Exploring Alternatives to Policy Search

Author: Carlos Güemes Palau

Thesis Supervisor: Albert Cabellos Aparicio - UPC

Tutor: Cecilio Angulo Bahón - UPC

MASTER IN ARTIFICIAL INTELLIGENCE

January 2022

UNIVERSITAT DE BARCELONA

FACULTAT DE MATEMÀTIQUES I INFORMÀTICA

UNIVERSITAT DE BARCELONA (UB)

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona

FIB

FACULTAT D'INFORMATICA DE BARCELONA (FIB)

UNIVERSITAT POLITECNICA DE CATALUNYA (UPC) – BarcelonaTech

UNIVERSITAT

ROVIRA i VIRGILI

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA (ETSE)

UNIVERSITAT ROVIRA I VIRGILI (URV)

Contents

List of Figures 4

List of Tables 4

1 Introduction 7

2 Background and State of the Art 9
- 2.1 Deep Reinforcement Learning (DRL) 9
- 2.2 Evolution Strategies 11
- 2.3 Graph Neural Networks 12

3 Thesis Objectives and Goals 15

4 Proposed solution 17
- 4.1 Evolution Strategies and RL 17
- 4.2 Modifications to the parallelized solution 21
  - 4.2.1 Reward regularization and fitness shaping 21
  - 4.2.2 Mirror sampling 21
  - 4.2.3 Adding noise to the action probabilities 22
  - 4.2.4 Precomputed noise 22
  - 4.2.5 Distribution of perturbations among workers 22
- 4.3 Design of our proposed solution 23

5 Use case: Computer Network Resource Allocation 26

6 Evaluation 28
- 6.1 Evaluation settings 28
- 6.2 Analysis of ES hyperparameters 29
- 6.3 Performance of ES versus PPO 33
- 6.4 Analyzing ES scalability over the number of workers 38
- 6.5 Summary of results 39

7 Conclusions 42

References 44

List of Algorithms

1 Natural Evolution Strategies [17] 17
2 Parallelized Natural Evolution Strategies [17] 18
3 Improved Parallelized Natural Evolution Strategies 23

List of Figures

1 Interactions between the agent and the environment in RL 9
2 Effect on choosing an adequate noise level for generating perturbations. . . 19
3 Diagram detailing the interactions between the agent and the environment in the network routing problem [22] 27
4 Representation of the graph used in the MPNN [22] 27
5 Effect on the number of perturbations 30
6 Effect of the standard deviation for the mutations 31
7 Effect of the standard deviation for the noise to be added to the obtained action probabilities 32
8 Effect of using ES with varying number of workers over the NSFNET topology 34
9 Effect of using ES with varying number of workers over the GÉANT2 topology 35
10 Effect of using ES with varying number of workers over both topologies simultaneously 37
11 Effect on the number of workers over the time spent per epoch performing episodes 38
12 Effect on the number of workers over the relative time spent per epoch performing episodes 39
13 Training speed-up achieved when switching from PPO to ES across different number of workers and topologies 40

List of Tables

1 Summary of the configurations for the evaluation 29

Abstract

The field of Reinforcement Learning (RL) has been receiving much attention during the last few years as a new paradigm to solve complex problems. However, one of the main issues with the current state of the art is their computational cost. Compared with other paradigms such as Supervised learning, RL requires constant interaction with the environment, which is both expensive and hard to parallelize. In this work we explore a more scalable alternative to conventional RL through the use of Evolution Strategies (ES). This consists in iteratively modifying the current solution by adding Gaussian noise to it, evaluating these modifications, and use their score to guide the improvement of the solution. The advantage of ES lies on that creating and evaluating these modifications can be parallelized. After introducing the network routing scenario, we used it to compare how ES performed against PPO, a RL policy gradient method. Ultimately ES took advantage of increasing its number of workers to eventually overtake PPO, training faster while also generating better results overall. However, it was also clear that for this to occur ES must have access to a considerable amount of hardware resources, hence being viable only within high perfomance computing environments.

Acknowledgements

I wish to thank my supervisor Alberto Cabellos and my tutor Cecilio Angulo. I also wish to thank Paul Almasan, author to the original proposal of applying DRL for network routing and which I expand upon on this thesis. I also wish to thank my research group colleagues at the Barcelona Neural Networking Center including, but not limited to, Albert Lopez, Jordi Paillissé, Guillermo Bernárdez and Spyros Garyfallos.

1 Introduction

Reinforcement learning is a paradigm within Machine Learning that has seen increased attention over the last decades. Originally its techniques like Q-learning were limited to toy problems, due to their inability to generalize knowledge across similar states or relying in memory taxing lookup tables [3]. While progress was done to amend this by encoding states and replacing lookup tables by linear functions, eventually a big resurgence came with the first applications of neural networks with reinforcement learning [4, 5].

The main advantage between RL algorithms over the other forms of learning like Supervised and Unsupervised learning, was the ability to learn from actively interacting with environment. This allowed RL to be more appropriate for tasks related to control, that is, where the model takes actions to control the environment's state. In research this usually consists in playing games, from toy examples like pole balancing, to more complex games like playing soccer or even videogames like StarCraft II [11, 10]. However, in real world applications this usually means control over machinery or even entire installations. For example, in 2016 Google used a DQN model to oversee energy consumption, which lead to "15% reduction in overall energy overhead" [12]. A more recent application is the idea of using a RL model to control traffic demands within a computer network to increase throughput and reduce the saturation of links [22].

However, while powerful, RL's biggest strength is also is biggest drawback: unlike in the other types of learning where samples are already available, with RL the agent needs to constantly interact with the environment to continue learning, which in turn takes time. Overall, this is the biggest factor affecting the speed of RL training. For example, the solution discussed earlier about network routing found itself limited by the training time, making it impractical to train for larger network topologies [22]. Another issue is that the actions of updating the parameters and further interacting with the environment are interweaved and must be done concurrently. This means that while it is interesting to perform several interactions with the environment in parallel there are some issues to resolve, like how is the learned experienced re-unified or how to avoid overlap of learned knowledge between different copies. That does not mean that it has not been tried, as approaches like A3C saw success until being overshadowed by better, single-threaded algorithms [8].

Evolutionary Strategies comes into play as a black-box optimization algorithm, part of the field of evolutionary computing [13, 14]. Evolutionary methods are based on Darwinian theory of natural selection. In a nutshell, these algorithms consist in iteratively improving the current solution by randomly generating small modifications to it and keeping those that improve the score given by a given fitness function to optimize. ES specifically uses these modifications, called perturbations, to make a sampling of alternative solutions, called mutations, which are then evaluated and used as a basis for applying changes to the current solution.

After looking at the issue of scalability and parallelization within RL, and the potential of ES, OpenAI released a paper which proposes training RL agents using ES instead of traditional RL algorithms [17]. The main benefit of this approach is that, unlike RL, here the interactions with the environment can be safely parallelized with marginal overhead. As a result, we wished to adapt and test this solution in uses cases like the network routing it was introduced earlier, and see if this way of training agents would allow for larger network topologies to be trained by taking advantage of high performance computing environments.

In section 2, *Background and State of the Art*, we will provide the theoretical background behind RL, ES and Graphical Neural Networks, the network model used in our use case, necessary to understand the rest of the paper. Next, in section 3, *Thesis Objectives and Goals*, we will go more into detail about the limitations behind scaling RL, as well as formalizing the goals of the report. In section 4, *Proposed solution*, we will explain the solution proposed by the OpenAI paper, as well as their optional modifications and some of our own. Ultimately, we will conclude with the final version of the algorithm to be evaluated, as well as justifying our reasoning behind including or dropping some of features discussed earlier. Afterwards, in section 5, *Use case: Computer Network Resource Allocation*, we will introduce the use case of network routing, and how it is represented as a Reinforcement learning problem, as well as understanding how the GNN is designed and applied within this context. In Section 6, *Evaluation*, we start by presenting how the algorithm is going to evaluated, followed by detailing the obtained results. The report ends with section 7, *Conclusions*, in which we discuss the results obtained and if they satisfy, or not, the objectives of the thesis. In it we will also propose ways in which our work can be further expanded.

2 Background and State of the Art

2.1 Deep Reinforcement Learning (DRL)

The field of Machine Learning can be split broadly into three paradigms: Supervised, Unsupervised and Reinforcement learning. Supervised learning may be thought as the more intuitive of the three: problems consist on a set of samples, each with an assigned output (also referred as labels), and the task is to build a system that is able to learn from this set of samples to then be able to predict the labels of new unseen ones. Hence, supervised models are mostly used for solving problems of classification or regression. Unsupervised learning differs slightly as the samples no longer have assigned outputs. Instead, unsupervised models try to find patterns within the data, in an attempt to summarize and better understand it (for example, through clustering).

In Reinforcement learning, on the other hand, instead of having a problem represented through a set of samples, the problem is framed as the interactions between the environment and agent, as shown in Figure 1. The environment starts with an initial state s_0, which the agent will observe. From the current environment, the agent will take an action a_0 within the environment. As a result, the environment will give feedback to the agent, also known as the reward r_0, and the state of the environment will be modified from s_0 to s_1. This will continue endlessly or until the environment reaches an end-state. We refer as an episode to the sequence of state, actions and rewards from the initial state until an end-state is reached.

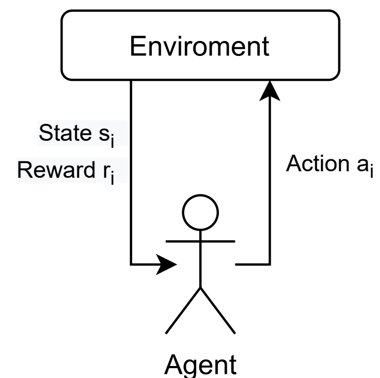


Figure 1: Interactions between the agent and the environment in RL

We refer as 'the policy' the way the agent chooses its actions from the observed state from the environment. Formally, we can define it as a mathematical function that maps states to actions. Hence, Reinforcement learning algorithms try to find the policy that maximizes the reward obtained.

One of the first known RL algorithms is Q-learning [2]. The algorithm consists in learning the Q-values $Q(a, s)$ for each state, action pair, which indicates the expected long-term reward from performing action a on state s. During training, the expected long-term reward is updated by allowing the agent to interact with the environment, record which states it visited and which actions it took in response. Then, the Q-values are updated through Bellman's equation, which takes into account the immediate reward obtained by taking action a on state s, as well as the expected-long term reward from other states s' currently available from the current one. Once trained, the optimal policy is trivial: given the current state choose the action which maximizes the current Q-value.

While originally the Q-values were stored in a lookup table, this made it inherently limited to toy problems with minuscule amount of possible states. Shortly after Q-learning was adapted so it can be applied as a function through linear approximation, where the state was encoded as a tile system or with a radial basis function [3]. Eventually, it was figured out how to represent Q-values through a neural network, in what was known as Neural Fitted Q-Iteration (NFQ), which later evolved into deeper networks hence creating Deep Q-learning and the field of Deep Reinforcement Learning as a whole [4, 5].

When the usage of neural networks in RL was introduced, some issues had to be addressed due to the nature of neural networks. One of them was the fact that Q-learning was an incremental algorithm: every time the agent interacted with the environment, the interaction is fed into the algorithm to update its policy. This proved to be problematic when applied to neural networks and gradient descent, as it broke the assumption data is independent and identically distributed (i.i.d) when applying gradient descent.

As a result, NFQ modified Q-learning for it to be done in batch learning rather than incremental. In order to do so, the algorithm was modified to include the addition of a previously proposed augmentation known as the experience replay buffer [7]. By storing the interactions, also called experiences, within a buffer, it allowed the algorithm to sample a batch of samples from it to perform training everytime it wanted to. While the original benefit of the buffer was to habilitate experiences to be used more than once during training, using experience replay also allowed for sampling to be done in such a way the i.i.d assumption was maintained during training.

As of today the most adequate RL algorithm will depend on the characteristics of the environment: if it follows the rules of a Markov Decision Process or not, if there is total or partial observability, if the rules of the environment are dynamic and changing, if there are multiple agents involved... However, one of the most popular general-purpose algorithms is Proximal Policy Optimization (PPO) [1].PPO is a policy search model, meaning that, unlike Q-learning where the model attempts to estimate the long-term reward of performing certain actions, the policy is directly represented as a parametrized mathematical function which takes states as input and returns probability distributions for choosing the next action. Since the policy is now represented by a parametrized function it is then possible to define a function that calculates its loss, and later optimize it through gradient descent. This is also referred as policy descent.

One of the original policy gradient methods was the actor-critic model, known by separating into two networks the act of evaluating the benefit of using each action (the critic) and then the final probability distributions for choosing each one (the actor). This helped reduce bias during training, and during evaluation only the actor model is kept [6]. Ultimately PPO is the accumulation of years of improvements upon the original actor-critic model.

2.2 Evolution Strategies

Evolution Strategies (ES) are a family of black-box optimization algorithms, first coined by Ingo Rechenburg and later expanded by Hans-Paul Schwefel [13, 14]. It is referred as such since it can be used to optimize a function without necessarily requiring to know its gradient or have any other underlying assumption.

As the name suggests, it is one of the branches of evolutionary algorithms. Evolutionary algorithms are conceptually based on the Darwinian theory of natural selection in biology: the algorithm starts with a set of initial solutions, each referred as an ‘individual’ and the set as a whole as a ‘population’. At each iteration, the individuals are evaluated according to a function to optimize, known as as the fitness function. Then, using the evaluation as a reference, part of the population is removed and replaced by new individuals. The new individuals are obtained by modifying one of the existing individuals (mutation) or by combining two or more individuals (recombination).

The long term consequences is that the worse performing individuals are systematically replaced by new ones derived from the best performing ones. Eventually, long-term progress is guaranteed: bad performing new individuals are quickly replaced, while those that perform best replace their parents. This strategy is what allows evolutionary algorithms to stand out as extremely effective optimization algorithms where gradient descent-based methods are not available. Even when applied to optimize problems that can be dealt by gradient descent, while slower, they also tend to be more resilient to local minima.

ES differentiates itself from other evolutionary algorithms in three major ways. The first is that is meant for arrays of natural numbers. The second is that, compared with other evolutionary algorithms, the population is almost replaced between iterations and the replacement is done almost exclusively through mutation. The third is that ES can adapt its own parameters: in order to generate a mutation from an individual you must draw noise from a probability distribution (usually Gaussian). Since these distributions can be parameterized, we can use the optimization mechanism ES uses to also adapt this parameters.

The most basic form of ES is the original ES, also known as Simple Gaussian Evolution Strategies. To begin with we need a defined fitness function F to optimize, a defined number of mutations μ to generate and later a refined number of mutations λ to choose from (such as $\mu > \lambda$), an initial solution θ and an initial standard deviation σ. At each iteration we perform the following

1. We sample a population made out of μ mutations. Mutations are generated by sampling values from a Gaussian distribution with mean θ and with standard deviation σ.

2. We evaluate each mutation θ' through the use of the fitness function F. For each mutation we will obtain a score $F(\theta')$. From here, we obtain a refined population by keeping the λ mutations with the highest fitness score (assuming we want to maximize the fitness function; otherwise we would keep the λ smallest instead).
3. Using the refined population we can update our current solution and standard deviation:
   - The updated solution will be the average of the mutations within the refined population.
   - The updated standard deviation will be standard deviation of the mutations within the refined population.

The underlying idea behind this algorithm is simple: we consider our current solution as a point within the space of potential solutions. While in the original proposal this space was uni-dimensional, we can assume any number of dimensions as required. Whenever we obtain mutations, what we are doing is exploring the solution space in the neighborhood near the current solution. From the neighbourhood we draw a better solution, hence moving closer to the optimum. However, since exploration is done through noise, no gradients are involved.

From here more advanced versions of ES sprout from this original idea. This can include, but is not limited to:

- Keeping some perturbations from one iteration to the rest, making learning more stable by guaranteeing perturbations will be at least as good as last iteration.
- In multidimensional solutions, draw noise from learned covariance matrices. By learning the correlations between dimensions the exploration can be done more efficiently [16]. Currently the most advanced example of this approach is the Covariance Matrix Adaptation Evolution Strategy algorithm (CMA-ES), although due to its complexity it is not recommended to be applied in problems with high-dimensionality. [38].
- Use of natural gradient ascent to better improve our current solution from the obtained mutations. This is one approach chosen by the subfamily of Natural Evolution Strategies algorithms [15].

2.3 Graph Neural Networks

Graph Neural Networks (GNNs) are a type of neural network modified to deal with data which can be represented through a graph. Originally introduced in a paper by Franco Scarselli and others, the motivation behind this type of network is for it to take advantage of the graph's structure to offer better performance with a lower number of parameters [20]. This idea is similar in how a Convolutional Neural Network takes advantage of an image's properties.

Since its inception several modifications to the original GNN have emerged, although the underlying concept is the same. First, data is introduced encoded by a graph, with a set of nodes and edges. Each node will have an internal state, which will be updated from neighboring nodes through some sort of neighborhood function. Once the states of the node of the graph have been updated, they will be aggregated and parsed through a final output function that returns the desired result. The first function is meant to share and process information based on locality (taking advantage of the graph structure), while the second function is meant to draw a conclusion from a global view of the graph.

In this project we will focus on Message Passing Neural Networks (MPNNs), as it is a type of network that has proven effective to tackle several graph-related problems. Originally it was used to predict properties from organic chemical compounds from their chemical structure [21]. It was also the model behind the solution for network routing problem we will focus later on. One strong advantage of MPNNs is that when its components are correctly defined, the resulting model is invariant to graph isomorphism. Formally, the MPNN is defined as follows:

1. As any other GNN, the data is received in form of a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is the set of edges.
   - Each node $v \in \mathcal{N}$ will have an internal state h_v, represented as a vector of real numbers, and a neighborhood $N(v)$ formed by the nodes which share an edge with.
   - While the original formulation considers undirected edges, it can be generalized to directed edges as well.
2. The message-passing phase occurs, where information is exchanged for a fixed number of iterations. For each node v at iteration t, we perform the following steps:
   (a) v will receive a message from all its neighbors $w \in N(v)$. The message is defined by a message function M which generates the message using the state of the receiver, the sender, and the edge connecting both $e_{vw} \in \mathcal{E}$:

   $$M(h_v^t, h_w^t, e_{vw})$$

   The message function M can be defined through any differentiable function (such the function defined by a neural network) and produces an output with the same dimension as the internal states of the nodes.

   (b) All the messages received by the node v from its its neighbors are aggregated. This can be done so through any commutative operation (as messages are sent all at once and they are received out of order), although by default this is done through element-wise addition:

   $$m_v^{t+1} = \sum_{w \in N(v)} M(h_v^t, h_w^t, e_{vw})$$

(c) The aggregated messages are then used to update the internal state of node v through an update function U. This function takes as input the current internal state h_v^t and the aggregated messages m_v^{t+1}, and produces the updated state h_v^{t+1}. This function can also be defined through any differentiable function, and is also suitable to be represented through a model with internal state (e.g. Recurrent Neural Networks).

$$h_v^{t+1} = U(h_v^t, m_v^{t+1})$$

3. The global state of the graph is obtained by combining the internal states of its nodes. This combination must be done in a way that does not implicitly order the nodes in any way, as to preserve invariance to graph isomorphism. By default, this is done by concatenating the states from each node into a single flattened vector.
4. Finally, the global state of the graph is fed to a readout function R, which takes the global state of the graph to produce the desired product. This function can be defined through any differentiable function.

3 Thesis Objectives and Goals

On one hand, we have established the usefulness of Reinforcement learning, specifically DRL, as a framework to be applied in problems related to control and continuous interactions between an agent and an environment. However, it has also proven extremely hard to scale upward. One hand, RL algorithms that internally uses neural networks ultimately will use gradient descent with backpropagation, which can be parellelized and even be run in a hardware accelerators, such as a GPU. However, the fact that RL work with experiences, and these must be sampled constantly throughout training poses an additional burden that is not present in Supervised and Unsupervised learning algorithms.

As a consequence, while training most of it is spent in interacting with the environment and drawing experiences to learn from. Consequently, logically the best way to accelerate training would consist in speeding up this process, for example through parallelization. However, this posses several coordination problems, such as how are experience shared between the different workers, how and where gradient descent is performed, and how it is guaranteed that all workers share the updated set of parameters at all times. As a result, this paradigm is costly and requires complex communication.

A more refined approach would be that one proposed in the paper "Asynchronous methods for Deep Reinforcement Learning" [8]. The idea of this paradigm is to have a main process along with a set of worker processes, all sharing a copy of the same network. Each worker interacts with the environment and learns independently. After a period of time each worker will send a partial gradient descent step to the main process, which combines updates from everyone else. The worker processes also periodically requests the updated parameters from the main process to stay up to date. While this paradigm was applied to several RL algorithms, it work best when combined with an actor-critic model, creating the method known as the Asynchronous Advantage Actor-Critic (A3C).

The main advantage of this method is that it allows the algorithm to scale to more than one process. Since A3C is asynchronous, workers do not have to update at the same time either, easing coordination. Another advantage is that it removes the need for a replay buffer: since samples from different workers are uncorrelated, sampling from the experience replay buffer is no longer necessary to preserve the i.i.d assumption of gradient descent.

However, A3C is not without its drawbacks. First, a fairly high number of workers are required to make sure samples are not too correlated during training. Furthermore, as RL research has continued it started to fall behind more current algorithms. For example, the synchronous version of A3C, known as Advantage Actor-Critic (A2C), was shortly introduced afterwards, with its main difference being in which all workers coordinate to perform the gradient descent at once, which proved to have better performance than the asynchronous version. Subsequently, both A3C and A2C were later dethroned by PPO, which proved to have a stronger performance even if done sequentially [1].

As a result, we aim to find a new method for training agents, specifically those represented by neural networks, that can compete with the state of the art algorithms, such as PPO, while also being scalable and suitable in high performance computing environments. We measure scalabity in three of ways: by measuring how performance is increased through increased hardware resources, how is decreased through increased number of worker processes, and how is decreased through increased problem size.

First, for the method to be scalable across hardware it should utilize all of the available resources at hand. To do so the method must be parallelizable and hence be run on multiple threads and/or processes, in different computers if it has to. The memory usage should also remain low enough so it does not become the limiting factor in the number of workers that can be launched at once in the same machine, and instead being the number of available cores in the CPU. Also, while not a requisite, it would be best if the algorithm also took advantage of any additional accelerator available, such as GPUs.

The second aspect of scalability refers to the amount of computational overhead generated by the number of workers launched. To accomplish a scalable method in this sense, the algorithm should aim to minimize the communication costs between workers, and even the amount of messages sent all together. The last aspect of scalability refers on how the computational cost of the algorithm increases depending on the problem size. In other words, the method should be invariant (or as close as it can be) to the the scale of the problem, which in this case would be measured through the network's size.

Ultimately, the objectives of this thesis are the following:

- Explore an appropriate parallelizable alternative to RL, and design an efficient implementation adequate for our use case.
- Evaluate the speed-up obtained by the algorithm when increasing the number of workers.
- Determine how the overhead increases as the number of workers increase, how many workers the algorithm can support at once, and which are the limiting factors.
- Ascertain how does the size of the network impacts the training time of the algorithm.
- Benchmark the performance of our implementation against a classical DRL algorithm.

4 Proposed solution

4.1 Evolution Strategies and RL

Recently OpenAI released a paper titled "Evolution Strategies as a Scalable Alternative to Reinforcement Learning" that dealt in how to apply ES to train RL agents [17]. In it, they specified how they adapted Natural Evolution Strategies, a variant of ES, and how it could be parallelized.

Unlike ES, NES updates the solution by using natural gradient ascent, rather than directly using the average of the perturbations [15]. This consists in obtaining an approximate of the gradient by computing the average of the product between the perturbations and their corresponding fitness function returns. Note that perturbations and mutations do not refer to the same concept: we refer as perturbations the normalized Gaussian noise before being multiplied by the standard deviation and added to the original sample to generate the mutation. Once the gradient is obtained, then the current solution is updated as if it were normal gradient descent. The pseudocode detailing the algorithm can be seen in Algorithm 1.

Algorithm 1 Natural Evolution Strategies [17]

Require: Learning rate α, noise standard deviation σ, initial policy parameters θ_0, number of mutations n, fitness function F

1: **for** $t = 0, 1, 2, ...$ **do**
2: Sample perturbations $\epsilon_1, \epsilon_2, ..., \epsilon_n \sim \mathcal{N}(0, I)$
3: Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, ..., n$
4: Update policy parameters $\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{i=1}^{n} F_i\epsilon_i$

For it to be applied in the context of RL, the main change to be done is how to define the fitness function to evaluate each mutation. Here, both the solutions and mutations represent a model as a one dimensional vector. In the context of neural networks, including GNNs, each solution will represent all the parameters of the network being flattened into a single vector. Then, the fitness score is extracted by obtaining the accumulated reward by the mutation across en entire episode within the environment. Consequently, the algorithm must perform gradient ascent, as we wish benefit those perturbations which maximize the obtained reward.

The main advantage of using ES is how it can be trivially parallelized. What the paper proposes is to initialize a number of workers n, each with an environment instance of its own. When working together, each worker can evaluate only a part of the perturbations, and then send its results to each other so everyone can update the parameters independently. As long as each worker share the same perturbations and fitness results across execution, all workers will converge to the same solution. The algorithm's pseudocode can be seen in Algorithm 2.

Algorithm 2 Parallelized Natural Evolution Strategies [17]

Require: Learning rate α, noise standard deviation σ, initial policy parameters θ_0, fitness function F
Ensure: n workers with known random seeds and initial parameters θ_0
1: **for** $t = 0, 1, 2, ...$ **do**
2:     **for** each worker $i = 1, ..., n$ **do**
3:         Sample perturbation $\epsilon_i \sim \mathcal{N}(0, I)$
4:         Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
5:     Send all scalar returns F_i from each worker to every other workers
6:     **for** each worker $i = 1, ..., n$ **do**
7:         Reconstruct all perturbations ϵ_j for $j = 1, ..., n$ using known random seeds
8:         Update policy parameters $\theta_{t+1} \leftarrow \theta_t + \alpha\frac{1}{n\sigma}\sum_{j=1}^{n} F_j\epsilon_j$

Not only the resulting version splits its workload across multiple threads and/or processes, but also does so at a minimal overhead. Through clever use of predetermined seeds for the noise generators, the communication cost can be kept to a minimum by generating each other's perturbations without needing to send anything. Hence, the only communication costs come from sending the fitness function returns to other workers, meaning the asymptotic cost of communication will be of $O(nki)$, where n is the number of workers, k the total number of perturbations and i the number of iterations, and the expected number of messages sent being $i \cdot (n-1)^2$.

Overall, this algorithm takes advantage of NES's properties to offer a more scalable alternative in form of a fully distributed solution compared to traditional RL algorithms. Additionally, it does so while requiring fewer hyperparameters. While algorithms like PPO requires making decisions about the replay buffer's characteristics (size, sampling rate), discount factor and exploration policy among others, NES only requires the user to decide the values of two hyperparameters:

- The number of perturbations n: The higher the number of perturbations, the more information the algorithm will have at each epoch to approximate the gradient correctly, making learning more stable. However, more perturbations to evaluate results in more time is spent interacting with the environment.
- Standard deviation σ for generating mutations: it controls how different the mutations will be compared with the current solution. The effect of this hyperparameter is similar to the learning rate in gradient descent, where the value will depend on how irregular is the "solution space". That is, it will depend on how much the quality of the solution will change over small changes in its value. This impact is illustrated in Figure 2 representing a simplified example where the solutions have only one parameter:
  - Point A shows the current solution θ.

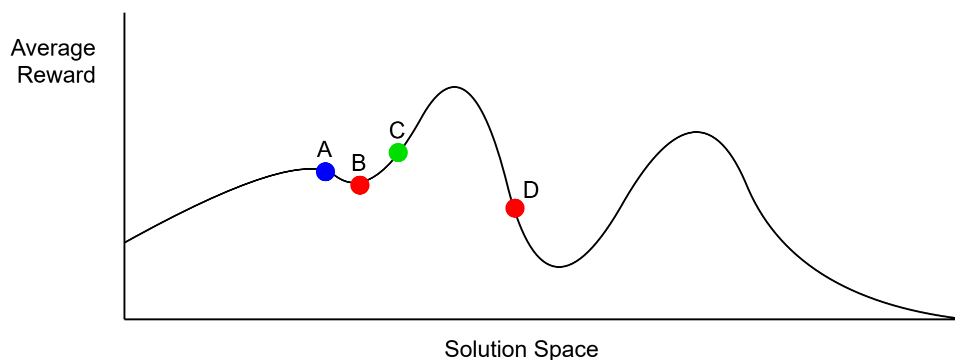


Figure 2: Effect on choosing an adequate noise level for generating perturbations.

- Point B shows a mutation being being generated from a sub-optimally small value of σ. By being too close to the solution it captures small irregularities that could lead to the solution to be more easily stuck in a local minimum.
- Point D shows a mutation being generated from an sub-optimally large value of σ. By being too far away from the current solution it no longer represents the neighborhood near it, and hence it will worsen the gradient approximation.
- Point C shows a perturbation generated from an ideal value of σ. It captures correctly the solution space near the current solution without capturing the irregularities that do not have a significant impact over the solution.

However, unlike the learning rate, the value of σ will not translate into larger gradient steps. This is because in the end the gradient is updated using the raw perturbations, which are drawn from the normalized Gaussian distribution, and standard deviation is only applied to then create the mutations.

ES also offers the advantage of not being affected by certain limitations of traditional RL algorithms [17]. The first one is that ES does not have to consider future discounting, a measure of how much future rewards are worth relative to immediate rewards. Since learning occurs at the level of individual interactions, a balance must be struck between maximizing the reward obtained with the next action and the expected long term reward that you can obtain afterwards. If this balance is not achieved agents may end up being myopic, choosing actions that maximize their short term rewards but at the cost of rewards later on. However, not applying any discount will avoid agents from converging, as those environments with potentially infinite episodes could have theoretically infinite expected rewards.

On the other hand, in ES each perturbation is evaluated by the accumulated reward across the entire episode. Since the evaluation is done at the episode level, the notion of taking into account expected long term reward from taking actions in individual states is no longer relevant for training the agent.

Another advantage of training agents through ES over traditional RL is also not being affected by environments with sparse rewards. In traditional RL, both Q-learning and policy approximation, learning is only triggered once the agent reaches states that award some reward. Otherwise, due to the definition of the loss functions used to train the agents, the obtained loss would be equal to 0 and so the network will not learn anything. This issue is not relevant in ES due to the same reason as before: since ES evaluate perturbations by evaluating entire episodes as a whole, it does not care how spread out the reward is within the episode itself.

Furthermore, due to the benefits of ES being a black-box optimization algorithm, it does not make any assumptions of the environment itself either, including if it can be defined as a Markovian Decision Process or not, assumption held by many traditional RL algorithms. Therefore, using ES may prove even more beneficial in these kinds of environments.

Before we move on, we must also clarify the disadvantages that ES has over traditional RL algorithms. The first one is that, as we clarified at our introduction to ES, evolutionary algorithms tend to perform slower than gradient based methods, and in this case we do not expect things to be different. Unlike DRL algorithms, ES tries to optimize all parameters at once and without using backpropagation, a major speed up in updating a network's parameters. This is exacerbated by the fact the current implementation of ES is not accelerated by being run in an accelerator such as a GPU.

Additionally, ES is a substantially more 'bruteforce' way of optimizing the gradient. In ES we depend on randomly generated perturbations for finding a better solution, while traditional RL algorithms like PPO have made massive improvements in how to sample interactions more effectively (e.g. through prioritized experience replay [9]). While using entire episodes to evaluate perturbations has significant benefits as we just discussed, it also means that ES takes many more interactions with the environment before it can adapt its current solution compared to traditional RL algorithms. Overall, we expect that the single-threaded performance for ES to be significantly slower than traditional RL algorithms like PPO.

Another important limitation to consider is that ES theoretically cannot run in environments with infinite episodes. Since ES uses entire completed episodes to evaluate the perturbations, it will always enforce a finite horizon on its episodes. In practice limiting the episodes length may not be a deal-breaker, but it will mean the obtained solution will not longer be reliable after the cut off mark in the episode. Also, allowing episodes to run for too long will affect the time taken to train the agent, as running the episodes is the current bottleneck in the agent's training.

4.2 Modifications to the parallelized solution

While the algorithm shown in Algorithm 2 is the proposed solution by the OpenAI paper, in the same paper they also discuss other modifications that could be done to improve algorithm, by either improving the training time or increasing the stability of learning.

4.2.1 Reward regularization and fitness shaping

Fitness shaping is a technique described in the paper introducing NES, although the idea is common place in evolutionary computing [15]. The idea is to evaluate the perturbations using a ranking according to their fitness return, rather than using the value directly. Once the ranking replaces their fitness values, then it is scaled so scores are assigned in the range of $[-0.5, 0.5]$. Using the rank instead of the values allows learning to be more stable. On one hand, it makes sure that the magnitude of change of the solution between iterations is more bounded by limiting the values of the returns, especially useful in environments where the obtained reward can drastically change [17]. This effect is more pronounced at the beginning of training, where the differences in reward between different mutations is more pronounced. This effect also helps avoiding learning to stall when reaching convergence, as it will amplify the differences in obtained reward between mutations if it becomes too small.

4.2.2 Mirror sampling

Mirrored sampling is a technique introduced to increase stability of ES algorithms by improving how perturbations are generated [19]. The idea is simple: for every perturbation ϵ_i we generate, we will also consider its inverse $-\epsilon_i$. This change improves stability by ensuring we are correctly exploring the solution space near our current solution, as for every perturbations we evaluate we will also take into account exploring the complete opposite direction.

Applying this change is also computationally inexpensive, barely increasing the cost of obtaining the gradient. First, it reduces the cost of generating the perturbations to half, as the other will be the mirrored version. Additionally, by taking advantage of how the gradient is obtained, we can reshuffle the operators to further minimize the computational cost of doing so. This is done as such: let us have n generated perturbations, and for each perturbation ϵ_i we have its mirrored version $-\epsilon_i$. We have also obtained the fitness returns for ϵ_i and $-\epsilon_i$ as F_i and F'_i respectively. Hence, the gradient would be obtained as:

$$\frac{1}{2n\sigma}\Big(\sum_{j=1}^{n} F_j\epsilon_j + \sum_{j=1}^{n} F'_j(-\epsilon_j)\Big) = \frac{1}{2n\sigma}\Big(\sum_{j=1}^{n} (F_j\epsilon_j - F'_j\epsilon_j)\Big) = \frac{1}{2n\sigma}\Big(\sum_{j=1}^{n} (F_j - F'_j)\epsilon_j\Big)$$

As such, by subtracting the obtained mirrored rewards from the original perturbations's returns, we can reduce by half the amount of dot products operations later on to obtain the gradient, making this change inexpensive computationally.

4.2.3 Adding noise to the action probabilities

Unlike traditional RL algorithms, exploration strategies are no longer necessary when running episodes to train the network. That is because the exploration is already done when generating the perturbations, as with them we are exploring the solution space. Instead, episodes are run to evaluate the perturbations, thus they must be run as if it were evaluation episodes in traditional RL methods, always choosing the action preferred by the model.

The paper exploring the use of ES in RL explores this argument from a more formal angle [17]. Specifically, they state that trying to apply gradient descent directly over the solution space is risky since it is not known how smooth the solution space really is. Therefore, as to add smoothness to the gradients Gaussian noise is added. In DRL this is done when exploration policies are considered and/or Gaussian noise is added to the action probability distributions drawn by the models. In ES, the noise added are the perturbations in the solution space.

Ultimately, they argue that in those scenarios where stronger smoothing is required for gradient descent to occur successfully, noise can be added to both the solution and actions. That is, whenever episodes are run in ES to evaluate a perturbation, noise should also be added to action distributions drawn by the model before committing to a specific action.

4.2.4 Precomputed noise

Another of OpenAI's proposals is to, rather than generating new noise vectors every epoch, to have a certain amount of pre-computed noise from a normalized Gaussian distribution and sample from it instead [17]. The argument is that sampling from the matrix is quicker than generating the noise, trading memory for a computational speed up. The main issue with this is that the drawn perturbations must be i.i.d, while sampling from the same precomputed set of noise vectors is not. As a result, bias is introduced into the search and hurting its performance, unless a significant amount of noise is pre-generated.

4.2.5 Distribution of perturbations among workers

One aspect that the original pseudocode glosses over is how is perturbations are divided among the workers . As it is written in Algorithm 2 it implies that there are many workers as perturbations, which in reality this is impractical as each worker requires its own overhead. In reality, workers are expected to sample several perturbations.

The most simple solution would mean to split the perturbations evenly across workers. However, since episodes could have varying length this does not guarantee that the workload is evenly divided. As a result, the proposal recommends a system in which all workers will train for a fixed number of interactions, and will send the results of the episodes it is able to complete in the process. This minimizes even further downtime from waiting threads [17].

4.3 Design of our proposed solution

Given the original parallelized algorithm we will explore the version of the algorithm we will use in this report after introducing some of the modifications introduced in the same paper, as well as some of our own. By combining all of these changes together, the final version of the algorithm to be used can be seen in Algorithm 3.

Algorithm 3 Improved Parallelized Natural Evolution Strategies

Require: Learning rate α, noise standard deviation for mutations σ, noise standard deviation for actions φ, initial policy parameters θ_0, fitness function F, number of mutations k

Ensure: n workers with known random seeds and initial parameters θ_0, one of them being designed as the coordinator

1: **for** each worker $i = 1, ..., n$ **do**
2: Obtain assigned number of mutations $k_i \leftarrow$ *AssignMutations(k)*
3: **if** worker i is coordinator **then** ▷ Coordinator thread
4: **for** $t = 0, 1, 2, ...$ **do**
5: Sample global perturbations $E \leftarrow [\epsilon_1, \epsilon_2, ..., \epsilon_k], \epsilon_j \sim \mathcal{N}(0, I)$
6: Retrieve local perturbations $E_i \leftarrow E[k_i, k_{i+1} - 1]$
7: Compute returns $F_i^{original} \leftarrow F(\theta_t + \sigma\epsilon_i; \varphi)$
8: Compute mirrored returns $F_i^{mirrored} \leftarrow F(\theta_t - \sigma\epsilon_i; \varphi)$
9: Receive returns other workers: $F^{original}$
10: Receive mirrored returns from other workers: $F^{mirrored}$
11: Combine returns and mirrored returns $F^{combined} \leftarrow F^{original} - F^{mirrored}$
12: Obtained ranked returns $F^{ranked} \leftarrow$ *ObtainRanking*$(F^{combined})$
13: ▷ Other forms of gradient descent can be used, regularization can be added
14: Obtain solution update $\Delta\theta \leftarrow \alpha\frac{1}{n\sigma}\sum_{j=1}^{n} F_j^{ranked}\epsilon_j$
15: Send $\Delta\theta$ to all worker threads
16: Update solution $\theta_{t+1} \leftarrow \theta_t + \Delta\theta$
17: **else** ▷ Worker thread
18: **for** $t = 0, 1, 2, ...$ **do**
19: Sample local perturbations $E \leftarrow [\epsilon_1, \epsilon_2, ..., \epsilon_{k_i}], \epsilon_j \sim \mathcal{N}(0, I)$
20: Compute returns $F_i^{original} \leftarrow F(\theta_t + \sigma\epsilon_i; \varphi)$
21: Compute mirrored returns $F_i^{mirrored} \leftarrow F(\theta_t - \sigma\epsilon_i; \varphi)$
22: Send $F_i^{original}$ and $F_i^{mirrored}$ to coordinator thread
23: Receive $\Delta\theta$ from coordinator thread
24: Update solution $\theta_{t+1} \leftarrow \theta_t + \Delta\theta$

Out of the five improvements above, we have decided to include fitness shaping, mirror sampling and adding noise to the action probabilities. Including fitness shaping and mirror sampling was an easy decision, due to their many benefits and no downsides. Having to add noise to the action probabilities will be more exceptional, but its code complexity is not high and it can be added without affecting the behaviour whenever no noise is added.

Contrary to the original paper, we decided to not pre-compute noise as the benefit it brought was not worth its cost in memory. On one hand, later executions of our code saw that the cost of generating Gaussian noise was insignificant in respect to the cost of executing the environments themselves. On the other hand, the amount of pre-genereted noise that was needed to avoid introducing too much bias in the generation of perturbations, combined by the fact that this noise had to be replicated in all execution threads, increased memory costs to a point it was limiting in the amount of workers a machine could run.

The other difference we have respect the original paper is how perturbations are distributed. Ultimately, we did not observe the predicted situation where some episodes would be significantly longer than others, as the authors worried. Even if some episodes were marginally longer than others, ultimately all workers had to evaluate several episodes, which further evens out their workload. Additionally, this method also requires the perturbations themselves to be sent, which significantly increases the communication costs, and therefore the overhead, of the algorithm. Instead, we opted with a fixed assigned number of perturbations per worker.

Another aspect we decided to modify is the decentralized nature of the algorithm. While decentralization may be attractive in some applications, we do not believe it was the case here. Distributed solutions that benefit from decentralization are those that want to avoid a single point of failure. However, due to the synchronous behaviour of our algorithm, it would fail if anything interrupted any of the workers involved.

Additionally, decentralization increased significantly the memory cost of the algorithm. For the environment to be purely decentralized every worker must be able to update their solutions independently and in order to do so it must store the perturbations of the others. As a result, each perturbation will be replicated in each worker, resulting in the memory needed for storing every copy increasing up to a total of $n \cdot k \cdot p$ 32-bit floating point numbers, where n is the number of workers, k is the number of perturbations, and p the size of the perturbations, the same as the size of the solution θ.

A counter proposal would be instead to develop a centralized version of the algorithm, where one coordinator is the only one responsible for performing gradient descent. As a result, the workers only have to worry about their own perturbations, and only the coordinator will have a copy of other's. By limiting to storing each perturbation twice (the worker's original and the coordinator's copy), rather than having n copies, it reduces the cost of storing them down to $2 \cdot n \cdot k$ 32-bit floating point numbers, independently of the number of workers. The biggest issue with centralization is the increased downtime by the threads/processes, as the coordinator must wait for the workers to finish before continuing and vice-versa.

Centralization also has an impact in communication overhead. On one hand, since the rewards for perturbations have to be sent to a single coordinator rather to several workers, the amount of messages sent is reduced. On the other hand, the updated parameters now must be sent from the coordinator to the workers every time they are updated (once per iteration). Consequently, the resulting asymptotic cost of communication therefore changes to $O(ip)$, where i the number of iterations and p the size of the perturbations, compared to the previous $O(ikn)$ where k the total number of perturbations and n is the number of workers. Usually the size of the solution θ is larger than the number of perturbations, but is hard to say if it is always larger than $k \cdot n$. What is known to decrease is the number of messages, from $i \cdot (n-1)^2$ to $i \cdot 2(n-1)$ messages sent.

Overall, the benefits outweigh the drawbacks. While not commented in the paper, this is the approach chosen in the official code implementation of the algorithm [18]. However, unlike the official implementation, we made our coordinator to also assume the role of a worker. This both virtually eliminates the downtime of the coordinator, and also slightly reduces the communication costs, as the coordinator does not have to send information to itself.

One final note to discuss is how gradient ascent is applied. This is not covered by the paper's pseudocode, but it reflected later on both in its evaluation section and in the published code [18]. Since gradient ascent is applied, even if not through backpropagation, that means that we can use other forms of gradient descent such as gradient descent with momentum, or even the Adam optimizer. It is also possible to add regularization to the gradient approximation. For example the authors in their implementation opted for Ridge regression [18].

5 Use case: Computer Network Resource Allocation

The use case we wish to cover with our method is the network routing scenario. As the name suggests, scenarios of this problem consists of a computer network, in which computers are connected through links with a given specified bandwidth, and a set of traffic demands that must go through the network, defined by a source, destination and the bandwidth they occupy. The task consists in determining the path each demand must take as to satisfy as many of them as possible. This scenario is an interesting problem, due to both its utility and difficulty. Correctly assigning bandwidth demands is key to minimize throttle and avoid bottlenecks. However, it was proven that finding the optimal configuration is a NP-hard problem [25].

This dichotomy is even more pronounced by the fact that many real applications of network routing wishes for on-line methods, meant to allocate demands in real time in a working network, and therefore any time deliberating on the routing is further delay added to each demand. Current solutions work with heuristics in order to balance optimally and quickness. For example, using the shortest available path between two nodes returns good results with minimal cost, while more complex optimizers such as DEFO are much more expensive [26, 28].

Among the proposals used to tackle this problem, one of them focuses in the use of DRL and GNNs to do so [22]. In this case, the environment will be represented by the network represented as a graph, initially empty. Then the agent must allocate the received demands in the network, and is rewarded according to how many demands it can allocate. In terms of RL, in each interaction the environment sends its current state by sending the current state of the network, the newest traffic demand to be allocated and the reward from the previously allocated demand, while the agent acts by choosing the path the newest demand must go through. The episode starts with an empty network, and ends when one of the links ends up saturated, that is, there is more allocated bandwidth than the available one. These interactions are illustrated in Figure 3.

In the proposal, the agent used a GNN, specifically a MPNN, to take advantage of the fact the problem takes place in a graph [22]. Specifically, the graph within the MPNN represents the links of the network, and edges connect links that share nodes in the network. This representation is shown in Figure 4. The internal state of the nodes within the graph represents three features of each edge:

- The original, maximum capacity of the edge.
- The betweenness of the link: a feature from graph theory that measures how many possible paths exist in the network that goes through that link [22].
- The amount of bandwidth allocated within that link. Since the units dealt with here are atomic, this value is represented through one-hot encoding. This also means that most states will also have zero-padding to accommodate for the link with the largest bandwidth.

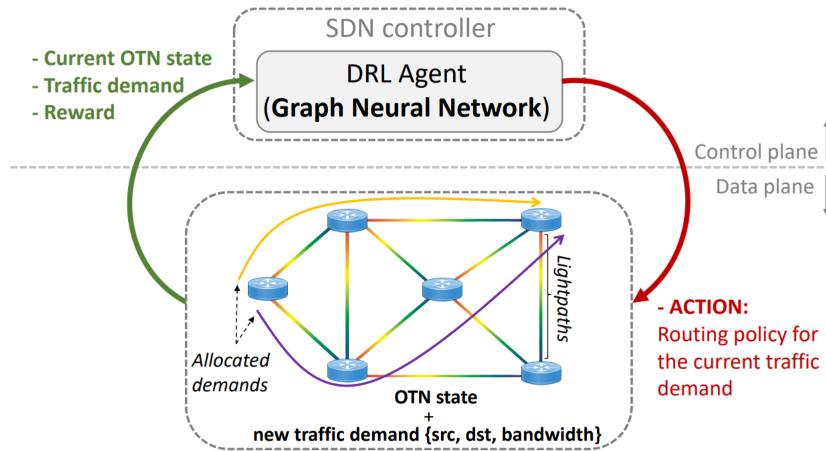


Figure 3: Diagram detailing the interactions between the agent and the environment in the network routing problem [22]

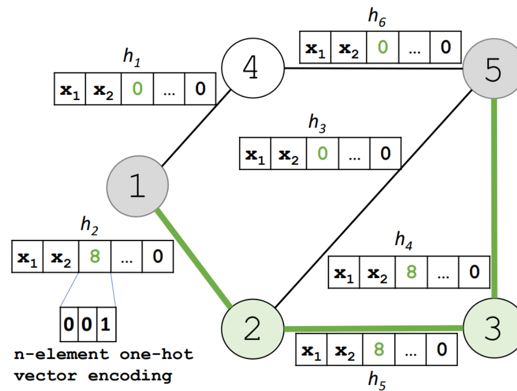


Figure 4: Representation of the graph used in the MPNN [22]

Hence, the MPNN must return a probability distribution indicating which action should be taken. As a remainder, each action represents a potential routing of the current demand on the network. Therefore, when an action was introduced in the network it had to be modelled as graph. This consisted on obtaining the current state of the network plus the reflection of performing the given routing on it. When introduced, the MPNN produced a score reflecting the desirability of taking that action.

This process was then repeated with all the possible routings between the source and destination of the demand. Although clever implementation eased computational cost, for example by evaluating all routings at once, the cost of the model increased heavily with the number of possible actions, which themselves increase exponentially with the graph's size. As a result, the authors limited the model so it only considered the 4 shortest paths between the each pair of nodes in the network. The agent was originally trained through DQN, and later on through PPO as training was quicker, more stable and obtained better results [22].

6 Evaluation

6.1 Evaluation settings

For the evaluation of the algorithm we had it run in two hardware environments:

- Local server: uses a 16-core AMD Ryzen 9 3950X CPU, although it supports execution of up to 32 concurrent threads, and 64 GB of RAM.
- Remote server: a remote server run in Amazon Web Services. Specifically, an c5a.16xlarge instance was used, which includes AMD EPYC 2nd generation processors with 64 cores and 128 GiB of RAM.

Since the use case runs on top a computer network, we must also choose its topology to test our algorithm. As such we considered two topologies released to the public: the topology of the National Science Foundation Network (NSFNET) and the topology of the GÉANT2 network, by the GÉANT project [23, 24]. The NSFNET network is made up of 14 nodes and 42 edges, while the GÉANT2 network is made of 26 nodes and 74 edges. We kept the nodes and links from the topologies, but changed the link's maximum bandwidth for it to be constant across all links. We then decided on testing three configurations, consisting in a combination of topologies, MPNN configurations and ES hyperparameters:

- A smaller MPNN trained with the NSFNET topology.
- A bigger MPNN trained with the GÉANT2 topology. The hyperparameters of the ES method are adapted to accommodate the larger network.
- The same as before, but now is trained using both the NSFNET and GÉANT2 topologies. This is achieved by using two environments, one for each, that are used interchangeably during training. During evaluation both environments are used.

Across all configurations the demands to be allocated in the graph will have one of three predetermined sizes, and all will share roughly the same MPNN architecture, as it is the one borrowed from the paper which introduces the use case [22]. The details of each configuration can be seen at Table 1, while the details of the MPNN are as follow:

- The states in the MPNN's graph (the edges in the original network), are represented through a vector with a size specified by the configuration.
- The message function is represented as a single dense layer, with the same number of dimensions as the internal states of the nodes.
- The update function is represented through a Recursive Neural Network, specifically a GRU RNN. The internal dimension of the network is the same as the dimension of the internal state in the graph.
- The readout function is represented as a dense neural network made out of three layers. The two hidden layers have the same dimension, which depends on the configuration. The output layer is made up by a single neuron, as it draws the score for the action introduced as an input to the network.

<table>
<tr><td colspan="2">Topologies</td><td>NSFNET</td><td>GÉANT2</td><td>NSFNET and GÉANT2</td></tr>
<tr><td rowspan="4">GNN Parameters</td><td>Number of message passing iterations</td><td colspan="3">5</td></tr>
<tr><td>Dimension of the graph’s internal state</td><td>20</td><td>35</td><td>35</td></tr>
<tr><td>Number of hidden layers in the readout function</td><td colspan="3">2</td></tr>
<tr><td>Dimension of readout function’s hidden layers</td><td>20</td><td>35</td><td>35</td></tr>
<tr><td rowspan="4">ES Parameters</td><td>Number of iterations</td><td>300</td><td>300</td><td>320</td></tr>
<tr><td>Number of perturbations per iteration*</td><td>128</td><td>640</td><td>640</td></tr>
<tr><td>Noise standard deviation for perturbations</td><td colspan="3">0.05</td></tr>
<tr><td>Noise standard deviation for actions</td><td>0</td><td>0.05</td><td>0.05</td></tr>
<tr><td colspan="2">Optimizer’s Parameters (Adam)</td><td colspan="3">$\alpha = 0.005$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$</td></tr>
<tr><td colspan="2">L2 Regularization Coefficient</td><td colspan="3">0.005</td></tr>
</table>

* Does not take into account perturbations generated by mirrored sampling; the true number of perturbations is double.

Table 1: Summary of the configurations for the evaluation

The code was written in Python with the aid of several libraries. Mathematical computation and representation of vectors was done through the NumPy library [29]. The MPNN was modelled through the Keras library with a Tensorflow backend, although the actual gradient descent was done through our own implementation of the Adam optimizer [31, 30, 34]. Communication between workers was achieved through the use of MPI. Specifically, the operating system used the OpenMPI library, and within the code we used the mpi4py interface [32, 33]. The RL environment was created using through OpenAI’s Gym library [35]. Other noteworthy libraries include NetworkX for working with graphs, and Kspath for efficiently finding the k-shortest paths between every two nodes in the topology [36, 37].

6.2 Analysis of ES hyperparameters

Before proceeding with the evaluation of our method’s performance, we first wish to run several experiments to understand the impact of ES’s hyperparameters, which are once again the number of perturbations, the standard deviation used for generating mutations, and the standard deviation of noise introduced into the action probability distribution. These executions consists on running the NSFNET configuration with changes to one of the hyperparameters. All these executions were run in the local server with 16 workers.

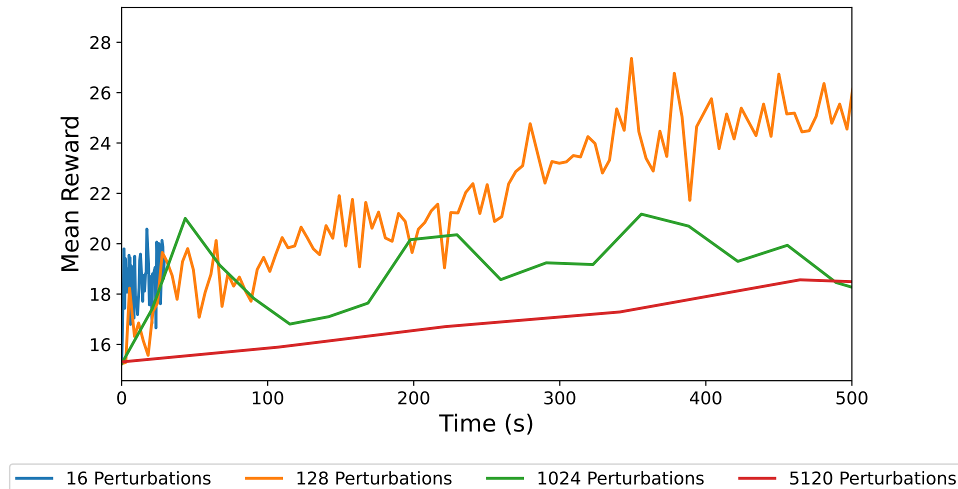


Figure 5: Effect on the number of perturbations

Figure 5 shows the effect on changing the number of perturbations we generate per iteration. This graph shows the state of the model at different points of its training. Each point in the line plot represents one epoch, the x-axis represents the amount of training time taken to reach that point, and the y-axis represents the mean accumulated reward per episode obtained across 60 episodes.

Overall, we can an observe that the more perturbations there are, the slower the training goes. This is reflected by the smaller spacing between each point in the curve as the number of perturbations increase, as well as the speed at which the mean reward increases over time. Out of the different configurations tested only the one with 16 perturbations end in the time frame shown in the graph, in spite running for the same number of epochs in all four scenarios.

On the other hand we see the higher the number of perturbations, the more stable training becomes. For example, the configuration with 16 perturbations is so unstable that is unable to learn, as its mean reward does not increase over its entire execution. With 128 perturbations the learning is still unstable, indicated by how the mean reward tends to fluctuate between epochs, but the model is able to learn overall. Finally, the two configurations with the highest number of perturbations show lower variance in the mean reward per epoch and the more sustained increase of the mean reward.

This experiments confirms our intuition about the number of perturbations. More perturbations are necessary to increase the stability of learning, but doing so will increase the cost of the algorithm, as it will also mean an increase in the number of interactions with the environment. Overall, a sweet spot must be achieved between speed and stability so as to minimize the training time of the algorithm. For example, while 128 perturbations is more unstable than 1024, the speed up it benefits from is more than enough to make up for it and ends up reducing training time overall.

There are also two other facts to point out. The first is that the number of perturbations will need to increase when dealing with larger networks, as the number of parameters, and therefore the size of the solution and the solution space, will increase and more perturbations will be needed to correctly cover the larger solution space. Second, increasing the number of perturbations will not lead to larger gradient descent steps. If we examine the gradient formula back in Algorithm 3 the gradient uses the average of the perturbation, so the number of perturbations will not affect its size.

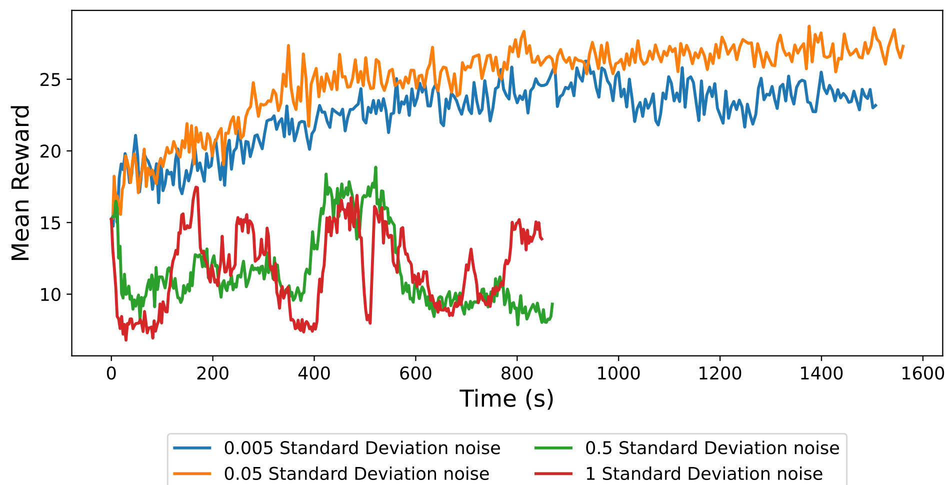


Figure 6: Effect of the standard deviation for the mutations

Figure 6 shows the impact of different values for the standard deviation for the mutations. As a remainder, this standard deviation is the one used for converting perturbations into mutations, also known as σ back in Algorithm 3. From the results, it is clear that using a value of 0.05 is the optimal choice, as it converges in the highest mean reward overall, a value of 26. One one hand, if the value of σ is increased any more the training becomes too unstable and is unable to learn, represented by the fact that the executions where $\sigma = 0.5$ and $\sigma = 1$ the mean reward did not increase by the end of training. On the other, if the value of σ decreases we end up converging in a worse solution, at around a mean reward of 22 with $\sigma = 0.005$.

This graph shows reinforces the ideas we presented back in section 4.1 about the impact of σ. Large values will make that the generated mutations are no longer representative of the solution space near the current solution, and therefore are inadequate for computing its gradient. Small values will make mutations similar to the current solution, which increases the chances of capturing small irregularities and getting stuck in a local maximum. Therefore, the procedure for choosing the value of σ would be to perform a grid search to find the optimal value. Is important to note that this hyperparameter showed no signs of affecting the training speed of the algorithm, as both curves followed a very similar trajectory.

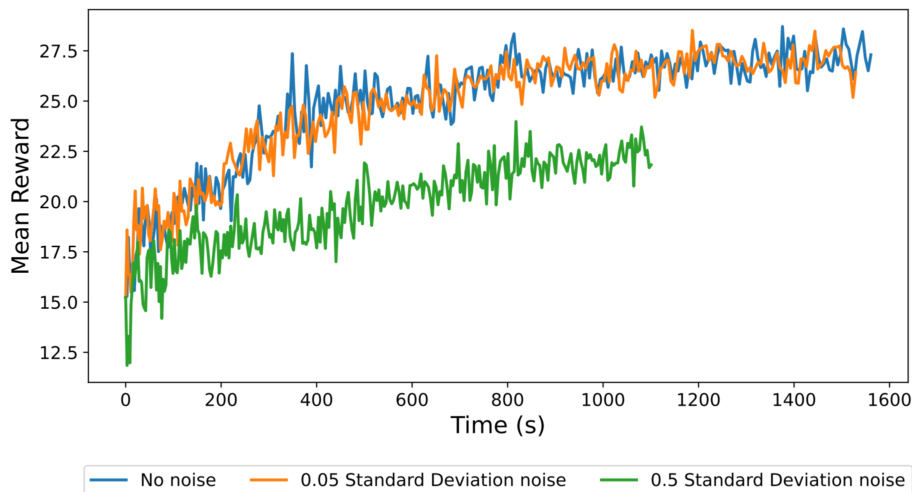


Figure 7: Effect of the standard deviation for the noise to be added to the obtained action probabilities

Figure 7 shows the impact of different values for the noise to be added to the obtained action probabilities. This is also referred as φ back in Algorithm 3. On one hand we observe that the effect of this parameter is very subtle. When moving from adding no noise to a minimal amount of noise, with $\varphi = 0.05$, we see that it did not have an impact over the results, with both curves moving in an identical trajectory. However, by increasing the amount of noise to $\varphi = 0.5$ the mean reward is negatively affected, reducing the point where it converges from a mean reward of 26 to roughly 20.

Ultimately, the addition of noise to the action distribution is not always necessary and it is more reasonable to consider it on a case by case basis. Small amounts of noise should be included in a grid search for finding the correct hyperparameter values of the algorithm, but only with the understanding that most likely no noise should be added.

Another observation to point out from the graph is how adding more noise increased the training speed of the algorithm. As a reminder, all three configurations tested had the same number of epochs, however the configuration with $\varphi = 0.5$ finished after 1100 seconds, while the other two curves took longer than 1500. However, we believe this occurs because when the algorithm performs worse its episodes are shorter: since it performs poorly, the graph is saturated in fewer allocations of demand, hence episodes end sooner. A similar situation can be found in Figure 6 with the curves from configurations with values of $\sigma = 0.5$ and $\sigma = 1$, whose mean reward did not increase over training and they also ended significantly sooner.

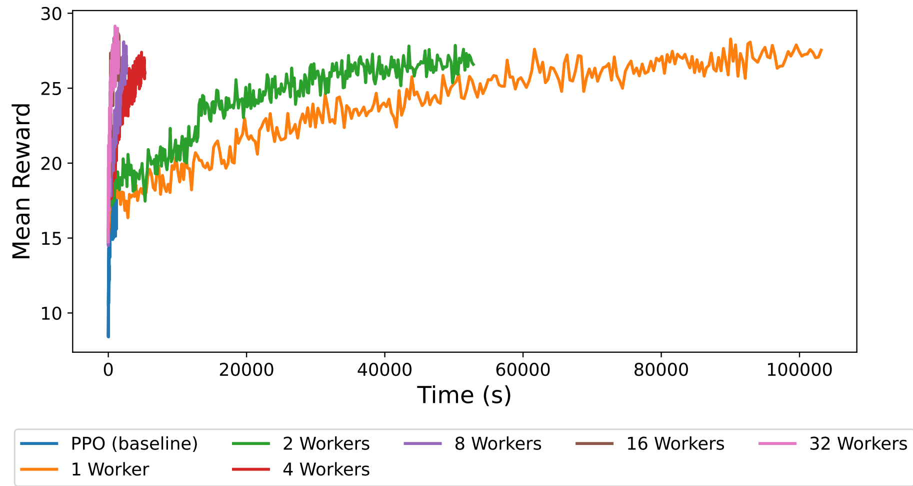
6.3 Performance of ES versus PPO

In the following section we will focus in analyzing the performance of ES against PPO as our reference RL algorithm. In order to make this comparison we will run our implementation of ES across the three configurations with a varying number of workers: one, two, four, eight, 16 and 32 in the local server, and 16, 32 and 64 in the remote AWS server. We will also run PPO, using the implementation offered by the authors of DRL solution for network routing, in both the local and remote server, to act as a baseline [22].

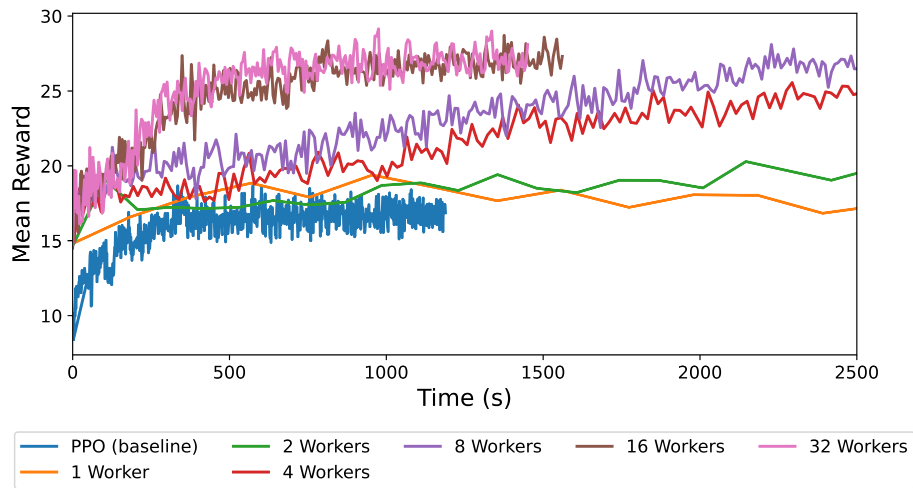
The first set of figures will show the raw results of our experiments. The plots will be similar to those from our previous section: each curve represents one configuration run with either PPO or ES with a specific number of workers. Each point in the line plot represents one epoch, the x-axis represents the amount of training time taken to reach that point, and the y-axis represents the mean accumulated reward per episode obtained across 60 episodes (in the case where we use both topologies we will obtain 60 episodes from each). We will begin with Figure 8, which shows the results of the experiments run with the NSFNET topology and configuration (revise Table 1 for details). Figures 8a and 8b show the results of the experiments run on the local server, where the latter represents a zoomed-in version of the former. Figure 8c shows the results of of the experiments run on the remote AWS server.

The first clear observation that, overall, the more workers we assign, the greater is the speed up the training receives. As all ES experiments use the same number of iterations, we can compare how long they took by simply seeing where the curve ends. Looking back at Figure 8a running ES with one and two workers only are clearly the slowest options, with a great margin from the third slowest being four workers. If we continue by looking into in Figure 8b we see this trend of increasing number of workers resulting in decreasing training times continues. Only when we reach 32 workers this speed up is halted, but it also coincides with the point where all 16 cores of the CPU are occupied, hence this is not due to the limitation of algorithm but of the machine. When moving to the remote server in Figure 8c we can confirm that the trend continues even further.

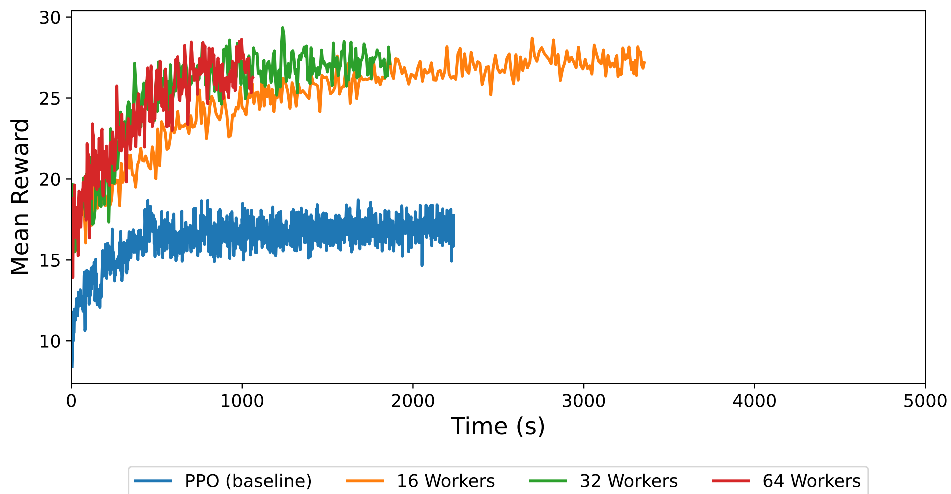
If we compare our results with PPO we can notice two interesting trends. The first one is that by looking at the shape of the curve PPO tends to converge faster than ES with any number of workers, even with 32 and 64 of them in Figures 8c: the fastest ES run, the one with 64 workers, converges at roughly 800 seconds, while PPO roughly at 500. However, across all ES executions we seen that it obtains significantly better solutions than PPO, converging at a mean reward of over 26 instead of PPO's 16. This is unexpected, as we were not anticipating any method to obtain inherently better policies. However, the properties of evolutionary algorithms being more resistant to local minima may give the edge to ES when obtaining a better policy of this specific scenario. What we were expecting, however, is that the number of workers do not affect the mean reward at which ES converges. Looking at graphs eventually all ES executions reach a mean reward of 26, confirming that its performance does not degrade when parallelized.



(a) Results of running 1 to 32 workers in the local server

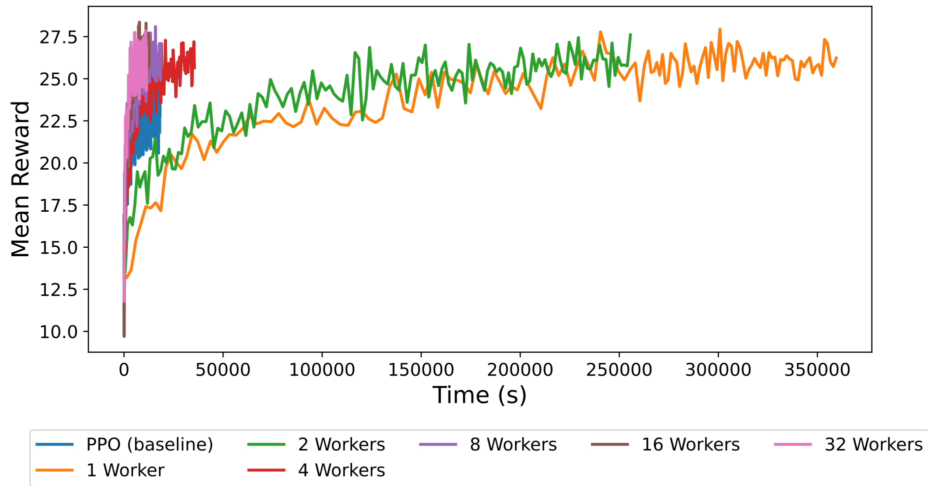


(b) Results of running 1 to 32 workers in the local server (zoomed in)

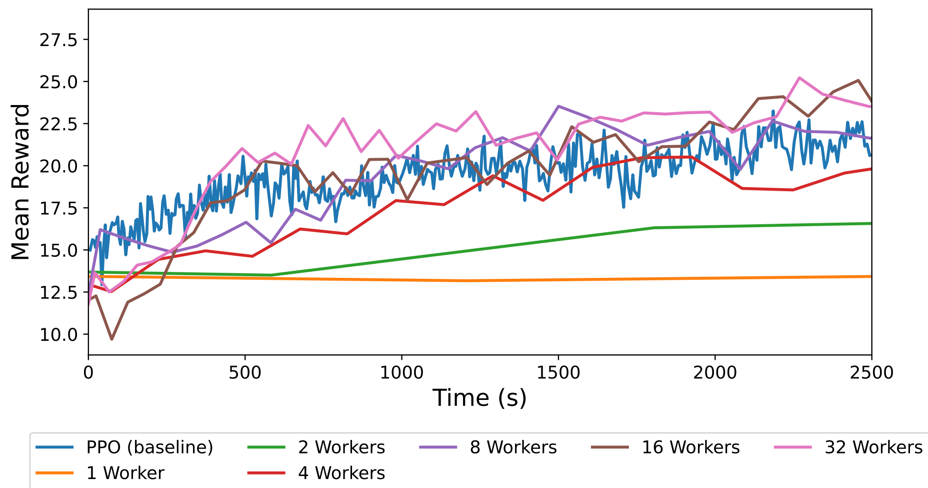


(c) Results of running 16 to 64 workers in the remote server

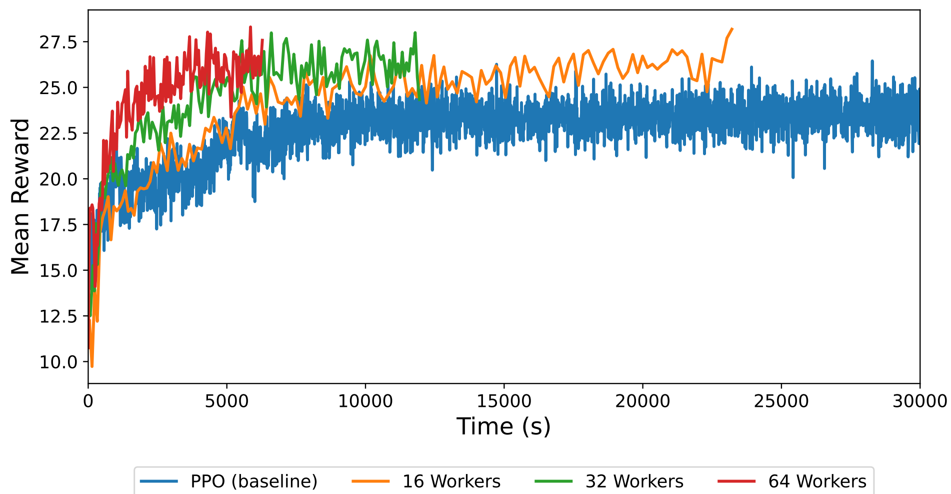
Figure 8: Effect of using ES with varying number of workers over the NSFNET topology



(a) Results of running 1 to 32 workers in the local server



(b) Results of running 1 to 32 workers in the local server (zoomed in)



(c) Results of running 16 to 64 workers in the remote server

Figure 9: Effect of using ES with varying number of workers over the GÉANT2 topology

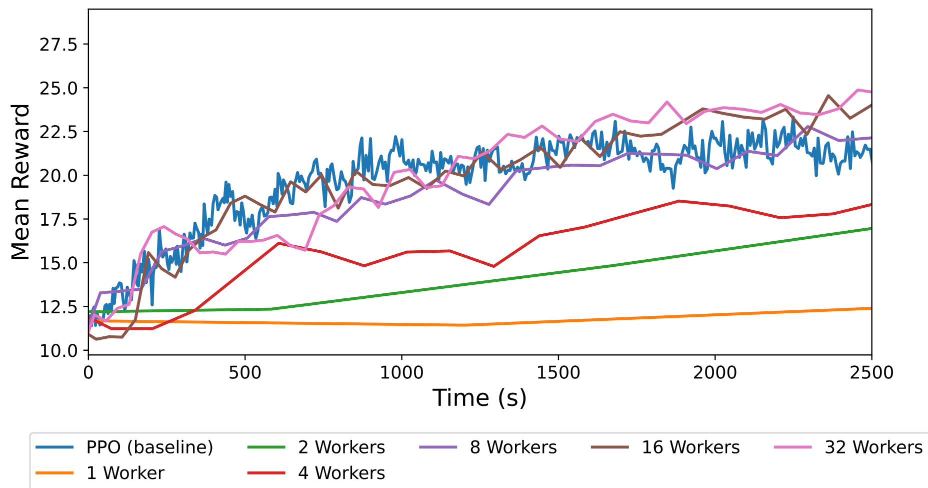
Figure 9 shows the results when run with the GÉANT2 topology and its respective configuration. Just as before, Figures 8a and 8b show the results of the local server, while Figure 8c of the remote server. From it results we can draw similar conclusions to the ones drawn by the results over NSFNET. First, increasing the number of workers speed ups training time without affecting the value at which the algorithm converges, with the exception of increasing from 16 to 32 workers in the local server (Figure 9b), just as before. ES also converges to a higher value than PPO: the former converges at a mean reward of around 26 while the latter at around 22, as seen clearly in Figure 9c.

However, unlike before we see that this time around the shape of the training curve of PPO and ES ends up being much closer to each other. In Figure 9c we can see that the slope of the curve of ES when using 16, 32 and 64 workers is more pronounced than PPO's curve during the first 5000 seconds. This confirms that ES was increasing its mean reward faster than PPO during this period, thus meaning it was learning faster. This contrast to the results in NSFNET, where the main advantage came from ES outperforming PPO's score by a significant margin. Besides this, the only remaining difference between the results in both topologies is that overall the GÉANT2 topology took longer longer to train both in ES and PPO, but that was due to the larger environment and MPNN model.

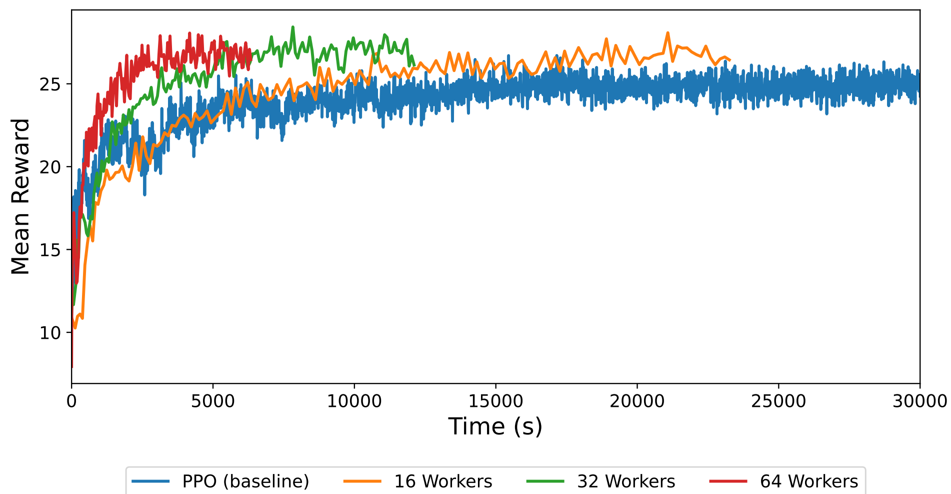
The final set of graphs, shown in Figure 10, show the results when running the model to optimize both topologies simultaneously. The idea with this configuration was not only to test the scalability of ES but also how it performed when training from more that one environment. To do is important to enrich the model by learning from more than one topology at once. This aspect is crucial when we consider the fact that the underlying model is a MPNN, which is invariant to graph isomorphism, meaning that any degradation in performance is more likely to come from the optimization algorithm that the model itself. However, by looking at the results in Figure 10 it is clear that ES works appropriately when trained with both graph topologies simultaneously, as its results mirror those from previous results, specifically those when trained with the GÉANT2 topology alone.

The only notable difference is by comparing the performance of ES with 16 workers against PPO in the remote server. By comparing Figures 9c and 10c we can see than when training only in the GÉANT2 topology and using ES with 16 workers converges slightly faster than PPO. However, when training in both topologies PPO seems to be slightly quicker instead. This may suggest that ES performs slightly worse with multiple topologies relative to PPO, however this result is marginal. A alternative explanation could be the innate instabilities of both methods (as they use randomness in some way or another) caused PPO to be more "lucky" when training in both topologies. Nevertheless the main conclusions related to ES converging at a higher score and being accelerated through additional workers still hold.

(a) Results of running 1 to 32 workers in the local server



(b) Results of running 1 to 32 workers in the local server (zoomed in)



(c) Results of running 16 to 64 workers in the remote server

Figure 10: Effect of using ES with varying number of workers over both topologies simultaneously

6.4 Analyzing ES scalability over the number of workers

With the figures showing the raw data out of the way, let us now examine more in detail the scalability of ES in respect to the number of workers. Figure 11 shows the effect of the number of workers in the time spent interacting with the environment. Each curve represents one of the configurations, the x-axis denotes the number of workers and the y-axis the time dedicated in each iteration to interact with the environment.

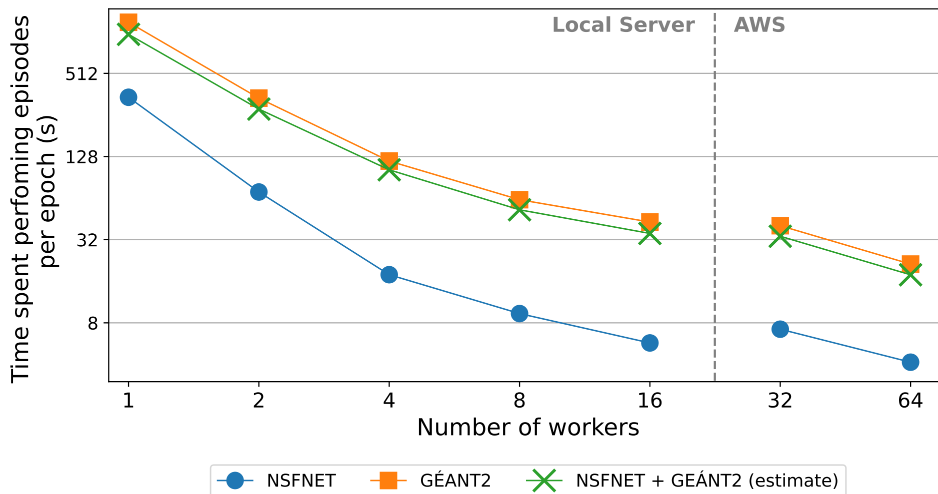


Figure 11: Effect on the number of workers over the time spent per epoch performing episodes

By looking at the downward slope it is clear that increasing the number of workers decreases the time spent by ES interacting with environment across all three configurations. Note that this does not mean that the number of mutations is reduced, but that more mutations are done simultaneously, as the workload is further spread out. We can also see that this relationship is linear as expected: doubling the number of workers will cut in half the time spent (for example, in GÉANT2 from 118 to 62 seconds when increasing from four to eight workers, in NSFNET being from 18 to nine seconds). In the remote server this effect continues, as the difference between 32 and 64 threads the time is reduced from 41 to 21 seconds for the GÉANT2 topology and from seven to four seconds for the NSFNET topology.

Unsurprisingly, overall NSFNET spends the least time interacting with the environment and GÉANT2 the most. In the first place the latter works with a larger network topology, which can take more demands before saturating and thus its episodes last longer. The GÉANT2 configuration also works with a larger model, so it will also take longer for it to compute its action probabilities distribution. Finally, referring back at Table 1, the GÉANT2 configuration generates six times as many perturbations as those in NSFNET, which means it performs six times the number of episodes per iteration.

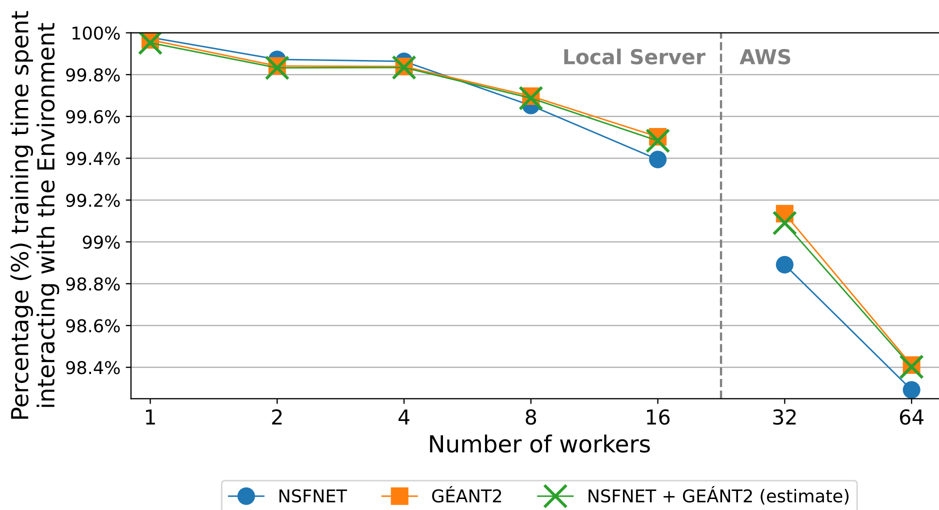


Figure 12: Effect on the number of workers over the relative time spent per epoch performing episodes

To these results we have to add those of Figure 12, which shows the percentage of training time dedicated to performing episodes according to the number of workers. This figure is the same as before, but the y-axis shows the time invested in interacting with the environment as a percentage of total training time. While increasing the number of workers reduces this value, as the raw amount time taken is reduced and the overhead due to communication costs is increased, across all configurations it is clear that interacting with the environment is the most time-consuming part of the training. Even with the configuration with the fastest episodes, NSFNET, and with the highest number of workers tested, it still spends over 98.2% of training interacting with the environment. This comes to show that we still have plenty of margin to increase the number of workers and further decrease the time taken to interact with the environment, solidifying the scalability of ES over the number of workers.

6.5 Summary of results

The final aspect to consider is evaluating how each configuration performs relative to PPO depending on the number of workers available. While we know from the previous figures that the algorithm does scale, the question now is if that is enough to justify its use over PPO. To do so, from the experiments run and presented in Figures 8, 9 and 10 the following steps were completed:

1. We recorded the score at which PPO converged, and established it as the "benchmark".
2. From the results, we extracted the time taken for both PPO and ES (with varying number of workers) to reach that benchmark.
3. We extracted the ratio between ES's time and PPO's time. This is repeated across the three configurations and the different number of workers for ES used.

The resulting graph is the one shown in Figure 13. The y-axis shows how fast ES training time was as a multiple of PPO's training time. For example, a value of "×2" meant that ES was twice as fast to reach the benchmark, or that it took half the time. A value of "×1/2" conversely means that ES took twice as long as PPO to reach the benchmark, or that it was twice as slow. Overall, the higher the y-value, the faster was ES relative to PPO. The horizontal dotted line at "×1" represents the break-even point where both techniques train just as fast. The first thing it can be seen is the confirmation of what it was already shown by Figures 11 and 12: an increase in workers resulted in a linear decrease in training time. However, the degree of the speed-up depended heavily on the configuration used.

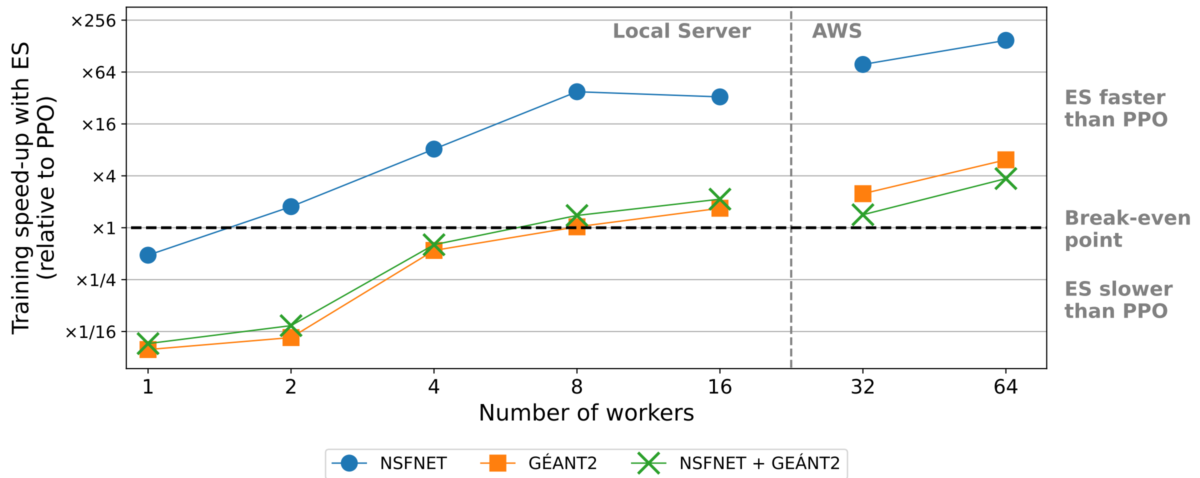


Figure 13: Training speed-up achieved when switching from PPO to ES across different number of workers and topologies

On one hand, NSFNET benefited massively by switching to ES. The only scenario where PPO is faster than ES is when it only has one worker, where it goes twice as fast. However, by increasing to two workers by we see the time taken to reach the benchmark fall by one quarter, from taking twice the time than PPO to slightly over half of the time. Further duplicating the number of workers continues this trend: with four workers the training time is one eighth of PPO's time, and with eight workers ES reaches the benchmark 32 times as fast as PPO. This streak is broken by running 16 workers, where the speed-up stalls. When changing to the remote server the results fall within our original expectations: doubling from 32 to 64 workers halved the time taken to reach the benchmark.

The question then is how duplicating the number of threads results in a four-fold decrease in the training time, as well as how did the speed up stalled when jumping from eight to 16 workers. The answer lies in the method's inherent instability due to its reliance in Gaussian noise. Thus, during those four first experiments not only each iteration was becoming shorter, but also the amount of epochs required to reach the benchmark score was also decreasing due to having better "luck" with the perturbations it sampled. Similarly, that randomness might explain why with 16 workers the speed up stalled, as poorer sampling resulted in taking more iterations that its predecessors to reach the benchmark.

On the other hand, both the GÉANT2 topology and running both topologies offered more moderate results. Compared to the NSFNET results, both curves had a more stable trajectory along our expected outcome of double the workers, half the time. At first the results appear to be abnormally bad, with both one and two workers taking 16 times longer than PPO respectively. However, by looking the trend between four and 16 workers we can observe the desired behaviour: with four workers the training time was roughly twice as slow as PPO, with eight was approximately just as fast, and with 16 around twice as fast.

When switching to the remote server the trend continued. The only notable fact here is how in the local server the GÉANT2 topology by itself was taking slightly longer to reach the benchmark relative to using both topologies at once, while in the remote server it was the other way around. While we discussed this when presenting the results from Figure 10, we believe that this is due to a slight increase in PPO's performance when training in both topologies due to the method's inherent instability.

Another important observation is the clear the gap that exists between the curves using NSFNET and the other two configurations. The main reason why the gap exists is due to the difference in the number of mutations. Looking back at Table 1 we can see that the NSFNET topology used 128 perturbations per iteration (doubled if counting the mirrored perturbations), while the other two episodes used 640, five times as many. As a remainder, the number of perturbations used per each iteration also equals the number of episodes in it. The increased number of perturbations is needed however, as the GÉANT2 and the combined NSFNET and GÉANT2 configurations use a larger MPNN, meaning its solution space is larger and more perturbations are needed to properly cover it when computing its gradient.

Still, even if the number of episodes increases five-fold, from the graph we can observe that roughly NSFNET takes an eight of the time to reach the benchmark compared to the other two configurations. On one hand this can be explained by the fact that NSFNET's initial scores are closer to the benchmark than GÉANT2, and therefore it requires less iterations to reach it. Looking back at Figure 8b we see when applying ES to NSFNET the initial solutions start with a mean reward of around 15 while PPO converges at around 17. On the other hand, looking at Figure 9b shows that when applying ES to GÉANT2 the initial solutions also start with a mean reward of 15 but PPO converges at around 23.5. The difference between the starting points may explain this massive difference, but also hints that NSFNET speedups are too optimistic for new untested scenarios, and instead it is more realistic to expect results similar to the ones obtained with GÉANT2. Other factors also include longer computation times from GÉANT2's larger MPNN, or longer episodes due to its larger topology. In the end, it is clear that problem's size will eventually affect the number of parameters of the network, which in itself will have a direct and indirect (through the effect on the number of perturbations) impact in the training time of ES.

7 Conclusions

In conclusion, in this thesis we have examined the viability of using ES to train RL models as a more scalable alternative to RL. On one hand, the algorithm has proven beyond reasonable doubt that it was able to scale to a high number of workers, more than enough to cover all available CPU cores in a machine, without being limited by memory consumption or degrading its performance due to its overhead. While in our experiments only reached to up to 64 workers, profiling results show that we still have plenty of margin to further increase the number of workers while causing a linear decrease in training time. Of course this growth is limited by the amount of CPU cores available, although they do not have to come from the same machine necessarily. Conversely, one aspect this method is unable to do is to take advantage of any accelerators such as GPUs.

While the algorithm scaled well with the number of workers, it was unclear how well it did with the size of the problem. In the tested use case it was clear that larger networks increased the cost of the algorithm: episodes take longer and require of larger models, which operate slower and requires more interactions to train. The fact that currently a network such as GÉANT2 with 26 nodes and 74 edges requires for 8 workers to match PPO shows that this method in its current state will not be appropriate for massive problems, at least without the adequate hardware to match it.

In spite of this, ES offers benefits besides scalability alone. One of them was unexpected, which better scoring models than those obtained by PPO. This means that even in the cases where ES does not provide improved training times, it may be worth the consideration to use it nevertheless due to its better results, at the very least in this the studied use case. Another advantage of ES it that it only depends on three hyperparameters, and arguably only two without considering noise in the action distributions which has a marginal effect, if any. As a result is easier to launch and optimize, especially compared to the many hyperparameters PPO has (size of mini-batches, experience replay horizon, discount factor, clipping values, GAE parameter...).

Overall, ES is in fact a viable alternative under certain conditions. First, just because of the better scores alone is a more than enough reason for considering using ES. ES is also interesting in those complex scenarios where the environment does not follow the assumptions that most RL algorithms depend on (as commented at the end of Section 4.1). Moreover, ES will increase its advantage over traditional RL algorithms when using models with fewer trainable parameters. Even in cases where none of the above apply, ES can still prove to be more adequate if the amount of hardware resources at disposal is enough to support enough worker instances, specially if accompanied with a correct selection of hyperparameter choices.

Additionally, we believe that the work done in this thesis can be expanded in several ways as to improve its competitiveness:

- Attempt to perform the analysis with more uses cases, to see if the higher scores translate to other scenarios and environments that the ones tested here. If so there is a strong argument to use ES over PPO for purely obtaining better models.
- Change the internal ES algorithm for a more complex one. Algorithms such as CMA-ES samples higher quality perturbations, meaning they are more effective at directing training towards the global maximum. The main issue lies that it does this sampling using a learned covariance matrix, while NES currently assumes no covariance, meaning the cost of the algorithm increases by $O(n^2)$, where n is the number of parameters in the solution. Nevertheless, using CMA-ES, or finding a way of lowering its cost, may reduce the number of perturbations generated per epoch and/or the number of epochs needed to reach convergence, compensating for its higher cost.
- Consider other approaches mentioned but untested in the original OpenAI paper to reduce the computational cost, such diving between workers not only the number of episodes but also the parameters it is affecting.
- Additionally considering other modifications to the ES algorithm. For example, while uncommon in ES, a population of perturbations could be kept between iterations. This could reduce the number of perturbations to be generated in iterations beyond the first, hence also reducing the number of interactions with the environment.

References

[1] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[2] Christopher John Cornish Hellaby Watkins. PhD thesis, Chris Watkins, 1989.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an Introduction*. Bradford Books. MIT Press, 2nd edition, 1998.

[4] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, pages 317–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, and et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[6] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180–1190, 2008. Progress in Modeling, Theory, and Application of Computational Intelligenc.

[7] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[10] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, and et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[11] Peter Stone, Richard S. Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13:165 – 188, 2005.

[12] Machine learning finds new ways for our data centers to save energy, Dec 2016.

[13] Ingo Rechenberg. *Evolutionsstrategie – Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Frommann-Holzboog, 1973.

[14] Hans-Paul Schwefel. *Numerische Optimierung von Computer-Modellen Mittels der Evolutions strategie mit einer Vergleichenden Einführung in die hill-climbing- und zufallsstrategie*. PhD thesis, 1974.

[15] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, and Jürgen Schmidhuber. Natural evolution strategies, 2011.

[16] Nikolaus Hansen. Invariance, self-adaptation and correlated mutations in evolution strategies. 10 2000.

[17] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[18] Cristopher Hesse and Jonathan Ho. Openai/evolution-strategies-starter: Code for the paper "evolution strategies as a scalable alternative to reinforcement learning".

[19] Dimo Brockhoff, Anne Auger, Nikolaus Hansen, Dirk V. Arnold, and Tim Hohm. Mirrored Sampling and Sequential Selection for Evolution Strategies. In *PPSN*, Parallel Problem Solving from Nature (PPSN XI), pages 11–21, Warsaw, Poland, September 2010.

[20] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[21] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry, 2017.

[22] Paul Almasan, José Suárez-Varela, Arnau Badia-Sampera, Krzysztof Rusek, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Deep reinforcement learning meets graph neural networks: exploring a routing optimization use case, 2020.

[23] Hans-Werner Braun. Nsfnet – the national science foundation network.

[24] Welcome to the géant2 website, Feb 2011.

[25] Miriam Di Ianni. *Efficient delay routing*, volume 196, pages 258–269. 01 2006.

[26] Deep Medhi and Karthik Ramasamy. Chapter 2 - routing algorithms: Shortest path, widest path, and spanning tree. In Deep Medhi and Karthik Ramasamy, editors, *Network Routing (Second Edition)*, The Morgan Kaufmann Series in Networking, pages 30–63. Morgan Kaufmann, Boston, second edition edition, 2018.

[27] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 15–28, New York, NY, USA, 2015. Association for Computing Machinery.

[28] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. 45(4):15–28, aug 2015.

[29] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser,

Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[30] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[31] François Chollet. keras. `https://github.com/fchollet/keras`, 2022.

[32] Software in the Public Interest, Inc. OpenMPI Project Version 4.1, 2021.

[33] Lisandro Dalcin and Yao-Lung L. Fang. mpi4py: Status update after 12 years of development. *Computing in Science Engineering*, 23(4):47–54, 2021.

[34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[35] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[36] NetworkX Developers. NetworkX, 2021.

[37] Chong Hon Fah and Lee Shangqian. Kspath, 2021.

[38] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *CoRR*, abs/1604.00772, 2016.