Faculdade de Engenharia da Universidade do Porto

Empirical Evaluation of Random Wired Neural Networks

João Mendes

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Luís Teixeira

Co-Supervisor: Cláudio Sá

Co-Supervisor: Žiga Emeršič

July 17, 2020

Empirical Evaluation of Random Wired Neural Networks

João Mendes

Mestrado Integrado em Engenharia Informática e Computação

July 17, 2020

Abstract

Artificial Neural Networks (ANN) are powerful and flexible models that perform learning tasks by considering examples, with a human-designed structure. However, architecture engineering of neural network models is one of the most time-consuming tasks. It involves trial and error which does not identify why a solution works and there are many hyperparameters to be adjusted.

Established networks like ResNet and DenseNet are in large part effective due to how they are wired, a crucial aspect for building machine learning models. For this reason, there have been some attempts to automate both the design and the wiring process. In this work, we study approaches for network generators, algorithms that bypass the human intervention in neural network design.

From this starting point, we are looking for alternative architectures which in practice would be more difficult to design by humans using randomly wired neural networks. We conduct an empirical study to evaluate and compare the results of different architectures. With our findings, while analyzing the network behavior during training, we hope to contribute to future work on this type of neural network.

Our implementation of randomly wired neural networks achieves state of the art results, confirming their potential. Our empirical study involves a grid search for the best parameters in the random graph models that are the foundation of these networks. The graph models generate random graphs that define the wiring of the network in three of its stages. We also perform a search for the best number of nodes present in the mentioned graphs. We find that a low number of nodes is enough to achieve similar performance to the state of the art results in our datasets. The range of parameters for the graph models is more extensive compared to the original study. The original results are replicable but not reproducible in the sense that if the original experiments were conducted in the same conditions, we suspect that the *stochastic* nature of the *network generator* would yield distinct results.

Our optimization approaches lead us to conclude that the wiring of a network is important to allow more operations but the weights in the connections are nearly irrelevant. Avoiding using these weights, by *freezing* them, saves some training time while still reaching a good performance.

The final thoughts of this work conclude that the *network generator* is one to be explored under the right resources and can contribute to the improvement of the field of NAS.

Keywords: Machine learning, Neural networks, Randomized search

Resumo

As Artificial Neural Networks (ANN) são modelos poderosos e flexíveis que executam tarefas de aprendizagem considerando exemplos, com uma estrutura definida por humanos. No entanto, *architecture engineering* de modelos de redes neuronais é uma das tarefas mais complexas. Envolve tentativa e erro, o que não permite identificar porque uma solução funciona e contém diversos hiperparâmetros sujeitos a ajuste.

Redes já estabelecidas como ResNet e DenseNet são em grande parte eficazes devido à forma como são conectadas, um aspecto crucial para a construção de modelos de *machine learning*. Por esse motivo, houve algumas tentativas para automatizar o *design* e o processo de ligação. Neste trabalho, estudamos abordagens para geradores de redes, algoritmos que ignoram a intervenção humana no *design* de redes neuronais.

Deste ponto de partida, procuramos arquiteturas alternativas que, na prática, seriam mais difíceis de obter por seres humanos, usando redes neuronais ligadas aleatoriamente. Realizamos um estudo empírico para avaliar e comparar os resultados de diferentes arquiteturas. Com as nossas descobertas, ao analisar o comportamento da rede durante o treino, esperamos contribuir para o trabalho futuro neste tipo de rede neuronal.

A nossa implementação de redes neuronais *randomly wired* alcança resultados dentro do estado da arte, confirmando o seu potencial. O nosso estudo empírico envolve uma *grid search* pelos melhores parâmetros nos modelos de grafos aleatórios que são a base dessas redes. Os algoritmos para modelar grafos geram grafos aleatórios que definem as ligações da rede em três dos seus estágios. Também realizamos uma busca pelo melhor número de nós presentes nos grafos mencionados. Concluímos que um número baixo de nós é suficiente para obter desempenho semelhante aos resultados de última geração nos nossos conjuntos de dados. A gama de parâmetros para os modelos de grafos aleatórios é mais extensa em comparação com o estudo original. Os resultados originais são replicáveis, mas não reproduzíveis, no sentido de que, se as experiências originais fossem conduzidas nas mesmas condições, suspeitamos que a natureza *estocástica* do *gerador de redes* produziria resultados distintos.

As nossas abordagens de otimização levam-nos a concluir que o *wiring* de uma rede é importante para permitir mais operações, mas os pesos nessas conexões são quase irrelevantes. Evitar o uso desses pesos, ao *congelá-los*, economiza algum de tempo de treino e, ao mesmo tempo, atinge um bom desempenho.

As considerações finais deste trabalho concluem que o *gerador de redes* deve ser explorado com os recursos certos e pode contribuir para melhorias no campo de Neural Architecture Search.

Acknowledgements

Thank you to the University of Ljubljana, especially the Faculty of Computer Science for hosting me for a full academic year, providing resources that made this work possible.

For my parents, for all the support.
For my sister, for all the (maybe constructive) criticism.

For my graduating friends back home, for all the memories.
For my Erasmus friends, for all the adventures.

For my supervisors, for all the patience.

To Edgar and Marie, in the back of the SAAB.

Thank you all.

Aos Finalistas!

João Mendes

*"Find what you love
and let it kill you."*

Charles Bukowski

Contents

List of Figures

List of Tables

Abbreviations

DL	Deep Learning
ML	Machine Learning
NN	Neural Network
NAS	Neural Architecture Search
ANN	Artificial Neural Networks
RNN	Recurrent Neural Network
CNN	Convolutional Neural Network
DNN	Deep Neural Network
WS	Watts-Strogatz random graph model
ER	Erdős and Rényi random graph model
BA	Barabási-Albert random graph model
CPU	Central Processing Unit
GPU	Graphic Processing Unit
AutoML	Automated Machine Learning
AI	Artificial Intelligence
MLP	Multi-layer Perceptrons
SGD	Stochastic Gradient Descent algorithm
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
TCN	Temporal Convolutional Networks
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
DAG	Directed Acyclic Graph
FLOPs	FLoating point OPerations

Chapter 1

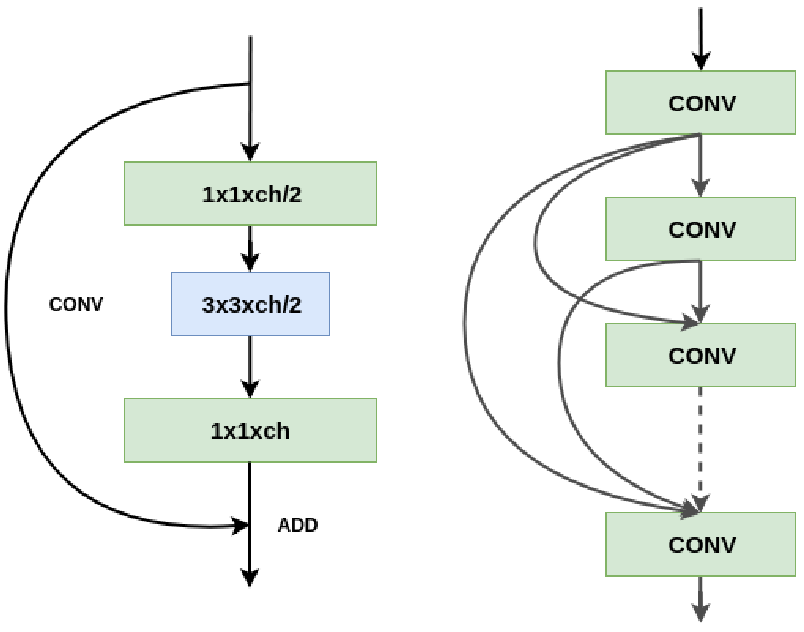
Introduction

Machine Learning evolved from pattern recognition and the idea that computers could learn without explicit programming [76]. Like with humans, or computers with machine learning, learning comes from experience in a certain task. This experience stems from analyzing large amounts of data and the quality of the data is important for the success of the machine learning model. Mathematical models are built by machine learning algorithms from training data. The model should be able to generalize from experience and make successful predictions on new examples. Several types of algorithms have been developed for machine learning and one of the most popular are Artificial Neural Networks.

Artificial Neural Networks are inspired by biological neural networks [68], having simple interconnected processing nodes. Nodes are usually organized into layers that are responsible for operating at a specific depth. The input layer receives data and the output layer produces an output given the information passed through the network. The hidden layers is usually where most of the learning happens and our knowledge of the behavior is weaker. Each node has a set of weights and biases for their inputs and computes an output using an activation function.

Deep learning, also known as deep neural learning, is too a subset of machine learning algorithms that uses a connectionist approach to develop networks [86]. Higher-level features are progressively extracted from a dataset using multiple layers. The neural network phenomena success has led to a focus transfer from feature engineering to a more abstract level, architecture engineering. In other words, although feature engineering has become simpler through deep learning, machine learning models architecture has become consequently more complex. Some problems may require such a specific architecture that it becomes a task similar to past feature engineering obstacles.

Current methods still significantly involve human effort and are short of achieving *AutoML*. The goal is not to automatize ML but to show the potential in *AutoML* by bringing us a step closer in its direction. Researchers are trying to automate the process of applying machine learning to real-world problems, aiming for simpler solutions and faster creation of models. Neural networks are built using building blocks that are usually small and manually designed, plus a set of constraints is applied to their architecture. We are a long way from having self-designed, adaptable networks that cover the complete pipeline. Google provides a service, “Cloud AutoML” that aims to fulfill these needs [29] [74], from the initial dataset to a fully working machine learning model. A lot of work is being done towards that goal and we hope that this research contributes to the field.



(a) ResNet [28] architecture [17] (b) DenseNet [34] architecture [17]

Figure 1.1: Comparison between building blocks of ResNet and DenseNet.

Established networks, like ResNet [28] and also DenseNet [34] – compared in Figure 1.1, perform well because of their innovative connections. ResNet introduced skip connections to prevent the degradation of the accuracy in deeper layers of the network. The method stacks additional layers as residual blocks which are identity mappings, implying no extra parameters to train. This process benefits the deeper layers to perform more similarly to the shallower ones. In DenseNet, layers are connected to all their preceding layers. The final classifier will then have information from all feature-maps because the information was preserved through these extra connections. Thus, how the computational networks are wired is crucial for their performance. Just like the human brain, where wiring and connectivity are important not only to prevent malfunction but to achieve peak performance [30], we can have an immeasurable amount of wiring combinations.

1.1 Context and Motivation

Neural Architecture Search is a technique for automating the design of neural networks. One of the first proposed algorithms [90], has a set of building blocks as a starting point and uses an Artificial Neural Network(ANN) to assemble them and searches for the best neural network architecture for the problem at hand. The network is subject to training and testing, and, based on the results, the blocks can be adjusted to obtain a better performing network.

The search space of NAS algorithms is too extensive to be explored manually [83], even though researchers have discovered some successful designs. The idea is to evaluate the potential of randomly wired neural networks to traverse the search space without human intervention to design the architectures. One of the motivations is that researchers have found that state-of-the-art NAS algorithms perform similarly to a random architecture selection [69] where it would be expected that an effective search policy significantly outperforms a random one.

The authors in [86] proposed future work in this area to explore new network generator designs that may yield new, powerful network designs. These network generators are inspired in classical random graph models [5] from random graph theory, used to reduced bias. They are based in different probabilistic distributions which could have predictable outcomes to some degree. The use of other tools to generate random graphs is an option to do comparisons with the currently used models.

This work extends the previous work in [86] through empirically testing their proposed approach and derive conclusions that could lead to the expansion of the NAS search space.

1.2 Objectives

The goal is to implement network generators by means of a random policy and use them to generate several architectures. Collecting these architectures, we evaluate the performance, compare the results with state-of-the-art solutions, and draw conclusions. From this empirical study, we attempt to gather useful insights that might support new solutions for the NAS. We use machine learning algorithms to predict part of the initial weights of the randomly wired networks before training to boost training in duration or an earlier convergence.

1.3 Document Structure

Firstly, the document exposes the literature related to this study in Chapter 2, where we go through definitions and related work. In Chapter 3, we describe our methods and different approaches to study randomly wired neural networks. In Chapter 4 we present our results with graphical aid and justify our thought process in the iterative experimental procedure. Finally, in Chapter 5 we submit our conclusions from our study and give some final thoughts about the future in the research.

Chapter 2

State of the Art

In this chapter, we start by giving a short introduction to how the field of Machine Learning evolved through the years. Then we present some concepts to contextualize neural networks and how they came to be. We focus on Randomly Wired Neural Networks in Section 2.3, providing an in-depth description of how they are built. Furthermore, we review the state of the art of Neural Network algorithms.

Neural architecture search (NAS) is, arguably, the next big challenge in the field of deep learning [76]. With NAS, one can use a model to look for better architectures given a specific type of data, rather than being limited to trial and error search. NAS algorithms can become even more powerful when we look for alternative architectures which in practice would be more difficult to design by humans. This has the potential for discovering truly novel architectures. In this section, we present these concepts and the state-of-the-art of the field.

2.1 Introduction

Machine Learning (ML) has contributed to the success in the ongoing digital change across industries, like transportation, healthcare, finance, agriculture, and retail. This favorable outcome is long overdue because machine learning is not new [52] since machine learning exists for over 70 years [40]. The explosion of computing power and hardware enhancement has allowed pursuing ideas that have been around for decades but were not feasible before [3]. One of the ideas was to make a machine learn: it performs a task by studying a set of examples – a training set. Then the same task is to be performed using unseen data. This comes hand in hand with the Artificial Intelligence (AI) definition where a machine can perceive its environment and takes action to achieve its goals [63]. Other examples of important machine learning findings, involve activation functions (*Leaky ReLU* [87]) and dropout, where some neurons are chosen at random to be *ignored* during training.

Since the early days of AI that problems have become ever complex, causing smaller subsets of machine intelligence to appear. More recently, one of these subsets of ML emerged and is establishing itself in the field for solving a multitude of computer science problems, known as Deep Learning (DL). It impacts areas including computer vision, speech recognition, robotics, networking, and gaming. DL is a branch of Machine Learning using algorithms that aim to represent high-level data abstractions by constructing – complex – computational models [19]. Deep neural networks are one of the most familiar deep learning structures, characterized by having multiple levels of non-linear mathematical operations [6].

Similarly to ML, deep learning can be categorized in unsupervised, supervised, or partially supervised approaches [3]. The first, unlike supervised learning, does not make use of labeled data and relies purely on its internal representation to discover patterns within the input data. As the name suggests, partially unsupervised approaches rely on partially labeled data. Deep Reinforcement Learning is a more recent development that successfully combines the Reinforcement Learning Paradigm of ML and artificial neural networks [57]. Supervised and Unsupervised Learning includes the use of Convolutional Neural Networks (CNN), Deep Neural Networks (DNN), and Recurrent Neural Networks (RNN), which are the most common types. Reinforcement Learning uses goal-oriented algorithms, operating in a delayed return environment, and allows us to solve complex decision-making tasks [20].

2.2 Neural Networks

Neural networks are a powerful and important technique in Machine Learning. In the early '40s, using simple logic gate operations, it was proven that the combination of neurons can construct a Turing machine [54]. In the late '50s, Rosenblatt [65] shows that if the learning data can be represented, perceptrons will be able to converge. Here the neural networks appeared and since then they have been around for decades. This was the earliest neural network and it was used for binary classification. Because of this, there was a tremendous amount of investment and development in neural networks for decades.

There was a common belief that these neural networks are mimicking some of the functionality in biological systems and animals, like humans. Both the neuroscience and computer science communities had an interest in what were the capabilities of these networks. Limitations in the data (size and labeling) and the computational power at the time, caused a delay in research in the late 70s and 80s in what is known as the "AI winter" [53]. Besides, in '69, Minsky [55] exposes the limitations of the neural network's foundation, the perceptron, and the research is frozen for almost a decade.

There were several developments in the '80s, the backpropagation algorithm was proposed by Hinton et al [1] which re-energizes the field. A couple of years later, the Neocognitron [21] hierarchical neural network emerges and is capable of recognizing visual patterns. In 1999, backpropagation is coupled with Convolutional Neural Networks and applied to image-based document analysis [45]. In 2006, the training problem for Deep Neural Networks is solved by the Hinton Lab [31] using a greedy algorithm.

There was an overhyping of capabilities and the actual architectures could not deliver because of limitations of training time. These fell out of fashion until 2012, with the ImageNet [67] challenge. Here, deep neural networks were trained on vast datasets in new computational architectures, GPUs (graphics processing units). The challenge evaluates algorithms for image classification and object detection at large scale, thus requiring complex solutions. This culmination of big data, big label datasets, and much more powerful computers that could facilitate the training of deep neural network architectures was the reviving point for neural networks. In 2012 the community was surprised by how significant the performance improvement in neural networks had become. AlexNet [44] reduced the error rate by roughly half (from 26.2% to 15.3%) of the previous state-of-the-art in this image classification challenge using Convolutional Neural Networks. Since then, in a very short period, neural networks have risen to the top of the performance metrics [28] [34] in many challenges both in speech recognition, natural language processing, machine translation, and image classification. Now they are setting the standard for what is possible in these and other applications where you have huge data sets and you want to do arbitrary function approximation or interpolation based on your training data set.

Neural networks are inspired in principle by biological systems [68]. Human motion and decision making are controlled by its brain, with a network of neurons that are performing incredibly complex tasks. There is a lot of research in neuroscience and computer science fields to see what kind of connections and comparisons we can make. What can we learn from biological systems that will help our neural networks? What can we learn when we build and design these neural networks that can teach us about these biological systems? This is inspired by visual systems [21], in particular this kind of multi-layered information processing architectures where you take very high-resolution information and pass it through layers of abstraction [49]. Humans can easily detect objects in motion and abstract what a certain collection of pixels represents, based on our previous experiences and abstractions. That is the kind of process we want our neural network to be able to do: take a complex input data and distill out through layers and processing different abstractions that we can use to build models.

2.2.1 What they are

Neural networks are at the lowest level consisted of neurons. These units take two or more inputs, perform mathematical operations using them – typically a weighted average – and produce one output. Each input is affected by its weight and the result of the sum is tuned with a *bias*. The final result is then used as an input to an activation function which normalizes it to a certain range. The *sigmoid* function is commonly used:

which defines the output to be in the range $[0,1]$. This process is known as *feedforward*, where the data flows in only one direction. This basic unit is called perceptron [55], a mathematical model of the biological neuron. A neural network is a group of connected perceptrons. There are an input and an output layer, and at least one hidden layer. Shallow neural networks have

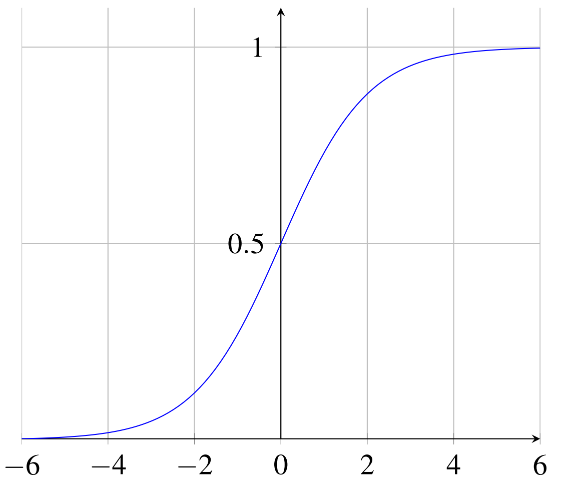


Figure 2.1: *sigmoid* function that is widely used in classification problems. It outputs values between 0 and 1 and is nonlinear, so we can combine it with other *sigmoid* functions and stack layers.

one hidden layer as opposed to Deep Neural Networks (DNN), which have several and often of different types. Multi-layer Perceptrons (MLPs) are a basic type of feedforward neural networks and are used for simple tasks. MLP can have several layers, whilst in DNN we assume there is a considerable number of layers and perceptrons.

Neural Networks use the gradient descent method to find an objective function's local minima. Having a long training time has a drawback, we use optimizers to train these networks, for example, the Stochastic Gradient Descent algorithm (SGD) [7]. Using backpropagation [66] the neural network knows how every node is responsible for the total loss and takes action updating weights accordingly [3]. The step size chosen during training is known as learning rate, it affects how much the weights are updated.

The human visual processing system is more similar to Convolutional Neural Networks (CNNs) than DNNs. They are mostly used for image/video tasks and use filters and pooling to extract features from the raw data. They make use of the convolutional layer, where filters are applied to an image in several windows, the sliding through the image is known as convolution. The pooling layer – or down-sampling – is a layer in which the dimension of the feature maps is reduced, having less computational demand and parameters. The final layer is the output layer, wherein the case of classification, the score of each class is computed.

Some popular state-of-the-art CNN architectures include:

- AlexNet [44], a large, deep CNN introducing the dropping out of nodes to reduce overfitting;
- VGGNet [70] asserts the depth of a network as a critical aspect to achieve better performance in CNNs;
- GoogLeNet [72] presented a less computational complex model with the concept of "Inception Layers" that within the same layer deploy in parallel multiple convolutions with

multiple pooling and filters layers simultaneously;

- DenseNet [34] was named because of the dense connectivity between layers. The outputs of each layer are connected with all successor layers;
- ResNet [28] is a traditional feedforward network with emphasis on the depth that uses residual functions concerning the layer inputs for learning, these blocks allow the flow of information from initial layers to last layers;
- MobileNets [33] are lightweight deep neural networks, using hyperparameters to work on different mobile phones while maximizing accuracy;
- EfficientNet [73] that creates a *compound scaling* hyperparameter affecting three dimensions of a network: width, depth and resolution;
- Xception [11] uses depth-wise separable convolution layers, while being based on GoogLeNet (also known as part of the *Inception* family of architectures);
- ResNeXt-50 [85] is related to ResNets and explores the concept of “cardinality” which describes the repetition of a building block with the aggregation of transformations of the same topology.

Human thinking has implicit previous knowledge drawn from the past. Traditional neural networks are not prepared to handle historical information. Both their input and output are fixed-size vectors and have their computational steps also fixed (number of layers). In Recurrent Neural Networks (RNNs), a loop passes the information from one step of the network to the next [37]. To compute an output, they use processed information from previous time steps. They are mostly used for sequential data, allowing input of any length. Plain RNNs suffer from the vanishing gradient problem, where for very big sequences the gradient is too small to update the weights [60]. Different solutions have been proposed to solve this issue. The following examples are two of the most commonly used:

- Long Short-Term Memory (LSTM) [23] is based on a cell that remembers values over different time intervals, using three gates to regulate information flow (input, output, forget gates);
- Gated Recurrent Unit (GRU) [13] is similar to LSTM with fewer parameters and less computationally expensive, using a simple *update* gate that combines the input and forget gates.

These are layers that can be combined with convolutional layers. Video analysis and text processing are some of the applications.

More recently, Temporal Convolutional Networks [4] (TCNs) have achieved good performances in sequence modeling tasks. TCNs take a sequence of a given length and produce an output of the same length whilst in RNNs it can also be larger or smaller. Even though they use

1D convolutions, these are causal, which implies that the information from the future to the past is not lost. These convolutions use elements from a moment in time *m* and earlier in the previous layer to operate. Because the filters are shared across a layer, parallelism is possible, and the required memory for training is reduced. More recently, the Transformers [79] were shown to be superior in quality, more parallelizable because they can process unordered sequential data and require less time to train.

2.2.2 How they work

The learning part of the process is the training of the neural network. In this research, mostly supervised learning methods with fully labeled data will be used so we focus on processes involving those techniques. Firstly, the network is fed with training data, that traverses through the layers until its final prediction is calculated.

2.2.2.1 Layers

All the neurons, or at least part of them, affect their respective inputs from the previous layer and pass them to the next layer. At the end of this process, the final layer will output a prediction for the input example.

Batch normalization [36] is a technique that for each mini-batch standardizes the inputs to a layer. The reasons for the effectiveness to improve stability, speed, and performance of ANN's remain under discussion but the effect is evident.

2.2.2.2 Activation Functions

Activation functions define the output of a neuron. We have mentioned the *sigmoid* function 2.1 but there are more:

- linear: identity function;
- tanh: hyperbolic tangent function, is a rescaling of the *sigmoid* function with range [-1,1];
- softmax: probability distribution of classes where the sum is 1 and the range is [0,1];
- ReLU(rectified linear unit): activates a node if the input is above a certain threshold.

These functions are not very complex because they are computed at every neuron. Their derivatives can also be used. This means activation functions have an impact on the computational cost of the network.

2.2.2.3 Losses

A differentiable loss function is defined to estimate the error, to compare the prediction to the correct result. An error of – in a perfect world – zero, means that our predicted values do not diverge from the real values. This is never the case in Machine Learning, although the goal is to minimize this function. To improve the predictions, the weights of the neuron connections will be fine-tuned during training. Once this loss is calculated, its value is propagated backward, thus backpropagation.

The total signal of the loss is distributed according to the relative contribution of each neuron. The weights of connections between neurons are adjusted based on this information using the gradient descent technique. This helps detect which direction to shift the weight, using the derivative of the loss function. The process can be applied to batches of data that we feed the network in each iteration eventually passing through all the dataset – epoch.

2.2.2.4 Optimizers

To minimize the loss, we can use optimizers to adjust weights and biases. These parameters are not solvable analytically but can be approached using optimization algorithms such as the mentioned stochastic gradient descent where a few samples are randomly selected in each iteration to reduce computational costs.

Parameters are configurations internal to the model whose values are estimated from the data whilst hyperparameters are configuration variables external to the model and are specified by the user. This is where the creation of architectures in Deep Learning becomes complex, but human learning – or experience – is key. Some of these hyperparameters include activation functions, number of neurons, and number of layers at the topology level. At the learning algorithm level for example the number of epochs, the batch size, the momentum, and the learning rate.

The learning rate decay decreases the learning rate epoch by epoch, which allows earlier faster learning. It is progressively fine-tuned to ease the training convergence to a loss function minimum. In real cases, this function is complex and the optimization process may be stalled on a local minimum, leading to sub-optimal results. Momentum addresses this problem by taking a weighted average – constant with range [0,1] determined by the user – of the previous steps to determine the direction of the gradient.

Although not exactly a hyperparameter, the initialization of parameter weights is important to reduce the number of epochs needed to train deep networks and stabilize the learning process.

2.2.3 Computer Vision

Computer Vision is a field of AI that makes use of computing systems to interpret the real world, by locating and identifying objects accurately through images and video. Deep Learning is widely used in Computer Vision [71] with applications in important areas like Medicine, Autonomous Vehicles, Military, and Machine Vision. Before deep learning, traditional computer vision methods relied on hand-crafted features, which are not feasible for the current problems the field tries to tackle.

Some Computer Vision problems approached by Deep Learning [80]:

- Image Classification: predict the probability of an object present in an image, labeling;

- Object Detection: detecting instances of objects inside images and/or video;
- Image Retrieval: a collection of images having at least the one same object;
- Semantic Segmentation: linking each pixel in an image with one class label;
- Object Tracking: following a specific object in a scene;
- Action and Activity Recognition: sports and urban settings using input video sequences or images;
- Various Biometrics Tasks: from how to recognize subjects, to pose estimation and tracking, crowd counting, gaze estimation, etc.

These are some of the problems currently approached with deep learning. CNN's are used in most of them because of feature learning as a unique capability although they rely heavily on labeled data [71]. Being one of the most popular fields to deal with real-world problems [80], computer vision is an optimal field to one who studies neural networks. Their extensive use in the field and the wide variety of datasets (Section 3.5) makes Computer Vision an interesting domain to study.

2.3 Randomly Wired Neural Networks

Neural networks are inspired by the brain, although when we are young we lack the experience to learn and we rely on innate behaviors that are already present at birth. This suggests that animals are born with highly structured brain connectivity, enabling them to learn rapidly [89], hence biologists suggest the importance of wiring when building Artificial Neural Networks.

Searching the possible space of wirings – or connectivity – of neural networks can lead to more efficient solutions [83]. In [83], the typical notion of layers is relaxed and, although the wiring is fixed, as weights are modified during training, the higher k weights are dynamically selected. This threshold edge swap allows learning the connectivity as it learns the network parameters. The importance of wiring is also emphasized when using randomly initialized weights in network architecture, performing no training, and still achieving success [22] in simple tasks.

Randomly wired hardware and its implementation in software (i.e. artificial neural networks) were an interest in the early stages of artificial intelligence. In the resurgence era of hardware, engineers were inspired by randomly wired hardware to apply them in software. Alan Turing proposed *unorganized machine* [75], a concept comparable to a model of randomly connected neural networks. In the 1950s, Rosenblatt [65] built a visual recognition machine based on an array of photocells that was randomly connected and Minsky [56], implemented, using vacuum tubes, one of the first learning neural network machines.

Random graphs are also generally studied in graph theory because they show different probabilistic behaviors that depend on the algorithm [5]. In graph theory, a general graph defines a triple

$G = (V, E, \phi)$ where V and E describe vertices (nodes) and edges (links), respectively. ϕ is a function that maps edges to an unordered pair of vertices. They are linked to natural phenomena and are thus an effective means to model real-world problems like a computer or social networks [86].

The Facebook AI Research team recently presented a study [86] exploring the design of network generators based on graph theory algorithms to generate random graphs. Searching for connectivity patterns at random is less design demanding than conducting a manual architecture construction.

Previously the main direction with ANN studies would rely on network engineering and learn millions of parameters (features, internal filters, and classifier weights) while this study uses network generator engineering to learn the networks themselves.

2.3.1 Automatic generation of networks

A *network generator* is defined as $g : \Theta \mapsto \mathcal{N}$, where g is a mapping from a parameter space Θ to a space of neural architectures $\mathcal{N}$. The parameters $\theta \in \Theta$ define a network instance. E.g., in a ResNet [28] generator some of the parameters could be the depth, width, number of stages, etc.

For $g(\theta)$, mapping is deterministic, meaning we get a consistent output (network architecture N) when introducing the same input (parameters $theta$). To build a random family of networks, a seed s is introduced. It is obtained through a pseudo-random number generator and its value is changed while keeping the parameters θ fixed. $g(\theta, s)$ is referred to as a *stochastic network generator*.

In [91] this type of network generators were explored but contained several generation rules applied to the network mapping coupled with the use of LSTM and other hyperparameters. The resulting network space is highly restricted by human design, involving a strong prior that reduces the subset of all possible graphs.

The (stochastic) network generator encapsulates different operations:

- Generating random graphs;
- Mapping graphs into a neural network;
- Placing node and edge operations;
- Introduce heuristic rules.

A *random graph* helps reduce human design in a neural network. Usually, CNN follows a sequential pre-determined order when performing its operations. In Figure 2.2, the different nodes would follow this rule. The input, e.g. an image, is passed to the first node and follows an order accordingly. Randomly wired neural networks have a different approach, where the network's flow is arbitrarily defined while still being sequential.

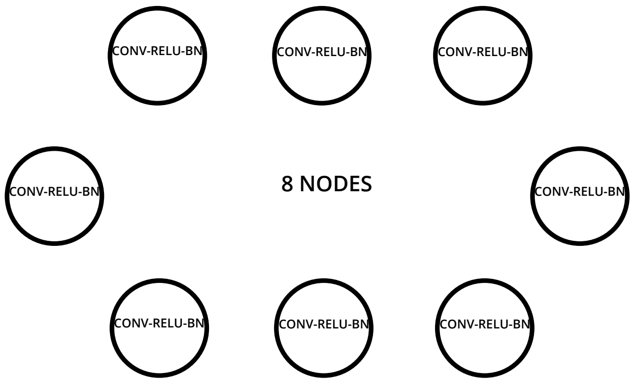


Figure 2.2: 8 nodes that represent layers

2.3.2 Random Graph Models

To generate general graphs, three classic graph generating algorithms from graph theory are used: ER (Erdős–Rényi) [18], BA (Barabási-Albert) [2] and WS (Watts-Strogatz) [81]. These random graph models generate undirected graphs with certain probabilistic characteristics inherent to each of them. Different graph output and network properties are expected using the same input. To follow a probability distribution is considered to be a prior of the network generator even though the neural networks are random.

The ER [18] model uses a probability p to (randomly) construct edges between pairs of nodes, each edge being completely independent of all other edges. Iterating through all pairs of nodes, any graph $\mathcal{G}$ with N nodes can be generated with this model, including disconnected graphs.

The BA [2] model adds new nodes sequentially. These are attached with m edges with a preferential attachment for existing nodes with a high degree. The degree is the number of edges linked to a node in a graph. Certain areas of the network might be heavily connected and can quickly accumulate more connections. The degree distribution converges to a Poisson distribution, unlike real-world scale-free networks that follow a power law.

The WS [81] algorithm receives as input an even number k. Each of the nodes in the ring is connected to $k/2$ neighbors on both sides, meaning if $k = 2$ only the immediate neighbors are connected. The graph $\mathcal{G}$ is then a ring lattice, having a regular wiring configuration. The other input, p, represents the probability of *rewiring*. It traverses along the edges in a clockwise direction, selects an edge with p probability, and randomly chooses where to rewire to.

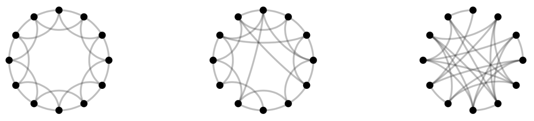


Figure 2.3: The impact of probability p on the WS model [81].

Figure 2.3 shows the effect of p in the WS model when the value k is set to 2 (independent). On the left $p = 0$, so the original ring remains unaffected. The middle graph has a p set to 0.15 and $p = 1$ for the one on the right. The graph becomes more tangled as we increase p, finally having all edges randomly wired with p set to one. Still, some of the edges might rewire to the same node.

After a graph iteration, we have an undirected connected graph $\mathscr{G}$. The nodes are sequentially labeled to convert the graph into a directed acyclic graph, more akin to a neural network structure.

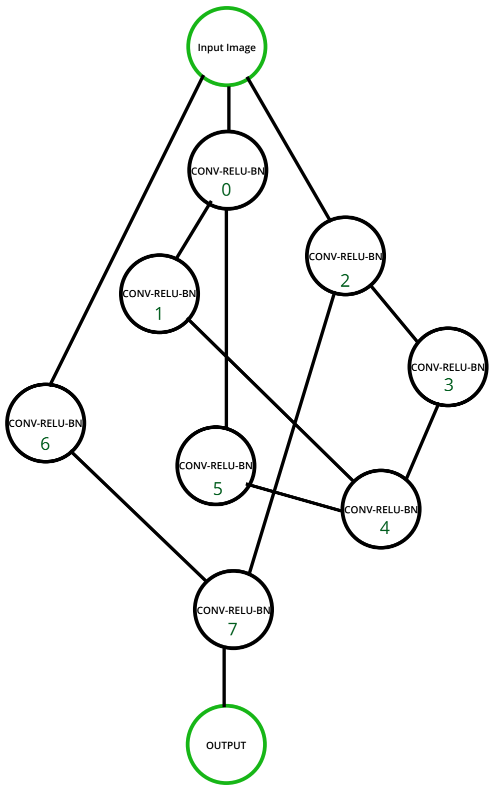


Figure 2.4: WS random graph model generated undirected graph.

Using the WS algorithm on the nodes in Figure 2.2, we obtain a small randomly wired neural network in Figure 2.4. We can observe that, regarding edge operations, the directed edges are responsible for data flow and the output of one node can be shared with multiple nodes. WS was the model that obtained the best results on the paper [86].

While BA can be described by *preferential attachment*, the WS model is limited by an unrealistic degree distribution and a fixed number of nodes. The network is fairly homogeneous topologically, where all nodes yield a kindred degree. Having these biases supports the six degrees of separation [81] theory, where an agent in a network is only 6 links away from any other agent in the network. [47]. This model is described as a *small-world* network where most nodes are not each other's neighbors but any node's neighbors are probably neighbors of each other, thus one node can reach any other through a short number of sequential connections.

After a general graph is generated there's a conversion from an undirected graph to a DAG (Directed Acyclic Graph). In this process, simple heuristics are used. All nodes in a graph are assigned an index – indexing strategy varies with random graph model – and the direction of every edge is set accordingly from the smaller indexed node to the larger. This operation excludes the possibility of having a cycle in the resulting DAG.

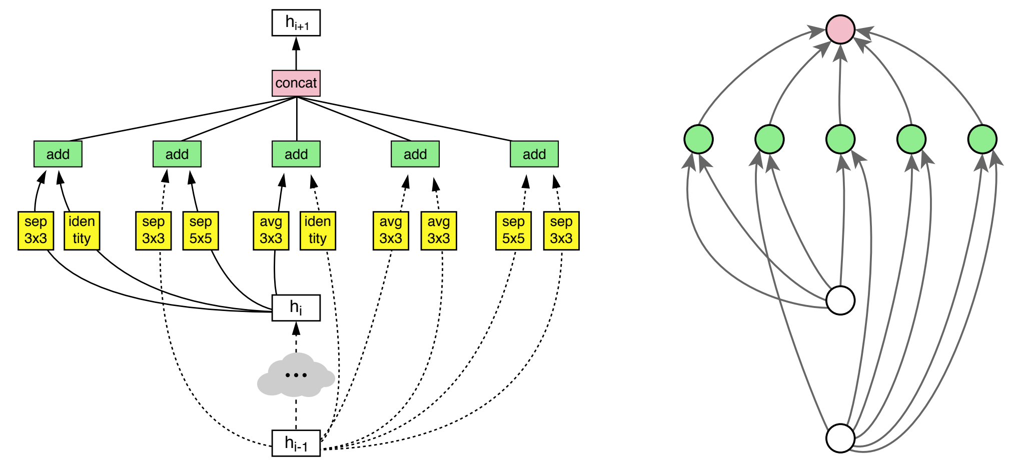


Figure 2.5: DAG mapping from a NAS cell [91]

In Figure 2.5 we see an example of a DAG on the right, containing no cycles and *unique* input/output nodes. On the left, we have a NAS cell, where the data flow is mapped as edges, and the transformations are mapped to nodes, akin to randomly wired neural networks.

2.3.3 Node Transformations

The node operations in the randomly wired neural networks include a weighted sum that combines the input edges. A *sigmoid* function (Figure 2.1) is applied to keep the weights positive – aggregation. Following, a ReLU-Conv-BatchNorm triplet transforms the previously aggregated data – transformation. And finally, the output edges send out the same copy of the obtained transformation – distribution.

There are some desirable properties in these blocks. The additive aggregation compared to concatenation along the channel axis makes sure that the focus of the architecture is on the wiring rather than having a large input degree [83]. To allow data combination between nodes, there should be the same output/input number of channels.

These blocks offer some fine properties:

- *aggregation* preserves the number of channels (input vs output), limiting the computation of the following convolution. The importance of a node can increase with its input degree due to computation, independent of wiring;
- *transformation* to allow distribution, the channel count is constant between input and output so the number of FLOPs (floating point operations) is unchanged;
- direct parameter influence on *aggregation* and *distribution* is reduced.

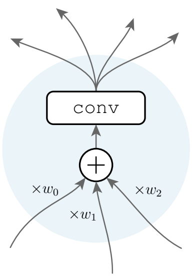


Figure 2.6: Node operations for random graphs [86]. Input edges produce an (additive) *aggregation*, *conv* performs a Conv-ReLU-BatchNorm *transformation* block and the output edges represent the data flow *distribution*.

The deviation of FLOPs is ≈2% between random network instances which are going to be different based on the pseudo-random generator's seed. This means we can compare graphs independent of their complexity and that the wiring pattern properties can be reflected in the performance.

2.3.4 Evaluation

We evaluate the predicted connection weights (labeled in Figure 2.6) with our machine learning algorithms through the Root Mean Squared Error (RMSE) metric:

$$RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}\left(\frac{d_i - f_i}{\sigma_i}\right)^2} \tag{2.1}$$

where a result closer to 0 suggests a better fit to the data. This is calculated using the scikit-learn [61] *Python* library that also provides the methods for building the Random Forest Regressor. Elastic net regularization and Support Vector Machines are also included in this library.

2.3.5 Structure Priors

Randomly wired neural networks are presented in two complexity regimes: *small* and *regular*. The difference between these is the number of stages that are represented through a random graph: 3 for the *small* complexity regime and 4 for the *regular* complexity regime. In common they have a *softmax* classifier that performs global average pooling and the first stage, a Conv-BatchNorm convolutional block with an input size of 224 by 224 given the ImageNet [67] usage.

Maintaining the input resolution through the stages is not ideal. The different stages downsample the original signal progressively and the channel count is also doubled when passing through each random graph.

Since the focus of the research in [86] relies on neural network's connections, some restrictions were applied:

- ReLu-Conv-BatchNorm triplet blocks are used;

- weighted sum is performed in the aggregation of tensors, being always positive;
- network stages remain the same in every random network instance.

The structure is analogous to ResNet [28] and is one of the restrictions of these networks. In Figure 2.7 we can see how the stages are connected by the generated random graphs and how the connection between stages is done. Because a general random graph is not a valid neural network – multiple input and/or output nodes possible, two extra nodes are added post graph generation. One node is the *unique* input node, connected to all original input nodes, and the other is the *unique* output node that aggregates all original output nodes. These are excluded on the node count N and have no convolution. The input node sends copies of itself to the original input nodes and the output node calculates the average of the original output nodes.

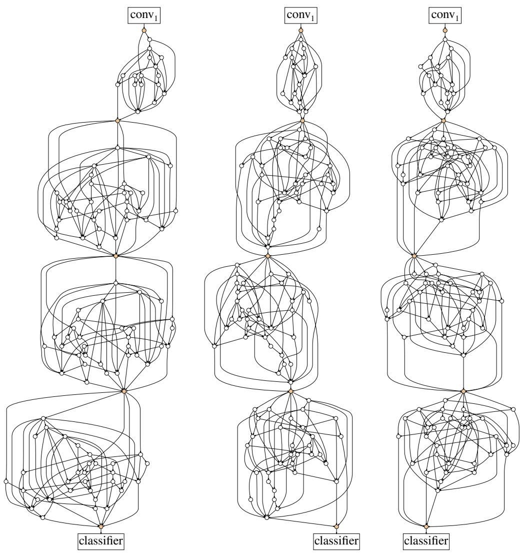


Figure 2.7: WS generated *regular* complexity regime networks [86]

The graphs included in the networks of Figure 2.7 were obtained using the same model, WS, but with different seeds between each network. In one single network, with the same generator parameters, the graphs are not similar to each other. This shows that even with the same parameters θ, including the seed s, a graph, and a network are not reproducible. Between the three networks, the seed was different while the performance on ImageNet [67] was still almost identical – $79,1\%$, $79,1\%$, and $79,0\%$ respectively. As the authors report [86], *random search* on the seed s is unfruitful due to a minimal accuracy variation.

2.3.6 Process automatization

Data analysis is an emerging tool in many organizations, which can be used to improve businesses in several areas: products, decisions, personnel, and client evaluations[76]. Machine learning models are frequently used for these tasks but require expert knowledge to be applied successfully, thus the automatization is a way to abstract difficult concepts. Automated Machine Learning

(AutoML) aims to reduce as much as possible the human intervention in producing machine learning models, including deep learning models [29].

As the process of building machine learning models is fundamentally iterative, through hyperparameter tuning, feature selection, and performance comparisons, it can benefit a great deal from automation [76]. With AutoML, a practitioner's focus shifts from tedious and slow tasks to highly creative tasks that deliver faster results, and allow earlier feedback of the architecture's predictive power [76].

2.4 Established Algorithms

Neural Architecture Search has been described as a joint optimization of both the operations in a node and the way the nodes are connected [90]. Usually, these aspects of a network involve human design and even in the NAS search space, e.g. NASNet [91], there is still a considerable bias from the designers.

Randomly wired neural networks relax even more the priors by producing more flexible networks through the *network generator* g. They create families of architectures with a larger space of possible neural networks by exploring the parameters of g.

Randomly wired neural networks [86] were partly created because of the success of the previously mentioned ResNet [28] and DenseNet [34] through their innovations in the wiring patterns.

2.4.1 NAS

NAS is categorized into two approaches: optimization algorithms and model structures [29]. For completeness, we reference model structures where a combination of a pool of primitive operations (pooling, convolution, concatenation, etc) generates a model. Representative structures include the generation of the entire structure, a cell-based structure, a hierarchical structure (leveling a cell structure), and network morphism by transferring a network's information into a new one.

The state-of-the-art NAS research emphasizes optimization methods, although the search space is not particularly widened in these approaches, thus the set of feasible solutions is limited [76]. The methods are useful for comparisons and further optimization but the search space is nonetheless defined by the network generator [91]. Some of those methods include:

- Reinforcement Learning [72] [62] [91] relies on RNNs to generate networks and a reward network, that trains and evaluates generated networks, updating the controller according to the reward (for example, accuracy);

- Progressive [50] using a sequential algorithm;

- Gradient-based [51] that can reduce hyperparameter search time [76];

- Weight-sharing [12] where filters are shared between parts of the image;

- Evolutionary [62] [78] [64] use genetic based algorithms to achieve state-of-the-art performance;
- Grid/Random Search [48] [69] being one of the most used methods when hyperparameter search started [76];
- Bayesian Optimization [82] that builds a probability model to propose more efficient choices in hyperparameters to evaluate [29].

In random wired neural networks, the focus turns to the grid and random search. Grid search defines regular intervals – a grid – to divide the space of hyperparameters, choosing the best performing values in model training. Random search can test more hyperparameters and since not all parameters have the same level of importance [10], the search is more practical and efficient. Grid search can be used for the exploration of the number of filters, nodes, and stages, which is essentially trial and error. Random search is performed by exploring values for the random seed of the network generator in [86] but was revealed to be unfruitful.

2.4.2 NAS state-of-the-art

Neural Architecture Search was first proposed in 2016 [90]. The main motivation for NAS is that architecture engineering is often heavily necessary to build successful neural networks, which can be hard to design. A Recurrent Neural Network with reinforcement learning was used to generate model descriptions of neural networks in [90]. The RNN is then trained to maximize the accuracy on a validation set, and it proved to rival the best hand-designed architectures.

NAS searches for the best possible neural network architecture from the pool of "building blocks" available. These are defined manually and the algorithm searches for an ideal combination that obtains a certain accuracy. It uses a gradient-based method to update the weights towards its goal, adjusting the building blocks as necessary. This means that we have a limited search space, and the networks end up being similar to the existing state-of-the-art.

In 2018 [91] appears an evolution of the previous algorithm where the idea is to, directly on the dataset of interest, learn the model architectures. Being a computationally demanding task, it is more feasible to search for a building block on smaller datasets. Later on, transfer that architectural block to a larger dataset, thus exposing the "NASNet" search space that enables transferability. With fewer parameters and fewer floating-point operations, it achieved state-of-the-art performances.

Progressive Neural Architecture Search (PNAS) [50], instead of reinforcement learning, uses a Sequential Model-based Optimization (SMBO). This could be viewed as a greedy algorithm, where the building blocks are searched in increasing complexity order, making the search smarter and it is significantly more efficient than the original although not reducing the search space.

Another approach is to use transfer learning. This means each time we train a model, we use transfer learning to converge faster. Efficient Neural Architecture Search (ENAS) [62] is an example, where the trained weights of each model are not wasted and instead are shared between models instead of training to convergence from scratch. The computational expense is also significantly lower than the previously mentioned methods.

Li and Talwalkar [48] defined NAS as a problem of hyperparameter optimization. A competitive baseline for this is random search, using early-stopping – removes the need to manually set the number of training epochs by saving the best model at the end of each epoch – and weight-sharing, achieving results comparable to ENAS. They point out the lack of experimental reproducibility as an obstacle to the slow progress of the research in the field due to the complexity of algorithms and computational costs.

2.4.2.1 Resources

AwesomeNAS [14] holds a select list of the NAS state-of-the-art, including paper references and code, when available. NAS is a field that is fastly growing and this is a tool for following the developments in the field. Recently, Google has created the NASBench [24] [88] dataset, which holds $\approx 400,000$ unique neural networks that obey certain constraints. It maps CNN architectures to their performances on CIFAR-10[42]. This dataset is continuously being updated as the field progresses, suffering its last update with NASBench201[16].

2.4.3 Other Definitions in Machine Learning

2.4.3.1 Machine Learning Algorithm - Random Forest

In a Random Forest, decision tree predictors are combined [8]. Decision trees define split nodes that test a feature and have a tendency for overfitting to the training set. An ensemble of individual decision trees cast a vote for a prediction and the majority result is the output. Each tree is uncorrelated and a large number of them outperform individual trees because they prevent each other's errors. Bagging allows each tree to sample from the dataset randomly with replacement. The sample size is constant across all trees.

2.4.3.2 Meta-learning

Using metadata obtained from machine learning experiments to perceive how learning occurs is part of the process of meta-learning. The goal is to improve an *inner* algorithm by inducing its behaviour with an *outer* algorithm [32]. The process then includes a learning subsystem. The learning instances of the learning subsystem provide the data necessary for the base learning algorithm to learn.

Chapter 3

Methodology

In this Chapter, we delve into the technical part of our work. First, we present the requirements and the environment of development. Following, the implementation is then described, as well as the datasets used in this study. Furthermore, we describe the experimental setup and the optimization approaches performed.

An empirical evaluation implies a study conducted through experimental means instead of a theoretical approach. A good study of this sort is sustained by proper design and execution of the evaluation procedures. In our case, we aim for the quantitative results of our observations of the behavior of the randomly wired neural networks.

The results will allow us to evaluate and compare with existing networks. The methodology then starts by searching for the established networks (Section 2.4), defining datasets for training (Section 3.5), and comparing those networks among themselves. We then implement a network generator, generating neural network architectures whilst collecting metadata, and finally derive conclusions. This metadata also involves gathering the weights and respective biases during training and other data that can help us understand better how these networks achieve the reported results [86].

Transfer learning can also be applied, where a neural network is set up with pre-trained weights, skipping the computational cost of training from scratch. The pre-trained model chosen should be of the same domain as the one we want to train and their use helps network generalization and speeds up convergence [76]. The technique involves freezing weights and/or decreasing the learning rate.

3.1 Developing Environment

The experiments were conducted using a Graphical Processing Unit (GPU) for faster training. Recent work shows methods in Neural Architecture Search that make efficient use of GPU's, e.g. in [15] four hours of GPU using a gradient-based approach achieved state-of-the-art results. In this research we use a NVIDIA GeForce GTX 1080 [59], with 8 *GiB* of Random-access Memory (RAM) and a memory speed of 14 *Gb/s*.

The GPU is installed in a Ubuntu [77] machine remotely accessible via SSH (Secure Shell). We use *PyCharm* on a local Linux machine for *Python* programming, bidirectional remote access and version control.

3.2 Requirements

Developed by Facebook's AI Research lab, as well as Randomly Wired Neural Networks [86], *PyTorch* is the main machine learning library used in this work. *PyTorch Tensors* are operable with CUDA, a parallel computing platform created by NVIDIA, supported by our GPU. Similar to *NumPy* arrays, these tensors handle multidimensional number arrays but with powerful acceleration.

Other than *Python* (version 3.7) itself, another main library is *NetworkX* [26]. This is a *Python* package for generating, manipulating, and studying networks on several levels – structure, function, and dynamics. It proves useful both at the implementation phase, where we use it to generate random graphs that define the *wiring* of the networks, and in the empirical evaluation process to obtain graph metrics that help us interpret the complexity of a network and are used as input to our optimizations.

3.3 Implementation

Randomly Wired Neural Networks do not have a publicly available source from the original authors [86]. Thus, we implement our own version based on third-party code available on GitHub [38].

In our code, we have different files to host different functionalities:

- *graph.py* - allows us to generate a random graph, independent of the model (WS/BA/ER) and their parameters – see Section 2.3.2. It is possible in this file to obtain information about a graph based on nodes and/or edges, convert from an undirected graph to directed and *vice-versa*, and load/store a graph from memory;

- *model.py* - defines a full randomly wired neural network, receiving the necessary parameters. *Small and regular* regime networks (see Section 2.3.5) are supported;

- *preprocess.py* - loads datasets from memory with the corresponding parameters for each and applies the necessary transformations;

- *randwire.py* - describes a single randomly wired neural network, corresponding to one stage of the full network. It generates a general random graph expanded with unique output and input nodes whose edges correspond to the data flow. The nodes are a class by themselves to host the node operations – Conv-ReLU- BatchNorm Triplets;

- *test.py* - runs experiments as the command of its arguments, including optimizing options;
- *train_utils.py* - utility functions for training manipulation and user interaction.

To create a randomly wired neural network we simply instantiate a *Model* with the user-defined parameters. These involve:

- number of nodes – fixed at 32 in the original study;
- input and output channel count;
- graph generation model to use – WS/BA/ER in 2.3.2.
- complexity regime – *small* has 3 randomly wired network stages (see Table 3.1) and *regular* has 4;
- seed s for randomization;
- p probability, for ER and WS (also uses k mean node degree);
- m initially connected nodes in case of BA algorithm;
- name – a string to represent identity.

The *Model* instructs the *RandWire* class to generate random general graphs that are converted to a Directed Acyclic Graph. After this, the extra input and output nodes that connect the stages (different random graph based networks) are added and operations within the nodes are built with the ReLU-Conv-BatchNorm computational blocks (see Section 2.3.5). The weights of node connections are all initialized to the value of 1. The stride of the nodes that are directly connected to each input node is 2 and the number of channels is progressively doubled through each stage.

When training a neural network, we record several aspects that are important for the study:

- the neural network object (including weights, biases, structure, etc). *PyTorch* allows us to save and load the models. This is useful when we want to further train or observe a network's (internal) state. Besides this, the file that holds the model records the epoch number and accuracy of the highest obtained test accuracy;
- a CSV (Comma Separated Value) file which holds training information for every epoch, including test accuracy, training loss, training accuracy, and the timestamp;
- a text file generated by NetworkX to save the graph structure. This is useful for visualization and to load a network because its structure is coupled with the graphs it contains.

3.4 Metrics

Our implementation makes use of a random graph generator to create a set of different neural networks. These networks are evaluated using several metrics:

- accuracy;
- loss;
- convergence rate;
- training time.

These metrics are recorded for every epoch of training for all our generated networks. The test accuracy tells us the percentage of correct classifications on unseen examples, whilst the training accuracy measures the accuracy on examples the model was constructed on. The training loss calculates the prediction error of the neural network. This allows us to judge the parameters used in each network but most of what happens within a neural network is still a black-box [48].

3.5 Datasets

We use image-classification datasets because this is one of the most interesting and challenging fields in deep learning. We had further support in this field by collaborating with the Computer Vision Laboratory at the Faculty of Computer and Information Science of the University of Ljubljana.

We selected mostly smaller datasets to allow faster training in the computational resources available. The datasets used in this work are:

- **MNIST**[46] is a widely used dataset for image classification models training and contains images of handwritten digits;
- Fashion MNIST[84] similar to MNIST, featuring grayscale clothing images;
- **CIFAR10** /CIFAR100 [42] [43]: 32x32 natural image dataset with 10/100 categories, this allows to quickly try different algorithms;
- ImageNet [67]: image database organized in accordance with the WordNet hierarchy with 14M images;
- in the laboratory will also be available a more challenging multi classification dataset.

In Appendix C we expose image samples for each of the mentioned datasets. MNIST, CIFAR10 and CIFAR100 contain 60000 images each whilst Fashion-MNIST has 70000 images.

The training on ImageNet requires extensive training and was performed during the final stages of this dissertation. We managed to train one network for 60 epochs, which is a fraction of the orignal study – 250.

3.6 Empirical Evaluation Experiments

Throughout our experiments, to make appropriate use of our computational resources we use the network parameters that use a smaller amount of resources, in particular, we keep the initial channel count C at 78 and we use the *small regime* complexity (see Section 2.3.5). The network architecture is described in Table 3.1. This keeps our work comparable to the original randomly wired neural networks [86], so we also use a constant number of nodes for every graph unless noted, $N = 32$ – this number excludes the *unique* input/output nodes (see Section 2.3.5) that connect the different *random wire stages*.

stage	output	operation
$conv_1$	112x112	3x3 $conv$, $C/2$
$conv_2$	56x56	3x3 $conv$, C
$conv_3$	28x28	***random wiring*** N, C
$conv_4$	14x14	***random wiring*** $N, 2C$
$conv_5$	7x7	***random wiring*** $N, 4C$
classifier	1x1	1x1 $conv$, 1280-d global average pool, 1000-d fc, $softmax$

Table 3.1: The general network architecture, as seen in the *small regime* [86].

Regarding training details, we use an initial learning rate of 0.1 and progressively lower it by $1/10$ every 30th epoch, as in [25]. The weight decay is $5e-5$ and the momentum is set to 0.9. Regarding label smoothing regularization, the coefficient is set to 0.1. Our default batch size is 128.

The default optimizer is SGD (Stochastic Gradient Descent [7]). An optimizer updates the weights after every training batch is processed. The optimizer takes the previously mentioned momentum, weight decay, and learning rate as parameters. The momentum given in the original study is 0.9, the weight decay is $5e-5$ and the initial learning rate is 0.1.

3.6.1 Network generator evaluation

To evaluate randomly wired neural networks we vary the network generator $g(\theta, s)$ parameters. In θ we observe the random graph model parameters, including the pair (p, k) for WS, probability p for ER akin to WS, and m initially connected nodes BA.

The authors in [86] report that random search on the seed s is not beneficial, so we do not procure results in this direction although our initial observations point towards the same conclusion.

We selected a range of values for the mentioned parameters that are similar yet broader than the original study:

- WS has two parameters (k, p). We use the range $0 \leq p < 1$, with step 0.15 and k (even number) with the range $2 \leq k \leq 10$, with step 2;

- BA has one parameter $m(1 \leq m < N)$, where N is the number of nodes. We use the range of $1 \leq m \leq 8$, with step 1;
- ER has one parameter p. We use the range $0 \leq p \leq 0.8$, with step 0.1.

For each of these experiments, we replicate them in the four datasets mentioned before (Section 3.5). The total number of experiments is $(8*5+8+9)*4$ which equals 228 neural networks. These were evaluated through several metrics exposed in Section 3.4.

3.6.2 Node number evaluation

Since the number of nodes is constant, it becomes interesting to explore it as part of the parameters in θ. Besides the authors not providing any explanation for this specific number, the datasets we use are considerably smaller than ImageNet [67] so we might obtain similar results on a lower node number.

We perform a grid search through N, using the values: $6, 9, 12, 16, 20, 24, 28$ and 32. This will result in a total number of $8*4 = 32$ networks in this scenario. The knowledge obtained in this part of the study attempts to reduce to some extent the complexity of randomly wired neural networks while still maintaining its performance on larger datasets.

In this evaluation, we use the paper's reported best parameters for each graph model:

Table 3.2: Default θ parameters used in the original study [86]

Model	p	k	m
WS	0.75	4	-
ER	0.2	-	-
BA	-	-	5

These values represent the best performance of the graph models in the experiments performed prior to this study. A grid search involving the node number and the θ parameters would be too extensive to be fit in the time frame of this work.

3.7 Optimization Approaches

Although the authors challenge researchers to explore the design of *network generator* within Neural Architecture Search, we opted to use techniques to improve their methods which provides additional insight into randomly wired neural networks. During the course of this dissertation, we made several attempts to optimize this type of neural network.

We consider meta-learning (see Section 2.4.3.2), an *outer* algorithm that aims to improve the *outer* objective by updating the model learned by the *inner* algorithm [32]. We use a machine learning algorithm as an *outer* algorithm, namely Random Forest to support in the training of a machine learning model (generated from randomly wired neural networks), or even, as is this case, optimization.

3.7.1 Prediction of connection weights

We try to predict the learnable weights seen in Figure 2.6 by using metadata from already trained networks. First, we train 10 neural networks with 10 different seeds ($s = [1, 2, 3, ..., 10]$) for 100 epochs. For the parameters θ of the *network generator*, we use, for all networks, the reported best performing graph generating parameters in the paper – WS graph model, with $(p = 0.75, k = 4)$. By using different seeds, we can also observe how much the seed impacts the *network generator*.

The training on the *first generation* is made with the CIFAR-10 [42] dataset. We can then observe if the connection weights are dependent on the problem by using the same model to predict with other datasets.

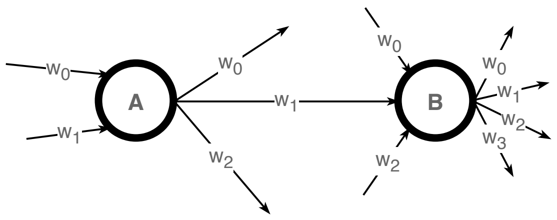


Figure 3.1: Focus on a single node connection, between nodes A and B through weight w_1

In Figure 3.1, we can see an example of a node in detail, where the graph *edges* w_i represent connections between several nodes and each *edge* has a weight and a bias associated with it. As explained in Section 2.3.3, node B's *incoming* weights are summed, passed into a ReLU function and the result is the input to the convolutional block in B. Focusing on the *edge* labeled w_1 that connects node A and node B, we predict the weight w_1 using graph metrics that involve node A, node B and the connection between them within the context of the graph that represents the current neural network stage. All weights with the same label are independent and were numbered sequentially per node input/output. Please note that the input w_0 and w_1 on node A is different from the output w_0 and w_1 of node B.

From these 10 networks, we extract their learned connection weights and build a dataset with network information. Part of this dataset includes:

- the seed s;
- the connection label defined by a string with the two involved source and destination nodes, e.g. node A and node B translate to connection “A-B”;
- the stage of the network the connection belongs to, in our case from 1 to 3 because we use the *small complexity regime* (see Section 2.3.5).

The latter is important due to the type of representation earlier stage nodes learn comparing to further in the architecture.

The rest of the dataset is built using network science metrics, either directly obtained from the *Graph* class or by using NetworkX's API. Some of those graph features include (full list in Appendix B):

- level - sequentially assigned number that describes the position of the graph in the network with possible values $1, 2$ and 3;
- *out_degree* - the number of edges in the output end of a connection, i.e. the successor node's edge count;
- *in_deg_c_b* - represents the in-degree centrality of node b – the successor of the connection. This value is the proportion of nodes its incoming links are connected to;
- *edge_bet_cent* - refers to the betweenness centrality of an edge. The value is obtained by summing the fraction of the shortest paths of all pairs that pass through the edge evaluated;
- *group_in_deg_cent* - uses both nodes of a connection as a group to calculate the group's in-degree centrality. It represents the proportion of outer group members linked to group members by incoming edges;
- *edge_load_cent* - constitutes the edge load in a graph. It counts the number of shortest paths that include the edge in question;
- *depth_a* - using the first node as a reference, we determine the shortest path to the node in question, a, which is the connection's predecessor node. This is a simple heuristic to obtain a node's depth in a graph.

To predict the weights, we need to use a regression model. We have a target or dependent variable – the connection weight – that is to be predicted from a set of predictors or the independent variables – our network characteristics described above. We use a Random Forest, a simple algorithm based on an ensemble of decision trees which is the most used *off-the-shelf*, ready to use, ML algorithm.

Before training a new neural network, we use a regressor with knowledge from our previous neural networks. This *regressor* predicts the weights for the new network, including all stages. The neural network is then initialized with these predicted weights. The desire is to observe a faster training process or to at least have higher initial accuracy.

In the features of the network, we do not use characteristics of the neural network because characteristics such as activation function or regularization layers are constant in every node.

The optimizer and its parameters are explored to some extent. The momentum given in the original study is 0.9 and we experiment with the default value of 0.0 whilst having our weight prediction activated. Another subject to experimentation is Adam (Adaptive Moment Estimation [39]). This algorithm also has a learning rate and weight decay as parameters, which we set to the same values mentioned in Section 3.6.

3.7.2 *Freezing* weights

To observe the influence of the connection weights (Figure 3.1), we *freeze* these weights during the training phase. This is a common term describing inhibiting the desired weights from changing during training, i.e. preventing them from learning. By freezing weights, we can observe how much we can boost the training, by simplifying the backward pass.

It is also interesting to perceive the training evolution when we freeze predicted weights. In this experiment, we hope to notice at least a slight initial improvement, with the best-case scenario being an immediate boost of the accuracy.

3.7.3 Weight shuffling

Another way of testing the weights is to randomly shuffle them. We take all the connections of a network and we reassign each of their weights to another connection in a stochastic fashion. We can then tell if our predictions are accurate and/or evaluate if the order of the weights in the network is important.

3.7.4 Weight distribution

Trying to track distribution in the *aggregation's* learnable connection weights can help us avoid the use of a machine learning algorithm. The weights can then be generated through the standard deviation and the mean of just one previously seen network. The average of the 10 mentioned networks in Section 3.7.1 could be seen as a stronger heuristic.

3.7.5 Node shuffling

In this experiment, we randomly access one of our 10 base networks and randomly extract one of their nodes. We respect the stage the nodes are in, not to combine different feature extraction properties that can be held in each of the stages. The depth of the network is respected but any node of the 32 in one stage can be selected and they can be repeated.

The weights and biases of the chosen node are transferred to the current node in the new network that is being subject to training. Similarly to previous experiments, we separately freeze these weights. It is interesting to observe if the behavior is considerably worse when the new network is training for a different problem.

Chapter 4

Results and Evaluation

In this Section, we present the results of our empirical study. We cover the experiments described in Section 3.6 and Section 3.7. Some of the experiments are omitted due to either repetition of information or redundancy, and exposed on Appendix A.

4.1 Problem State of the Art

The four datasets used in this study have been extensively used in research. Despite the accuracy being high, these are still relevant datasets in research and are regularly used as benchmarks for performance. To put our results into perspective, we introduce the best results in each of them in the preceding works:

Dataset	*Accuracy*	Reference
MNIST	99.84	Homogeneous Filter Capsules [9]
Fashion-MNIST	96.36	FMix [27]
CIFAR10	99	GPipe [35]
CIFAR100	93.51	Big Transfer (BiT) [41]

Table 4.1: Dataset SotA survey

These results were obtained without Neural Architecture Search and are the result of hand-designed architectures.

4.2 Empirical Study

Our empirical study comprises searches in three different directions. Firstly we focus on the *network generator* $g(\theta, s)$, exploring the parameters separately. Inside the θ parameters, we evaluate the performance of the random graph models displayed in Section 2.3.2. Secondly, we evaluate the impact of the seed s and lastly the node count in each stage of the network.

4.2.1 *Network Generator* Hyperparameter Tuning

In Section 3.6.1 we describe this section's experiments. We perform grid search for the three random graph models in our datasets (Table 4.1).

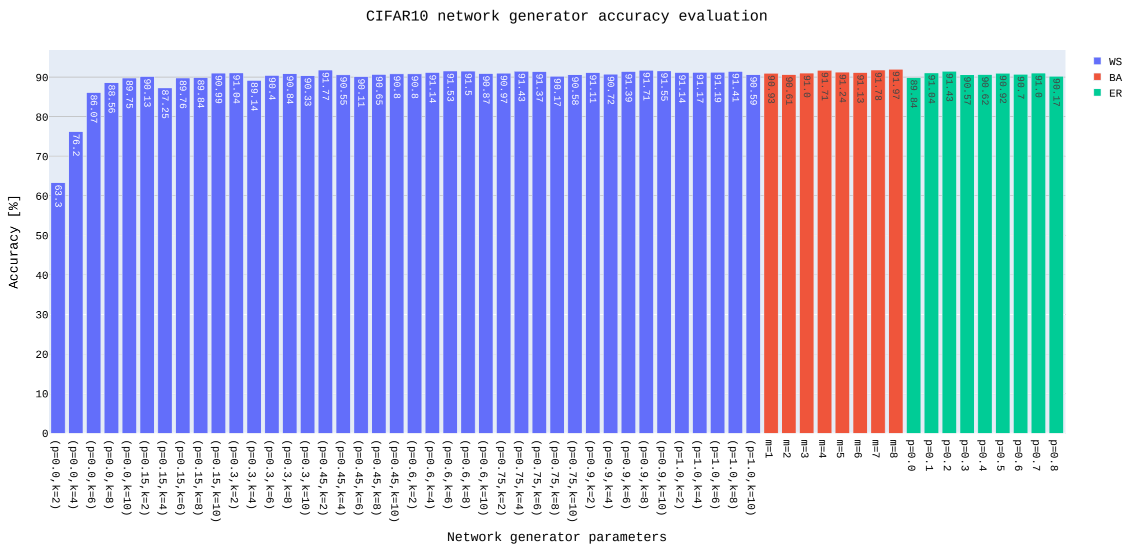


Figure 4.1: *network generator's* θ parameters variation impact on CIFAR10's accuracy

The Watts-Strogatz model can be interpreted by analyzing its two arguments, and looking at Figure 4.1:

- probability p: with a higher value of p, the performance of the network is progressively improving and this improvement seems to be independent of k. With $p = 0.0$ we have a simple network which does not randomize links between nodes which might cause a shallower feature extraction process;
- number of initial connections k: defines the density of the ring at the random graph's generation, which explains that the combination $(p = 0.0, k = 10)$ holds an accuracy similar to the highest accuracy obtained – parameter pair $(p = 0.45, k = 2)$. This also explains why in Figure 4.2 the time of execution is considerably smaller for lower values of k, being around $2.5x$ faster comparing $p = 2$ to $p = 10$.

In the Barabási–Albert random graph model, m is the initial number of nodes in the graph. The nodes are progressively added until reaching the intended number of nodes C in a stage of the network. When attaching a new node, m edges are assigned to it with a preference for higher degree nodes. This discloses the increasing training time for higher values of m, seen in Figures 4.2, 4.4 and 4.6.

The neural networks generated with the BA model have an almost constant performance in these datasets. Minimal differences in the accuracy can be observed, so it is preferable to use a lower level of m since the training time is smaller.

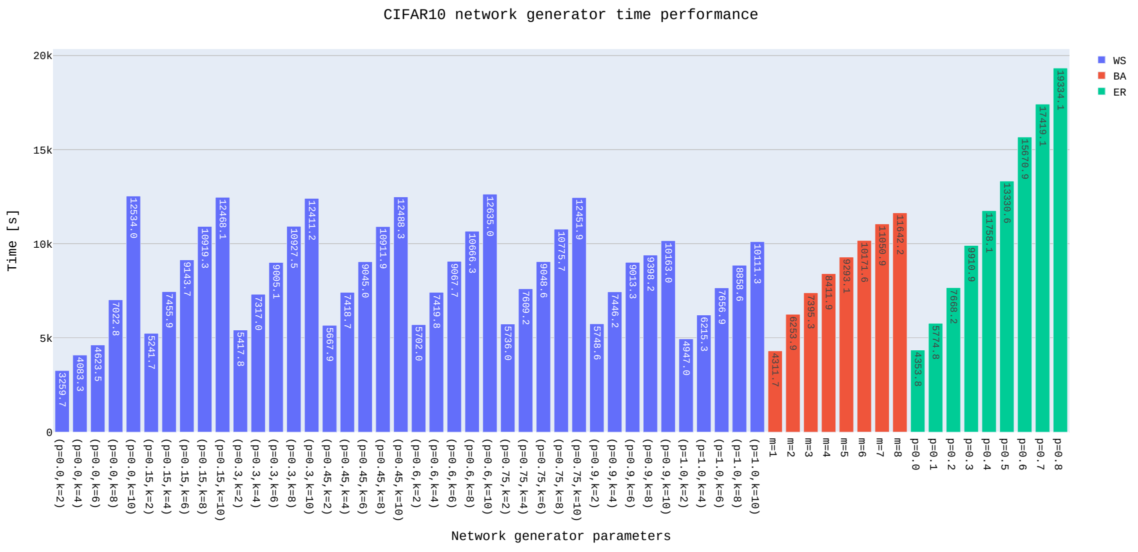


Figure 4.2: *network generator's* θ parameters variation reflected on CIFAR10's training time.

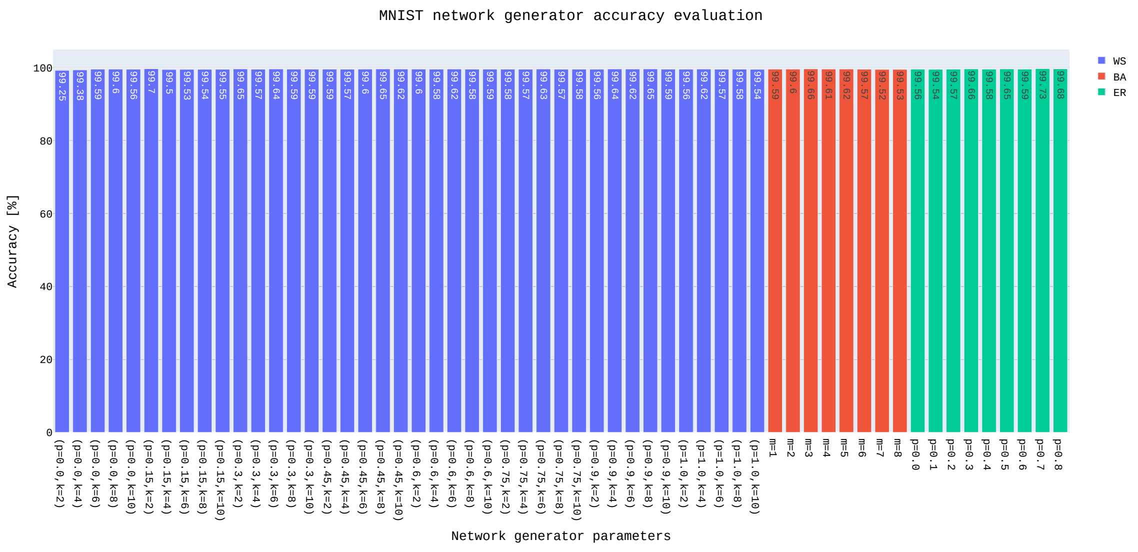


Figure 4.3: θ *network generator's* parameters variation impact on MNIST's accuracy.

The Erdős–Rényi algorithm has only one parameter, probability p. This probability represents the chance for each possible edge to exist. The graph is constructed randomly and the probability of each new edge is independent of the others.

The higher the probability p in ER, the more likely it is for connections to be formed. With p closer to 1, the graph becomes denser, implying more aggregation operations in the network. The training times for ER increase with p, showing that the graph becomes more complex. The duration is higher than the other models due to the superior number of operations in the network. The graph generation algorithm is likewise complex, with $\mathcal{O}(n^2)$.

The original study reports $p = 0.2$ as being ER's best parameter, which is in line with the results obtained with CIFAR10 and Fashion-MNIST, in Figures 4.1 and 4.5. This value should

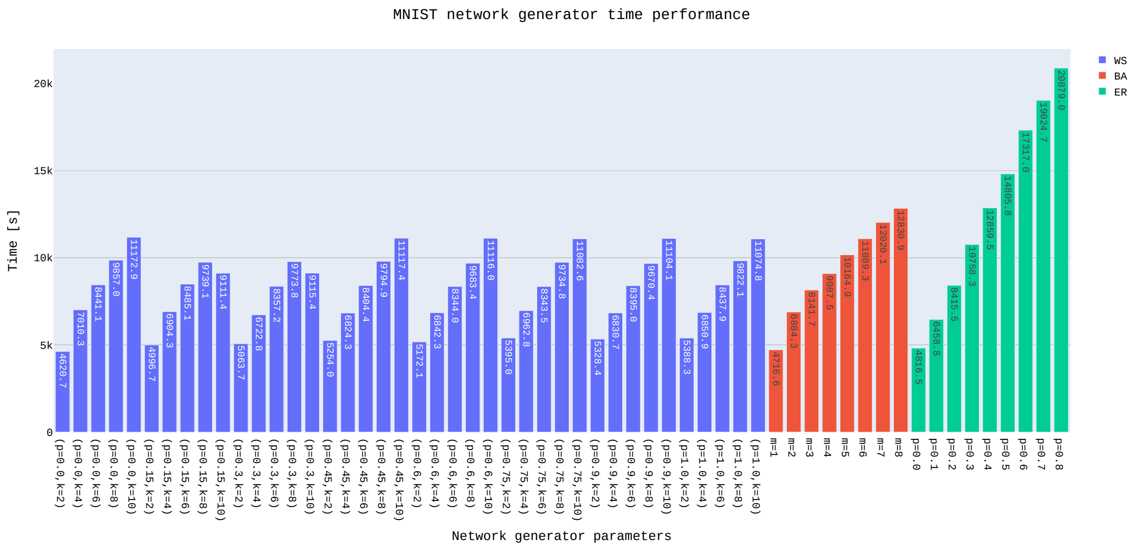


Figure 4.4: θ *network generator's* parameters variation impact on MNIST's training time.

represent a network with vaguely dense graphs that have a minimum amount of random connections which allows to obtain a high accuracy with moderate training times, comparable to other models as seen in Figures 4.2 and 4.6.

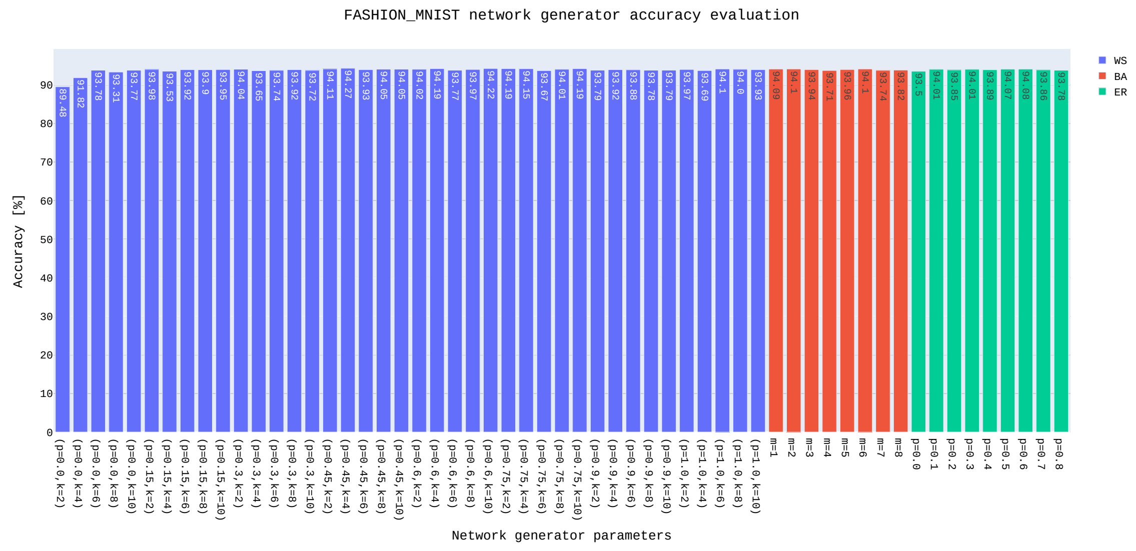


Figure 4.5: θ *network generator's* parameters variation impact on Fashion-MNIST's accuracy.

The MNIST dataset is the most undifferentiated one, as seen in Figure 4.3. The dataset is considered an easy task for having small, gray-scaled images and containing no background. The feature extraction is not a complex problem for any network we subjected MNIST to classification.

On CIFAR100 [43] we obtain a more diverse set of results. Being a more complex dataset, using $p = 0.0$ implicates poor results. These networks contain graphs generated with no randomness and are too simple to learn how to extract features from the input well. With a p of 0.3 the network starts approaching its potential even with a low value of k.

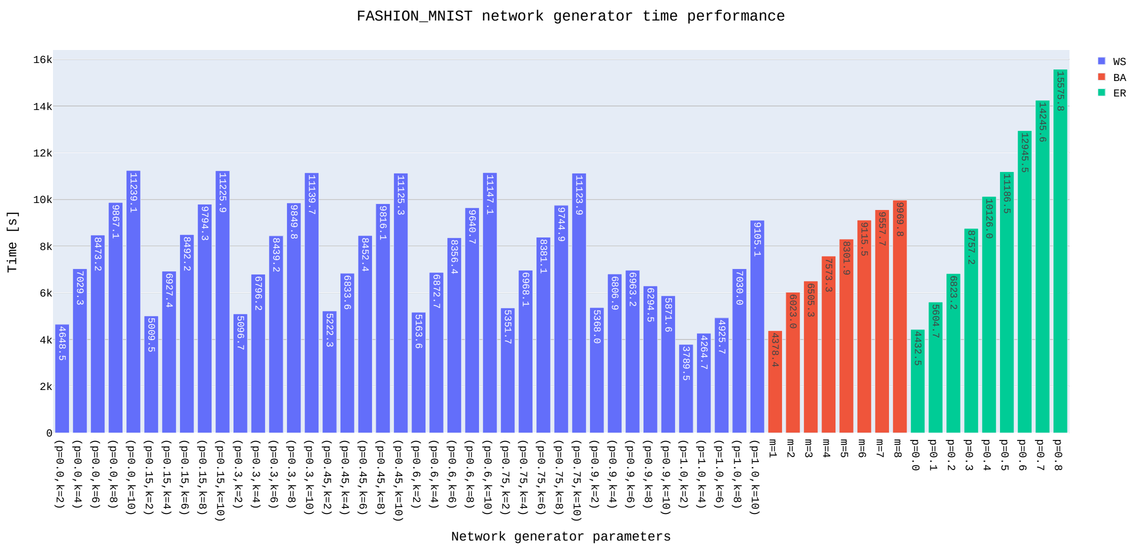


Figure 4.6: θ *network generator's* parameters variation impact on Fashion-MNIST's training time

In Figure 4.7, the best parameter combination for WS is $(p = 0.6, k = 2)$. This network has less initial connections because k is at a minimal value; $p = 0.6$ implies that the majority of those initial connections will be rewired. This results in a fairly random graph that contributes to the performance of the network that is based on 3 graphs generated with these parameters.

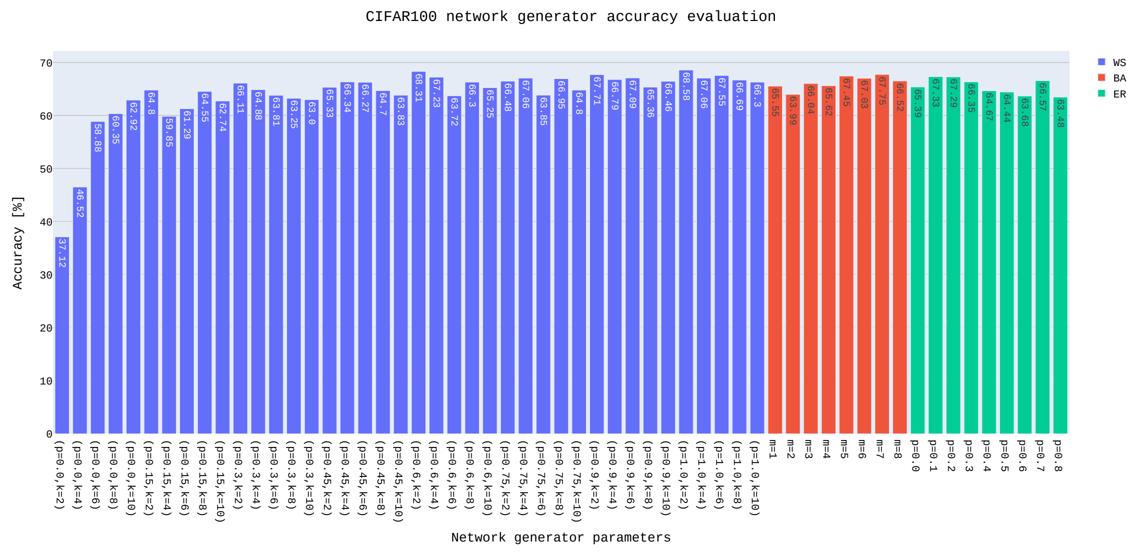


Figure 4.7: θ *network generator's* parameters variation impact on CIFAR100's training time

For ER on the CIFAR100 dataset, the best values for p are 0.1 and 0.2. This is in line with the original paper, reporting $p = 0.2$ (see Table 3.2) as the best ER performing network. BA recorded a lower accuracy for lower values of m. This dataset is more complex than the previous ones, having 10 times more classes which explains the better performance on more complex (denser) networks.

The training times for CIFAR100 can be seen in Figure A.1 and are similar to the other datasets. Since in the best network $k = 2$, the complexity of the network is lower, thus the training duration is short compared to other values of k. The training is completed in just around 70 minutes.

As expected, we can then conclude that the training duration is closely related to the complexity generated in the random graph models. This defines the number of connections and consequently, the number of values and operations computed.

4.2.2 Random Search

In the 10 original networks described in Section 3.7.1, we use 10 different seeds for the parameter s of the *network generator* $g(\theta, s)$. The authors of the randomly wired neural networks [86] report no random search due to small differences between the networks. When detailing their best set of *network generators* they overlook the seed s due to its ineffective contribution.

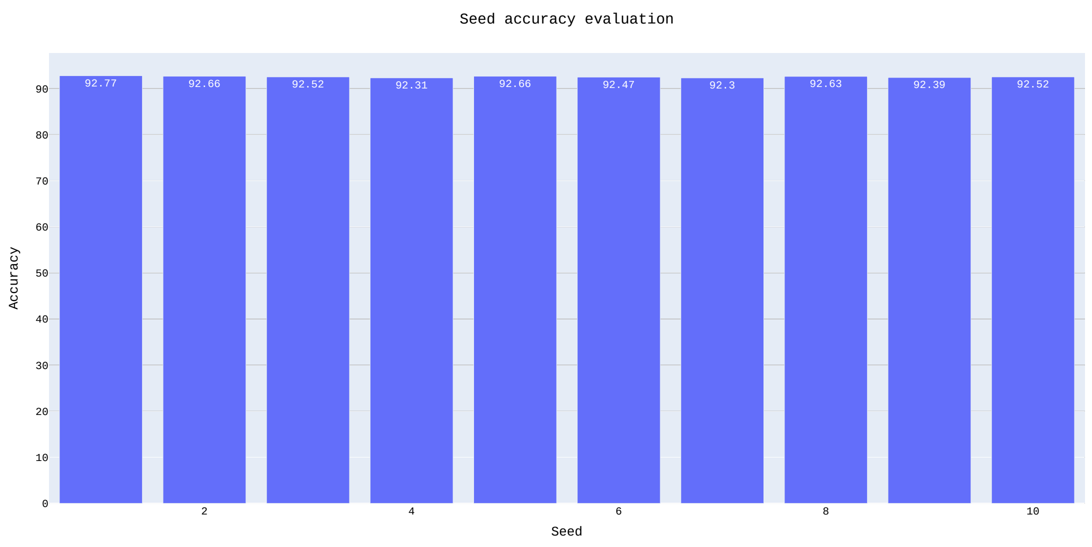


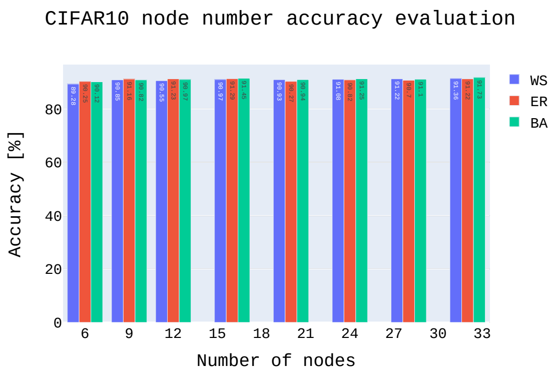
Figure 4.8: Seed variation impact on CIFAR10's accuracy. These are the *first generation* networks used on the optimization approaches in Section 3.7

In our implementation, we use NetworkX [26] to create the graphs. The functions allow us to use an argument *seed* to generate the graph. Unless the argument is passed, a global random number generator is used. We describe it if a specific seed is used in our networks.

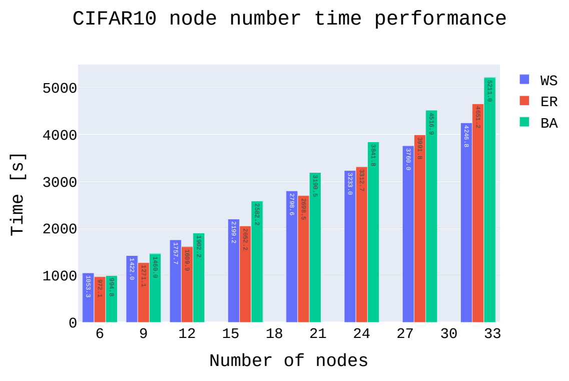
In Figure 4.8 we confirm that the seed s is not substantial enough to make a difference in the performance nor to motivate further analysis. The *network generator* described has $g(\theta, s)$ is not entirely accurate because its randomness stems mostly from the random graph models generating the network, present in the θ parameters.

4.2.3 Node Grid Search

We perform a grid search to evaluate the node count C in every dataset 3.5 as described in Section 3.6.2. The parameters θ are constant through all experiments, respective to the graph model in observation. The seed is not studied in this evaluation, the value given is $s = 1$.



(a) Accuracy (%)



(b) Time (s)

Figure 4.9: CIFAR10's node count C study: accuracy (left) and time (right)

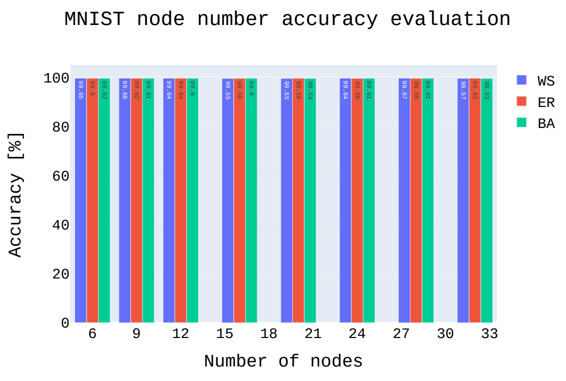
CIFAR10 holds a consistent performance except with 6 nodes, especially using the WS model. The model in that scenario does not have enough nodes to generate sufficient edges to provide a decent number of operations. The superior graph model is traded between the three models throughout the different C values. ER dominates lower values of C whilst BA the higher values on the left side of Figure 4.9.

On the right side of the same Figure (4.9), we can see the training times. These are very similar in all datasets, where the time increases with the size of the network. Note that each network has 3 graphs with the specified node count C.

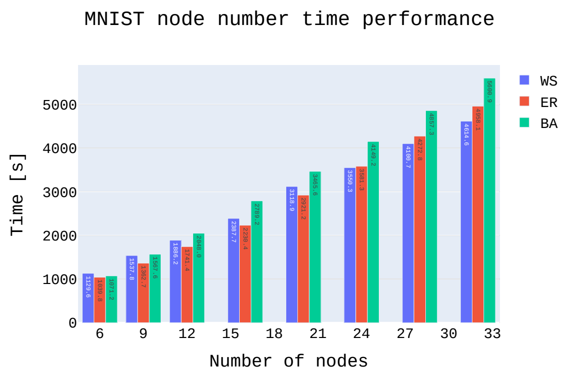
BA's training time is consistently higher than the other graph models, except for 6 nodes. Having such a low number of nodes, with $m = 5$ initial nodes, translates into a very simple network of only 5 connections per stage. This occurs because only one node is added to the algorithm. As the node count C increases, so does the difference $C - m$, which implies a growth in the number of connections and, consequently, the complexity of the network rises.

MNIST has a homogeneous accuracy through all tested node counts C, achieving a virtual state of the art performance. This result was expected in a simple dataset. The number of nodes is then inconsequential in this scenario if we disregard the training time. In Figure 4.10b, the duration of the learning process grows along with the increase of C.

Using 6 nodes in MNIST translates to a small, 18 node (plus 3 unique nodes to connect stages) network that achieves 99.6% accuracy with a training time of around 17 minutes. Although the dataset is not the hardest, this shows the potential of this type of network considering the wiring of each stage is defined randomly.

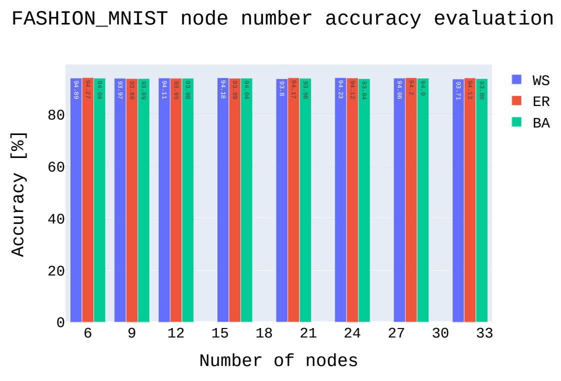


(a) Accuracy (%)

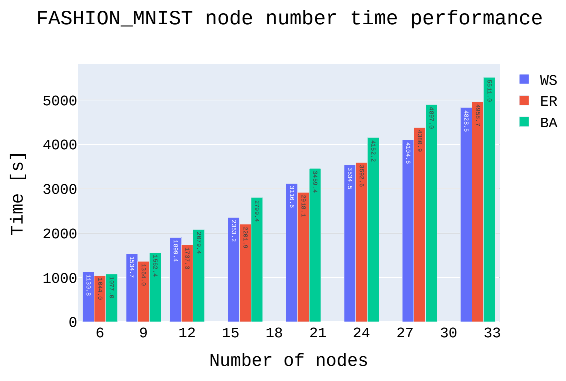


(b) Time (s)

Figure 4.10: MNIST's node count C study: accuracy (left) and time (right)



(a) Accuracy (%)



(b) Time (s)

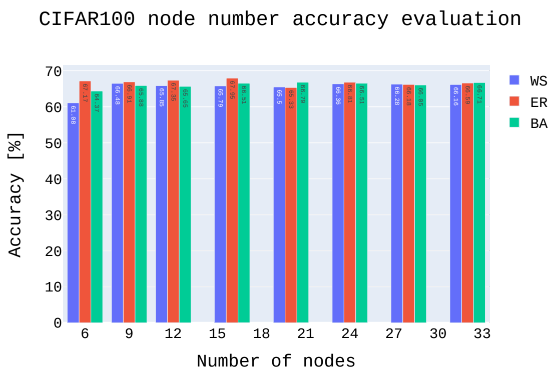
Figure 4.11: Fashion-MNIST's node count C study: accuracy (left) and time (right)

Analyzing the node count variation behavior in the Fashion-MNIST dataset, the BA graph model is trailing slightly behind its counterparts. The optimal combination is using ER with just 6 nodes, having both the least training time and the highest attained accuracy.

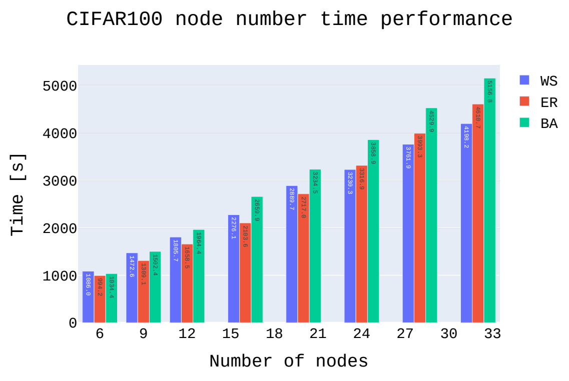
Has mentioned before MNIST is an easy dataset and Fashion-MNIST is an attempt to replace it. While images still have gray-scale images, no background, and 10 classes, the content requires a more complex feature extraction process. Figure 4.11a shows that our networks can still obtain results just 2 percentage points below the state of the art.

In Fashion-MNIST we start observing a tenuous dominance of the ER model, where $p = 0.2$. WS shows an almost identical effort but its training time is higher for a small number of nodes. Having $k = 4$ as the number of initial connections (2 on each side) to form a ring and a $p = 0.75$ results in a denser network than BA for a small number of nodes. This implies that the number of connections will be higher, hence the more complex network with a longer time for learning.

On CIFAR100 [43] we have a more complex dataset, portraying 100 classes that contain 600 images each. The state of the art within NAS is 86.4%, obtained with XNAS [58]. In that study, the



(a) Accuracy (%)



(b) Time (*s*)

Figure 4.12: CIFAR100's node count *C* study: accuracy (left) and time (right)

networks are trained for 1500 epochs (30 times as much as our method) and perform optimization for the learning rate. Our best node study network obtained 67.95%, which is within the top-30 reported results on the dataset. Considering our low network complexity and short training, this is still a decent result.

In Figure 4.12a, ER's dominance is asserted throughout the different number of nodes, except for when $C = 20$. ER creates every edge with an independent probability p, which in this case is equal to 0.2. The algorithm applies this probability to every edge possible, using all nodes. When C is low, the connection possibilities are scarce and combined with a low p, the network has a shallow complexity. This explains the low training times up until $C = 20$, where ER surpasses WS and coincidentally its performance is also lower.

The higher number of nodes might overwhelm the low probability p, outputting more complex networks, increasing training times. The WS parameters define a smoother curve in training times than ER, which has a steeper increase. The BA model describes a similar curve which is influenced by the increase of C.

The training durations are akin in every dataset, leading us to believe that the problem's context is independent and the graph generation model used and resulting network complexity are mostly responsible for the training time.

4.3 Optimization results

In this Section, we present our results for the optimization approaches described in Section 3.7. This was an iterative process, where the previous result defined the next step. We acknowledge that at each step there were numerous directions we could have taken.

4.3.1 Weight prediction

We first use a machine learning algorithm to predict the weights of the connections, shown in Figure 3.1. We use Grid Search to build our Random Forest Regressor. This tests the model with several parameters that we specify and returns the best one. The process can be done many times for parameter tuning. The parameters involved include:

- *n_estimators*: number of trees in the random forest;
- *max_features*: when splitting, this describes the number of features to be considered;
- *max_depth*: the limit for the tree expansion;
- *min_samples_split*: when in a node, this is the minimum number of samples needed to split;
- *min_samples_leaf*: the number of samples required for a node to be a leaf node;
- *bootstrap*: defines if the whole dataset is used.

In Figure 4.13 the importance of the main features is described, importances under 0.005 are not regarded and listed in Appendix B. Feature importance is a score assigned to input features that represents their usefulness in predicting the target variable. This score is obtained by permutation and reevaluation of the model. The highest importance is for *group in-degree centrality*, which evaluates a connection's relation to the rest of the graph. These features are described in Section 3.7.1.

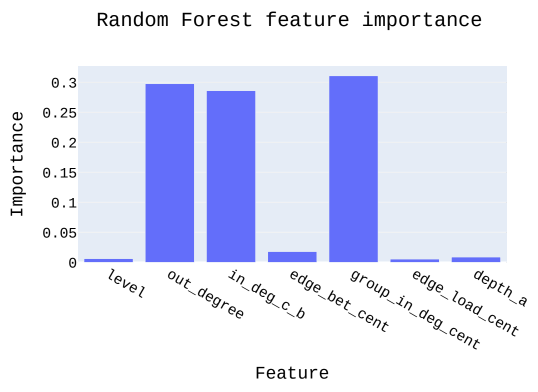


Figure 4.13: Considering features with an importance larger than 0.005

The regression model seems to favor features related to the flow of the connection, but also its role on the network. The level of the connection, i.e. its origin stage is important due to the different feature extraction tasks in each stage which result in a variation on the weights as scalars.

Exploring other Machine Learning algorithms like ElasticNet and Support Vector Machines yielded a higher Mean Root Squared Error. We opt to use Random Forest for the following experiments.

We first evaluate the model's performance on the same context problem, CIFAR10, while evaluating the optimizer in Figure 4.14. The default weight initialization is using 1.0 for all weights, its curve represents a normal training process. We compare this to three different optimizers, SGD with default 0.9 momentum, SGD with $momentum = 0.0$, and Adam.

These three curves use the model to predict their connection weights. The weights are placed in their respective connections and the training is done ordinarily.

We notice a convergence around the 30th epoch in all networks, this coincides with the learning rate scheduled to decrease. This is where the highest accuracy section is achieved and suggests that the learning rate and the learning rate schedule are strong candidates for hyperparameter tuning. The learning rate does not appear to be too low, the convergence rate is not slow, reaching a good performance in just 30 epochs. The scheduling of the learning rate is open to discussion, in this case, it may be too far apart. Using e.g. annealing could be an option but the behavior is not predictable and schedulers are not adaptable dataset-wise.

The SGD optimizer with default momentum is very similar to the default training (see Section 3.6). There are some "above the original curve" moments but in general it seems that there is no advantage in using predicted weights. The final accuracy has minimal differences.

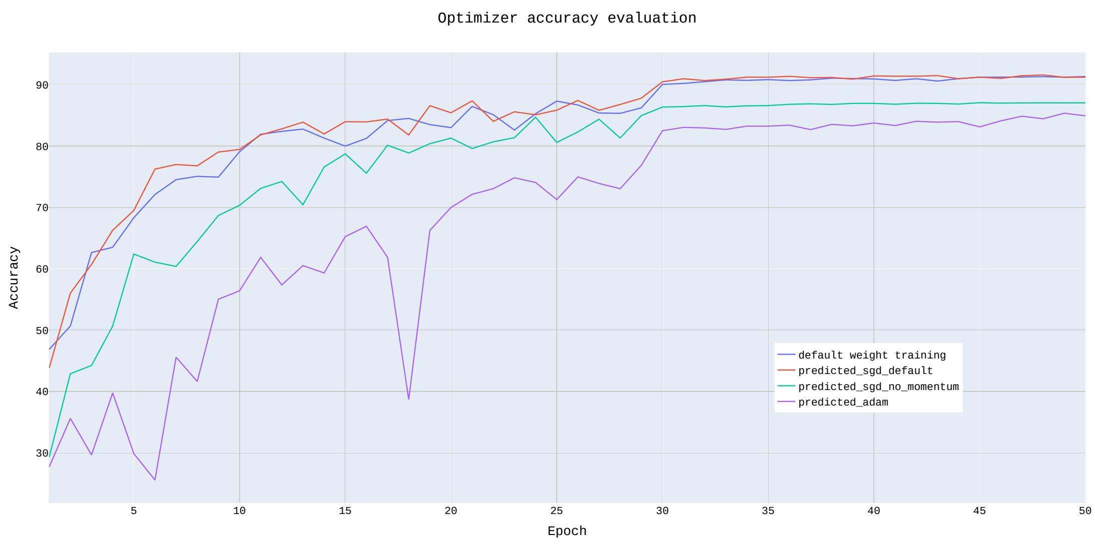


Figure 4.14: Optimizer influence on accuracy on the CIFAR10 [42] dataset

The absence of momentum encourages the optimizer to make slower progress along the local optima, resulting in a slower evolution. SGD with no momentum struggles to learn as quickly but still reaches a high accuracy. Performing a (grid) search on the momentum would be another possible hyperparameter to search on.

In this section we have now seen some training limitations defined by human design:

- learning rate;
- learning rate schedule;
- momentum.

Even though the authors do not mention directly the optimizer used, it is also another limitation to the original study. To perform hyperparameter tuning on these parameters would be an extensive task, which we do not have resources to perform.

4.3.1.1 Weight *shuffling*

To further evaluate the prediction of connection weights, we conduct an experiment where we create a new model for CIFAR10 and evaluate it. Then, we replace the weights with predicted weights and evaluate the model. Lastly, we run 100 iterations where we shuffle the same predicted weights throughout the connections and evaluate each combination.

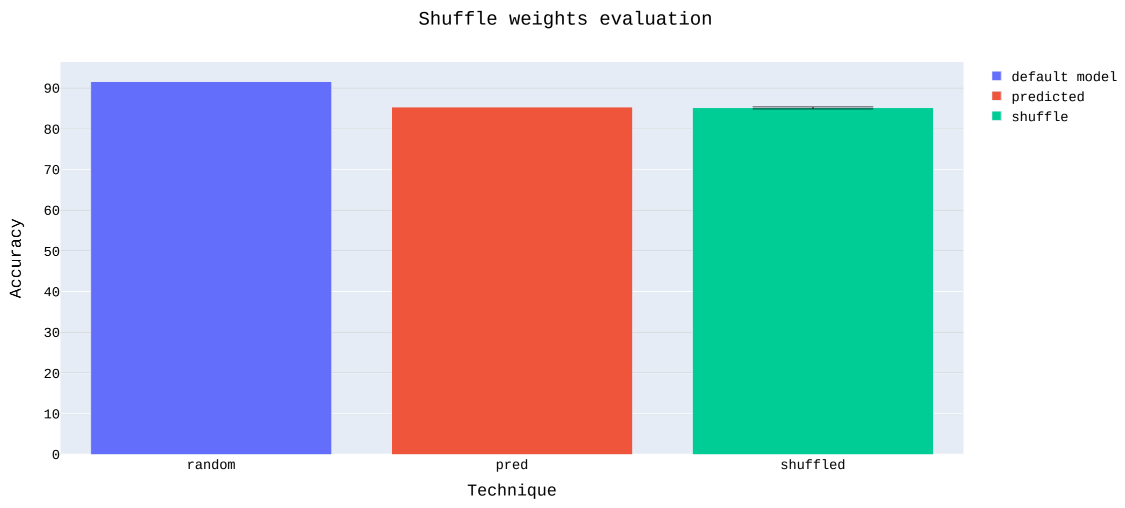


Figure 4.15: Evaluation for *shuffling* a connection weight prediction for 100 iterations in CIFAR10. The standard deviation in the shuffled bar is 0.25.

Figure 4.15 shows that the prediction evaluation is underperforming slightly compared to the trained network. The 100 shuffled instances have an almost identical performance as the predicted. The order of the weights is perhaps unimportant and their influence on the network minimal.

If we perform untrained evaluations on the models, the predicted weights have no advantage compared to a network with default initialization of 1 in the connection weights as shown in Figure A.5. This supports the statement that the impact of the connection weights in the model is insubstantial.

4.3.2 Weight variation

Our networks have on average about 180 connections, between the 3 graphs present which are dissimilar. The weights can vary between $[-1, 1]$ and their distribution might be different from graph to graph inside the same network. An aspect that influences the weight is its depth, i.e. the stage it belongs to. The features learned in each stage are distinct, so the weights between them cannot be compared.

In Figure 4.16 we have an histogram of the weights. They follow an approximately normal distribution with a mean of 0.109 and a standard deviation of 0.283. Using a normal distribution, we can initialize a network's connection weights with the generated values.

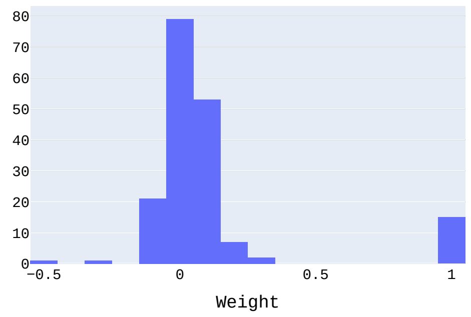


Figure 4.16: Example of weight distribution in a network, with standard deviation of 0.283 and mean of 0.109

The weights with a value equal to 1.0 represent the input connections of the *unique* output node. Since this node performs no convolution, its edges simply perform data flow. The weight 1.0 works as an identity function and when either using a normal distribution or shuffling the weights through all connections we keep these specific connections with weight equal to 1.0.

4.3.2.1 On the original dataset

Predicting weights for the same problem as the regression model was trained for is expected to have analogous results. In Figure 4.17 we observe just that. While most of the training has variations, after the learning rate update the curves are nearly indistinguishable.

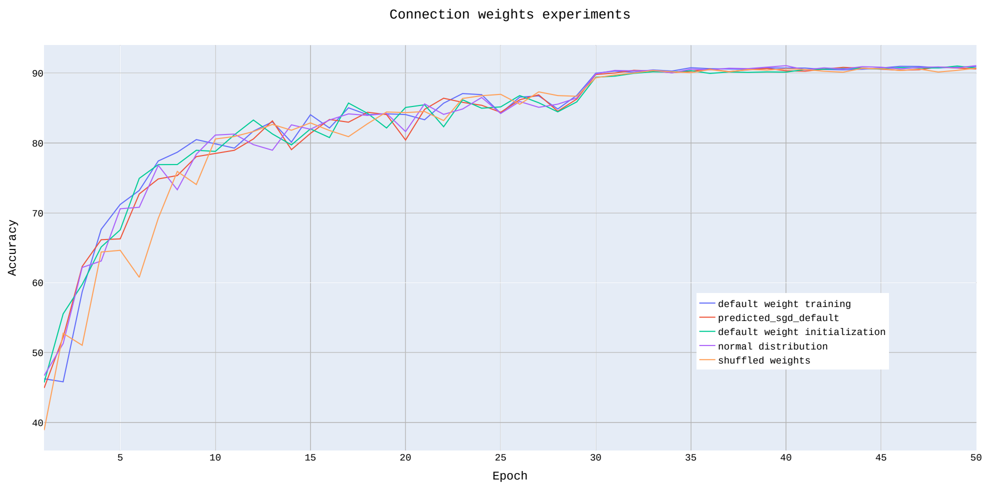


Figure 4.17: Accuracy evolution of several approaches on the CIFAR10 [42] dataset

The networks observed in Figure 4.17 are subject to weight *freezing* except the “default weight training”. The connection weights (predicted or not) are not learned throughout the training. The networks present are:

- *default weight training*: a normal training procedure is carried;
- *predicted_sgd_default*: connection weights are predicted and placed before training with a default SGD optimizer ($momentum = 0.9$);
- *default weight initialization*: connection weights are *frozen* in their default initialization of 1.0;
- *normal distribution*: connection weights are initialized from a normal distribution generator extracted from the *first generation* networks described in Section 3.7;
- *shuffled weights*: connection weights are taken from the *first generation* networks and are randomly *shuffled* and assigned.

The shuffled weights have a vague training variation compared to the others, which can be justified by existing the possibility of mixing weights from deeper stages with shallower stages. Other than that, we note no distinguishable differences in the training. We conclude that predicting the connection weights has a small impact on the network:

- random weight variation methods can still perform compared to weight prediction;
- *frozen* weights achieve good results, which means the influence of the connection weights on the network is minimal.

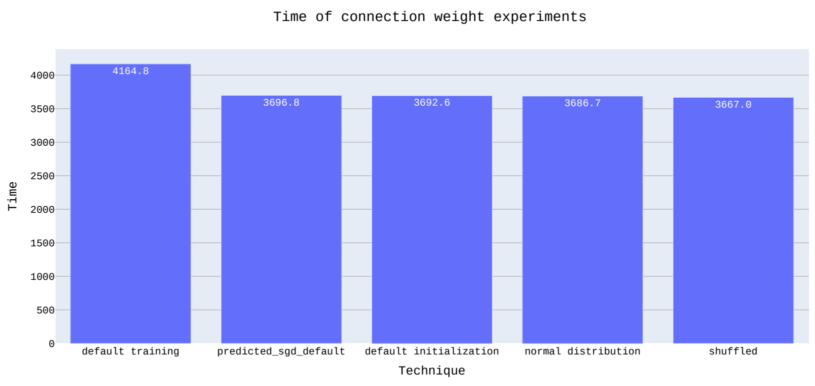


Figure 4.18: Training time of several approaches on the CIFAR10 [42] dataset

Considering the training times in Figure 4.18, we notice that the networks where we *freeze* the connection weights have a 10% drop. The presence of these weights in the network is fractional. Coupling this with their original operation, *data flow*, where no convolution is performed, the values of the weights are practically irrelevant. Either using random, predicted, or 1 value weights, the networks have equivalent training duration.

One take away from this experiment is that by *freezing* weights with value 1, which requires no previous computation or knowledge, can lead to a decent network. Bearing in mind that in the original study we have 250 epochs, reduced training time can have more impact than in our study.

4.3.2.2 On another problem

To test further our machine learning algorithm, we use the same procedure in all the datasets mentioned in Section 3.5. Here we present results for Fashion-MNIST, while the results of the other datasets are in Appendix A.

Compared to CIFAR10, in Fashion-MNIST we also obtain a result, after the 30th epoch (learning rate update), parallel in all approaches. The earlier training stages are quite varied.

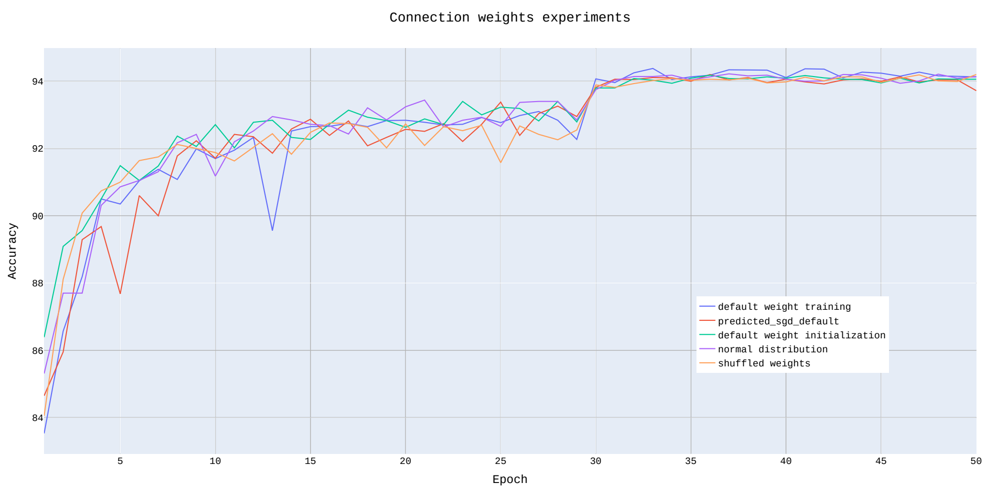


Figure 4.19: Training accuracy evolution of several approaches on the Fashion-MNIST [84] dataset

In Figure 4.19, all the techniques have a cut above start than the default weight training. For the most part, the training has homogeneous accuracy variations. Surprisingly, the default weight initialization of 1 weight values holds the best start. Although converging eventually, our regression model underperformed in the first 10 epochs. The fact that this approach still converges to a good performance confirms the small influence of the weights of the connections in the network.

The *shuffled* weights had an improvement compared to its performance in CIFAR10, which is interesting considering the weights used were trained on CIFAR10 with the 10 *first generation* networks. Similarly, the normal distribution curve has an “ordinary” performance, strengthening our deduction that the connection weights are innocuous.

Figure 4.20 supports the previous claim on CIFAR10 that the presence of connection weights on the networks is fractional, again around 10%. It is counterproductive to predict weights for a fragmentary time gain. Likewise, other approaches as the normal distribution and the weight shuffling require previous knowledge. Nevertheless, using a default initialization and *freezing* the connection weights we obtain good results anyway.

In Table 4.2 we list the results obtained from the 4 datasets combined with the 4 techniques plus the default network training. The results quote the last epoch’s accuracy (50th) and not the highest attained accuracy during training. We can see that in each dataset the differences between each approach are almost negligible. Having good results in other datasets should not happen

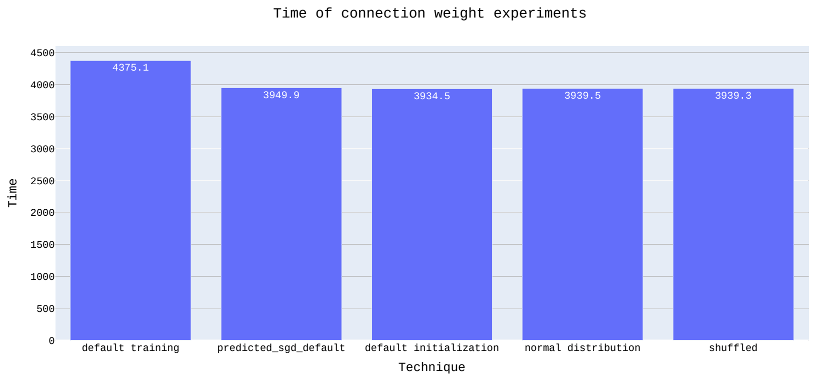


Figure 4.20: Training time of several approaches on the Fashion-MNIST [84] dataset

approach	**MNIST**	**Fashion-MNIST**	**CIFAR10**	**CIFAR100**
default weight training	99.68	94.13	90.93	67.52
predicting weights	99.60	93.72	90.60	66.95
default weight initialization	99.55	94.06	90.73	67.22
weight distribution	99.65	94.14	91.08	66.63
shuffled weights	99.68	94.20	90.72	67.09

Table 4.2: Performance of several attempted optimization approaches

in theory, however, this supports the statement that the influence of the connection weights is unimportant.

The default weight initialization proves to be the most efficient method if we bear in mind that it requires no previous computation like predicting the connection weights.

4.3.3 Weight *sharing*

In this experiment, we reuse nodes from CIFAR10 into the networks. The nodes match each stage of the network to prevent to some extent the differences in the learned features. We call these networks “cooked” weights.

For Fashion-MNIST, in Figure 4.21, we *freeze* connection weights in all networks except *“cooked” weights* and *default weight training*. The other datasets hold the same approaches and are present in Appendix A.

The “cooked” network does not have an evolution comparable to the other networks although obtaining a similar final score. This indicates that the network still learns the node weights (convolutional blocks) despite the learning process in the *first generation* network.

Freezing these node weights results in a significantly worse training evolution and outcome. This is represented by the yellow line, where the only trained weights are the first two stages in the Table 3.1 and the connection weights in the *random wiring* stages. The (poor) evolution in training is mostly influenced by the first two stages which perform convolutions. The influence of the connection weights in this case is also minimal.

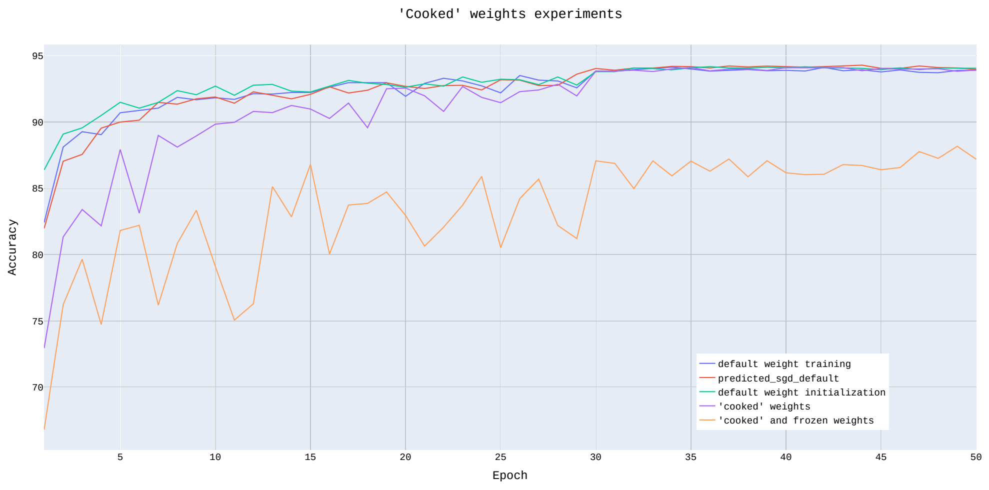


Figure 4.21: Training evolution of *shared* weights on the Fashion-MNIST [84] dataset

4.3.4 Main results

At the *network generator* level, we can conclude that parameters that make the network more complex translate into extensive training. Less complex and faster-trained networks can achieve a state of the art performance in our datasets. Some of the best parameters we found differ from the original study in [86]. We believe using different datasets has an influence but if their experiments were reproduced in the same conditions, the *stochastic* nature of the *network generator* would result in different best-performing parameters.

Regarding the node study in Section 4.2.3, due to the size of our datasets, we show that a small fraction of the original node number $C = 32$ can achieve the same results. The training time is also significantly smaller because the complexity of the networks is also reduced. We believe, since there is no foundation for the original node number, that the number of nodes in a stage should be explored in ImageNet. The training time would be reduced and we would be able to determine an optimal node number for randomly wired neural networks.

Our optimizations did not improve the accuracy or convergence rate of the neural networks but they reduced slightly the training time. Similar performance using different techniques, including randomly generated weights for the connections, lead us to believe that the weights of the connections have a negligible influence on randomly wired neural networks.

In this final version of the document, we would like to report that combining these conditions:

- connection node weight prediction;
- ER graph generator model because it was the most consistent in our study;
- $C = 6$ nodes for a lower complexity network;

we have obtained 54% accuracy on the ImageNet [67] dataset in just 61 epochs, which is a fraction of the 250 epochs of the original study. We will try to continue to obtain results on this dataset and write a paper. The training is still extensive because due to low GPU memory we can only use a batch size of 32 – considerably low than the original 128.

Chapter 5

Conclusions and Future Work

The base work is embedded in one of the several NAS techniques – random search –, taking a different direction. Instead of using established NAS algorithms as baselines, different "truly" random generator designs are implemented – the *network generator* is based on random graph models from graph theory. In line with those approaches, the goal is to design networks that learn features instead of designing features. The networks are not random from scratch, only the wiring design suffers variation while the other aspects are fixed. The complete randomization is far from being an achievement. However, the search space is less constrained than in other works.

5.1 Conclusions

The NAS network generator constraints wiring patterns to a small subset of possible graphs. The research we evaluate explores the design of new network generators by loosening that constraint using randomly wired neural networks sampled from a random generation process. The desire is a shift from the traditional network engineering to a network generator engineering, which requires less effort in the overall design process.

The method proposed obtained interesting results [86] considering the low human intervention, thus we first and foremost reproduce and confirm those results and, through our experiments, improve our knowledge on why and how they happen. It is still hard nowadays to understand how neural networks work, how they make decisions, and how they learn.

Our empirical study has results that differ from the authors, in terms of which parameters to use for the graph generating models. There is an influence of the datasets because we do not make observations on ImageNet [67]. However, we used datasets that are mostly alike in size and difficulty and we concluded that the best performing parameters also vary between them. Thus, the randomness of the graph models is not predictable and the results reported by the authors could have been achieved by chance, i.e. if reproduced several times the high-performing parameter combination could be inconsistent.

Our node study touches one of the weak points of the design, where the default node count is 32 for all networks. In our datasets, we observe that with a significantly reduced number (6, 9, or 12 nodes) we obtain the same performance as we do with 32. Certainly, using smaller datasets has a role in this conclusion however, we expose the weakness in the original study where the node count is not explored.

Our optimization approaches help us understand how the networks function internally. We conclude that the connection weights in Section 3.1 have a reduced impact on the network. The nodes or computational blocks conduct the majority of the important tasks in the network. The wiring is however still important because from our empirical study we see that networks containing graphs with small complexity and little to no randomness do not perform well. We conclude then that the weights in the connections are not relevant, as using a value of 1 achieves identical results as a normal training process.

5.2 Contributions

Our research gave us a different feeling from the original paper, where the authors emphasize how constraints are loosened in randomly wired neural networks. The node operations are uniform and do not affect the input size. The heaviest constraint is the definition of stages. Although having two *regimes*, this process is manual and has an implicit bias from the authors. Exploring another set of *regimes* could reveal different results.

As we concluded before, the **residual** connections are the only true relaxation offered by this type of network. The graph generation model used and consequent wiring patterns is the only real degree of freedom. We explored it and conclude that the results may vary in function with the context of the problem. Nevertheless, being a *stochastic* network generator suggests that any results we obtain are hard to reproduce.

5.3 Future Work

Firstly, we recommend exploring the node count C for the ImageNet [67] dataset. Using a lower number of nodes reduces the training time while keeping a similar performance to the reported achievements.

Additional hyperparameter optimization should also be considered and we believe it would be beneficial. This would be an extensive study that we do not have the resources to perform but we believe the outcome to be worthy.

Regarding our optimization approaches, several investments can be made:

- we can use optimizer characteristics and the optimizer itself as a feature for the machine learning algorithm. This can affect the weight to optimize;
- evaluating the dataset influence (the domain of the problem) could see the domain liable to use as a feature.

Attempting to expand the machine learning algorithm to the nodes themselves could be a possibility. Trying *clustering* as a classification problem to extract types of kernels present in the nodes. This would have a larger impact on the network and a significant boost in training would be expected, at least in the same domain.

Exploring the behavior of the network by removing 1 and/or 2 of the *random wire* stages would help define new *regimes* or at least give new knowledge about how these networks learn.

Even though our primary focus is on Computer Vision, we acknowledge that other scientific fields can be influenced by this research and we could have made use of them. On a similar note, other types of neural networks beyond CNN's can be focused on, such as the mentioned 2.2 Recurrent Neural Networks and Feedforward networks.

Regardless of limitations, we confirm that the use of a *network generator* with less human design involved is a viable solution. To the best of our knowledge this is the first known study in this direction, the research is still fresh but the results are encouraging. The continuous exploration of the concept of a *network generator* has the potential to yield constant state of the art results in the future.

Appendix A

Appendix 1

In this Appendix, we present some extra results we have obtained.

A.1 Empirical Study

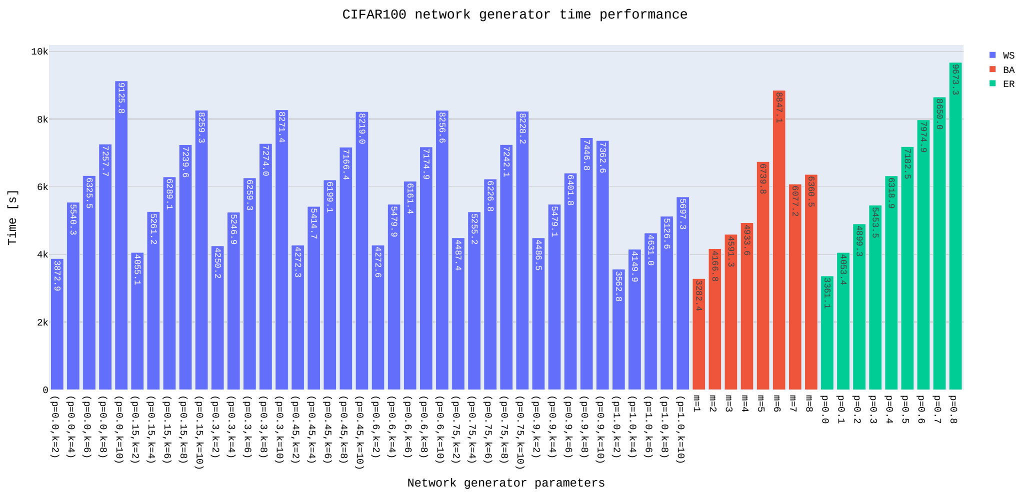


Figure A.1: θ *network generator's* parameters variation impact on CIFAR100's accuracy

A.2 Optimization Approaches

A.2.1 Weight Variation

A.2.1.1 Weight *shuffling*

A.2.2 Weight *Sharing*

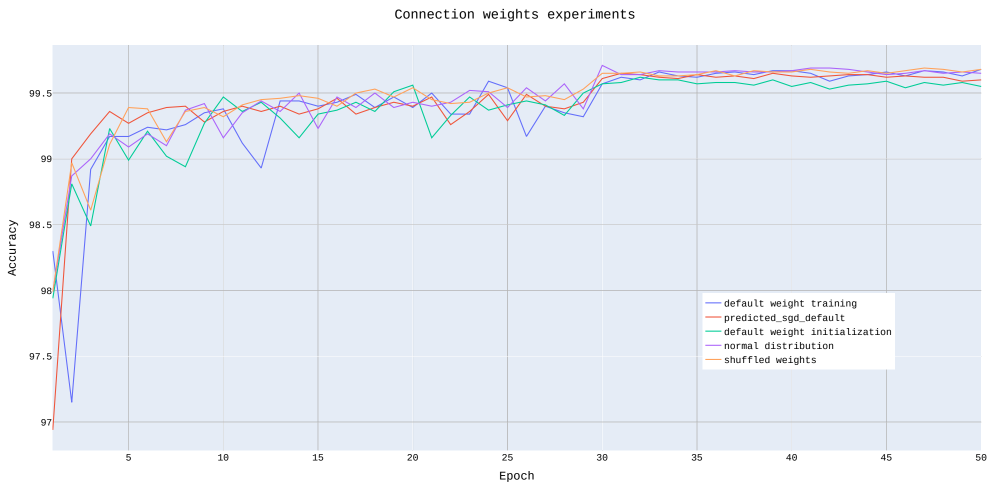


Figure A.2: Training accuracy evolution of several approaches on the MNIST [46] dataset

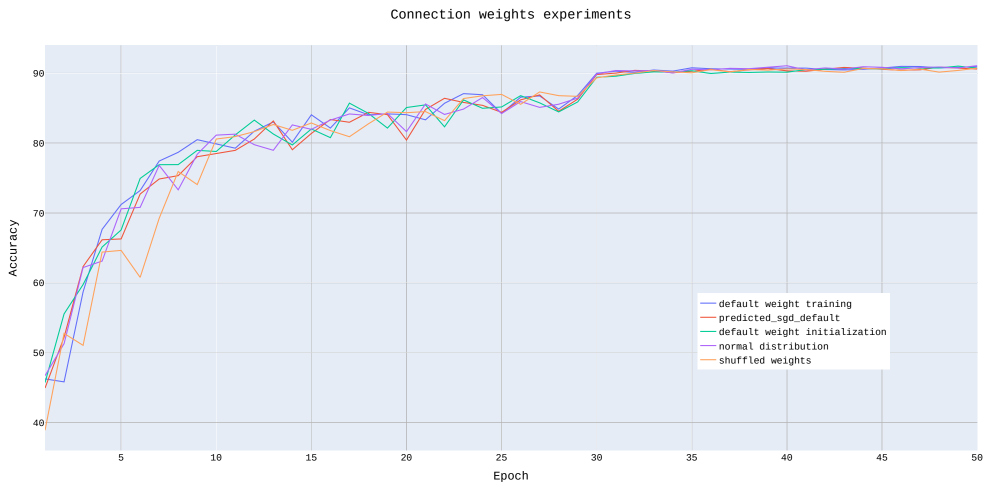


Figure A.3: Training accuracy evolution of several approaches on the CIFAR10 [42] dataset

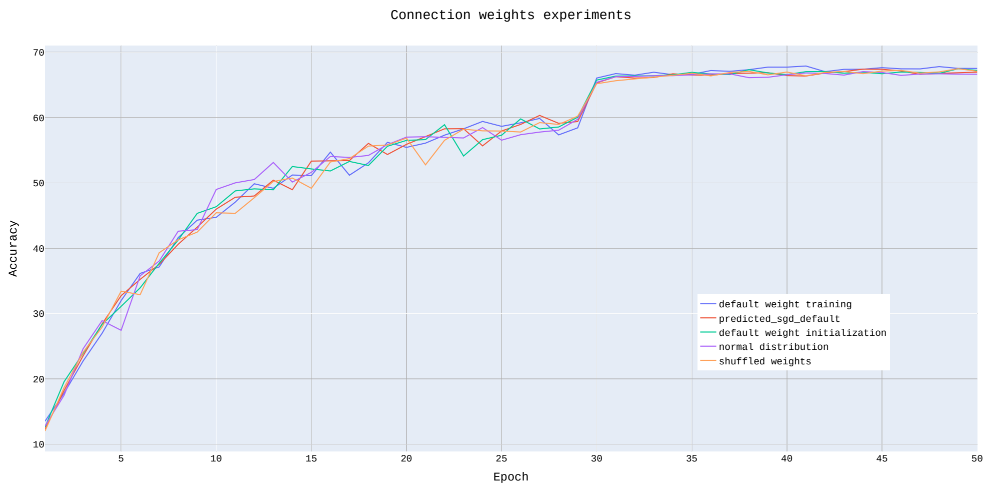


Figure A.4: Training accuracy evolution of several approaches on the CIFAR100 [43] dataset

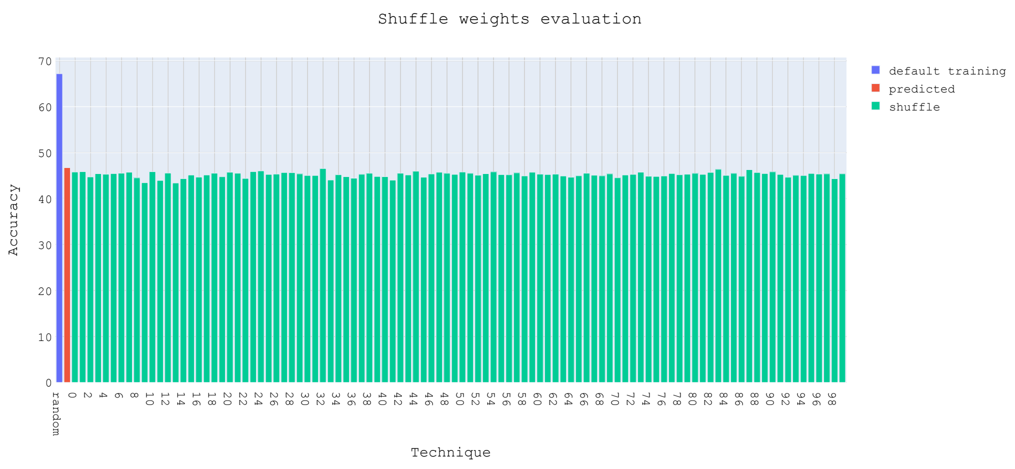


Figure A.5: Evaluation for *shuffling* a connection weight prediction for 100 iterations in CIFAR100

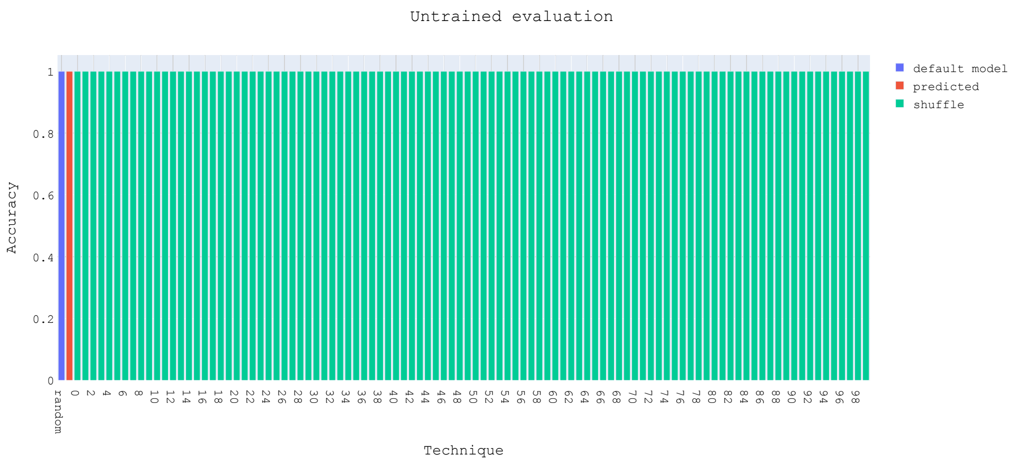


Figure A.6: Untrained valuation for *shuffling* a connection weight prediction for 100 iterations in CIFAR100

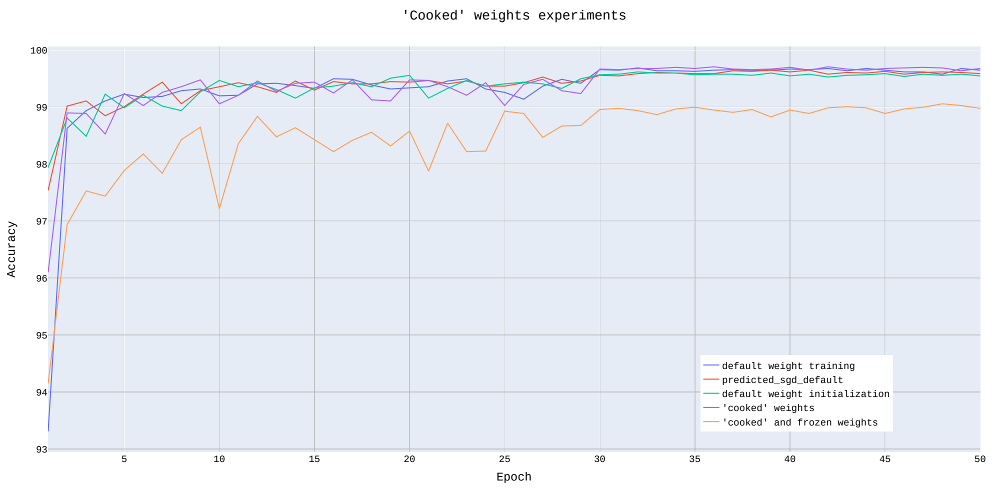


Figure A.7: Training evolution of *shared* weights on the MNIST [46] dataset

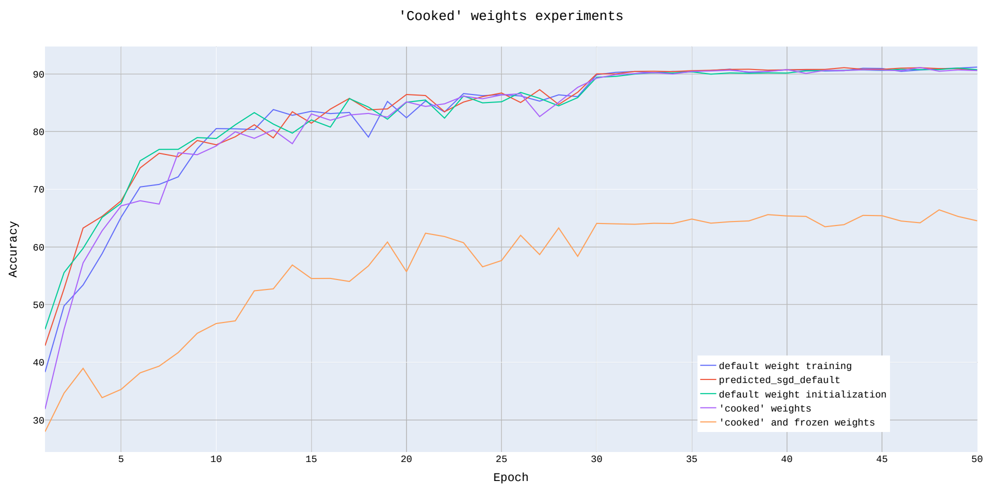


Figure A.8: Training evolution of *shared* weights on the CIFAR10 [42] dataset

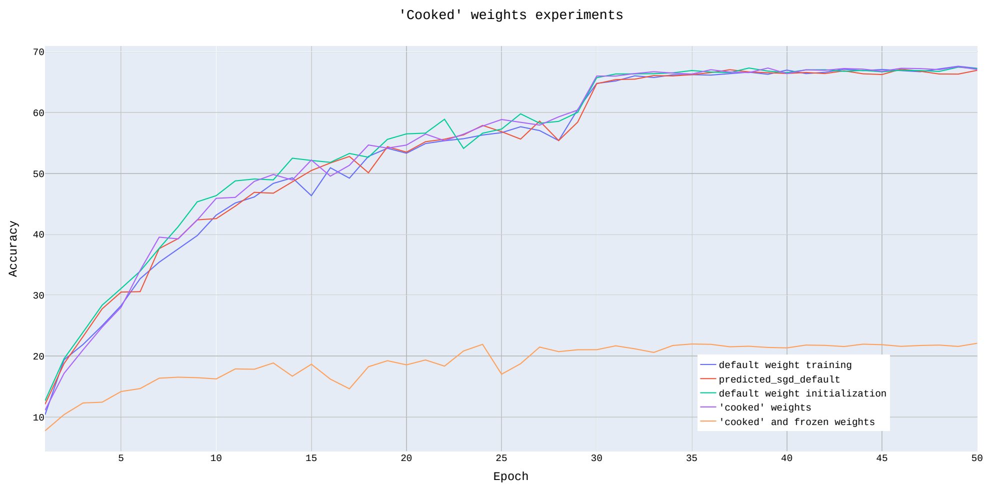


Figure A.9: Training evolution of *shared* weights on the CIFAR100 [43] dataset

Appendix B

Appendix 1

In this Appendix, we present the full features used in our Random Forest regressor, mostly calculated using NetworkX [26].

B.1 Graph Metrics

Feature	**Description**	**NetworkX**
Level	Stage of the connection	
In-degree	Input degree of node A	
Out-degree	Output degree of node B	
Degree centrality A	Proportion of nodes node A is connected to	x
Degree centrality B	Proportion of nodes node B is connected to	x
Closeness centrality A	Average distance from the shortest path to node A	x
Closeness centrality B	Average distance from the shortest path to node B	x
Betweenness centrality A	Number of paths that use node A	x
Betweenness centrality B	Number of paths that use node B	x
Current flow centrality A	Electrical-current model for closeness in node A	x
Current flow centrality B	Electrical-current model for closeness in node B	x
Current flow betweenness centrality A	Electrical-current model for betweenness in node A	x
Current flow betweenness centrality B	Electrical-current model for betweenness in node A	x
Eigen centrality A	Measure the influence of node A in the graph	x
Eigen centrality B	Measure the influence of node B in the graph	x
Katz centrality A	Measure the influence of node A in the graph through neighbours	x
Katz centrality B	Measure the influence of node B in the graph through neighbours	x
Communicability betweenness centrality A	Centrality of node A rooted on communicability betweenness	x
Communicability betweenness centrality B	Centrality of node B rooted on communicability betweenness	x
Load centrality A	Proportion of shortest paths that use node A	x
Load centrality B	Proportion of shortest paths that use node B	x
Page rank A	Evaluates quantity and quality of connections to node A	x
Page rank B	Evaluates quantity and quality of connections to node B	x
Dispersion	Evaluates the connection between node A and node B	x
Communicability	Number of different cycles possible between node A and node B	x
Node connectivity	Number of nodes needed to be removed to disconnect the graph (node A: source, node B: target)	x
Edge connectivity	Number of edges needed to be removed to disconnect the graph (node A: source, node B: target)	x
Average neighbor degree A	Computes the average degree of node A's neighborhood	x
Average neighbor degree B	Computes the average degree of node B's neighborhood	x
In-degree centrality A	Proportion of nodes node A's incoming edges are connected to	x
In-degree centrality B	Proportion of nodes node B's incoming edges are connected to	x
Out-degree centrality A	Proportion of nodes node A's outgoing edges are connected to	x
Out-degree centrality B	Proportion of nodes node B's outgoing edges are connected to	x
Edge betweenness centrality	Edge frequency in shortest paths	x
Edge current flow betweenness centrality	Electrical-current model for betweenness in the edge A-B	x
Group betweenness centrality	Proportion of pairs shortest path passing through vertexes in group (A,B)	x
Group closeness centrality	Measures how close group (A,B) is to other nodes	x
Group degree centrality	Proportion of outer group elements connected to group (A, B) members	x
Group in-degree centrality	Proportion of outer group elements connected to group (A, B) members by incoming edges	x
Group out-degree centrality	Proportion of outer group elements connected to group (A, B) members by outgoing edges	x
Edge load centrality	Adds the number of shortest paths which cross edge (A, B)	x
Simrank similarity	Metric that computes if two nodes are similar (A,B) by analyzing their references	x
Volume	Sum of the out-degrees of group (A,B)	x
Depth A	Distance from input node to A	
Depth B	Distance from input node to B	

Table B.1: This table describes the features used in the machine learning algorithm. The features marked with "x" in the NetworkX column were directly obtained from the NetworkX library [26]. The remaining features were calculated.

Appendix C

Appendix 3

In this Appendix, we present some image samples from the datasets we used.

C.1 Dataset Samples

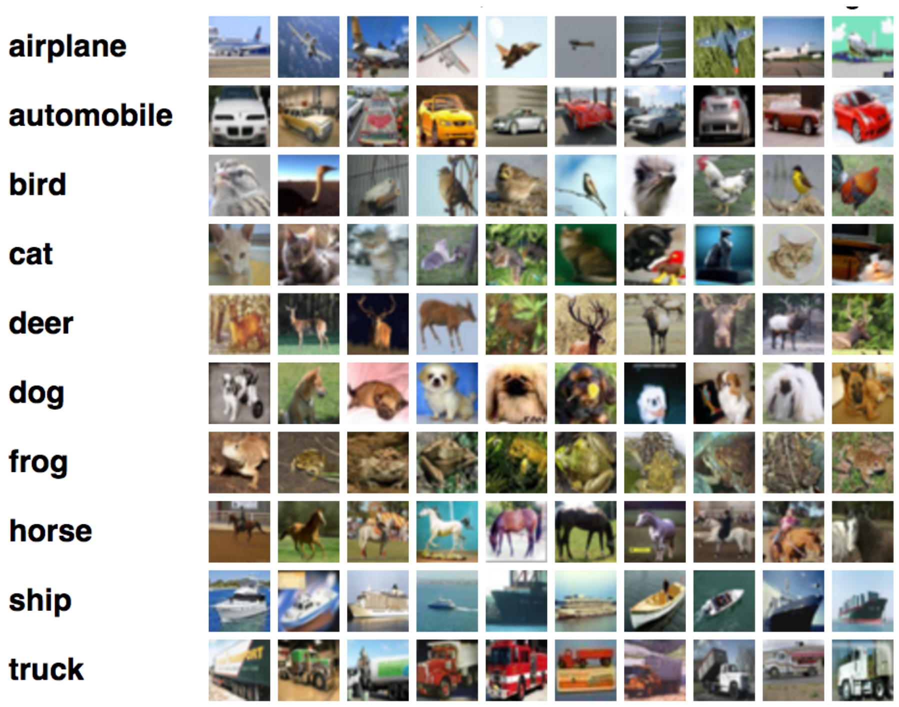


Figure C.1: CIFAR10's [42] image samples

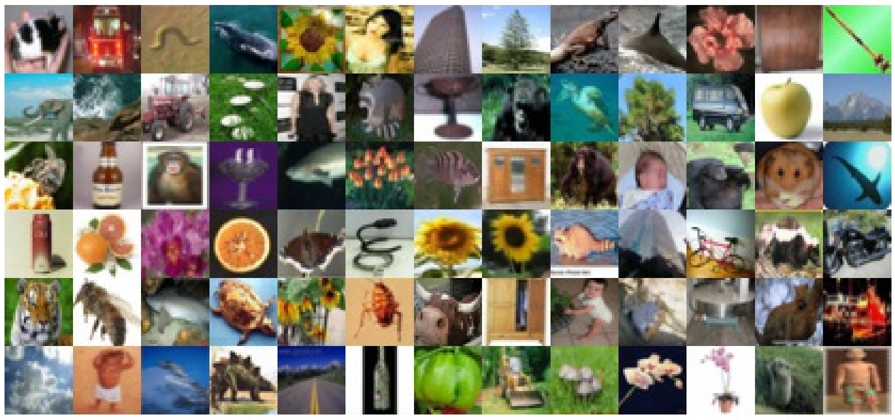


Figure C.2: CIFAR100's [43] image samples

Figure C.3: MNIST's [46] image samples

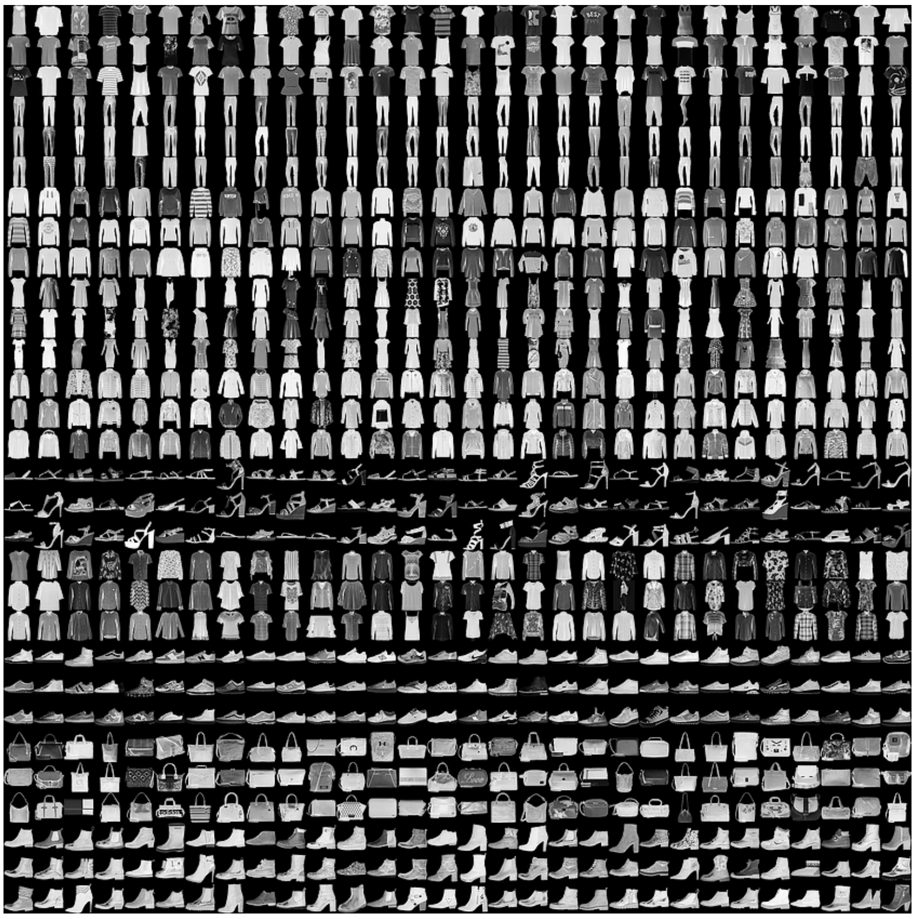


Figure C.4: Fashion-MNIST's [84] image samples

References

[1] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147 – 169, 1985.

[2] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002.

[3] Md. Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Mahmudul Hasan, Brian C. Van Esesn, Abdul A. S. Awwal, and Vijayan K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, abs/1803.01164, 2018.

[4] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *CoRR*, abs/1803.01271, 2018.

[5] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 71:036113, 04 2005.

[6] Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[7] Léon Bottou. *Stochastic Gradient Descent Tricks*, volume 7700, pages 421–436. Springer, 01 2012.

[8] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[9] Adam Byerly, Tatiana Kalganova, and Ian Dear. A branching and merging convolutional network with homogeneous filter capsules, 2020.

[10] X. Cao, F. Zhou, L. Xu, D. Meng, Z. Xu, and J. Paisley. Hyperspectral image classification with markov random fields and a convolutional neural network. *IEEE Transactions on Image Processing*, 27(5):2354–2367, May 2018.

[11] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[12] Xiangxiang Chu, Bo Zhang, Ruijun Xu, and Jixiang Li. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. *CoRR*, abs/1907.01845, 2019.

[13] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[14] D-X-Y. D-x-y/awesome-nas, Jan 2020.

[15] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four gpu hours. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2019.

[16] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2020.

[17] Narsi Reddy Donthi Reddy, Ajita Rattani, and Reza Derakhshani. Comparison of deep learning models for biometric-based mobile user authentication. In *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, 10 2018.

[18] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. In *On the evolution of random graphs*, 1984.

[19] Z. M. Fadlullah, F. Tang, B. Mao, N. Kato, O. Akashi, T. Inoue, and K. Mizutani. State-of-the-art deep learning: Evolving machine intelligence toward tomorrow's intelligent network traffic control systems. *IEEE Communications Surveys Tutorials*, 19(4):2432–2455, Fourthquarter 2017.

[20] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *CoRR*, abs/1811.12560, 2018.

[21] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*, 1:119–130, 1988.

[22] Adam Gaier and David Ha. Weight agnostic neural networks. *CoRR*, abs/1906.04358, 2019.

[23] F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194 vol.3, July 2000.

[24] Google-Research. google-research/nasbench, Nov 2019.

[25] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[26] Aric Hagberg, Pieter Swart, and Dan Schult. Graph generators, Oct 2019.

[27] Ethan Harris, Antonia Marcu, Matthew Painter, Mahesan Niranjan, Adam Prügel-Bennett, and Jonathon Hare. Fmix: Enhancing mixed sample data augmentation, 2020.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[29] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *ArXiv*, abs/1908.00709, 2019.

[30] Eloisa Herrera. *Rodent Zic Genes in Neural Network Wiring*, volume 1046, pages 209–230. Springer, 02 2018.

[31] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[32] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey, 2020.

[33] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[34] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[35] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018.

[36] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[37] Michael I. Jordan. *Chapter 25 - Serial Order: A Parallel Distributed Processing Approach*, volume 121 of *Advances in Psychology*. North-Holland, 1997.

[38] Myeongjun Kim. Randomly wired neural networks pytorch. `https://github.com/leaderj1001/RandWireNN`, May 2019.

[39] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[40] Ron Kohavi and Foster Provost. Glossary of terms. *Machine Learning*, 2:271–274, 01 1998.

[41] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning, 2019.

[42] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). *Canadian Institute for Advanced Research*, 2009.

[43] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 (canadian institute for advanced research). *Canadian Institute for Advanced Research*, 2009.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[45] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[46] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. *NaN*, 2010.

[47] Jure Leskovec and Eric Horvitz. Planetary-scale views on an instant-messaging network, 2008.

[48] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. *CoRR*, abs/1902.07638, 2019.

[49] Grace W. Lindsay and Kenneth D. Miller. How biological attention mechanisms improve task performance in a large-scale visual system model. *bioRxiv*, 2018.

[50] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *CoRR*, abs/1712.00559, 2017.

[51] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[52] P. Louridas and C. Ebert. Machine learning. *IEEE Software*, 33(5):110–115, Sep. 2016.

[53] John McCarthy. Professor sir james lighthill, FRS. artificial intelligence: A general survey. *Artif. Intell.*, 5(3):317–322, 1974.

[54] Warren S. McCulloch and Walter Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity*, page 15–27. MIT Press, Cambridge, MA, USA, 1988.

[55] Marvin L. Minsky and Seymour A. Papert. *Perceptrons: Expanded Edition*. MIT Press, Cambridge, MA, USA, 1988.

[56] M.L. Minsky. *Theory of Neural-analog Reinforcement Systems and Its Application to the Brain Model Problem*. Princeton University., 1954.

[57] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[58] Niv Nayman, Asaf Noy, Tal Ridnik, Itamar Friedman, Rong Jin, and Lihi Zelnik-Manor. XNAS: neural architecture search with expert advice. *CoRR*, abs/1906.08031, 2019.

[59] NVIDIA. Introducing nvidia geforce gtx 1080 graphics card.

[60] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[62] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018.

[63] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, Inc., USA, 1997.

[64] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 4780–4789, 2019.

[65] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[66] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[67] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[68] Izabela Samborska, Vladimir Aleksandrov, Leszek Sieczko, Bożena Kornatowska, Vasilij Goltsev, Magdalena Cetner, and Hazem Kalaji. Artificial neural networks and their application in biological and agricultural research. *NanoPhotoBioSciences*, 02:2347–7342, 01 2014.

[69] Christian Sciuto, Kaicheng Yu, Martin Jaggi, Claudiu Musat, and Mathieu Salzmann. Evaluating the search phase of neural architecture search. *CoRR*, abs/1902.08142, 2019.

[70] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv 1409.1556*, 09 2014.

[71] Rajat Kumar Sinha, Ruchi Pandey, and Rohan Pattnaik. Deep learning for computer vision tasks: A review. *CoRR*, abs/1804.03928, 2018.

[72] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[73] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[74] Google Dev Team. Cloud automl documentation | google cloud.

[75] Christof Teuscher. *Intelligent Machinery*, pages 17–62. Springer London, London, 2002.

[76] Lukas Tuggener, Mohammadreza Amirian, Katharina Rombach, Stefan Lörwald, Anastasia Varlet, Christian Westermann, and Thilo Stadelmann. Automated machine learning in practice: State of the art and recent results. *CoRR*, abs/1907.08392, 2019.

[77] Ubuntu. Ubuntu 18.04 release.

[78] Gerard Jacques van Wyk and Anna Sergeevna Bosman. Evolutionary neural architecture search for image restoration. *CoRR*, abs/1812.05866, 2018.

[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[80] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational Intelligence and Neuroscience*, 2018:1–13, 02 2018.

[81] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[82] Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search, 2019.

[83] Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. Discovering neural wirings. *ArXiv*, abs/1906.00586, 2019.

[84] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[85] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.

[86] Saining Xie, Alexander Kirillov, Ross B. Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. *CoRR*, abs/1904.01569, 2019.

[87] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[88] Chris Ying, Aaron Klein, Esteban Real, Eric Christiansen, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. *CoRR*, abs/1902.09635, 2019.

[89] Anthony M. Zador. A critique of pure learning: What artificial neural networks can learn from animal brains. *bioRxiv*, 2019.

[90] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[91] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.