, 60, 62

[4] W. Chen, X. Gong, Y. Wei, H. Shi, Z. Yan, Y. Yang, and Z. Wang, "Understanding and accelerating neural architecture search with training-free and theory-grounded metrics," *CoRR*, 2021. xxix, 21, 83, 85, 86, 87, 89, 122

[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, 2015. 1, 9

[6] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015. 1, 9

[7] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5455–5516, 2020. 1, 9

[8] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022. 1, 9

[9] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary intelligence*, vol. 1, pp. 47–62, 2008. 1, 10, 18

[10] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated Machine Learning*. Springer, 2019. 1, 2, 10

[11] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2015, pp. 2962–2970. 2

[12] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019. 2, 3, 4, 10, 12, 15, 17, 18, 19, 21, 22, 65

[13] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-parameter Optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2011, pp. 2546–2554. 2, 19

[14] J. Bergstra and Y. Bengio, "Random Search for Hyper-parameter Optimization," *Journal of machine learning research*, vol. 13, no. Feb, pp. 281–305, 2012. 2, 86

[15] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2017. 2, 3, 11, 15, 17, 20, 65, 93, 113, 117

[16] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," in *International Conference on Learning Representations (ICLR)*, 2018. 2, 11, 15, 17, 18

[17] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter, "Understanding and Robustifying Differentiable Architecture Search," in *International Conference on Learning Representations (ICLR)*, 2020. 2, 11, 18, 89, 91, 93

[18] C. Peng, Y. Li, R. Shang, and L. Jiao, "Recnas: Resource-constrained neural architecture search based on differentiable annealing and dynamic pruning," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2022. 2, 11

[19] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei, "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 2, 11, 16

[20] C. Liu, P. Dollár, K. He, R. B. Girshick, A. L. Yuille, and S. Xie, "Are labels necessary for neural architecture search?" in *European Conference on Computer Vision (ECCV)*, 2020. 2, 11

[21] Y. Chen, T. Yang, X. Zhang, G. Meng, C. Pan, and J. Sun, "Detnas: Neural architecture search on object detection," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. 2, 11

[22] X. Gong, S. Chang, Y. Jiang, and Z. Wang, "Autogan: Neural architecture search for generative adversarial networks," in *International Conference on Computer Vision (ICCV)*, 2019. 2, 11, 14

[23] C. Gao, Y. Chen, S. Liu, Z. Tan, and S. Yan, "Adversarialnas: Adversarial neural architecture search for gans," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 2, 11

[24] Q. Zhou, B. Zhong, X. Liu, and R. Ji, "Attention-based neural architecture search for person re-identification," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 11, pp. 6627–6639, 2022. 2, 11

[25] G. Wang, L. Lin, R. Chen, G. Wang, and J. Zhang, "Joint learning of neural transfer and architecture adaptation for image recognition," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 10, pp. 5401–5415, 2022. 2, 11

[26] M. Verma, M. S. K. Reddy, Y. R. Meedimale, M. Mandal, and S. K. Vipparthi, "Automer: Spatiotemporal neural architecture search for microexpression recognition," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 11, pp. 6116–6128, 2022. 2, 11

[27] N. Klyuchnikov, I. Trofimov, E. Artemova, M. Salnikov, M. Fedorov, A. Filippov, and E. Burnaev, "Nas-bench-nlp: Neural architecture search benchmark for natural language processing," *IEEE Access*, vol. 10, pp. 45 736–45 747, 2022. 2, 11, 14, 24, 25

[28] Y. Wang, Y. Yang, Y. Chen, J. Bai, C. Zhang, G. Su, X. Kou, Y. Tong, M. Yang, and L. Zhou, "Textnas: A neural architecture search space tailored for text representation," in *AAAI Conference on Artificial Intelligence*, 2020. 2, 11

[29] J. Kim, J. Wang, S. Kim, and Y. Lee, "Evolved speech-transformer: Applying neural architecture search to end-to-end automatic speech recognition." in *INTERSPEECH*, 2020, pp. 1788–1792. 3, 11

[30] A. Mehrotra, A. G. C. Ramos, S. Bhattacharya, Ł. Dudziak, R. Vipperla, T. Chau, M. S. Abdelfattah, S. Ishtiaq, and N. D. Lane, "Nas-bench-asr: Reproducible neural architecture search for speech recognition," in *International Conference on Learning Representations (ICLR)*, 2021. 3, 11, 14, 24, 25

[31] S. Wang, H. Tang, B. Wang, and J. Mo, "A novel approach to detecting muscle fatigue based on semg by using neural architecture search framework," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–12, 2021. 3, 11

[32] Y. Wan, Y. Zhong, A. Ma, J. Wang, and L. Zhang, "E2scnet: Efficient multiobjective evolutionary automatic search for remote sensing image scene classification network architecture," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2022. 3, 11

[33] Q. Gao, Z. Luo, D. Klabjan, and F. Zhang, "Efficient architecture search for continual learning," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–11, 2022. 3, 11

[34] Y. Li, Z. Chen, D. Zha, K. Zhou, H. Jin, H. Chen, and X. Hu, "Automated anomaly detection via curiosity-guided search and self-imitation learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 6, pp. 2365–2377, 2022. 3, 11

[35] B. Lyu, Y. Yang, S. Wen, T. Huang, and K. Li, "Neural architecture search for portrait parsing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 3, pp. 1112–1121, 2023. 3, 11

[36] M. Wistuba, "Transfer neural architecture search," *International Conference on Learning Representations (ICLR) Workshop on Neural Architecture Search*, 2020. 3

[37] M. Wistuba, A. Rawat, and T. Pedapati, "A Survey on Neural Architecture Search," *arXiv preprint arXiv:1905.01392*, 2019. 3

[38] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2018. 3, 11, 13, 15, 17, 20, 65, 84, 89, 91, 93, 103, 104, 113, 118

[39] M. Guo, Z. Zhong, W. Wu, D. Lin, and J. Yan, "Irlas: Inverse reinforcement learning for architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 9021–9029. 4

[40] A. Yang, P. M. Esperança, and F. M. Carlucci, "Nas evaluation is frustratingly hard," in *International Conference on Learning Representations (ICLR)*, 2020. 4, 13, 23, 26, 40, 41, 42, 65, 72, 93, 94, 105, 106, 107, 109, 111, 115, 116, 117, 124

[41] V. Lopes, B. Degardin, and L. A. Alexandre, "Are neural architecture search benchmarks well designed? a deeper look into operation importance," *CoRR*, vol. abs/2303.16938, 2023. 4, 5

[42] X. Dong and Y. Yang, "Nas-bench-201: Extending the scope of reproducible neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020. 4, 23, 24, 70, 71, 72, 84, 85, 109, 114, 115, 122, 124

[43] V. Lopes and L. A. Alexandre, "Towards less constrained macro-neural architecture search," *arXiv preprint arXiv:2203.05508*, 2022. 4, 7, 93

[44] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," *Journal of Machine Learning Research*, vol. 21, no. 243, pp. 1–18, 2020. 4, 22, 23, 26, 40, 41

[45] X. Wan, B. Ru, P. M. Esperança, and Z. Li, "On redundancy and diversity in cell-based neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2022. 4, 13, 14, 23, 26, 40, 41, 72, 88, 93, 106, 109, 111, 115, 124

[46] V. Lopes and L. A. Alexandre, "Auto-classifier: A robust defect detector based on an automl head," in *International Conference on Neural Information Processing (ICONIP)*. Springer, Cham, 2020, pp. 137–149. 6

[47] V. Lopes, A. Gaspar, L. A. Alexandre, and J. Cordeiro, "An automl-based approach to multimodal image sentiment analysis," in *International Joint Conference on Neural Networks (IJCNN)*, 2021. 6

[48] V. Lopes, S. Alirezazadeh, and L. A. Alexandre, "EPE-NAS: efficient performance estimation without training for neural architecture search," in *International Conference on Artificial Neural Networks (ICANN)*, 2021. 6, 89, 114

[49] Lopes, Vasco and Santos, Miguel and Degardin, Bruno and Alexandre, Luís A, "Guided evolution for neural architecture search," in *Advances in Neural Information Processing Systems (NeurIPS) New In ML Workshop*, 2021. 6

[50] ——, "Efficient guided evolution for neural architecture search," in *The Genetic and Evolutionary Computation Conference (GECCO)*, 2022. 6, 19, 122

[51] ——, "Guided evolutionary neural architecture search with efficient performance estimation," *arXiv preprint arXiv:2208.06475*, 2022. 6, 86, 88

[52] Lopes, Vasco and Carlucci, Fabio Maria and Esperança, Pedro M and Singh, Marco and Gabillon, Victor and Yang, Antoine and Xu, Hang and Chen, Zewei and Wang, Jun, "Manas: Multi-agent neural architecture search," *Springer Machine Learning*, 2023. 6, 17

[53] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural computation*, vol. 29, no. 9, pp. 2352–2449, 2017. 9

[54] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai *et al.*, "Recent advances in convolutional neural networks," *Pattern recognition*, vol. 77, pp. 354–377, 2018. 9, 49

[55] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016, vol. 1. 9

[56] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012. 10

[57] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015. 10, 47, 48, 116

[58] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9. 10, 13, 116

[59] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. 10, 13, 47, 48, 49, 71, 88, 115, 116, 117, 123

[60] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. 10, 47, 48, 49, 103, 116

[61] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning (ICML)*, 2019. 10

[62] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. 10, 17

[63] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," in *IEEE Congress on Evolutionary Computation (CEC)*, vol. 4, Dec 2003. 10, 18

[64] S. Whiteson, P. Stone, K. O. Stanley, R. Miikkulainen, and N. Kohl, "Automatic feature selection in neuroevolution," in *The Genetic and Evolutionary Computation Conference (GECCO)*, 2005, pp. 1225–1232. 10, 18

[65] J. Huizinga, J. Clune, and J.-B. Mouret, "Evolving neural networks that are both modular and regular: Hyperneat plus the connection cost technique," in *The Genetic and Evolutionary Computation Conference (GECCO)*, 2014, pp. 697–704. 10, 18

[66] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the nero video game," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 653–668, 2005. 10, 18

[67] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 24–35, 2019. 11

[68] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2016. 11, 17, 20, 91, 117

[69] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 11, 14, 17, 20

[70] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient Neural Architecture Search via Parameters Sharing," in *International Conference on Machine Learning (ICML)*, 2018. 11, 14, 17, 65, 71, 88, 89, 91, 93, 102, 103, 115, 117, 123

[71] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *International Conference on Computer Vision (ICCV)*, 2019. 11, 17

[72] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," in *AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789. 11, 15, 18, 20, 70, 71, 72, 73, 84, 86, 89, 91, 93, 103, 104, 113, 115, 122, 123, 125

[73] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical Representations for Efficient Architecture Search," in *International Conference on Learning Representations (ICLR)*, 2018. 11, 16, 18, 93

[74] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, "A survey on evolutionary neural architecture search," *IEEE Transactions on Neural Networks and Learning Systems*, 2023. 11, 12, 17, 18

[75] X. Xie, Y. Liu, Y. Sun, G. G. Yen, B. Xue, and M. Zhang, "Benchenas: A benchmarking platform for evolutionary neural architecture search," *arXiv:2108.03856*, 2021. 11, 18

[76] P. Ren, Y. Xiao, X. Chang, P. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Computing Surveys*, 2021. 12, 14

[77] S. Lee and B. C. Song, "Fast filter pruning via coarse-to-fine neural architecture search and contrastive knowledge transfer," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–12, 2023. 12

[78] H. Zhao, H. Zeng, X. Qin, Y. Fu, H. Wang, B. Omar, and X. Li, "What and where: Learn to plug adapters via nas for multidomain learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 11, pp. 6532–6544, 2022. 12

[79] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L. Li, L. Fei-Fei, A. L. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *European Conference on Computer Vision (ECCV)*, 2018. 14, 20

[80] J. Dong, A. Cheng, D. Juan, W. Wei, and M. Sun, "Dpp-net: Device-aware progressive search for pareto-optimal neural architectures," in *European Conference on Computer Vision (ECCV)*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds., 2018. 14

[81] D. Zeng, Y. Huang, Q. Bao, J. Zhang, C. Su, and W. Liu, "Neural architecture search for joint human parsing and pose estimation," in *International Conference on Computer Vision (ICCV)*. IEEE, 2021. 14

[82] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search," *CoRR*, vol. abs/1905.01392, 2019. 14

[83] C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter, "Neural architecture search: Insights from 1000 papers," *CoRR*, vol. abs/2301.08727, 2023. 14, 19, 21

[84] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *International Conference on Machine Learning (ICML)*, 2017, pp. 2902–2911. 15, 18, 22, 93, 117, 118

[85] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing, "Neural Architecture Search With Bayesian Optimisation and Optimal Transport," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 2016–2025. 15, 19

[86] X. Dong, L. Liu, K. Musial, and B. Gabrys, "NATS-Bench: Benchmarking nas algorithms for architecture topology and size," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021. 15, 24

[87] Y. Duan, X. Chen, H. Xu, Z. Chen, X. Liang, T. Zhang, and Z. Li, "Transnas-bench-101: Improving transferability and generalizability of cross-task neural architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 15, 24, 25, 70, 73, 122, 124

[88] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing Neural Network Architectures using Reinforcement Learning," in *International Conference on Learning Representations (ICLR)*, 2017. 15

[89] L. Xie and A. L. Yuille, "Genetic CNN," in *International Conference on Computer Vision (ICCV)*, 2017. 15

[90] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," in *International Conference on Learning Representations (ICLR)*, 2019. 15, 22, 84, 117

[91] R. Ru, P. Esperanca, and F. M. Carlucci, "Neural architecture generator optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 16

[92] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, 2019. 16

[93] Y. Jaafra, J. L. Laurent, A. Deruyver, and M. S. Naceur, "Reinforcement learning for neural architecture search: A review," *Image and Vision Computing*, vol. 89, pp. 57–66, 2019. 17

[94] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992. 17, 71, 86, 123

[95] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu, "Graph neural architecture search," in *International joint conference on artificial intelligence*, 2021. 17

[96] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. 17, 73, 125

[97] A.-C. Cheng, C. H. Lin, D.-C. Juan, W. Wei, and M. Sun, "Instanas: Instance-aware neural architecture search," in *AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 3577–3584. 17

[98] S. Xie, H. Zheng, C. Liu, and L. Lin, "Snas: stochastic neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2018. 17, 18, 89, 91, 93, 103, 104

[99] F. M. Johner and J. Wassner, "Efficient evolutionary architecture search for CNN optimization on GTSRB," in *IEEE International Conference On Machine Learning And Applications (ICMLA)*, M. A. Wani, T. M. Khoshgoftaar, D. Wang, H. Wang, and N. Seliya, Eds. IEEE, 2019, pp. 56–61. 18

[100] G. Kyriakides and K. Margaritis, "Evolving graph convolutional networks for neural architecture search," *Neural Computing and Applications*, vol. 34, no. 2, pp. 899–909, 2022. 18

[101] C. Hu, C. Wang, X. Ma, X. Meng, Y. Li, T. Xiao, J. Zhu, and C. Li, "Ranknas: Efficient neural architecture search by pairwise ranking," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2021, pp. 2469–2480. 18

[102] J. Prellberg and O. Kramer, "Lamarckian evolution of convolutional neural networks," in *Parallel Problem Solving from Nature (PPSN)*, 2018. 18

[103] O. Kramer, "Evolution of convolutional highway networks," in *Applications of Evolutionary Computation (EvoApplications)*, K. Sim and P. Kaufmann, Eds., 2018. 18

[104] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Evolving deep convolutional neural networks for image classification," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 394–407, 2019. 18

[105] Y. Sun, B. Xue, M. Zhang, G. G. Yen, and J. Lv, "Automatically designing cnn architectures using the genetic algorithm for image classification," *IEEE transactions on cybernetics*, vol. 50, no. 9, pp. 3840–3854, 2020. 18

[106] D. Song, C. Xu, X. Jia, Y. Chen, C. Xu, and Y. Wang, "Efficient residual dense block search for image super-resolution," in *AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 12 007–12 014. 18

[107] M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, and M. Sjödin, "Deepmaker: A multi-objective optimization framework for deep neural networks in embedded systems," *Microprocessors and Microsystems*, vol. 73, p. 102989, 2020. 18

[108] Z. Lu, I. Whalen, Y. Dhebar, K. Deb, E. D. Goodman, W. Banzhaf, and V. N. Boddeti, "Multiobjective evolutionary design of deep convolutional neural networks for image classification," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 2, pp. 277–291, 2020. 18

[109] Z. Yang, Y. Wang, X. Chen, B. Shi, C. Xu, C. Xu, Q. Tian, and C. Xu, "Cars: Continuous evolution for efficient neural architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1829–1838. 18, 20

[110] Z. Lu, I. Whalen, V. Boddeti, Y. D. Dhebar, K. Deb, E. D. Goodman, and W. Banzhaf, "Nsga-net: Neural architecture search using multi-objective genetic algorithm," in *The Genetic and Evolutionary Computation Conference (GECCO)*, 2019. 18, 117

[111] Z. Lu, K. Deb, E. Goodman, W. Banzhaf, and V. N. Boddeti, "Nsganetv2: Evolutionary multi-objective surrogate-assisted neural architecture search," in *European Conference on Computer Vision (ECCV)*. Springer, 2020. 18

[112] C. Schorn, T. Elsken, S. Vogel, A. Runge, A. Guntoro, and G. Ascheid, "Automated design of error-resilient and hardware-efficient deep neural networks," *Neural Computing and Applications*, vol. 32, pp. 18 327–18 345, 2020. 18, 22

[113] X. Chen, L. Xie, J. Wu, and Q. Tian, "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation," in *International Conference on Computer Vision (ICCV)*, 2019. 18, 89, 91

[114] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong, "Pc-darts: Partial channel connections for memory-efficient architecture search," in *International Conference on Learning Representations (ICLR)*, 2020. 18, 65, 89, 91

[115] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 7816–7827. 18

[116] A. Brock, T. Lim, J. Ritchie, and N. Weston, "SMASH: One-shot model architecture search through hypernetworks," in *International Conference on Learning Representations (ICLR)*, 2018. 19, 117

[117] C. Zhang, M. Ren, and R. Urtasun, "Graph hypernetworks for neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2019. 19

[118] L. Li, M. Khodak, N. Balcan, and A. Talwalkar, "Geometry-aware gradient algorithms for neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2021. 19

[119] R. Garnett, *Bayesian optimization*. Cambridge University Press, 2023. 19

[120] A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter, "Learning curve prediction with bayesian neural networks," in *International Conference on Learning Representations (ICLR)*, 2017. 19, 21

[121] G. Dikov and J. Bayer, "Bayesian learning of neural network architectures," in *International Conference on Artificial Intelligence and Statistics*, 2019. 19

[122] M. Parsa, J. P. Mitchell, C. D. Schuman, R. M. Patton, T. E. Potok, and K. Roy, "Bayesian multi-objective hyperparameter optimization for accurate, fast, and efficient neural network accelerator design," *Frontiers in neuroscience*, vol. 14, p. 667, 2020. 19

[123] C. White, W. Neiswanger, and Y. Savani, "Bananas: Bayesian optimization with neural architectures for neural architecture search," in *AAAI Conference on Artificial Intelligence*, 2019. 19, 21, 103

[124] C. White, S. Nolen, and Y. Savani, "Exploring the loss landscape in neural architecture search," in *Uncertainty in Artificial Intelligence*, 2021, pp. 654–664. 19

[125] H. Shi, R. Pi, H. Xu, Z. Li, J. Kwok, and T. Zhang, "Bridging the gap between sample-based and one-shot neural architecture search with bonas," *Advances in Neural Information Processing Systems*, vol. 33, 2020. 19

[126] T. Den Ottelander, A. Dushatskiy, M. Virgolin, and P. A. Bosman, "Local search is a remarkably strong baseline for neural architecture search," in *Evolutionary Multi-Criterion Optimization (EMO)*. Springer, 2021. 19

[127] H. Jin, Q. Song, and X. Hu, "Auto-keras: Efficient neural architecture search with network morphism," *arXiv preprint arXiv:1806.10282*, vol. 5, 2018. 19, 22

[128] D. Eriksson, P. I.-J. Chuang, S. Daulton, P. Xia, A. Shrivastava, A. Babu, S. Zhao, A. A. Aly, G. Venkatesh, and M. Balandat, "Latency-aware neural architecture search with multi-objective bayesian optimization," in *8th ICML Workshop on Automated Machine Learning (AutoML)*, 2021. 19

[129] L. Ma, J. Cui, and B. Yang, "Deep neural architecture search with deep graph bayesian optimization," in *IEEE/WIC/ACM International Conference on Web Intelligence*, 2019. 19

[130] C. Wei, C. Niu, Y. Tang, Y. Wang, H. Hu, and J. Liang, "Npenas: Neural predictor guided evolution for neural architecture search," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2022. 19, 21, 103

[131] A. Zela, A. Klein, S. Falkner, and F. Hutter, "Towards automated deep learning: Efficient joint neural architecture and hyperparameter search," in *International Conference on Machine Learning (ICML) Workshop on AutoML*, 2018. 19, 20

[132] Y. Shen, Y. Li, J. Zheng, W. Zhang, P. Yao, J. Li, S. Yang, J. Liu, and B. Cui, "Proxybo: Accelerating neural architecture search via bayesian optimization with zero-cost proxies," *CoRR*, vol. abs/2110.10423, 2021. 19

[133] T. Elsken, B. Staffler, J. H. Metzen, and F. Hutter, "Meta-learning of neural architectures for few-shot learning," in *IEEE/CVF conference on Computer Vision and Pattern Recognition*, 2020. 20

[134] Y. Xu, L. Xie, W. Dai, X. Zhang, X. Chen, G.-J. Qi, H. Xiong, and Q. Tian, "Partially-connected neural architecture search for reduced computational redundancy," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 9, pp. 2953–2970, 2021. 20

[135] B. Moser, F. Raue, J. Hees, and A. Dengel, "Less is more: Proxy datasets in nas approaches," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp. 1953–1961. 20

[136] Z. Zhong, Z. Yang, B. Deng, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Blockqnn: Efficient block-wise neural network architecture generation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 7, pp. 2314–2328, 2020. 20

[137] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," in *International Conference on Machine Learning (ICML)*, 2018. 20, 71, 123

[138] R. Ru, C. Lyle, L. Schut, M. Fil, M. van der Wilk, and Y. Gal, "Speedy performance estimation for neural architecture search," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, pp. 4079–4092, 2021. 20

[139] D. Zhou, X. Zhou, W. Zhang, C. C. Loy, S. Yi, X. Zhang, and W. Ouyang, "Econas: Finding proxies for economical neural architecture search," in *IEEE/CVF Conference on computer vision and pattern recognition*, 2020, pp. 11 396–11 404. 20

[140] D. O'Neill, B. Xue, and M. Zhang, "Evolutionary neural architecture search for high-dimensional skip-connection structures on densenet style networks," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 6, pp. 1118–1132, 2021. 20

[141] S. Liu, H. Zhang, and Y. Jin, "A survey on computationally efficient neural architecture search," *Journal of Automation and Intelligence*, vol. 1, no. 1, p. 100002, 2022. 20

[142] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015. 21

[143] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *Artificial intelligence and statistics*. PMLR, 2016, pp. 240–248. 21

[144] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, pp. 185:1–185:52, 2017. 21

[145] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction," in *International Conference on Learning Representations (ICLR)*, 2018. 21

[146] A. Rawal and R. Miikkulainen, "From nodes to networks: Evolving recurrent neural networks," *arXiv preprint arXiv:1803.04439*, 2018. 21

[147] Z. Li, T. Xi, J. Deng, G. Zhang, S. Wen, and R. He, "Gp-nas: Gaussian process based neural architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020. 21

[148] H. Tan, R. Cheng, S. Huang, C. He, C. Qiu, F. Yang, and P. Luo, "Relativenas: Relative neural architecture search via slow-fast learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 1, pp. 475–489, 2023. 21

[149] H. Benmeziane, H. Ouarnoughi, K. El Maghraoui, and S. Niar, "Multi-objective hardware-aware neural architecture search with pareto rank-preserving surrogate models," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 2, pp. 1–21, 2023. 21

[150] Y. Sun, H. Wang, B. Xue, Y. Jin, G. G. Yen, and M. Zhang, "Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 350–364, 2019. 21

[151] A. Krishnakumar, C. White, A. Zela, R. Tu, M. Safari, and F. Hutter, "Nas-bench-suite-zero: Accelerating research on zero cost proxies," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. 21, 22, 25

[152] X. Ning, C. Tang, W. Li, Z. Zhou, S. Liang, H. Yang, and Y. Wang, "Evaluating efficient performance estimators of neural architectures," in *NeurIPS*, vol. 34, 2021. 21, 23

[153] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," in *International Conference on Machine Learning (ICML)*, 2021, pp. 7588–7598. 21, 74, 85, 87, 88, 114, 122

[154] M. S. Abdelfattah, A. Mehrotra, L. Dudziak, and N. D. Lane, "Zero-cost proxies for lightweight nas," in *International Conference on Learning Representations (ICLR)*, 2021. 21

[155] Y. Li, C. Hao, P. Li, J. Xiong, and D. Chen, "Generic neural architecture search via regression," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. 21, 122

[156] W. Chen, X. Gong, and Z. Wang, "Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective," in *International Conference on Learning Representations (ICLR)*, 2021. 21, 71, 86, 87, 88, 89, 91, 123

[157] N. Lee, T. Ajanthan, and P. H. S. Torr, "Snip: single-shot network pruning based on connection sensitivity," in *International Conference on Learning Representations (ICLR)*, 2019. 21

[158] M. Lin, P. Wang, Z. Sun, H. Chen, X. Sun, Q. Qian, H. Li, and R. Jin, "Zen-nas: A zero-shot nas for high-performance deep image recognition," in *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021. 21

[159] M.-T. Wu, H.-I. Lin, and C.-W. Tsai, "A training-free neural architecture search algorithm based on search economics," *IEEE Transactions on Evolutionary Computation*, pp. 1–1, 2023. 21

[160] L. Xiang, L. Dudziak, M. S. Abdelfattah, T. Chau, N. D. Lane, and H. Wen, "Zero-cost proxies meet differentiable architecture search," *CoRR*, vol. abs/2106.06799, 2021. [Online]. Available: https://arxiv.org/abs/2106.06799 21

[161] Y. Akhauri, J. P. Muñoz, N. Jain, and R. Iyer, "EZNAS: evolving zero-cost proxies for neural architecture scoring," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. 21

[162] C. White, A. Zela, R. Ru, Y. Liu, and F. Hutter, "How powerful are performance predictors in neural architecture search?" *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, 2021. 22, 23, 65

[163] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," in *International Conference on Learning Representations (ICLR) Workshop*, 2018. 22

[164] T. Wei, C. Wang, Y. Rui, and C. W. Chen, "Network morphism," in *International Conference on Machine Learning (ICML)*, M. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 564–572. 22

[165] P. Radiuk and H. Kutucu, "Heuristic architecture search using network morphism for chest x-ray classification," in *International Workshop on Intelligent Information Technologies & Systems of Information Security*, ser. CEUR Workshop Proceedings, T. Hovorushchenko, O. Savenko, P. T. Popov, and S. Lysenko, Eds., vol. 2623, 2020, pp. 107–121. 22

[166] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *AAAI Conference on Artificial Intelligence*, 2018. 22, 117

[167] H. Hu, J. Langford, R. Caruana, E. Horvitz, and D. Dey, "Macro neural architecture search revisited," in *Advances in Neural Information Processing Systems (NeurIPS) Workshop on Meta-Learning*, 2018. 22, 117, 118

[168] H. Hu, J. Langford, R. Caruana, S. Mukherjee, E. J. Horvitz, and D. Dey, "Efficient forward architecture search," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019. 22, 117, 118

[169] T. Elsken, B. Staffler, A. Zela, J. H. Metzen, and F. Hutter, "Bag of tricks for neural architecture search," *CoRR*, vol. abs/2107.03719, 2021. 22

[170] K. Yu, C. Sciuto, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the search phase of neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2019. 22, 94, 105, 106, 107

[171] X. Ning, W. Li, Z. Zhou, T. Zhao, Y. Zheng, S. Liang, H. Yang, and Y. Wang, "A surgery of the neural architecture evaluators," *CoRR*, vol. abs/2008.03064, 2020. 23

[172] Y. Mehta, C. White, A. Zela, A. Krishnakumar, G. Zabergja, S. Moradian, M. Safari, K. Yu, and F. Hutter, "NAS-bench-suite: NAS evaluation is (now) surprisingly easy," in *International Conference on Learning Representations (ICLR)*, 2022. 23, 25, 26, 40, 41, 85

[173] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "NAS-Bench-101: Towards Reproducible Neural Architecture Search," in *International Conference on Machine Learning (ICML)*, 2019. 23, 24, 70

[174] I. Radosavovic, J. Johnson, S. Xie, W. Lo, and P. Dollár, "On network design spaces for visual recognition," in *International Conference on Computer Vision (ICCV)*, 2019. 23, 24

[175] A. Zela, J. Siems, and F. Hutter, "Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020. 23, 24

[176] L. Dudziak, T. Chau, M. S. Abdelfattah, R. Lee, H. Kim, and N. D. Lane, "BRP-NAS: prediction-based NAS using gcns," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 24

[177] X. Su, T. Huang, Y. Li, S. You, F. Wang, C. Qian, C. Zhang, and C. Xu, "Prioritized architecture sampling with monto-carlo tree search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 24, 25

[178] S. Yan, C. White, Y. Savani, and F. Hutter, "Nas-bench-x11 and the power of learning curves," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, 2021. 24, 25

[179] C. Li, Z. Yu, Y. Fu, Y. Zhang, Y. Zhao, H. You, Q. Yu, Y. Wang, C. Hao, and Y. Lin, "Hw-nas-bench: Hardware-aware neural architecture search benchmark," in *International Conference on Learning Representations (ICLR)*, 2021. 24

[180] T. C. P. Chau, Ł. Dudziak, H. Wen, N. D. Lane, and M. S. Abdelfattah, "BLOX: Macro neural architecture search benchmark and algorithms," in *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2022. 24, 25

[181] A. Bansal, D. Stoll, M. Janowski, A. Zela, and F. Hutter, "JAHS-bench-201: A foundation for research on joint architecture and hyperparameter search," in *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2022. 24

[182] A. Zela, J. N. Siems, L. Zimmer, J. Lukasik, M. Keuper, and F. Hutter, "Surrogate NAS benchmarks: Going beyond the limited search spaces of tabular NAS benchmarks," in *International Conference on Learning Representations (ICLR)*, 2022. 24, 25

[183] M. Ding, Y. Huo, H. Lu, L. Yang, Z. Wang, Z. Lu, J. Wang, and P. Luo, "Learning versatile neural architectures by propagating network codes," in *International Conference on Learning Representations (ICLR)*, 2022. 24, 25

[184] R. Tu, N. Roberts, M. Khodak, J. Shen, F. Sala, and A. Talwalkar, "NAS-bench-360: Benchmarking neural architecture search on diverse tasks," in *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2022. 25

[185] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. 35

[186] A. Agresti, *Analysis of ordinal categorical data*. John Wiley & Sons, 2010, vol. 656. 38

[187] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1251–1258. 45, 48

[188] M. Soleymani, D. Garcia, B. Jou, B. Schuller, S.-F. Chang, and M. Pantic, "A survey of multimodal sentiment analysis," *Image and Vision Computing*, vol. 65, pp. 3–14, 2017. 47

[189] C. C. Aggarwal and C. Zhai, "A survey of text classification algorithms," in *Mining Text Data*. Springer, 2012, pp. 163–222. 47

[190] B. Scholz-Reiter, D. Weimer, and H. Thamer, "Automated surface inspection of cold-formed micro-parts," *CIRP annals*, vol. 61, no. 1, pp. 531–534, 2012. 47, 59

[191] N. T. Siebel and G. Sommer, "Learning defect classifiers for visual inspection images by neuro-evolution using weakly labelled training data," in *IEEE Congress on Evolutionary Computation (CEC)*, 2008. 47, 59

[192] X. Xie, "A review of recent advances in surface defect detection using texture analysis techniques," *ELCVIA: Electronic Letters on Computer Vision and Image Analysis*, pp. 1–22, 2008. 47

[193] C. J. Hutto and E. Gilbert, "Vader: A parsimonious rule-based model for sentiment analysis of social media text," in *International Conference on Weblogs and Social Media (ICWSM)*, 01 2014. 47, 49

[194] S. Loria, "textblob documentation," Technical report, Tech. Rep., 2018. 47, 49

[195] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. 48, 49

[196] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International Conference on Learning Representations (ICLR)*, 2015. 48, 49, 50

[197] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, vol. 18, no. 5, pp. 602–610, 2005, iJCNN 2005. 48, 49, 51

[198] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990. 48, 49, 50

[199] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *AAAI Conference on Artificial Intelligence*, 2015. 48, 49

[200] Y. Kim, "Convolutional neural networks for sentence classification," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Oct. 2014. 48, 49, 51

[201] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," in *European Chapter of the Association for Computational Linguistics (EACL)*, Apr. 2017. 48, 49

[202] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations (ICLR)*, 2021. 48

[203] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1492–1500. 49, 116

[204] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *British Machine Vision Conference (BMVC)*, R. C. Wilson, E. R. Hancock, and W. A. P. Smith, Eds. BMVA Press, 2016. 49, 116

[205] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," in *European Chapter of the Association for Computational Linguistics (EACL)*, 2017. 49

[206] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *International Conference on Learning Representations (ICLR)*, 2013. 49

[207] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," in *International Conference on Learning Representations (ICLR)*, 2017. 50

[208] H2O.ai, *H2O AutoML*, June 2017, h2O version 3.30.0.1. [Online]. Available: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html 53

[209] P. Gijsbers, E. LeDell, J. Thomas, S. Poirier, B. Bischl, and J. Vanschoren, "An open source automl benchmark," in *International Conference on Machine Learning (ICML) Workshop on Automated Machine Learning*, 2019. 53

[210] L. Vadicamo, F. Carrara, A. Cimino, S. Cresci, F. Dell'Orletta, F. Falchi, and M. Tesconi, "Cross-media learning for image sentiment analysis in the wild," in *IEEE International Conference on Computer Vision Workshops (ICCVW)*, Oct 2017, pp. 308–317. 56, 62

[211] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2013. 56

[212] Q. You, J. Luo, H. Jin, and J. Yang, "Building a large scale dataset for image emotion recognition: The fine print and the benchmark," in *AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016. 56

[213] D. Weimer, B. Scholz-Reiter, and M. Shpitalni, "Design of deep convolutional neural network architectures for automated feature extraction in industrial inspection," *CIRP Annals*, vol. 65, no. 1, pp. 417–420, 2016. 59

[214] F. Timm and E. Barth, "Non-parametric texture defect detection using weibull features," in *Image Processing: Machine Vision Applications IV*, 2011. 59

[215] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015. 59

[216] K. Meena and A. Suruliandi, "Local binary patterns and its variants for face recognition," in *IEEE International Conference on Recent Trends in Information Technology (ICRTIT)*, 2011, pp. 782–786. 61

[217] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34. 65, 89, 93, 103, 104

[218] Y. Shu, W. Wang, and S. Cai, "Understanding architectures learnt by cell-based neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2020. 65

[219] K. Jing, J. Xu, and Z. Zhang, "A neural architecture generator for efficient search space," *Neurocomputing*, vol. 486, pp. 189–199, 2022. 65

[220] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012. 70, 71, 73, 123, 124, 125

[221] X. Dong and Y. Yang, "One-Shot Neural Architecture Search via Self-Evaluated Template Network," in *International Conference on Computer Vision (ICCV)*, 2019. 71, 123

[222] H. Shi, R. Pi, H. Xu, Z. Li, J. T. Kwok, and T. Zhang, "Bridging the gap between sample-based and one-shot neural architecture search with BONAS," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 73, 122, 125

[223] J. Wu, X. Dai, D. Chen, Y. Chen, M. Liu, Y. Yu, Z. Wang, Z. Liu, M. Chen, and L. Yuan, "Stronger NAS with weaker predictors," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. 73, 122, 125

[224] M. Huang, Z. Huang, C. Li, X. Chen, H. Xu, Z. Li, and X. Liang, "Arch-graph: Acyclic architecture relation predictor for task-transferable neural architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 73, 124, 125

[225] X. Chen, Y. Duan, Z. Chen, H. Xu, Z. Chen, X. Liang, T. Zhang, and Z. Li, "CATCH: context-based meta reinforcement learning for transferrable architecture search," in *European Conference on Computer Vision (ECCV)*, 2020. 73, 125

[226] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 78, 116

[227] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 78

[228] E. Imani, W. Hu, and M. White, "Representation alignment in neural networks," *Transactions on Machine Learning Research*, 2022. 81, 83

[229] D. Kopitkov and V. Indelman, "Neural spectrum alignment: Empirical study," in *International Conference on Artificial Neural Networks (ICANN)*. Springer, 2020, pp. 168–179. 81, 83

[230] A. Jacot, F. Gabriel, and C. Hongler, "Neural tangent kernel: Convergence and generalization in neural networks," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018. 82, 83

[231] J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington, "Wide neural networks of any depth evolve as linear models under gradient descent," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019. 82, 83

[232] S. Arora, S. S. Du, W. Hu, Z. Li, R. R. Salakhutdinov, and R. Wang, "On exact computation with an infinitely wide neural net," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019. 82

[233] L. Chizat, E. Oyallon, and F. Bach, "On lazy training in differentiable programming," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019. 83

[234] L. Xiao, J. Pennington, and S. Schoenholz, "Disentangling trainability and generalization in deep neural networks," in *International Conference on Machine Learning (ICML)*, 2020, pp. 10 462–10 472. 83

[235] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *International Conference on Computer Vision (ICCV)*, 2015, pp. 1026–1034. 83

[236] I. Rafegas, M. Vanrell, L. A. Alexandre, and G. Arias, "Understanding trained cnns by indexing neuron selectivity," *Pattern Recognition Letters*, vol. 136, pp. 318–325, 2020. 85

[237] X. Chen, R. Wang, M. Cheng, X. Tang, and C. Hsieh, "Drnas: Dirichlet neural architecture search," in *International Conference on Learning Representations (ICLR)*, 2021. 88, 89

[238] X. Zhang, P. Hou, X. Zhang, and J. Sun, "Neural architecture search with random labels," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 88, 123

[239] P. Ye, B. Li, Y. Li, T. Chen, J. Fan, and W. Ouyang, "β-darts: Beta-decay regularization for differentiable architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 88, 89, 91

[240] P. Hou, Y. Jin, and Y. Chen, "Single-darts: Towards stable architecture search," in *International Conference on Computer Vision (ICCV)*, 2021, pp. 373–382. 88, 89

[241] H. Zhou, M. Yang, J. Wang, and W. Pan, "Bayesnas: A bayesian approach for neural architecture search," in *International Conference on Machine Learning (ICML)*, K. Chaudhuri and R. Salakhutdinov, Eds., 2019. 89

[242] Q. Yao, J. Xu, W. Tu, and Z. Zhu, "Efficient neural architecture search via proximal iterations," in *AAAI Conference on Artificial Intelligence*, 2020. 89, 104

[243] G. Li, G. Qian, I. C. Delgadillo, M. Muller, A. Thabet, and B. Ghanem, "Sgas: Sequential greedy architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1620–1630. 89

[244] X. Chen and C. Hsieh, "Stabilizing differentiable architecture search via perturbation-based regularization," in *International Conference on Machine Learning (ICML)*, 2020. 89, 103

[245] Y. Gu, L. Wang, Y. Liu, Y. Yang, Y. Wu, S. Lu, and M. Cheng, "DOTS: decoupling operation and topology in differentiable architecture search," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 12 311–12 320. 91

[246] X. Chu, X. Wang, B. Zhang, S. Lu, X. Wei, and J. Yan, "DARTS-: robustly stepping out of performance collapse without indicators," in *International Conference on Learning Representations (ICLR)*, 2021. 91, 103

[247] S. Lu, Y. Hu, L. Yang, Z. Sun, J. Mei, Y. Zeng, and X. Li, "DU-DARTS: decreasing the uncertainty of differentiable architecture search," in *British Machine Vision Conference (BMVC)*, 2021, p. 133. 91

[248] Z. Lu, G. Sreekumar, E. Goodman, W. Banzhaf, K. Deb, and V. N. Boddeti, "Neural architecture transfer," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 9, pp. 2971–2989, 2021. 91

[249] K. Maile, E. Lecarpentier, H. Luga, and D. G. Wilson, "On constrained optimization in differentiable neural architecture search," *CoRR*, 2021. 91

[250] G. Bender, H. Liu, B. Chen, G. Chu, S. Cheng, P.-J. Kindermans, and Q. V. Le, "Can weight sharing outperform random architecture search? an investigation with tunas," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 14 323–14 332. 93

[251] R. Ru, P. M. Esperança, and F. M. Carlucci, "Neural architecture generator optimization," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 12 057–12 069, 2020. 93

[252] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "Gambling in a rigged casino: The adversarial multi-armed bandit problem," in *IEEE 36th Annual Foundations of Computer Science*, 1995. 94, 99

[253] D. Bouneffouf, I. Rish, and C. Aggarwal, "Survey on applications of multi-armed and contextual bandits," in *Congress on Evolutionary Computation (CEC)*, 2020. 94

[254] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *International Conference on World Wide Web*, 2010, pp. 661–670. 94

[255] Y. Abbasi-Yadkori, P. Bartlett, V. Gabillon, A. Malek, and M. Valko, "Best of both worlds: Stochastic & adversarial best-arm identification," in *Conference on Learning Theory (COLT)*, 2018. 100

[256] P. Auer, N. Cesa–Bianchi, Y. Freund, and R. E. Schapire, "The nonstochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 48–77, 2002. 101

[257] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009. 102, 114

[258] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009. 102

[259] L.-J. Li and L. Fei-Fei, "What, where and who? classifying events by scene and object recognition," in *International Conference on Computer Vision (ICCV)*, 2007, pp. 1–8. 103

[260] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," *Computer Vision and Image Understanding*, vol. 106, no. 1, pp. 59–70, 2007. 103

[261] X. Dong and Y. Yang, "Searching for a robust neural architecture in four GPU hours," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 103, 104, 123

[262] B. Wang, B. Xue, and M. Zhang, "Surrogate-assisted particle swarm optimization for evolving variable-length transferable blocks for image classification," *IEEE Transactions on Neural Networks and Learning Systems*, 2021. 103, 104

[263] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation strategies from data," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 104, 117

[264] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 6848–6856. 104

[265] G. Larsson, M. Maire, and G. Shakhnarovich, "Fractalnet: Ultra-deep neural networks without residuals," in *International Conference on Learning Representations (ICLR)*, 2017. 115

[266] M. Lindauer and F. Hutter, "Best practices for scientific research on neural architecture search," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 9820–9837, 2020. 115, 116

[267] I. Loshchilov and F. Hutter, "SGDR: stochastic gradient descent with warm restarts," in *International Conference on Learning Representations (ICLR)*, 2017. 115

[268] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, "On the importance of initialization and momentum in deep learning," in *International Conference on Machine Learning (ICML)*, 2013. 115

[269] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017. 115

[270] H. Wang, R. Yang, D. Huang, and Y. Wang, "idarts: Improving darts by node normalization and decorrelation discretization," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 4, pp. 1945–1957, 2023. 115

[271] "Torchvision," https://pytorch.org/vision/0.8/models.html, 2021. 116

[272] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1097–1105. 116

[273] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 116

[274] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet v2: Practical guidelines for efficient cnn architecture design," in *European Conference on Computer Vision (ECCV)*, 2018, pp. 116–131. 116

[275] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016. 116

[276] T. Veniat and L. Denoyer, "Learning time/memory-efficient deep architectures with budgeted super networks," in *CVPR*, 2018. 117

[277] S. Saxena and J. Verbeek, "Convolutional neural fabrics," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2016. 117

[278] J. Perez-Rua, M. Baccouche, and S. Pateux, "Efficient progressive neural architecture search," in *British Machine Vision Conference (BMVC)*, 2018. 117

[279] S. Zaidi, A. Zela, T. Elsken, C. C. Holmes, F. Hutter, and Y. W. Teh, "Neural ensemble search for uncertainty estimation and dataset shift," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. 120

[280] S. Zaidi, A. Zela, T. Elsken, C. Holmes, F. Hutter, and Y. W. Teh, "Neural Ensemble Search for Performant and Calibrated Predictions," *arXiv:2006.08573*, 2020. 120

[281] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, "NAS-Bench-101: Towards Reproducible Neural Architecture Search," in *International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 7105–7114. 122

[282] L. Wang, S. Xie, T. Li, R. Fonseca, and Y. Tian, "Sample-efficient neural architecture search by learning actions for monte carlo tree search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021. 122

[283] B. Ru, X. Wan, X. Dong, and M. Osborne, "Interpretable neural architecture search via bayesian optimisation with weisfeiler-lehman kernels," in *International Conference on Learning Representations (ICLR)*, 2021. 122

[284] S. Yan, K. Song, F. Liu, and M. Zhang, "Cate: Computation-aware neural architecture encoding with transformers," *arXiv preprint arXiv:2102.07108*, 2021. 122

[285] H. Tanaka, D. Kunin, D. L. K. Yamins, and S. Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. 122