ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

A COMPARATIVE STUDY OF OPTIMIZATION ALGORITHMS IN PYTHON FOR NEURAL ARCHITECTURE SEARCH

Διπλωματική Εργασία

του

Λιαντζάκη Κωνσταντίνου

Θεσσαλονίκη, Φεβρουάριος 2021

A COMPARATIVE STUDY OF OPTIMIZATION ALGORITHMS IN PYTHON FOR NEURAL ARCHITECTURE SEARCH

Λιαντζάκης Κωνσταντίνος

Πτυχίο Εφαρμοσμένης Πληροφορικής, Πανεπιστήμιο Μακεδονίας, 2018

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής:
Μαργαρίτης Κωνσταντίνος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την

Μαργαρίτης Κωνσταντίνος	Ρεφανίδης Ιωάννης	Σιφαλέρας Άγγελος
....	....	....

Λιαντζάκης Κωνσταντίνος

....

Περίληψη

Η συνεχής ανάπτυξη του τομέα του Deep Learning έχει οδηγήσει σε εντυπωσιακές εφαρμογές νευρωνικών δικτύων αλλά και την αυτοματοποίηση διαδικασιών σε διάφορους τομείς. Μέχρι πρότινος, η σχεδίαση αρχιτεκτονικών νευρωνικών δικτύων ήταν μία διαδικασία που υλοποιούταν χειροκίνητα από εξειδικευμένους και καταρτισμένους επαγγελματίες. Η ανάδυση του ερευνητικού πεδίου της Αναζήτησης Αρχιτεκτονικών Νευρωνικών Δικτύων έρχεται να αντιμετωπίσει το πρόβλημα της αυτοματοποίησης της εύρεσης καλών αρχιτεκτονικών για νευρωνικά δίκτυα. Σε αυτή τη μελέτη, ξεκινάμε με τη χρήση της Ενισχυτικής Μάθησης για την Αναζήτηση Αρχιτεκτονικών Νευρωνικών Δικτύων, αξιοποιώντας τον αλγόριθμο Deep Q-Learning. Έπειτα, συνεχίζουμε την έρευνά μας με την υλοποίηση ενός εξελικτικού αλγορίθμου. Τέλος, χρησιμοποιούμε ένα σύγχρονο εργαλείο για τη γλώσσα προγραμματισμού Python, το οποίο περιλαμβάνει υλοποιήσεις μεθευρετικών αλγορίθμων, με σκοπό να διεξάγουμε πειράματα και να συγκρίνουμε τρεις από αυτούς τους αλγορίθμους: Ant Colony Optimization, Particle Swarm Optimization και Artificial Bee Colony. Για να επισπεύσουμε τα πειράματά μας, χρησιμοποιούμε μερικά από τα πιο πρόσφατα ευρήματα στον κλάδο αυτό, όπως το σύνολο σημείων αναφοράς NASBench-101. Μέσω της έρευνάς μας, θα αξιολογήσουμε τις επιδόσεις των αλγορίθμων, θα τους συγκρίνουμε μεταξύ τους και με την Τυχαία Αναζήτηση και θα αναλύσουμε τις δυνάμεις αλλά και ανεπάρκειές τους.

Λέξεις Κλειδιά: Νευρωνικά Δίκτυα, Deep Learning, Αναζήτηση Αρχιτεκτονικών Νευρωνικών Δικτύων, Ευρετικοί Αλγόριθμοι, Ενισχυτική Μάθηση, Deep Q-Learning

Abstract

The continuous development of Deep Learning has led to impressive applications of neural networks and automation of processes in various domains. Until recently, designing neural architectures has been a manual task for specialized and knowledgeable professionals. The emergence of Neural Architecture Search as a research field has come to tackle the problem of automating the procedure of finding good neural architectures. In this study, we begin with the usage of Reinforcement Learning for Neural Architecture Search, using the Deep Q-Learning algorithm. Then, we continue our research with the implementation of an evolutionary algorithm. Finally, we utilize a modern framework for the Python programming language that includes implemented versions of metaheuristic algorithms, in order to conduct experiments and compare three of these algorithms: Ant Colony Optimization, Particle Swarm Optimization and Artificial Bee Colony. To expedite our experiments, we use some of the latest findings in the field, such as the NASBench-101 benchmark. Through our research, we assess the performance of each algorithm, evaluate how they compare to each other and the Random Search and analyze their strengths and inadequacies.

Keywords: Neural Networks, Deep Learning, Neural Architecture Search, Metaheuristic Algorithms, Reinforcement Learning, Deep Q-Learning

Acknowledgements

The completion of this thesis is an urge to rewind and look back at all the valuable lessons this Master’s Programme has taught me and the importance of patience, perseverance, diligence, consistency, and hard work. It is also a great opportunity to reflect on the people that have helped me the most, supported me, inspired me to set the bar high and achieve difficult and ambitious goals.

I would like to express my gratitude to my supervisor, Professor Konstantinos Margaritis, and the PhD Researcher, George Kyriakides, for their invaluable contribution to this M.Sc. thesis. Their continuous guidance, constructive feedback and attention to detail have been unmeasurably helpful and inspiring to me.

Finally, I would also like to thank my family, my close friends and those who have been there for me, supported and encouraged me over the past years.

Contents

Table of Figures

Tables of Tables

Abbreviations

ABC = Artificial Bee Colony

ACO = Ant Colony Optimization

GACO = Extended Ant Colony Optimization

GRU = Gated Recurrent Units

LSTM = Long Short-Term Memory

MDP = Markov Decision Process

NAS = Neural Architecture Search

NORD = Neural Operations Research & Development

PPO = Proximal Policy Optimization

PSO = Particle Swarm Optimization

PyGMO = Python Parallel Global Multi-Objective Optimizer

ReLU = Rectified Linear Unit

RNN = Recurrent Neural Network

Tanh = Hyperbolic Tangent

1 Introduction

1.1 Motivation

Nowadays, with the exponential technological development in software, but mostly in hardware, we have seen various scientific fields' galloping improvement exceed any expectation. The field of Deep Learning, using Artificial Neural Networks, has achieved impressive tasks and excelled at certain fields and applications in a way that was unfathomable in the past. Some characteristic examples are image recognition [1], [2], natural language processing [3], [4], [5], speech recognition [6], [7], or even more compound and ensemble applications of all of the above, such as the popular self-driving cars [8], [9].

We are living in the era of the 4th industrial revolution [10]. The construction of neural architectures has so far been a task that is performed by scientists in a way that is not well automated. The emergence of Neural Architecture Search as a research sub-field of Deep Learning has come to solve this problem. As it is also stated in [11], it has come like an effect of the ripple of natural evolution that we are now trying to automate the construction of the best possible neural architectures, just like with other similar inventions and findings in the human history; a similar occurrence to every industrial revolution we have experienced up until today. Therefore, it has become very interesting to explore the various approaches, methods and algorithms to achieve the automation of optimal neural architecture construction.

1.2 Research Aims

In this thesis, our main purpose is the comparison of certain methods we use to implement Neural Architecture Search solutions. We use a Reinforcement Learning algorithm and various metaheuristic algorithms. Some of the metaheuristic algorithms are pre-implemented through a specific Python framework. Through our research, we analyze the adequacies and inadequacies of each of the implemented methods and proceed with a more analytic comparison of their performance and results on some very specific datasets.

1.3 Approach

In our introductory chapters, we go through some basic terminology and theoretical background that is necessary to understand the methods and algorithms we use in order to implement Neural Architecture Search solutions. We cover Neural Networks, Reinforcement Learning, all the Metaheuristic Algorithms we use and some more details about Neural Architecture Search.

The next section of this thesis discusses the first Neural Architecture Search implementation, which uses a Reinforcement Learning algorithm, Deep Q-Learning [12]. We present different implementations, with the main difference between them being the search space. We present results that are indicative of how the algorithm performs in smaller spaces, but also some much wider ones.

Further on, we proceed with the implementation of an evolutionary algorithm inspired by [13]. We implement the constructed algorithm as it is presented in the paper and experiment with various parameters in order to improve its performance.

The final part of our experiments includes the implementation of three metaheuristic algorithms. To be more specific, we use a Python framework, called PyGMO [14], which contains pre-implemented versions of the metaheuristic algorithms. We formulate our problem as an optimization problem and feed it into the various algorithms (with or without constraints). We conclude with a detailed comparison of the three metaheuristic algorithms and Random Search.

1.4 Study Contribution

The main contribution of this thesis is the detailed comparison of various metaheuristic algorithms and Random Search, showing the adequacies and inadequacies of each one of them. Our experiments also show any inadequacies of the relatively weak algorithm of Deep Q-Learning. Moreover, the utilization of various state-of-the-art tools and scientific findings, such as NASBench-101 [15], NORD [16] and PyGMO [14], show how many different tools we have in our arsenal, ready to be utilized and combined in any way we can imagine and consider feasible to be flexible and efficient in our approaches and implementations.

1.5 Study Outline and Structure

Chapter 2 is an introduction to some necessary terms related to Deep Neural Networks. We go over some basic terminology and functionality specifications of Neural Networks that help us understand how they are functioning and performing computations.

Chapter 3 is an introduction to Reinforcement Learning. We go through some basics of Reinforcement Learning and refer to an initial implementation of Reinforcement Learning. We also include more details about Deep Q-Learning, which is the algorithm we are using in our first major implementation for Neural Architecture Search.

Chapter 4 presents in more detail about the metaheuristic algorithms we use in our implementations. We cover Evolutionary Algorithms [17], Extended Ant Colony Optimization [18], [19], Particle Swarm Optimization [20], [21] and Artificial Bee Colony [22], [23].

Chapter 5 presents in more detail Neural Architecture Search. We discuss the need for Neural Architecture Search in Deep Learning and refer to some related work.

Chapter 6 introduces us to the environment of our implementations. We discuss certain tools that we use in all of our implementations and some specific limitations that the usage of these tools ultimately entails.

Chapter 7 includes the first group of our implementations, using the Deep Q-Learning algorithm. We present different implementations that include various versions of the agent/controller and the search space. We include experiments of the parameter experimentation phase and present results of the performance of the algorithms in each one of them.

Chapter 8 covers the implementation of an evolutionary algorithm. We implement an algorithm that is inspired by [13] and we proceed into experimenting with the various parameters and present some results of the algorithm’s performance.

Chapter 9 presents the use of PyGMO. We formulate the problem of Neural Architecture Search, subject to certain limitations, so that it is PyGMO compliant and allows us to use certain pre-implemented metaheuristic algorithms of PyGMO on solving it. We present some detailed experiments in which we compare the performance of three different metaheuristic algorithms compared to the performance of Random Search.

Chapter 10 is the concluding section. We summarize the findings of our experiments and include some suggestions for future work and experiments.

Finally, any related code to the implementations that we present in this thesis can be found in the following repository: https://github.com/liantzakis-it/NAS-MSc-Thesis .

2 Deep Neural Networks

2.1 Introduction

In this section, we get into more detail about what Deep Neural Networks are. We explain some fundamental terminology that revolves around neural networks, such as the neurons, the activation functions and the layer types. Neural networks are the cornerstone of various popular applications in the modern world, such as the ones described in section 1.1. Ergo, it is imperative to understand the role of its constituents and the various calculations that take place during the learning process of a neural network.

2.2 Artificial Neurons and Activation Functions

Artificial neurons are essentially the atoms of an Artificial Neural Network. They are the blocks from which layers are built, which eventually construct a neural network. Artificial neural networks aim to imitate the way the human brain functions. On that ground, artificial neurons' purpose is to imitate the functionality of the actual human brain neurons as calculation units to perform complex calculations characterized by nonlinearity.

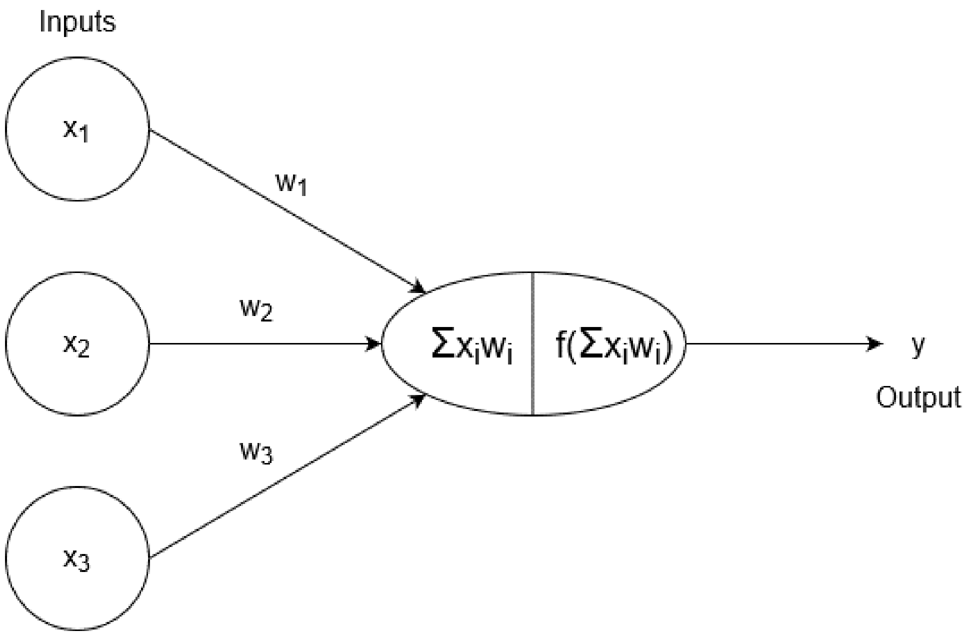


Figure 1: Artificial Neuron with 3 inputs

On a high-level, the entire calculation process of an artificial neuron is relatively simple. An artificial neuron receives a certain input (or a set of inputs), and based on that input and an activation function, it concludes to and produces an output (Figure 1). An activation function is merely a function which will determine the output of the layer [24]. There are various activation functions, among which: Linear [25], Softmax [2], Rectified Linear Unit [26], [27] or ReLU and Hyperbolic Tangent [28], [29] or tanh (Table 1). These are 4 common activation functions that we also used in our experiments. The artificial neuron's performance will be improved by training it, which is a procedure that would update its weights. We can see in Figure 1 how the artificial neuron receives 3 inputs, calculates their weighted sum based on the defined weights for each input, and then feeds that result in the activation function in order to produce the final output. These are the weights that will be optimized in order to minimize the cost function. The cost function is essentially a function which evaluates the performance of our network, based on the actual produced values and the target ones. For instance, if we were to use the Mean Squared Error function as our cost function (which happens to be the one we are using as a cost function for our controller networks in the Reinforcement Learning implementations), we would practically calculate the mean squared difference between the actual and the desired outputs and proceed by updating the weights in order to minimize the cost via a method that is called backpropagation. The speed our weights will get updated also depends on a hyper-parameter called learning rate, which basically dictates the pace that our network will learn.

Linear	$f(x) = x$
Softmax	$f(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, x \in \mathbb{R}^k$
Rectified Linear Unit (ReLU)	$f(x) = \max(0, x)$
Hyperbolic Tangent (tanh)	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Table 1: Activation functions and formulas

2.3 Neural Networks

We have already mentioned in the previous section that artificial neurons group up and create a structure that is called layer. Consequently, a neural network consists of multiple layers. It is important to differentiate the three different categories of layers. Firstly, there is the input layer. This is a layer that will feed the network with the necessary inputs so that it can perform all the necessary calculations and produce the output. Secondly, the input layer is followed by the so called “hidden” layers. The hidden layers are the ones that perform all the intermediate calculations that will lead to the neural network’s output. Finally, following the hidden layers, comes the output layer, which is essentially the layer that outputs the neural network’s computed values. It is important to note that the layers do not need to be sequentially connected (e.g. the input layer feeds the 1st hidden layer, the 1st hidden layer feeds the 2nd hidden layer etc.), but a more abnormal structure can be constructed (e.g. the input layer feeds the 1st and 3rd hidden layers, the inputs of the 4th hidden layer come from the 1st and 2nd hidden ones, and the output layer’s inputs comprise the outputs of three different hidden layers). We mostly explore these architectures in our experiments and we prove that, in certain scenarios, sequential neural architectures constitute a pathway to inadequacy.

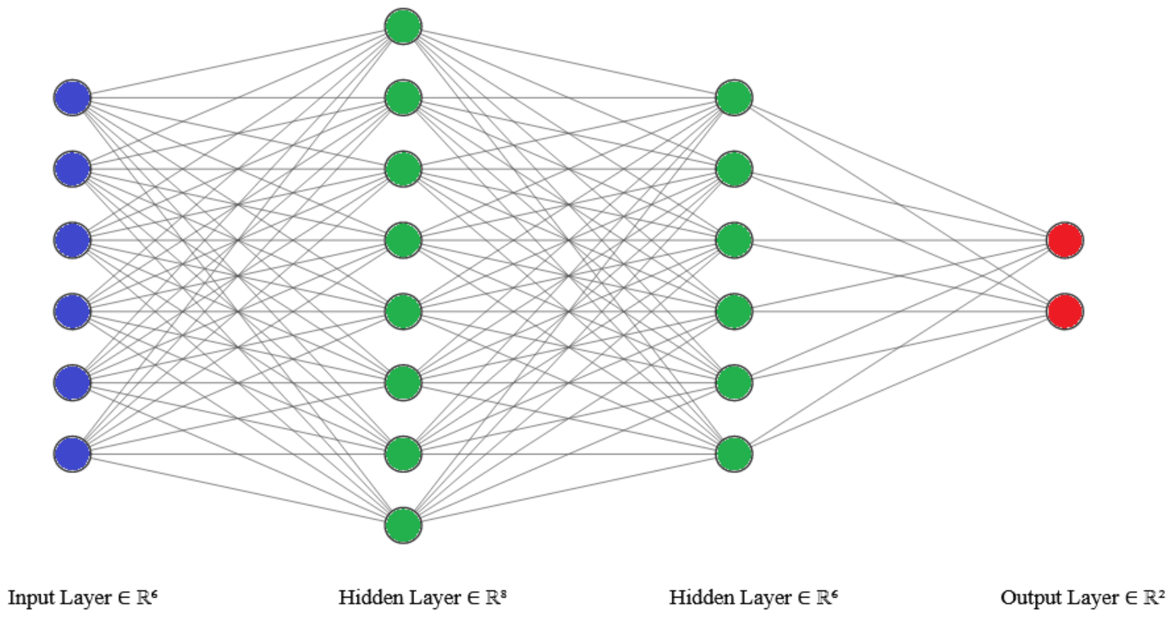


Figure 2: Neural network with 2 hidden layers

The training process is something we also mentioned in the previous section when we were talking about artificial neurons. Now that we have a better picture of what an actual neural network looks like (Figure 2), the description of the training process can be easier to understand. A feed-forward process will be followed in order to carry the inputs of the network all the way to the output. The inputs will be fed to the first hidden layer, the neurons of which will perform the required calculations and feed the second hidden layer, as depicted in Figure 2. Then, the second hidden layer will perform the necessary calculations and feed its outputs to the output layer, which are now ready to produce their values. At this point, we have what we need to calculate the cost, using the cost function we mentioned in the previous section, starting from the output layer for which we know the actual values (the ones it produced) and the target ones. Using backpropagation, in a backward-transfer manner, we will propagate the results backwards and update the weights of the neurons so as to minimize the cost. Backpropagation is a method that, using algorithms like gradient descent or variants of it, like stochastic gradient descent, computes the gradient of the loss with respect to each weight. This is a process that is performed layer by layer, starting from the output one, propagating the results and updating the weights in a backward manner.

2.4 Layer Types

In this section, we talk about a few different layer/neuron types that exist. We focus on some specific ones, which we used in our implementations, such as convolutional, LSTM and pooling layers.

To begin with, convolutional layers [30], [31], and ultimately convolutional networks, are mostly used in tasks related to image recognition and anything that revolves around it. These layers basically excel at the recognition of certain patterns and the extraction of certain special features. For instance, a convolutional layer might be good at detecting edges or round objects or sharp corners. A common example in the case of image classification is the cat vs dog one. Certain convolutional layers might detect certain features to manage to properly classify the images, starting from very abstract ones, like long edges, and moving on to more complete ones, like eyes. Convolutional layers base their calculation process on an *n x n* kernel and the defined stride. For example, assume a 3x3 kernel, a stride of 1, and an input image 9x9. The process is the following:

- Start from the top left corner of the 9x9 image and “select” a 3x3 part of it

- Perform element wise multiplication for the 3x3 selected section and the 3x3 kernel
- Sum the elements of the resulting matrix
- Move 1 position to the right (since stride = 1)
- Repeat the 3 initial steps until you have reached the end of the columns
- Move 1 row down (since stride = 1)
- Repeat the 4 initial steps until you have reached the bottom right corner of the 9x9 input

It is worth noticing here if this specific process is followed, then the effect of dimensionality reduction will occur. Specifically, a 3x3 kernel and a stride of 1 applied on a 9x9 input will produce a 7x7 result. There are certain techniques to prevent dimensionality reduction, such as the one of zero-padding [32].

The pooling layers' calculation process is very similar to the one we have described step-by-step in the previous paragraph. The major difference is that we do not possess a kernel/matrix of values we will use to perform element-wise calculations, but just a filter-size and an operation to perform, such as getting the average or the maximum of the elements. There are different types of pooling, such as max-pooling [33] and average-pooling [34]. Pooling layers constitute an approach that assists in down sampling feature maps by abstracting the features of greater areas into smaller, more compact ones. It also makes the network translation invariant. We present one max and one average pooling example to make the calculation process more understandable. Assume the following:

- Filter size: 2x2
- Stride: 2
- Input size: 4x4

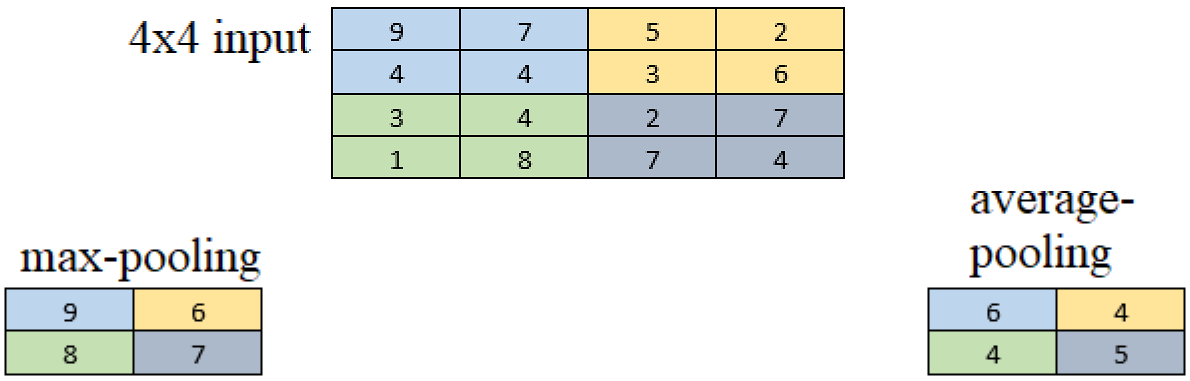


Figure 3: Max & average pooling

Having a stride of 2 and using a 2x2 filter size on a 4x4 input, results in a 2x2 output, as depicted in Figure 3.

Finally, when it comes to LSTMs [35], [36], which stands for Long Short-Term Memory, we are talking about an artificial recurrent neural network that differs from the regular feed-forward networks due to the fact that it has certain feedback connections, enabling it to be performing very well in sequences of data (e.g. speech and video) [37]. The fact that LSTM cells are composed of various gates, with extra emphasis on the forget gate, is what differentiates them from regular recurrent units. Recurrent Neural Networks, in their regular form, are networks optimized to remember things from their past experiences. However, when the dependencies become very long-term, regular RNNs tend to not perform that well. For example, if we depend on a regular RNN to predict the next word in a sentence, the dependencies of which are relatively “close” (e.g. one sentence away), it is going to do just fine. If the related dependencies, though, are some long-term ones (e.g. they are distanced at about 10 sentences away), the performance of regular RNNs is not going to perform well. With LSTMs, the long-term dependencies do not constitute a problem. As we have said before, LSTMs are composed of multiple gates; four, specifically. Technically, when we are talking about an LSTM layer, we are talking about four different layers that each one of them performs a different task. An important mechanism of LSTMs is the mechanism of “cell state”, which represents the internal state of the LSTM, or in other words, what it currently remembers and knows. The four layers we have been talking about decide the following, in this exact order:

- What the LSTM is going to forget (remove from the cell state)
- What the LSTM is going to change in its cell state
- What the LSTM is going to add in its cell state
- What the LSTM is going to output

As we can see, through their initial three gates, LSTMs tend to filter information (based on the input they have just received) and decide what they need to discard, modify and add. This way, they have the luxury of remembering what is useful and discarding information that is not of use to them anymore. Their ability to remember what is useful from their past experience is what enables them to perform so well in long sequences of data while maintaining good performance standards [38]. Finally, the output of the LSTM is going to be a part of the cell state that the LSTM is going to pick. The usage of the Sigmoid [25] and Hyperbolic Tangent functions (Table 2) is extensive in the LSTM gates we have just mentioned.

Sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$
Hyperbolic Tangent (tanh)	$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

Table 2: Sigmoid and Hyperbolic Tangent

The reason that Recurrent Neural Networks, and more specifically LSTMs, are useful in our implementations is because we are treating the constructed neural architectures as sequences, in which the order of the layers matters, just like the order of words matters in the formulation of a sentence.

3 Reinforcement Learning

3.1 Introduction

Reinforcement Learning is a sub-field of Machine Learning which differs from other types of learning. In the other types of learning, we might have cases where we leverage labeled data, which is the case of Supervised Learning [39], or data that are unlabeled and need to be properly structured and clustered in a way, which is the case of Unsupervised Learning [40]. There might also be a combination of the above, which in turn is the case of Semi-Supervised Learning [41], [42]. The essence of Reinforcement Learning revolves around an entity performing actions and not around labeled or unlabeled data. In recent years, we have seen some fantastic applications of Reinforcement Learning that have achieved miraculous tasks, such as playing board games [43], [44] and video games [45], [46] and defeating the best players in the world. Other interesting applications include Natural Language Processing (NLP) tasks that even achieve proper dialogue generation [47].

In the following sections, we explain Reinforcement Learning, the algorithm of Deep Q-Learning, which is an algorithm we used in one of our implementations and make a slight reference to a small introductory application we examined and implemented. We base our analysis and explanation on various surveys and overviews that currently exist in the literature about Reinforcement Learning [47], [48], [49], [50].

3.2 The essence of Reinforcement Learning

To begin with, we define Markov Decision Processes (MDP), which are related to planning and decision-making problems. Afterwards, we relate MDPs to Reinforcement Learning. There is a set of elements that can help us concisely define an MDP: (*S*, *A*, *T*, *R*), with *S* being the state space, *A* being the action space (all available actions), *T*, or more properly defined *T(s,a,s')* is the transition mapping for our state space. *T(s,a,s')* is essentially a function that defines the probability of moving to state *s'*, if action *a* is taken when the agent is on state *s*. This can be deterministic or non-deterministic. For example, the agent might be in a certain position and select an action that moves him upwards in a grid. We might be in a deterministic environment, in which the agent, when choosing to move upwards, has 100% chance to move upwards. However, we might be in a non-deterministic environment in which, when the agent selects to move upwards (action *a*), from state *s*, there is only 80% chance that it moves upwards and 20% chance that it moves downwards. Finally, *R*, or *R(s)*, represents the reward function, which dictates the immediate reward the agent gets from being in a state *s*. The reward can also be defined not just based on the state, but on the combination of the action and the state, too. It is also worth noting two more elements: An element that is not necessary to define in an MDP, but can definitely be useful, is s_0 which defines the starting state of our agent. Finally, the discount factor $\gamma \in [0,1]$ is a parameter that dictates the emphasis our agent puts on immediate or future rewards. A gamma discount factor equal to 0 means that our agent will only emphasize on the immediate rewards. In Figure 4, we can see an overly simplistic MDP, in which we have two states, s_1 and s_2, two possible actions, a_1 and a_2, $R(s_1) = 10$ and $R(s_2) = 0$, and certain probabilities.

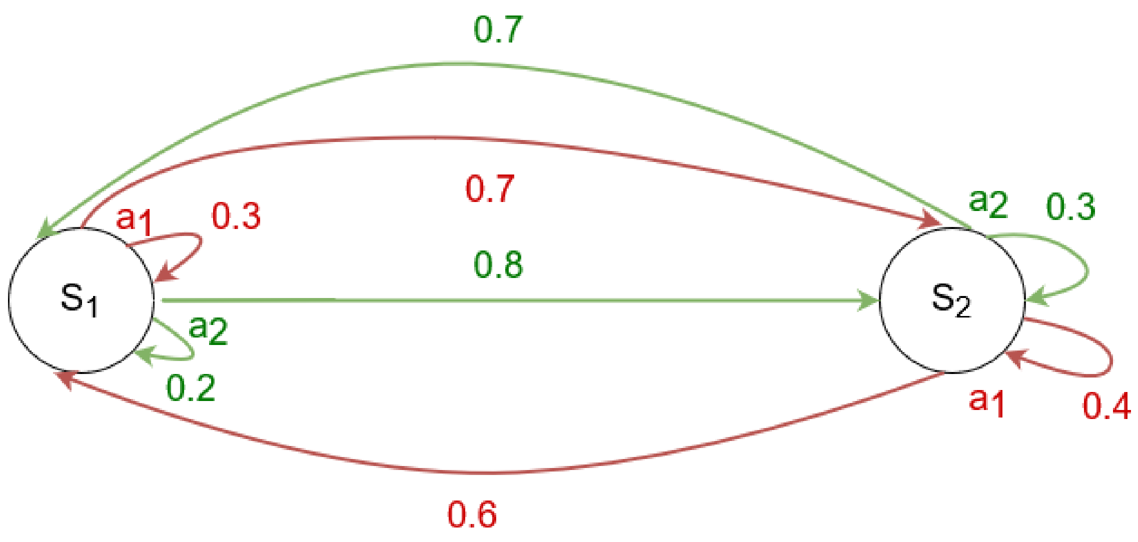


Figure 4: Markov Decision Process example

The figure shows that transitions are stochastic. For example, if the agent is in state s_1 and takes action a_1, there is a 70% chance that it will move to state s_2 and 30% chance that it will remain where it is. So, $T(s_1, a_1, s_2) = 0.7$. The ultimate goal of solving an MDP, is to find the policy π^* from all policies π (which are essentially a set of instructions that tell the agent what action to take based on the state it is in), which maximizes the reward the agent will receive.

When it comes to Reinforcement Learning, our agent is not aware of the transitions and rewards of its environment. Therefore, the agent needs to explore the environment and start understanding its surroundings. What is important to note is that we need to enforce a strategy in which the agent will be able to maintain a balance between the exploration of the environment and the exploitation of its prior knowledge. The exploration and exploitation tradeoff [52] is crucial and we need to address it in our Reinforcement Learning implementations. In our particular implementations, we use an ε-greedy strategy [53], in which the agent is encouraged to take more random moves in the first few episodes (exploration). An episode is the set of interactions between the agent and the environment from its initial state s_0 until it has reached a terminal state. To emphasize the importance of the exploration-exploitation tradeoff, let us examine Figure 5.

0	0	-10	0
2	0	0	10
0	0	0	0
X	0	2	0

Figure 5: Exploration vs Exploitation

The starting position of the agent is position (4,1), marked with an ‘X’. If it weren’t for the exploration-exploitation tradeoff and the certain enforced strategies, the agent could eventually find one of the light blue boxes, which reward it with 2 points, and stop exploring the environment from that point, fully exploiting its existing knowledge. That would be sub-optimal since the agent would always move to the light blue square it has discovered and never try to explore the environment more to try to locate the best possible reward, which is 10, on square (2,4). Notice also the existence of a “losing” terminal state on block (1,3), which is a block that terminates the episode with the agent’s loss.

In summary, in Reinforcement Learning, our agent interacts with the environment sequences that have termination conditions, called episodes. The ultimate goal of the agent is to find the strategy that maximizes its cumulative reward over its series of actions throughout an episode.

Having already talked about policies, we can expand this into mentioning that there are on-policy and off-policy methods. The major difference of the two is that the on-policy methods emphasize on the current policy in order to use it and generate new experiences, while the off-policy ones follow a strategy which reflects on the exploration paradigm and use any feasible policy to adequately generate new experiences and utilize them into forming better ones. This is important to know since we need to make a transition towards the algorithm of Q-Learning and ultimately introduce the Deep Q-Learning algorithm in the next section.

The Q-Learning method is aligned with a “looking-ahead” paradigm, in which the algorithm tries to emphasize on “what could be” and not “what is”. This is done by calculating the Q values which indicate the expected reward of an action *a* when in state *s* like so [54]:

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s', a)$$

This practically means that the Q value of taking action *a* when in state *s*, is the immediate reward *r(s,a)* the agent will receive plus the maximum Q value of all its possible destination state-action pairs it could end up to, affected by the gamma parameter (the higher the gamma, the more emphasis the agent puts on the future rewards). As one can imagine, the Q value of a state-action pair is dependent on the Q values of its possible successors, whose Q values are also dependent on their own successors and so on. Thus, this is a recursive process. While exploring the environment, the agent will be discovering new information, which need to be accumulated to the already existing data. The formula of updating the existing Q values for a state-action pair is the following [54]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

In this equation, *a* is the learning rate and is needed to control the updating pace of the Q values. In the following section, we explain the details of the Deep Q-Learning algorithm, some of the difficulties in it and how they can be properly dealt with.

3.3 The Deep Q-Learning Algorithm

In Deep Q-Learning [12], [54], we are utilizing a neural network in order to approximate the Q value function. It is imperative to understand the differences between Q-Learning and Deep Q-Learning: In Q-Learning, we are essentially using a state-action pair as input in order to calculate a Q value. In Deep Q-Learning, we are using just the state as input, and using a neural network, we approximate the Q-values associated with each possible action. In Figure 6 and Figure 7 (which is an example with 3 possible actions), we can see a descriptive illustration of the Q-Learning and Deep Q-Learning processes.

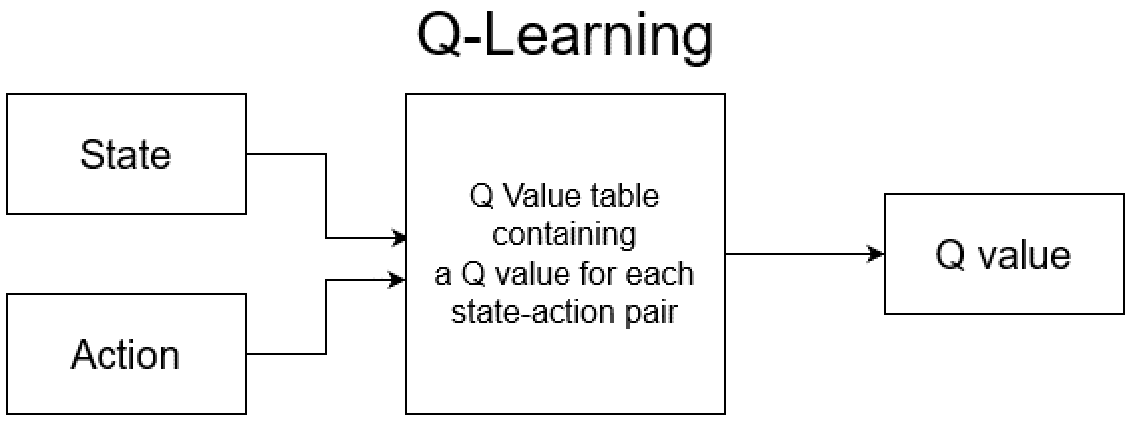

Q-Learning
State
Action
Q Value table containing a Q value for each state-action pair
Q value

Figure 6: Q-Learning

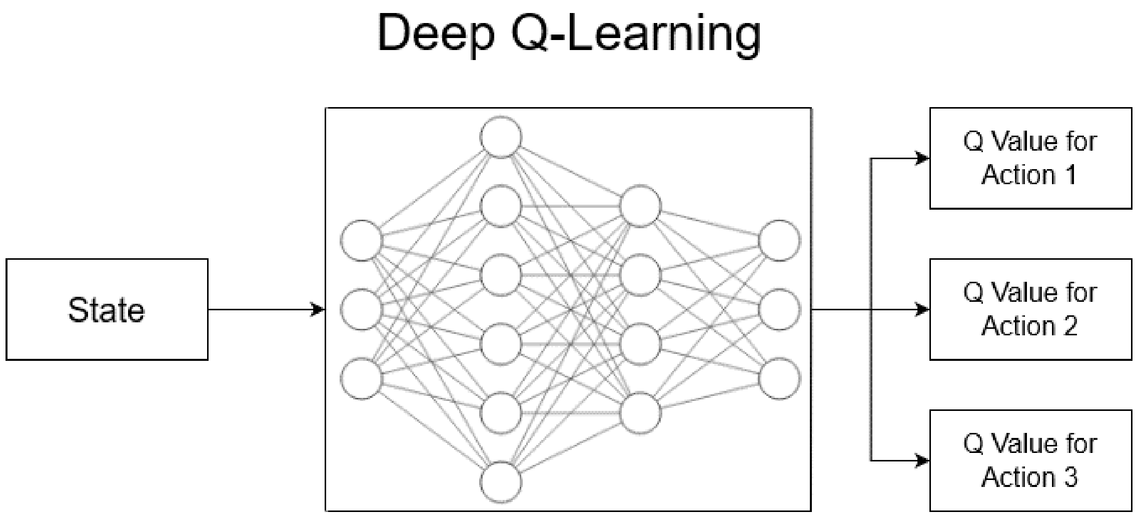


Figure 7: Deep Q-Learning

One would wonder how that neural network will be trained in order to make more accurate Q value approximations. In reality, the loss function of the network is related to the difference of the predicted Q value and the target Q value, which is then backpropagated to adjust the weights. There is something that seems to be problematic here, and that is that the target Q value will be changing on every iteration, which is not what happens in Deep Learning. The target Q value, as it can be observed in the following equestion, is also dependent on the maximum Q value of future state-action pairs. The Q values of these pairs are being constantly updated, on every iteration, resulting in the constant updating of the target Q value too [54]:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

This non-stationary target Q value will basically result in our agent chasing a non-stationary target. This can be casually parallelized with a case of a dog chasing its own tail; every step the agent would be making to move closer to its target would still be moving the target away from it. In addition, one more problem that may arise is that if we update the Q values only based on the agent's trajectory, we will be missing on past experience that could potentially be very important into the re-evaluation of certain steps. There is, however, a solution to both of these issues.

The first issue that has been described can be solved by using two different networks, one that will predict the Q values (and be trained) and one that will only be having the target Q values as its output. The "target network" will not be trained in order to annihilate the effect of the constantly moving target, while the "prediction network" will be normally trained at the end of each episode. In order to maintain realistic target outputs, the "target network", which is not having its weights updated at all, will have its weights become identical to the ones of the "prediction network" every few iterations. For instance, if we define this updating parameter to be equal to 20, then every 20 episodes, the weights of the "target network" will become identical to the weights of the "prediction network" which is trained at the end of each episode.

The second issue that has been described above can be tackled with the mechanism of Experience Replay memory [55]. This mechanism is based on storing certain experience tuples into a set memory from which we will be sampling to train the prediction network. The tuples that will be saved in the experience replay memory will be of the following form:

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

- e_t: the experience entry
- s_t: the state the agent was in
- a_t: the action the agent took
- r_t: the reward the agent received
- s_{t+1}: the state the agent resulted after taking action a_t from state s_t

Whenever the agent takes a certain action, an experience entry will be generated and pushed into the memory. When the “prediction network” needs to be trained, a random sample of a certain size (user-defined parameter) will be picked from the memory and will be used to train it and adjust its weights. On this specific sample, the “prediction network” will be producing the Q value approximations, while the “target network” will be producing the target Q values. The combination of the values that have just been produced will be utilized for the training process of the “prediction network”. This is a process that will occur at the end of each episode. The training process, though, would ideally begin after gathering enough experiences to provide a full sample. For instance, if we have a memory size of 20000 experiences and a sample size of 1024, it is more appropriate to begin the training process of the “prediction network” after gathering 1024 samples and not before that, since inadequate samples would be used [55].

The Deep Q-Learning algorithmic calculations can be well-summarized in the following steps [54]:

1. Feed the current state into the prediction network so that it can produce the Q value approximation of all possible actions based on the current state.
2. Take the action with the highest Q value approximation from the prediction network’s outputs.
3. Add the experience tuple of *(state, action, reward, next state)* into the memory.
4. If there are enough experiences to take a sample, take a random one from the memory.
5. Calculate the loss as the difference between the predicted and the target Q values on that selected random sample.
6. Update the weights of the prediction network to minimize the loss.
7. Every *C* iterations (user-defined parameter) make the target network’s weights identical to the prediction network’s ones.

Steps 1-7 will be repeated for a user-defined number of episodes.

3.4 The example of Frozen Lake

In order to get familiar with the Deep Q-Learning algorithm, we made a small implementation of the Frozen Lake [56] inspired by OpenAI’s Gym [57]. Taking an example grid from [56], we have a 4x4 problem formulation, with:

- S being the starting point (safe)
- F being a frozen block (safe)
- H being a hole (losing terminal state)
- G being the goal (winning terminal state)

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

0	0	0	0
0	-5	0	-5
0	0	0	-5
-5	0	0	5

Figure 8: Frozen Lake grid and rewards

In Figure 8, we see how we can take the Frozen Lake problem and translate the 4x4 (S,F,H,G) grid to a 4x4 grid that indicates the rewards. The starting (S) and frozen (F) blocks, which are safe, do not reward the agent with anything, the goal block (G) rewards the agent with 5 points and is the winning terminal state and all the hole blocks (H) have a negative reward and constitute a losing terminal state. The goal of the agent in this case is, starting from the top left block (0,0) to meander through its environment and find the bottom right block (4,4).

We only mention a few details about this implementation and do not analyze it furtherly, since this is not the purpose of this section. We implemented the Deep Q-Learning algorithm using just one network, the “prediction network”. The dimensionality of a problem that is depicted on a 4x4 grid is way too small. For this reason, the two networks (prediction & target) are not required. In smaller search spaces, the prediction network can operate sufficiently just by itself. This is something we also cover in the actual Deep Q-Learning implementations’ section. Additionally, we used an ε-greedy strategy to introduce the exploration-exploitation tradeoff in the problem formulation. Due to the low dimensionality of the problem, the usage of Experience Replay was also not rendered necessary, since the algorithm proved to be operating well without it. All things considered, without the usage of two networks and/or experience replay memory, the implemented algorithm performed quite well in the Frozen Lake problem. It always managed to solve the problem and converge, usually around 700 episodes in. As a slight clarification note for reference: This particular implementation should by no means be compared to the implementations we are about to present later for Neural Architecture Search. The low dimensionality of the problem makes the algorithm operate really quickly and have no noticeable performance issues. The amount of 700 episodes might sound like a lot, but in such a low dimensionality problem, it has been covered in just a few seconds, which is nowhere near close the time 700 episodes would take in a higher dimensionality problem, like the ones we will deal with later. Higher dimensionality, wider search spaces and the introduction of experience replay memory, occasionally with really large samples, will result in much slower execution per episode.

4 Metaheuristic Algorithms

4.1 Introduction

By definition, metaheuristic algorithms have been designed in order to solve problems in a faster and more efficient manner, compared to more traditional methods, occasionally with the tradeoff in optimality, precision or accuracy [58], [59]. We briefly present the evolutionary algorithms since we have implemented an evolutionary algorithm, which is quite similar to a genetic one, but with some noticeable differences that we explain. Also, we discuss three different algorithms that belong to the broader family of swarm intelligence algorithms [60].

4.2 Evolutionary Algorithms

In the introduction, we mentioned evolutionary and genetic algorithms literature [61], [62], [63], [64], [65], [66]. Evolutionary algorithms, by definition, constitute a generic population-based metaheuristic optimization paradigm, which are inspired by biological evolution mechanisms, such as reproduction and mutation [67]. Genetic algorithms are a sub-category of the evolutionary algorithms. Generally, the algorithmic implementation of an evolutionary algorithm has the following steps:

1. Generate a certain number of individuals that will form the initial population.
2. Calculate the fitness of the individuals.
3. Select a number of individuals, based on various criteria, for reproduction. These individuals will be the parents.
4. Perform various operations (e.g. crossover, mutation) to the parent(s) in order to lead to the creation of offspring.
5. Based on certain criteria, remove individuals from the population to maintain a certain number of individuals in it.

After the initial population has been randomly generated, the evolutionary process can begin by repeating steps 2-5 for a certain number of evolution cycles. The selection of the parents can happen in many different ways: It can be done through some tournament selection process, through the fitness evaluation of each individual or even through a simple random selection. It can certainly be a combination of the above, [13], in which, initially, we select a random sample from the population and out of this sample we select the best individual in order to mutate.

Following the selection of the parents, comes the part of the child creation. If there is only a single parent, as in our implementation, it can be as simple as just performing a mutative operation on that single parent. For instance, in order to look at this through the prism of neural network evolution, a mutative operation on a neural network could add a layer in it or modify the existing connections among its existing layers. In the case of multiple parents, which is something very common in genetic algorithms, we would have to find a way to properly pair the parents in order to create a child (or more than just one). There are various crossover operations that dictate the procedure of the offspring creation.

One final step would be to have a way to maintain a stable number for the size of the population. This means that since new individuals have joined the population, some others need to be discarded. This can be performed in various ways, which depend on different things. For example, one very common way to discard individuals off the population is based on their fitness evaluation; the individuals with the lowest fitness evaluation are removed. Another way would be to discard the oldest individuals in the population [13].

Finally, it is worth mentioning that the process of selecting parents and removing individuals from the population should be treated carefully. Based on the criteria we are using to select parent or discard individuals, we might never take into consideration some important features (genes of these individuals) of lower-fitness individuals since we might not take them into account during the reproduction phase and eventually discard them. For example, if we are only selecting the best-fit individuals for the process of reproduction, we might never consider some other features/characteristics of a lower-fitness individual, which, when combined with the characteristics of the best-fit individuals, lead to an even better individual we have not constructed yet.

4.3 Ant Colony Optimization

Ant Colony Optimization’s (ACO) [19] fundamental principles are based on the way biological ants work and communicate. The general idea behind the ACO algorithm is to model every problem as one of finding the shortest path in a graph. The artificial ants in this case represent some agents that will have to wander through the search space of all possible solutions and find the optimal one [67], [68], [69], using the knowledge that the rest of the ants have already discovered.

The key element in the functionality of an ant colony is pheromone. Pheromone is a chemical substance that ants leave on their trails when moving. The way this substance affects the way the ant colony works is that, in the beginning, ants start wandering randomly around the search space to find the optimal solution, just like when they are looking for their food in the real world. If an ant finds a path that has pheromone on it, it is more likely to follow this path. It is also more likely to prefer to follow a path that has more pheromone on it than some other. The pheromone-trail-following process results in the finding of the shortest paths, since the longer one path is, the more time pheromone has to evaporate. As a result, the next ants that will wander through the same area, will not consider this path as a good solution, resulting in them not following it, which in turn will mean that little to no amount of pheromone will be left on this path. However, if a short path is found, the ant that has found it will be able to return to the colony quite fast (since it is a short one). The other ants, beginning from the colony, will notice this path having a lot of pheromone and follow it, adding even more pheromone on it. This series of events results in more and more ants following the shortest path, which will eventually be elected as the optimal solution. All in all, the pheromone mechanism leads to shorter paths being preferred to longer ones by the ants (due to the pheromone evaporation).

For example, let us examine Figure 9. It is quite evident that Path 2 is the shortest path of all, with Path 1 being the second shortest and Path 3 being the longest one. Assume the following scenario: 3 ants, 1 for each path (ant 1 for path 1, ant 2 for path 2, and 3 for path 3), begin from the nest at the same time and start heading towards the "Food" node. Ant 2 will be the one that will reach the "Food" node first and start returning to the nest, while the other two are still heading towards the "Food" node. At some point, Ant 2 will return back to the nest, while the other two ants will still be on their way either to the "Food" node or back to the nest. Any ant that will begin from the nest from that point and on will notice that Path 2 has the largest amount of pheromone (this is because it is the shortest path and Ant 2 managed to leave much more pheromone on it since, not only did it reach its destination, but also returned). Thus, the other ants will follow this path, leaving even more pheromone on it, attracting more and more ants.

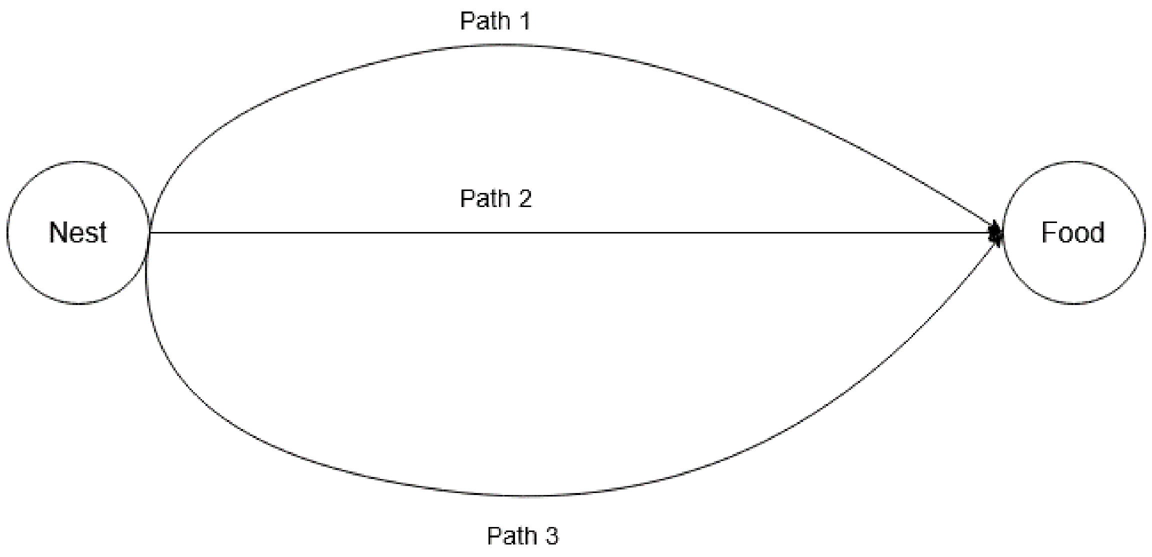


Figure 9: Ant Colony example

It goes without saying that the pheromone mechanism is not the only motivation for the ants' movement choice (e.g. there is also extra visibility and heuristics per problem), otherwise the algorithm will be way too prone to being trapped into local optima and never really getting close to global ones. This is of similar sense to the exploration and exploitation reference in the section of Reinforcement Learning.

It is worth mentioning that the pre-implemented version of ACO that we have used in our experiments is an extended version, called Extended Ant Colony Optimization (GACO) [18] of PyGMO [14]. The main difference between the original ACO and GACO, according to the PyGMO documentation [71], is that the future ant generations are produced by using a multi-kernel Gaussian distribution, which takes 3 parameters into consideration (i.e. pheromones), 2 of which are calculated differently (the weights and the standard deviation).

Finally, if we were to summarize the ACO algorithm into some steps, these would be:

- Initialize some necessary problem parameters (e.g. perform pheromone initialization).
- Generate a set of solutions.
- Compare and evaluate the different paths found by the ants.
- Update the pheromones.
- Repeat the previous 3 steps until there is a termination condition that is met.

- Return the solution that the ants have converged on (via the pheromone mechanism).

4.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is another swarm intelligence algorithm and we talk about it in this section. There is an extensive amount of sources, some of which we use to describe the algorithm [71], [72], [73], [74], and others that even evaluate the existing publications and applications of PSO [76].

The PSO algorithm is based on having a population of candidate solutions, the particles, which will be moving and meandering around the search space with the ultimate goal to converge on the optimal solution. Each particle has a specific position and some velocity that defines the speed it is moving towards a certain direction. The performance of the particles is constantly evaluated by a specified fitness function. One PSO feature is the maintenance of some restricted memory for each particle, which contains the best position in the search space that this specific particle has been in. This is useful information for reasons that we explain in the following paragraphs.

When it comes to moving around the search space, the algorithm needs to decide the direction each particle is going to move towards and its velocity. The calculation of a particle's velocity is dependent on its current velocity and a balance between the prior knowledge of this specific particle and the knowledge of the best particle of the swarm. A particle could be:

- Moving towards the position of the best particle in the swarm
- Moving towards its own best position (using the memory)
- Moving towards a different selected path

The existence of this particular memory, which we had talked about previously, does also not allow the particle to deviate way too much from its best existing position.

Before we talk about a very important constituent of the algorithm, which is the neighbouring types, we need to talk about a few problems that have occurred and the solutions that have been found to solve them. Firstly, the velocity mechanism had certain drawbacks in regard to the lack of certain bounds. This could potentially result in exploding velocity numbers, especially for particles that were really far from the best particle in the swarm. To tackle this problem, the concept of a maximum velocity constraint was introduced (the “max_vel” parameter of PyGMO). This is yet another mechanism to ensure that there is going to be a good exploration-exploitation balance, just like in many of the methods we had analyzed before. Secondly, it has been noticed that were some occurring difficulties for the algorithm to converge, or in other words, for all the particles to gather around the best solution that has been found. For this purpose, the inertia weight parameter has been introduced (the “omega” parameter of PyGMO, which represents the inertia weight or constraint coefficient, depending on the PyGMO “variant” variable). The purpose of this parameter is to control the impact of the current velocity of a particle to its velocity value on the next iteration. This parameter needs to be treated delicately since it can affect the speed of the algorithm convergence and even determine if it is going to converge on a local or global optimum.

The neighbouring types (“neighb_type” parameter [77] of PyGMO) define the visibility each particle has and can potentially affect its velocity and trajectory, apart from its own findings. Two very characteristic neighbourhood types are the “lbest” and “gbest”. In the “gbest” neighbourhood type, every particle is connected to every other particle in the swarm. With this topology, the particles’ velocities are affected by the velocity of the best particle in the swarm. In the “lbest” neighbourhood type, each particle is connected to its closest particles with respect to the neighbourhood size parameter (“neighb_param” parameter of PyGMO). For example, if the neighbourhood size is equal to 5, then each particle will be connected to its 4 closest particles. Thus, in this topology, the velocities of the particles are affected only by the information received by their neighbours.

As we can see in Figure 10, in the “gbest” neighbourhood type, every particle is connected to every other particle of the swarm, while in Figure 11, which is the “lbest” neighbourhood type with a neighbourhood size of 3, each particle is connected only to its 2 closest particles.

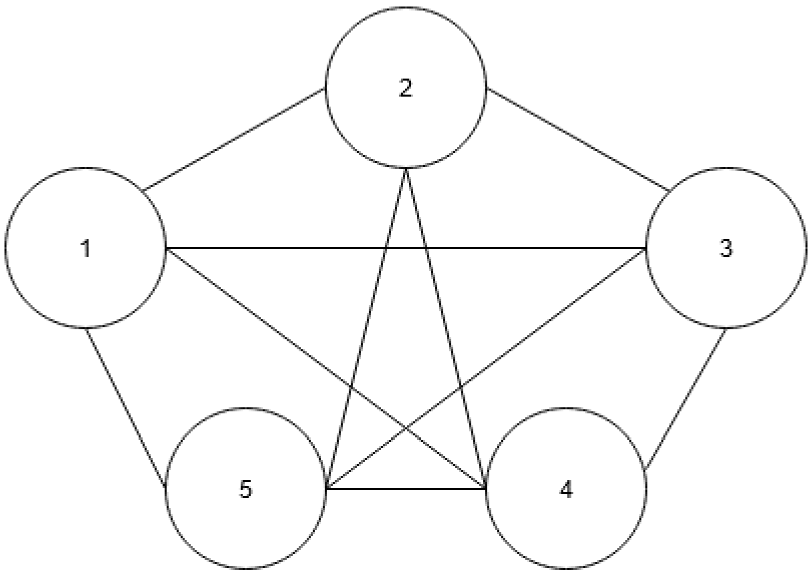


Figure 10: gbest neighbourhood type

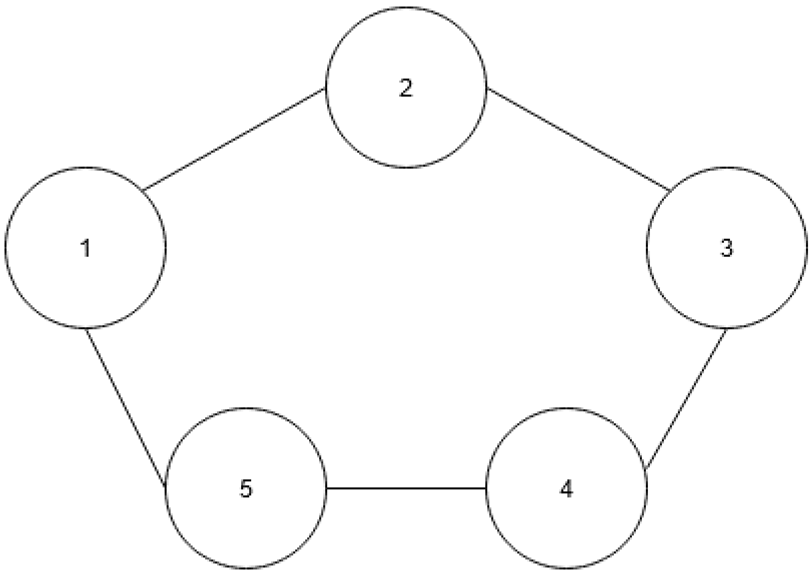


Figure 11: lbest neighbourhood type with neighbourhood size of 3 particles

It is important to ensure that the initial particle positions are well distributed all around the search space so that we can disregard the scenario of certain areas being completely unexplored. Having said all of the above, we can summarize the algorithmic process of Particle Swarm Optimization like so:

1. Initialize the position of the particle.
2. Initialize the particle's best-known position (memory) to its initial one.

3. Update the best position in the swarm if this new particle’s position is better than it.
4. Initialize the particle’s velocity.
5. Repeat steps 1-4 for every particle in the swarm.
6. Then, for every iteration, update the particle’s velocity.
7. Update the particle’s position.
8. Update the particle’s best-known position if its new position is better than it.
9. Update the best position in the swarm if this new position of the particle is better than it.
10. Repeat steps 6-9, for every particle, until a termination condition is met.
11. Return the solution that the particles have converged on, based on their position in the search space.

4.5 Artificial Bee Colony

The Artificial Bee Colony (ABC) is the last swarm intelligence algorithm we talk about. After the initial introduction of the motivation and the idea behind the algorithm [78], [79], some extra clarifications have been made by the author [80], with a lot of additional material on analyzing and evaluating the introduced algorithm [22], [81], [82].

The algorithm of Artificial Bee Colony aims to simulate the way bees work together in a biological beehive. There are three different types of artificial bees in the artificial hive: the employed, the onlooker and the scout bees. One very basic convention is that there is one employed bee per food source. Food sources represent the possible solutions and the nectar of each of them represents the fitness of the solution. The search process begins with the employed bees, which have their source position in their memory, performing a modification in that source to produce a new food source position. After this point, they will perform a greedy selection approach on the old and new food sources: They will evaluate the amount of nectar in both food sources (i.e. the fitness of the solutions) and keep the one with the highest nectar quantity. This means that if the new constructed solution is better than the existing one, it will replace it, otherwise the old source will be maintained. Having performed this step, the employed bees move to the hive and dance, trying to attract the onlooker bees to their sources. The onlooker bees, following a probabilistic selection process, which will generally favour the sources with the highest nectar amounts, will select one of the food source positions and perform a similar modification to it, which in turn will be followed by the greedy selection between the new and the old sources. These two types of bees are related to the largest part of the algorithm. We have not talked about the scout bees yet. After a food source has not been improved after a certain amount of iterations (the "limit" parameter [83] of PyGMO), or in other words, the modifications performed on that food source have not resulted in a solution that evaluates to higher fitness, then the employed bee responsible for that food source abandons it and plays the role of a scout bee, which needs to discover a new solution in a random fashion.

We could summarize the functionality of the Artificial Bee Colony algorithm and the bee communication protocol in the following steps:

1. Initialize the randomly distributed set of food source positions, one position for each employed bee.
2. Each of the employed bees goes to the food source in her memory, selects a close source, performs a modification based on it and evaluates both the existing and the new solution. If the new solution is better than the old one, it replaces it in the memory of the bee.
3. The employed bees move to the hive dance area and dance to attract onlooker bees.
4. Onlooker bees, based on the dance of the employed bees, select a source position, select one that is close to it and perform modifications on the source position based on the selected neighbouring position. Again, if the newly constructed source position is evaluated to higher fitness than the old one, it replaces it in the memory of the onlooker bee.
5. If a food source position has not been improved (through the modifications) after a limited amount of iterations, then it gets abandoned and its employed bee plays the role of a scout one, which will randomly discover a new food source position.
6. Register the best food source position found so far in the entire hive.
7. Repeat steps 2-6 until certain requirements (i.e. termination conditions) are met.
8. Return the solution with the best fitness evaluation (highest nectar amount).

5 Neural Architecture Search

5.1 NAS Motivation and Introduction

As we have already mentioned in section 1.1 , the automation of the construction of neural network architectures is not deemed as a surprising event, but more like an inevitable outcome of natural evolution [11]. History repeats itself and it is yet another industrial revolution [10] in which we use the latest technological findings to alleviate pressure by automating tasks that, up until that point, were being performed manually. It is the same case in Neural Architecture Search [84], where the construction of the best possible architectures for neural networks was a task that was being performed by very specialized individuals.

The field of Neural Architecture Search is attracting more and more interest over the years. The exponential increase of the Neural Architecture Search related papers is very indicative of that. With only 10 papers being related to NAS from 1988 until 2014, based on [85] and just one paper in 2015, we can see a rising trend from 2016 and on. 2016 and 2017, with 7 and 17 papers respectively, were followed by 2018, which was the year that we started seeing a very evident increase in the popularity of the field, with 59 papers (over 8 times higher than two years before it). In the following 2 years, 2019 and 2020, the increase of NAS papers was astonishing, with 226 and 505 papers respectively. In Figure 12 we can see a visual representation of the papers from 2015 until 2020.

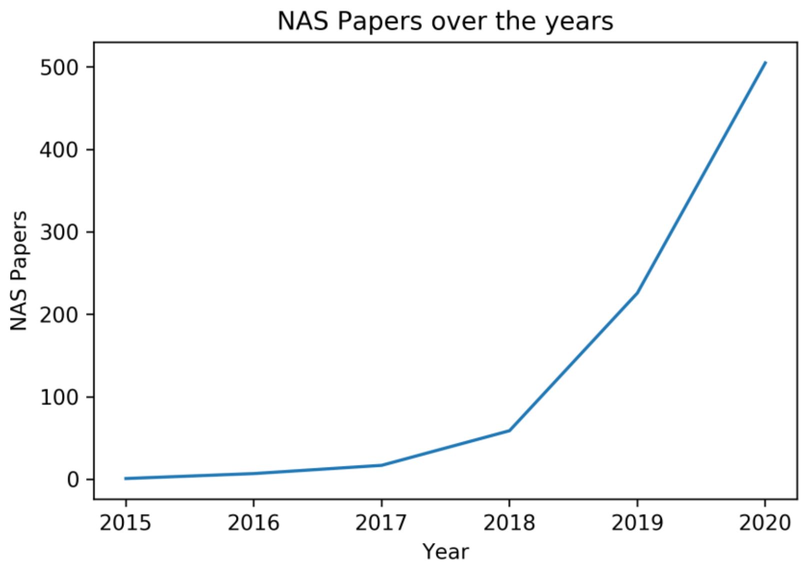


Figure 12: NAS Papers over the years

In order to properly understand the concept of Neural Architecture Search, we are going to explain some basic terms, like “search space” and “optimization method” [11], [85]. The search space basically defines the spectrum of neural architectures our algorithm is supposed to meander through in order to result to the optimal one. In terms of search space paradigms, there are two that properly split the search procedures throughout the networks, the global (or macro) search space and the cell (or micro) one [11]. The global search space refers to the case in which we allow the optimization method to create arbitrary networks. Therefore, the various layer types will be decided, the connections among them and the related hyper-parameters of the network. A characteristic and recent example is presented in [86], in which through the formulation of neural networks and their layers via blueprints and modules respectively, a global search space is utilized to tackle Neural Architecture Search. The cell search space has the identical characteristic of being structured by repeated blocks of layers that fit in a pre-defined neural network skeleton. For example, let us suppose that we have defined a skeleton for a network that instructs that 3 cells of one cell type (“type A”) will be followed by 1 cell of another cell type (“type B”), which will then be followed by 3 cells of the first type. What we are looking for here is not the general structure of the network; this is something that has already been defined. In reality, we are looking for the architecture of two different cells (“type A” and “type B”) by knowing the sequence in which they will be used to assemble the neural network. An approach very similar to the presented example is followed in NASNet [87], where there are two different types of cells, a “normal” and a “reduction” one. A concrete topology is enforced, instructing that *N* number of “normal” cells will be followed by one “reduction” cell, with every cell receiving its inputs from 2 of the previous cells.

We also mentioned optimization methods, which practically are the methods dictating how the algorithm will wander around the architecture search space, be it global or cell. Some examples of optimization methods can be Reinforcement Learning and Evolutionary Algorithms; two methods that we do use in this thesis. Furthermore, optimization methods are assisted by evaluation functions. These functions are needed in order to evaluate the candidate architectures and assist the optimization method into properly exploring the search space.

5.2 Related Work

Over the years, there have been various approaches to tackle the problem of Neural Architecture Search. Reinforcement Learning is one of the approaches that have been presented for NAS. Neural Architecture Search has been solidly posed as a policy gradient problem; a Reinforcement Learning problem [88]. At this initial state, the NAS formulated problem was at a state where gigantic amounts of computing power and training time was needed in order to have efficient results, also utilizing the feature of skip connections that has proved to be very efficient in presented networks like ResNet [89] and DenseNet [90]. Still, approaching the problem as a Reinforcement Learning one, there have been drastic performance improvements to this approach [91], which introduces the idea of sharing parameters among the networks, reducing the training time from thousands of days (using thousands of GPUs) to just a single day (using a single GPU). Reinforcement Learning is also met in NASNet [87], which is an approach we have followed for one of our implementations.

There have also been different approaches to the matter, trying to escape the entire Reinforcement Learning formulation and boundary, such as this Progressive NAS [92] approach. In this paper, we see an interesting take on the subject in which the networks follow an evolutional paradigm in order to evolve and progress (by predicting the performance of the offspring and selecting future parents), something that can come to a very close resemblance to genetic algorithms. There have also been interesting approaches, like DARTS [93], in which the problem of NAS is formulated as an optimization one and the optimal solutions are found by the continuous relaxation of this discrete optimization problem. This approach outperforms the Efficient and Progressive approaches we had mentioned before and also proves that it can be beneficial not to forcefully cast a problem as a Reinforcement Learning one.

Having spoken about PNAS [92], which definitely reminds us of the intuition behind genetic algorithms, we also need to refer to important significant genetic algorithm approaches. An initial idea, the one of NEAT, has been initially presented back in 2002 [94] and has become a more compound approach in a more modern version many years later [86] with the usage of genetic algorithms. Another approach, which we have also used in our implementations, is an aging evolutionary algorithm that has been introduced in this paper [13].

6 Implementation Approach and Limitations

For the scope of the experiments of this thesis, we have decided to utilize some of the latest findings in the field in order to expedite the procedure of evaluating our networks. So, in order to do that and avoid training our networks, we use some of the published benchmark results. Concretely, we use NASBench-101 [15]. NASBench-101 contains neural architectures that have been trained on the CIFAR-10 image classification dataset. In particular, the number of the architectures present in the benchmark is 423624. To compound this, all of these architectures have been trained for 4, 12, 36 and 108 epochs, three times each. As a result, the total amount of trained models is a little over five million (423K architectures * 4 epoch options * 3). The search space of our implementations is a NASNet-like search space [87]. Essentially, using this search space, we are defining a scope by the possible actions that can be taken to construct our network architectures. Specifically, we are enforcing a set of possible layer additions (in the case of NASNet, 12, but this will be tuned to be NASBench-101 compliant) and the connection capability to previous layers.

Using the NASBench-101 benchmark comes with a few constraints, which are related to the architectural limitations that have been enforced to the networks that have been trained. Firstly, the architectures included in this benchmark are constructed only by three possible layer types: 1x1 convolution, 3x3 convolution and 3x3 max pooling. Secondly, the number of total layers of the formed networks has a maximum of seven, including the input and output layers. To put it simply, the number of hidden layers a network is allowed to have, is five. Finally, the last constraint of NASBench-101 dictates that the number of the layer connections within the network cannot be greater than nine. In summary, the networks we will construct will only contain 1x1 convolution, 3x3 convolution and 3x3 max pooling layers, up to five hidden layers (or seven in total if the input and output layers are taken into consideration) and up to nine connections between them in total.

In order to furtherly enhance our ability to expedite the procedure of evaluating our constructed neural networks, we add to our arsenal an open source deep learning architectural research framework called “Neural Operations Research & Development”, or in short, NORD [16]. NORD aims to make the implementation of neural networks easier for the developers and accelerate the process of finding the best neural architectures, and this is exactly how it is going to be of assistance in this thesis. In addition, due to the fact that the amount of computational resources required is quite large when it comes to training and evaluating neural architectures, NORD utilizes distributed computing techniques to accelerate this process even more.

This neural architectural framework provides us with multiple useful and luxurious utilities. One of the modules that stands out and we use in pretty much any implementation is the NeuralDescriptor class. NeuralDescriptor is a class that renders describing a neural network's topology and structure something relatively easy. Moreover, various evaluator classes provide us with the convenience of evaluating a neural network's performance very quickly. For instance, using the class BenchmarkEvaluator, we are capable of evaluating a neural architecture by utilizing a specific benchmark, such as NASBench-101. With the process that has just been described, we are provided with the luxury of avoiding the arduous, long and time-consuming procedure of training our networks from scratch. Consequently, we do not need a large amount of computational resources to achieve our goals and also save a lot of time by not waiting for the networks to be trained, allowing us to focus on the implementation of the optimization methods more.

As a final note, when evaluating our networks in the implementations of this thesis, we are querying the validation accuracy from NASBench-101 and not the test one. As a point of reference, in order to have some ground to evaluate how good the results of our implementations are (or what percentile they belong do), we mention here that the best validation accuracy met in NASBench-101 is equal to approximately 95.15%. On the following table (Table 3), we can see the analysis of the validation accuracies included in NASBench-101 (a total of 1293208 validation accuracies):

Mean	Std	Min	25%	50%	75%	Max
0.9082867	0.05815191	0.09445112	0.9008414	0.9162660	0.9274840	0.9515224

Table 3: NASBench-101 validation accuracies' statistics

7 Deep Q-Learning

7.1 Introduction

The first method that is used for Neural Architecture Search is a Reinforcement Learning one and is called Deep Q-Learning [12]. Our agents in this specific method utilize neural networks, the Deep Q-Networks. Deep Q-Learning can be considered quite a simple optimization method. We go through three different approaches that differ from each other in the architecture of the controller/agent, the possible actions and search space, or both. In all cases, our agent is trained in order to learn the best neural architecture possible.

7.2 Deep Q-Learning using a dense-layer controller

7.2.1 Introduction

In our first Deep Q-Learning implementation, the neural network agent is a simple sequential network, composed of multiple dense hidden layers [95] and one output layer. Additionally, the actions of the agent are limited to the all possible layer types that can be added with respect to the limitations described in section 6. Therefore, our agent's actions are limited to adding a 1x1 convolution, a 3x3 convolution or a 3x3 max pooling layer.

7.2.2 Methodology

To begin with, we need to define the form of our states (*S*), actions (*A*) and rewards (*R*) for each state-action pair. Considering that we are constructing neural architectures, we need a way to depict the network's topology. Recalling that in this very first implementation the only action that our agent can perform is add a layer and not connect two existing ones, we are merely referring to constructed networks that are sequential. Therefore, a data structure that includes the layer type of each of the hidden layers (if it exists) is sufficient. Specifically, our network state is a 5x4 array, each row of which is initialized as [1, 0, 0, 0]. This is essentially a one-hot encoded vector of each layer, showing its type. Index zero indicates the 'no-layer' type, meaning that the agent has either not chosen an action for this layer yet (just like in the initial 5x4 array) or that the agent has chosen to add no layer and terminate this particular episode of our execution.

This brings us to the next topic, which is the agent's possible actions, which in this case are four, considering that the agent can add only three different types of layers. The fourth option our agent has is to add none of the three available layers and terminate the neural architecture building at that specific point. Finally, our algorithm has two possible terminal states. The first occurs when our constructed network has reached the total of five hidden layers (or seven in total, input and output layers included). The second terminal state is the one in which our agent has chosen not to add any layer, thereby terminating the network building at that point. Having reached a terminal state, we have a neural architecture in hand that needs to be evaluated. The validation accuracy, which is obtained by evaluating the constructed neural architecture, is the reward for this specific state-action pair. Any other state-action pair before reaching a terminal state is matched to a reward equal to zero.

In order to store the agent's experiences, we use the mechanism of experience replay memory. The instances that are stored for every action the agent takes are of the form: [current state, action taken, reward received, next state]. We still need to define two very basic parameters: the size of the memory and the size of the samples that will be taken from the memory each time we would like to train our agent. After some experimentation with these two parameters, keeping in mind that this is a problem with quite a small search space of 363 states (3 possible layer types/states and up to 5 layers), we ended up using a memory size of 2500 and a sample size of 128 experiences. Any values used that were greater than these, either for the memory capacity or the sample size, were rendered unnecessary and even ended up slowing down the process of training.

An ε-greedy strategy is what we use to maintain a balance between exploration and exploitation. The initial value of epsilon is 0.8, meaning that in the very first run, our agent has 80% chance of taking a random action rather than following the action of its own estimated Q-value. As for the epsilon decaying policy, instead of using a ratio to decay the epsilon parameter (e.g. reduce its value by 1% of its current value every time), we merely subtract a flat amount each time. Various values were tested for the epsilon decay parameter: 0.0005, 0.001, 0.002, 0.004, 0.006, 0.008, 0.01, 0.012, 0.014. The one that emerged as prevalent was 0.008, giving enough room for exploration, but allowing exploitation early enough, considering that our agent would not take much time to learn the best possible architecture in a scenario with such a small search space.

Using two separated networks in the scope of Deep Q-Networks, the prediction and the target network, entails the periodical update of the weights of the target network based on the weights of the prediction network. It is important to reiterate that only the prediction network is be trained throughout the episodes, while the target network has its weights modified to be identical to the prediction network's ones every few iterations, which is a parameter defined by us. Through experiments, it has been found that thirteen (13) iterations is a good number of iterations to update our target network's weights. Any values that were much smaller than that (e.g. between 1 and 5) or much greater (e.g. greater than 25) showed signs of instability during the training and the results were questionable.

The gamma or discount rate used was equal to 0.999 (γ = 0.999). Our Deep Q-Networks were trained for 1000 episodes. During these episodes, every 13 iterations, the target network's weights were being updated based on the prediction network's weights. Finally, it is worth mentioning that we have decided to subtract the value of 90 from the validation accuracy of the evaluated architectures, which is a number between 0% and 100%. This modification is being performed to avoid having high deviation in the returned rewards. Therefore, we define a baseline reward equal to 90, which is the mean of all accuracies and is subtracted from the returned rewards in order to emphasize on the best performance architectures and not any mediocre ones (e.g. around 83%) that would still stand out compared to zero-reward architectures. Modifying the rewards that are being stored in our experience entries in the way described above, we aid our Deep Q-Networks to make better decisions and evaluate the situation better, due to the usage of the hyperbolic tangent activation function in the output layer, which is something that is analyzed in the next section.

One final thing that is worth mentioning is that, from an algorithmic perspective, in order to be consistent with the limitations enforced by the usage of NASBench-101 [15], it has been decided not to allow our agent to build neural architectures whose constituents were more than seven layers (input, output and five hidden). The aforementioned constraint was performed by checking the number of hidden layers in every iteration. In case the agent's constructed neural architecture was at a state of five hidden layers, the agent would be forced to choose action 0, which is the action that instructs no layer addition, hence leading to a terminal state and eventually to the evaluation of our agent’s constructed architecture.

7.2.3 Implementation

The implementation of this method was carried out using the Python programming language. Specifically, a deep learning library called Keras [96] was used to implement the networks of our agents, while the NORD [16] framework described earlier was used in order to implement and evaluate the neural architectures that were constructed by our controllers/agents. Keras provides us with a luxurious toolkit of classes of neural network layer types, as well as optimizers and many more important components of the networks, which ultimately make the implementation of the agent’s neural networks a relatively easy-to-perform task.

As a reminder, the input of our Deep Q-Network agents is a 5x4 array, 1 row for each hidden layer and 1 column for each possible layer type (no layer, 1x1 convolution, 3x3 convolution, 3x3 max pooling). The architecture of our agents starts with a Flatten layer which flattens the input 5x4 array. Following the Flatten layer, there are Dense layers [95], of 512, 256 and 128 units. The Dense layers are using the Rectified Linear Unit [26], [27] (or ReLU in short) as their activation function. Finally, the agents’ architecture is completed by a final Dense layer of 4 units (one for each of the four possible actions of our agent), our output layer, which uses the hyperbolic tangent [28], [97] (or tanh in short) as its activation function.

An alternative to Dense layers which was initially tested was the one of 2D separable convolution layers [98]. The tests were performed by having both the Separable Conv2D layers work on their own and by having them before a Flatten [99] layer and two Dense layers. Neither of the above showed any promising results. Making the agent’s architecture much simpler by using only a Flatten and three Dense layers proved to be liberating since they contributed to a vast improvement in our agent’s performance. The performance of one or two intermediate dense layers instead of three was not extremely underperforming, but it was quite evident that three dense layers were a much better choice. Additionally, there were various options when it came down to the activation function of the output Dense layer. The softmax [2] activation function was the one that was initially tested and despite the meticulous modification of the outputs of the function and/or the network efficiency that was passed to the network when fitting, the results were sub-optimal. The linear [25] activation function, despite showing some slight improvement compared to the softmax one, did not have great results either. However, trying out the hyperbolic tangent activation function, combined with the reward reduction to enforce a baseline of 90, as described in the previous section, leads to a miraculous improvement in the network's performance, dissolving any instabilities that were noticeable in the cases of the other activation functions. We can see the Keras summary of the dense-layer controller in Figure 13.

Finally, the batch size when training the network was set to 64. Essentially, with the sample size being set to 128, the entire process would be finished in 2 passes. Given the simplicity of the problem and the quite small search space, it was not rendered necessary to train the network for more than 1 epoch on each episode. In this implementation, two different networks were used, one prediction network and one target network. Our prediction network would be first trained once our algorithm has gathered enough experience to provide a sample of the defined sample size (in this case, 128). It is also worth reiterating that according to the parameters we have set, every 13 iterations, the target network's weights would become identical to the prediction network's ones.

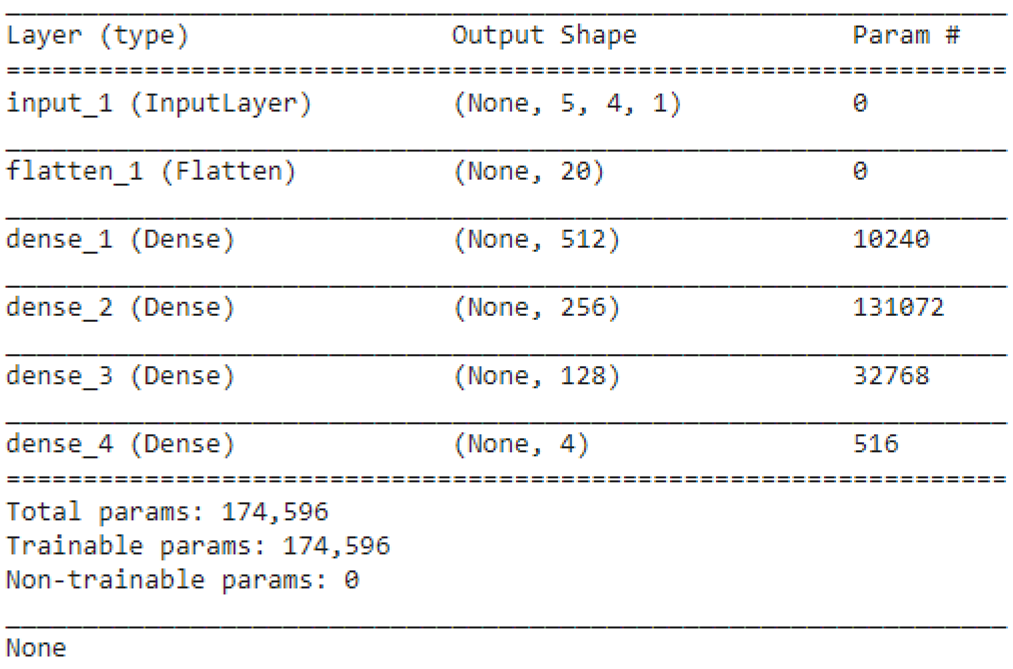

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 5, 4, 1)           0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 20)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 512)               10240     
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131072    
_________________________________________________________________
dense_3 (Dense)              (None, 128)               32768     
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 516       
=================================================================
Total params: 174,596
Trainable params: 174,596
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 13: The Keras Summary of the dense-layer controller's neural network

7.2.4 Experiments and Results

In this specific implementation, because of the small size of the search space, our Deep Q-Networks manage to perform very well. On account of the current circumstances, considering that the formulated problem is relatively easy to tackle and our algorithm finds the best possible neural architecture (based on the enforced limitations), we have chosen to compare the various parameter setups based on the percentage of the resulting architectures that are over 93%. Through our experiments, we found that the best architecture that can be constructed with the current implementation's restrictions that lead to sequential networks, has an accuracy of about 93.9%, hence the choice of the 93% baseline. Below there is a table (Table 4) with the performance of a few of the setups that we experimented with and the average percentage of the constructed architectures that evaluate to over 93% accuracy, for 1000 episodes:

Epsilon Decay	Target Update	Sample Size	Activation function	Average % over 93%
0.004	10	128	linear	26.70%
0.01	20	192	tanh	41.50%
0.008	15	192	tanh	91.10%
0.008	40	192	tanh	49.90%
0.008	10	128	tanh	93.70%
0.008	13	128	tanh	94.60%

Table 4: Experimental Results with a dense-layer controller

Finally, as we have mentioned before, the networks in the scenario of smaller search spaces tend to converge quite easily. In fact, it ended up being a matter of the epsilon and epsilon decay parametrization, giving our agent just enough time to explore its environment and then exploit its gained knowledge as soon as possible. In Figure 14, we can observe the convergence of a dense-layer controller:

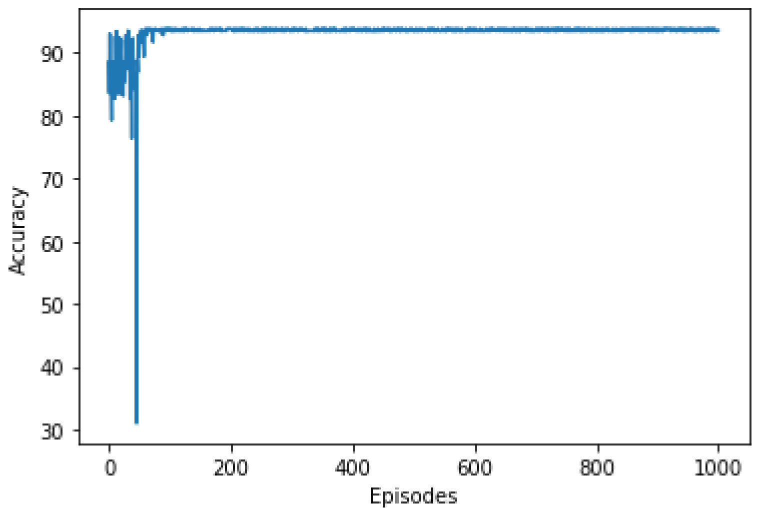


Figure 14: Accuracies of the constructed neural architectures (dense-layer controller)

As a side note, in this particular implementation, no significant performance differences were noticed between the version with only the prediction network and the version with both the target and the prediction network. The analysis and the results that have been included here are the ones corresponding to the two-network version. It might not be visible in the graph (we deliberately chose the accuracy to be presented on the 0 to 100 scale and not on the -90 to 10 scale), but it is crucial to remember that we are using a baseline reduction of the agent's reward of 90, which is what unlocked the potential of the hyperbolic tangent activation function in our approach. Thanks to the contribution of NORD [16] and NASBench-101, we managed to construct and evaluate, in just a few minutes, architectures that would normally require total computational time that exceeds 23 days.

7.2.5 Takeaways

Despite having mentioned in the introductory part of this chapter that the Deep Q-Networks algorithm is a relatively simple one, we do see that it has managed to find some well-performing neural architectures and has converged. However, it is of vital importance to remember that currently, providing our agent only with the option to add layers and not connect any already existing ones, we have limited our constructed networks to the spectrum of sequential networks. As a result, we end up constructing networks that will never achieve the optimal accuracy that is met in benchmark NASBench-101 because of the aforementioned limitation. Also, the fact that we are limiting the search space of the algorithm so much, does not reveal any weaknesses of the Deep Q-Networks that are obviously capable of performing very well in such a small search space of discrete actions. The transition from a small-scale search space of discrete actions to a larger scale, continuous one, could have a significant impact on the performance of our Deep Q-Networks. This is something that is discovered in the following sections.

Finally, an important takeaway from this formulated problem and its implementation, is that for the way we have decided to represent our network's state and the possible actions, the hyperbolic tangent function combined with a reward reduction modification to our returned accuracy, were the turning point to go from an average performing model to a very reliable and accurate one. This is an important point that will help in the future Deep Q-Network implementations, since the way the states and actions are represented will not change in anything else but its size.

7.3 Deep Q-Learning using an RNN controller

7.3.1 Introduction

In the second implementation that includes Deep Q-Networks we do not have any drastic changes compared to the first one. This one mostly serves the purpose of a transition implementation in order to introduce a Recurrent Neural Network (RNN) [85], [100] controller. The majority of the parameters that have been set in the initial formulated problem remain the same. Having said that, we are mostly going to experiment with the layer type included in the controller (LSTM [36] or GRU [101]) and the number of its units, while the actions our agent is able to perform remain the same (four possible actions, instructing no layer addition or the addition of one of the three following layer types: 1x1 convolution, 3x3 convolution, 3x3 max pooling).

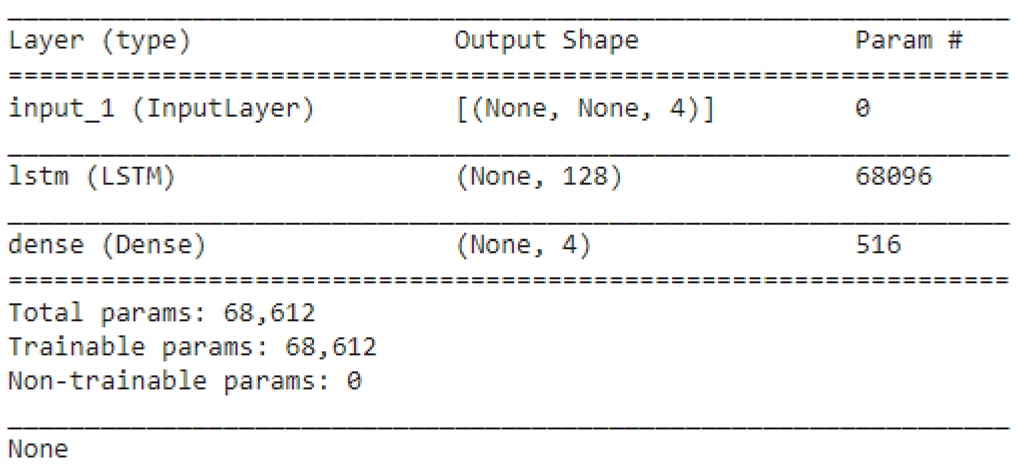
7.3.2 Methodology

As mentioned in the introductory part of this section, there are very few changes in this modified version of the first implementation. In fact, the states, actions and rewards (and the way we represent them) are the very same as the previous ones. In addition, very few to no changes are met when it comes to the parameters despite experimenting with them yet again. Concretely, an epsilon equal to 0.008 was prevalent (tried epsilon values of 0.002, 0.004 and 0.008), the memory size remained 2500 and despite trying values 64, 128, 192 and 256 for the sample size parameter, 128 was chosen again. Additionally, extra parameter experimenting was conducted for the target update parameter and after trying out the values of 5, 10, 13 and 17, the winning one was 13 again. Also, trying larger or smaller numbers for the batch size parameter while training the agent, like 32 or 128, did not show any improvement, so 64 was the value that was kept. One of the differentiators to the previous implementation is that for this one we have modified the number of training epochs. Compared to the initial implementation of a dense-layer [95] agent, the only difference in this one is that we are making the transition towards an RNN controller, using either an LSTM or a GRU layer. However, the agent with an RNN controller shows repetitive signs of instability, partially failing to converge at various points, always depending on the randomness induced in every experiment due to the ε-greedy strategy. Training the agent for a larger number of epochs resulted in a more stable and reliable performance. The agent's performance was tested while being trained for 5, 10, 15 and 20 epochs. Training the agent for 5 epochs barely had any noticeable differences compared to the instability met at the 1-epoch training, while training the agent for 15 or 20 epochs showed different signs of instability too, thereby resulting in us choosing to train our agent's network for 10 epochs. To finalize, the reward reduction to enforce a baseline of 90 has been maintained and the necessary restriction not allowing our agent to produce any neural architectures containing over 5 hidden layers (or 7 in total), is still utilized to avoid any invalid architectures.

7.3.3 Implementation

This implementation was also done using the Python programming language. The combination of the deep learning framework previously used, Keras [96], and the distributed deep learning neural architecture framework, NORD [16], is still useful for this version of the implementation.

The inputs for our agent's neural network are of the same dimension, a 5x4 array. In this particular case, our agent has a much simpler architecture than the previous dense-layer [95] one. Specifically, our agent is composed of two layers, an LSTM or GRU layer, and a dense one which is the output of the network. The choice between LSTMs and GRUs has been quite tough since the differences between the two are not that significant. However, after multiple runs (1000-episode executions), it has been found that some inconsistencies in the performance of the network and noticeable instabilities were mostly happening in the presence of GRU layers. As a result, the LSTM layer type was the one that was chosen. Four different numbers of units were tested on the LSTM and GRU layers, 64, 128, 192 and 256. The choice of 128 units for our LSTM layer was easy to make since any other option of the aforementioned ones caused some jittering in the performance of our agent. There has been no activation function that was specified. So, the default activation function of the LSTM class in Keras, which is the hyperbolic tangent (or tanh), was used. As for our output layer, the single dense layer of this network, the number of units that are being used is 4 (once again, one for each possible layer type plus the no layer addition option) and the activation function is the hyperbolic tangent here, too. It is reminded that the hyperbolic tangent function, combined with the reward reduction strategy applied to the rewards/accuracies of the constructed networks, is the one that leads to much greater results than any of the previously tested activation functions. In Figure 15, we can see the Keras Summary of the LSTM controller.


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None, 4)]         0

_________________________________________________________________
lstm (LSTM)                  (None, 128)               68096

_________________________________________________________________
dense (Dense)                (None, 4)                 516
=================================================================
Total params: 68,612
Trainable params: 68,612
Non-trainable params: 0

_________________________________________________________________
None
```

Figure 15: The Keras Summary of the LSTM controller's neural network

7.3.4 Experiments and Results

We do know that this second implementation does not differ much from the previous one, and neither do its results. We are yet another time producing mere sequential networks which have a certain ceiling. Our agents do manage to find the best possible sequential neural architecture in this implementation, too. As in the previous one, it is an architecture that corresponds to an accuracy of 93.9%. We present a sample of the parameter combinations that we experimented with, some that are indicative of the values that affected, either negatively or positively, the overall average percentage over 93%. The results presented below (Table 5) are for a subset of the parameters. For the rest of the parameters that are not included, the optimal value that has been found is inferred (13 for the target update, 10 training epochs, 64 batch size), merely because the different values of those did not lead to any noticeable differences in the performance.

Epsilon Decay	GRU/LSTM	GRU/LSTM Units	Sample Size	Average % over 93%
0.008	GRU	128	128	89.80%
0.008	GRU	256	128	88.10%
0.004	GRU	128	192	72.70%
0.008	LSTM	256	256	65.30%
0.008	LSTM	64	128	91.50%
0.008	LSTM	128	128	95.10%

Table 5: Experimental Results with an RNN controller

Generally speaking, experimenting with an RNN controller was a process that included a lot of instability in the network's performance. As a matter of fact, the cases reported in the table above, are also experiments that were unstable. In the implementation with the dense-layer [95] controller we rarely had any signs of unstable behaviour from our agent. However, in the case of the RNN controller, instability in the network's performance occurs more often than not. On a generic basis, the agent manages to find the optimal architecture and be led to convergence, even in the cases where sub-optimal networks are generated for many consecutive episodes. Figure 16 is an example of the network's performance for 1000 episodes:

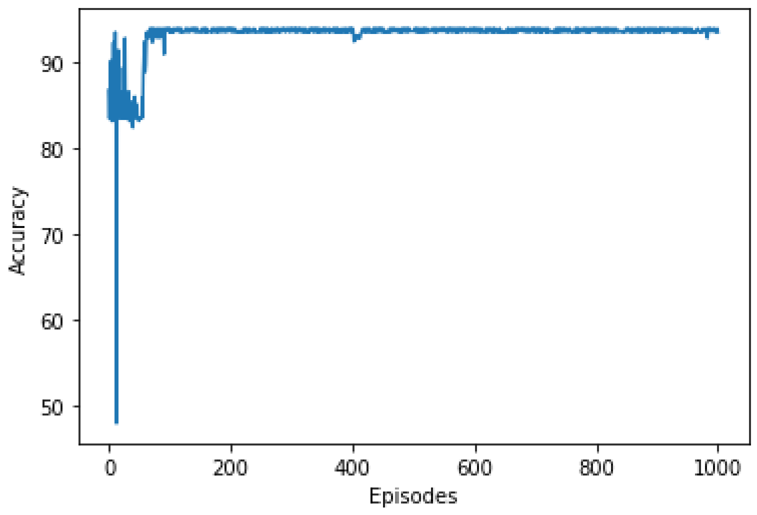


Figure 16: Accuracies of the constructed neural architectures (RNN controller)

Do notice the instability, which is arguably quite random, at around 400 episodes. By that point, the epsilon has already fully decayed and our agent is making decisions solely based on its own judgement, or in other words, the predicted and target Q-values. We do know that by that point the agent has already learnt the best possible neural architecture (a noticeable stability and convergence approximately a little after 120 episodes). Obviously, in the presented scenario, this minor instability does not cause any malfunction in the overall performance of the agent, and neither do greater instabilities. However, there have been cases where the instabilities were concerning and kept the agent in sub-optimal architectures for a long time. The silver lining is, that in the vast majority of the cases, our agent did not end up converging on a sub-optimal architecture (one around 83%). It is worth mentioning, though, that there were a few cases where our agent failed its task, resulting in average percentages of constructed architectures over 93% being 22%, 26.40%, 39.30% or even below 1%.

We do need to reiterate that a baseline reward reduction of 90 is enforced, tremendously improving the performance of our network. This is yet another implementation where the performances of the single/prediction network and the one of the target and prediction networks combination are indistinguishable. If we would like to include some extra differentiators compared to the previous implementation, other than the signs of instability being more often, we would have to mention that the training process was noticeably slower. However, once again, thanks to the contribution of NASBench-101 [15] and NORD [16], in just a few minutes we constructed and evaluated a certain number of architectures which normally, according to NASBench-101, would require over 25 days of total computational time.

7.3.5 Takeaways

In this implementation, which is very slightly modified compared to its predecessor, there are very few differences. The performance of the Deep Q-Networks is still not disappointing at all, but we ought to keep in mind that we are dealing with a problem of low dimensionality. The search space is limited to a total of 363 possible states, which is considerably small. Looking at the modifications we have made, which are mostly related to the agent's network architecture, we can keep as a note that the LSTM-layer controller required more epochs in order to stabilize and remove any signs of instability in its performance and jittering in the accuracy of the constructed networks' architectures. It is not surprising, though, that the agent is still able to converge, by locating and learning very decent architectures. It is reminded yet again, though, that due to the current representation of the states and the enforced limitations to our networks, the accuracy of the constructed networks have a set ceiling value, which can be exceeded only if we expand the search space. This will be done by allowing the algorithm to connect already existing layers, which we furtherly analyze in the next implementation.

7.4 Deep Q-Learning using an RNN controller and a wider search space

7.4.1 Introduction

The following implementation is much different from its predecessors. We make radical changes to the way we have been approaching this problem. There are changes not only in the architecture of our agent's network, but also in various parts of the algorithm. The greatest change of them all, though, is that we are widening the search space to a much larger portion of the NASBench-101 [15] benchmark. Now, our agent does not only have the option to add a new layer for up to five hidden layers, but is also allowed to choose which of the previous layers it is going to connect it to. This will have a great impact on the accuracies that the constructed networks will achieve, since now our agent is aiming to find the good architectures in a much wider search space due to the fact that it is not limited to sequential neural architectures. The reason that the updated search space is not equal to the entire NASBench-101 search space is because we have enforced some limitations to the search process, which we discuss later and specifically define the search space of this implementation. We also discuss various experiments that were conducted in order to find the best possible parameters, algorithmic modifications and alternations in the agent’s architecture that are needed for this approach, which is one that will put our Deep Q-Networks to the test.

7.4.2 Methodology

First and foremost, it is quite necessary to update the way we represent our states in order to formulate this problem in a different way. Reiterating, we are now allowing our agent to not only add layers to the constructed neural architecture, but also connect them to any of the previously added ones. This way, we are technically giving our agent the opportunity to build any of the 423624 architectures that are included in the NASBench-101 benchmark. However, this number is reduced due to some extra limitations we enforce. We now have a new way of depicting the state of our network. Previously, we would use a one-hot encoded vector of 4 elements, indicating the type of each specific layer (or the absence of a layer). Also, having a maximum of 5 hidden layers resulted in our state representation array being of size 5x4. In this implementation, we need to use vectors of 12 elements instead of 4. The initial state of each of these vectors is: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]. The number of the total vectors is not 5 in this particular case, but 6. The reason for adding an extra row in this case is because our constructed networks are not sequential, meaning that we can connect each layer to any of the previously added ones. Therefore, we now also care about the connections of the output layer, which are represented by the 6th row. The reason for using a total of 12 elements in each of our rows is because each one of these rows does not only represent the layer type, as it used to, but also the connections to the previous layers. The first 5 indices are related to the layer type(Figure 17): index 0 indicates the absence of a layer (as it used to), indices 1 to 3 indicate the layer types they used to (1x1 convolution, 3x3 convolution, 3x3 max pooling respectively) and index 4 represents the output layer. The remaining 7 indices (from 5 to 11) represent the layers that the current layer is connected to. Indices 5 to 10 (6 in total) correspond to the input layer and the 5 hidden layers, while the last index represents the state of having no connection. For example, if the 4th row of our 6x12 representation array is the following [0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0], it means that our 4th hidden layer is a 3x3 convolution layer which is connected to the 1st and 2nd hidden layers of our network. As a second example, if the 5th row of our state representation array is the following [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1], it means that the 5th hidden layer of our network is a 3x3 max pooling layer, which is only connected to the 2nd hidden layer. The possible actions of our agent remain the same as in the previous implementations. Our agent is able to pick any option from 0, 1, 2 and 3, which correspond to no-layer, 1x1 convolution, 3x3 convolution and 3x3 max pooling. When one of these layers is added, our agent will also know which of the previously added layers to connect it to. As for the rewards in this problem, they will also be exactly the same as in the previous problems, basically being the accuracy of the constructed network.

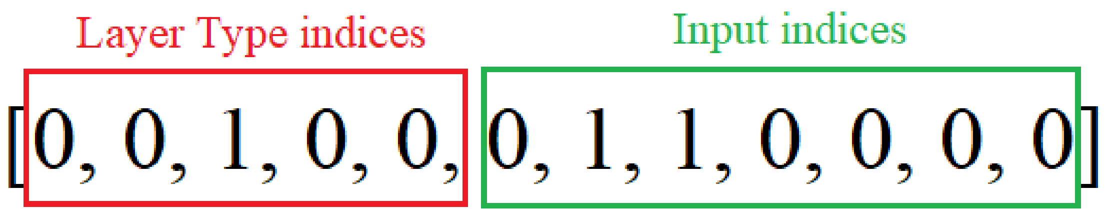


Figure 17: Layer Type and Input indices vector example

Moving on, from an algorithmic standpoint, a few changes were required in order for us to be NASBench-101 compliant. It is important to reiterate the constraints enforced by the aforementioned benchmark, which dictate a maximum of 7 layers (or 5 hidden, excluding the input and output layers) and a maximum of 9 connections in-between these layers. So far, it has been quite easy to enforce the limitation of the 5 hidden-layer maximum. We were doing this by instructing our agent to take the 'no-layer' action whenever we reached the hidden layer maximum, leading the episode to a terminal state. It has been decided that in order not to violate the limitation of the maximum connections in-between the layers, we have made the decision to enforce a number of maximum connections per layer, which is 2. This practically means that each layer can be connected to 0, 1 or 2 of the previous layers. This is a way to try to limit cases where the number of connections would exceed 9, ergo being rendered invalid. As a result, enforcing a maximum of 2 connections per layer and having a maximum of 7 possible layers (or, if depicted through a graph, layers would be the vertices and connections would be the edges), the search space of this implementation is limited to a total of 182947 possible states (or 182947 possible graphs, with respect to the limitations we have just described). As a reminder, it is worth mentioning that the entire search space of NASBench-101 is about 423000 architectures, which is about 2.3 times higher than the one enforced in our implementation. Nonetheless, with some simple Mathematic calculations, we can figure out that this will not be enough. The number of 9 connections could be reached just by the 5 initial hidden layers (2 connections each), without even counting the possible connections of the output layer in. For this reason, we have added an extra condition for our terminal state triggering. Previously, we had mentioned that we would force our agent to take action ‘0’ whenever we reached the maximum of 5 layers. Now, we also enforce this behaviour when we reached a total of 7 connections or more. Algorithmically, this check is satisfied only when we have 7 or 8 connections, allowing for at least 1 spare connection seed for the final layer, the output layer.

When it comes down to connecting layers, there have been various decisions that have been made in terms of the design of the algorithmic approach. It is important to make a quick note here about the outputs of our agent, which is analyzed more thoroughly in the following chapter. Our agent’s neural network has 3 different output layers, 1 for the layer type or action our agent will take, and 2 more for the 2 possible outputs of the layer our agent has been instructed to add. Having said that, inspired by the original paper, we decided that in terms of a slight processing of the network’s outputs, we will redirect any invalid connection suggestion to the input layer. For instance, if our agent’s connection outputs for hidden layer 3 are hidden layer 1 and hidden layer 4, it is quite obvious that the suggestion of hidden layer 4 is invalid since it has not been added yet. Therefore, this invalid suggestion is redirected to the input layer. After this slight processing of the network’s outputs, we check if the 2 suggested connections are the same. First, if the 2 connection suggestions are the same and both of them are either for the input-layer option or for the no-connection option, then one of them is redirected to the input-layer option and one of them to the no-connection option. Second, if the 2 connection suggestions are the same, but they do not happen to be either input-layer or no-connection (e.g. both are connections to hidden layer 3), then we have decided to apply the following approach: If the ratio of the total number of connections added up until that point over the total number of layers up until that point is not greater than 1.5 (which is a ratio parameter defined by us), then one of the connections is redirected to the input-layer connection and the second one is kept the same. But, if the previously mentioned ratio is greater than or equal to 1.5, then the connection that would be redirected to the input-layer index is now redirected to the no-connection index. This is merely enforced to dictate a balanced rate of adding connections and not exceed the limit of 9 connections way too quickly in the process of constructing our neural architectures.

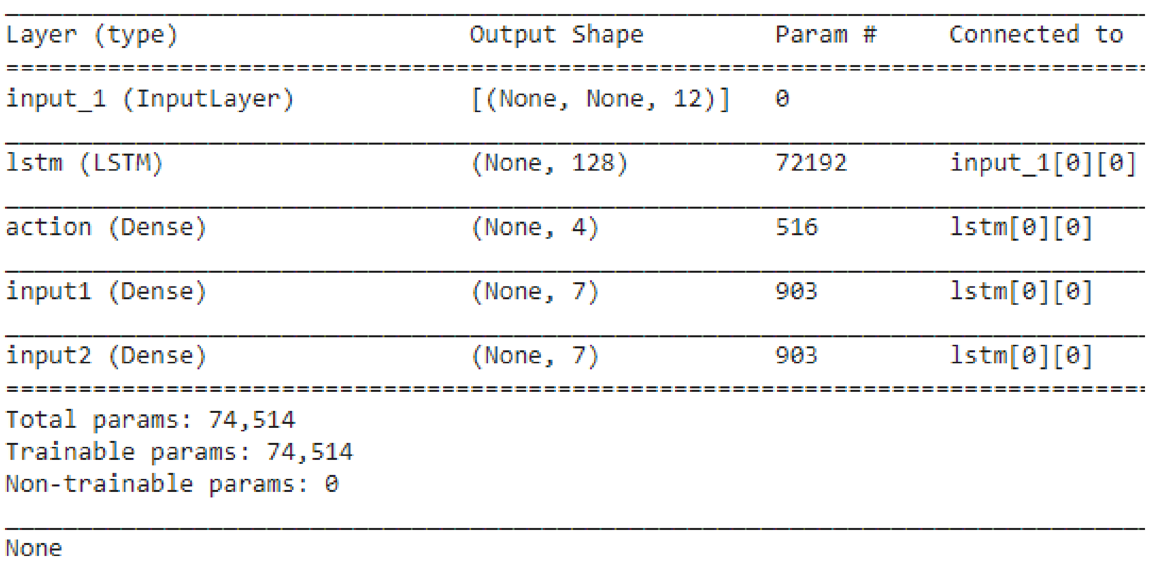
Having analyzed the necessary algorithmic modifications above, we can move on to the experimentations regarding the hyper-parameters of our formulated problem. It is incontrovertible that we now need to decrease the decaying amount of epsilon for our ε-greedy strategy approach. This is because we have turned the search space from quite a small one to an enormous one and it would be much better for our agent to have more time taking random actions in order to properly explore this wide search space. The values we tested for epsilon were 0.001, 0.0015 and 0.002 and, based on the conducted experiments, we decided that using 0.001 and allowing the agent to explore for longer was beneficial. To compound this, we increased the number of episodes in order to have more concrete results and be able to see if our agent has learnt anything and what that might be. We increased the number of episodes to 2000 from 1000, which was sufficient for epsilon to decay and have many episodes in which the agent would make a decision on its own, without any random actions. Phenomenally, we also needed to increase the capacity of the experience replay memory and the size of the samples that are taken from the memory in order to train the network. Between 10000 and 15000 for the capacity of the memory, 10000 was the one chosen since the alternative was way too large and unnecessary. As for the sample size parameter, after trying 512, 1024 and 2048, we decided that 1024 was the best option out of the three, on account of decent functional results and not much impact on the performance of the network. We need to remember that this sample size defines the number of experiences that are fed into the network to train it at the end of each episode. The larger the size of the sample, the longer the training time. Still, the prediction network’s weights are copied to the target network’s weights every 13 episodes. The batch size when training the network has been kept to 64, but the number of epochs was reduced to 1. This was done due to the fact that no significant difference in the results has been noticed when comparing the results of the implementation of 1 and 10 epochs. Additionally, the amount of time it takes to train the network for such a large sample size for 10 epochs, makes the experimentation with it quite unfeasible. Thus, we reduced the number of epochs to 1.

7.4.3 Implementation

For this implementation, the Python programming language was used, alongside with Keras [96] framework and the useful deep-learning distributed framework, NORD [16]. There are quite a few things to mention in the implementation process of our agent's network this time.

To begin with, throughout the testing of the parameters explained in the previous section, we have also been trying both GRU and LSTM layers, with the LSTM ones emerging as prevalent yet again. The number of units, though, was kept the same, 128, without any further experimentation. The input of our network is now an array of greater dimensions, a 6x12 one. The greatest differentiator of this current implementation to the previous ones, is that we now have 3 output layers instead of one. Our 3 output layers are 3 dense layers [95], all using the hyperbolic tangent as their activation function, and having 4, 7 and 7 units respectively. Specifically, the first output layer is responsible for the actions (4 possible actions: 3 possible layer type additions or no addition at all), while the following two are responsible for the two inputs of the layer to be added; the 2nd output layer makes the choice for one of the two inputs and the 3rd makes the second one. All three of our output layers have the one and only LSTM layer as their single input.

We experimented with various modifications and techniques to improve the performance of our agent. One of them was splitting our agent's network to make separated decisions; we used one network that would be responsible only for the actions (so, one output layer with 4 units) and another one that would be responsible only for the inputs (so, two output layers, 7 units each). This modification ended up overcomplicating things without aiding into improving our network and thus was discarded. Another modification that we tried was including an intermediate Dense layer, between the LSTM and the three output layers, meaning that the output layers would have this additional Dense layer as their single input. The aforementioned modification did not improve the performance of our agent at all. In fact, it had questionable results and possibly worsened the performance of our network and was discarded, too. Experimenting with various optimizers (instead of just Adam [102], [103] that we have been using so far) did not help much. Adamax [104] and Stochastic Gradient Descent [105], with various learning rates, did not seem to affect the performance of our network positively. Therefore, Adam with a learning rate of 0.001 was chosen yet another time. However, a modification which has positive results, when it comes down to the performance of our agent, was the specification of the loss weights in our Keras [96] model. Essentially, we are manually enforcing the way each output layer's outputs' loss impacts the training of our network during back-propagation. After a few experiments, we came to the conclusion that minimizing the impact of the two output layers responsible for the possible inputs had great results in the performance of our network, mitigating any occurring instabilities in its performance. Concretely, the values we set to the weights for layer were: 1 for the layer responsible for the actions and 0.01 for each of the layers responsible for the outputs. The value of 0.01 looks like quite radical depletion of these two output layers, but it is what worked best for us in this specific scenario. In Figure 18, we can see the Keras Summary of the LSTM controller's neural network.


```
________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================
input_1 (InputLayer)            [(None, None, 12)]   0

________________________________________________________________________________
lstm (LSTM)                     (None, 128)          72192       input_1[0][0]
________________________________________________________________________________
action (Dense)                  (None, 4)            516         lstm[0][0]
________________________________________________________________________________
input1 (Dense)                  (None, 7)            903         lstm[0][0]
________________________________________________________________________________
input2 (Dense)                  (None, 7)            903         lstm[0][0]
================================================================================
Total params: 74,514
Trainable params: 74,514
Non-trainable params: 0

________________________________________________________________________________
None
```

Figure 18: The Keras summary of the LSTM controller's neural network (larger search space implementation)

Through the Keras summary of our network we can also observe how the three dense output layers are all connected to the LSTM layer.

7.4.4 Experiments and Results

We have finally come to a point where we can deduce certain things about the algorithm and our model, after the embodiment of the more advanced algorithmic part that allows the exploration in a much wider search space than the one in the previous implementations. We began in this section with the hypothesis that the algorithm we are using in our approach is relatively weak, while evaluating the results of this third and final implementation verifies this initial hypothesis.

Since this final formulated problem unveils the suspected weakness of the Deep Q-Learning algorithm, we need to reassess the way we evaluate the performance of our network. If we were to do that by looking at the average percentage of the constructed architectures that evaluate to over a certain percentage (which would be more than 93% since now that we are in a wider search space, we expect better accuracies from the constructed networks), we would not be very fair to what we have done in this final approach. Therefore, we focus only on the best generated architecture, since despite all our efforts and the parameter experimentation and testing, we never managed to aid the agent to converge.

On the following table, we see some of the most indicative parameter combinations and the best generated architecture. Something worth noting here is that due to the wider search space it was a necessity to increase the sample size by a lot. As a result, the execution time increased significantly. All of the following results and episodes relate to executions of 2000 episodes. For the following table (Table 6), the parameters not included are set to their optimal values based on the conducted experiments (target network's weights updated every 13 iterations, a 128-unit LSTM layer for the controller, no intermediate dense layer between the LSTM and the 3 output layers of the controller, hyperbolic tangent activation function for the output layers).

Epsilon Decay	Memory Size	Sample Size	Loss weights (on the 'input' output layers)	Optimizer	Best accuracy
0.001	10000	1024	0.4	Adam	94.20%
0.0015	10000	1024	0.2	Adam	94.10%
0.001	20000	2048	0.1	Adam	94.33%
0.001	10000	1024	0.01	SGD	93.90%
0.001	10000	1024	0.01	Adamax	94.10%
0.001	10000	2048	0.01	Adam	94.37%
0.001	10000	1024	0.01	Adam	94.72%

Table 6: Experimental results with an RNN controller and a wider search space

The general tendency of the agent in this current approach is to converge on a sub-optimal architecture that evaluates to an accuracy of around 83%, despite having managed to explore and find much better architectures among which, one that has a validation accuracy of 94.72%, which is really high.

On the following table, we present the average accuracies of the constructed neural architectures in some combinations related to the epsilon and the epsilon decay parameters. We show the results for 6 combinations in total (2 possible epsilons: 0.85 and 0.75 and 3 possible epsilon decay values: 0.001, 0.004, 0.008) in Table 7. We also present two different cases for the mean calculations, one in which we include the invalid architectures that evaluate to an accuracy of 0, and one in which we do not. Due to the complexity of this formulated problem and the way we have enforced the certain limitations, generating invalid architectures proved to be noticeably more common than in the previous two implementations.

Epsilon	Epsilon Decay	Best accuracy	Mean accuracy	Mean accuracy (without 0% accuracies)
0.75	0.001	94.72%	77.42%	84.27%
	0.004	94.72%	69.69%	83.59%
	0.008	93.80%	52.29%	83.42%
0.85	0.001	94.72%	73.85%	83.78%
	0.004	94.72%	52.59%	83.48%
	0.008	94.71%	55.91%	83.73%

Table 7: Epsilon and epsilon decay experimentation using an RNN controller and a wider search space

As it is quite evident from the table above, most of the epsilon and epsilon-decay combinations yield the same results when it comes to the best accuracy that has been found and the mean accuracy when invalid architectures are not taken into account for the mean calculation. However, we do notice a trend in generating way more invalid architectures when the epsilon decay is larger. As we have seen, though, due to the weakness of the algorithm and the inadequacies that come out, no matter what we do in terms of parameter optimization, our algorithm fails to converge on the better architectures it has obviously found before. This is also quite easy to observe in Figure 19, which happens to be one of the quite successful executions due to the fact that we have had plenty of architectures in the first few episodes that were not evaluating to 83% accuracy. Based on the stochasticity of every execution, we have also had executions where pretty much from the very beginning the algorithm has almost already converged on an 83% architecture, failing to even sample any architecture higher than that.

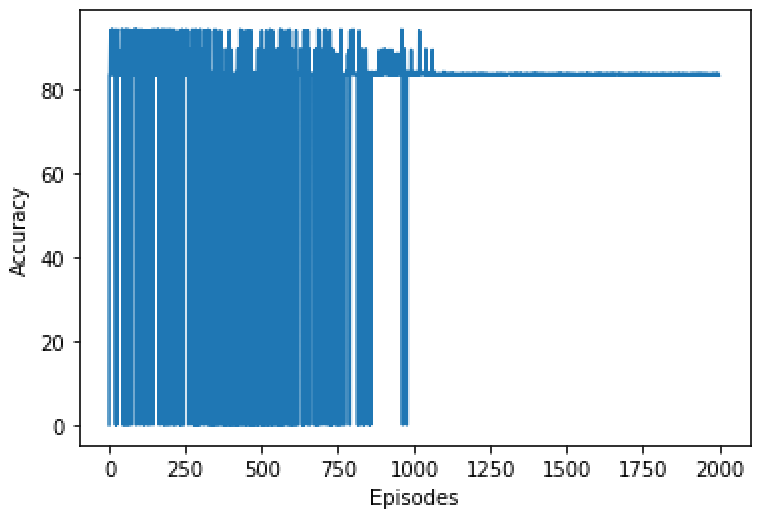


Figure 19: Accuracies of the constructed neural architectures (RNN controller, wider search space)

Again, this is one of the executions that we have a high density of architectures that evaluate over 94% for the first 800-850 episodes. Unfortunately, though, we observe the convergence on an architecture that evaluates to about 83% accuracy at the point that the epsilon, hence any chance of stochasticity, has decayed. Finally, this is the first time in these three Deep Q-Learning implementations that the two-network setup (prediction and target) outperforms the single-network (just the prediction network) one. Concretely, the signs of gradual instability throughout the entirety of the episodes or the inability to even sample any architectures over 83% were more common and noticeable in the single-network setup than in the two-network one.

7.4.5 Takeaways

As it is quite evident, the initial suspicion that Deep Q-Networks would not be of adequate performance for a wider search space has been verified. As it has been said in the very beginning of this section, the algorithm we have been using here is a relatively simple one. By enhancing our agents with the capability to connect layers and not just add them, hence widening the search space and unlocking the possibility of constructing non-sequential neural architectures, some architectures that evaluate to really high accuracies (like 94,72%) have been found. Despite the note of success thanks to the fact that we have found these well performing architectures, this final implementation proved that Deep Q-Networks are not that competent an algorithm for problems with wider search spaces.

One of the takeaways of this implementation is that it was quite necessary for our algorithm to have enough exploration time to evaluate various architectures. This way, we prevented the algorithm from converging way too early on a sub-optimal neural architecture. Unfortunately, though, with the current implementation, we do not seem to have the power to make the agent truly learn anything valuable from the training process. In spite of succeeding in exploring the search space and exploiting the knowledge we gained through the exploration enough to find architectures that evaluate to really high accuracies, our agent networks do not manage to converge.

7.5 Conclusion

In summary, it is unequivocal that, as we have initially mentioned, Deep Q-Networks are definitely not a very competent algorithm to tackle the problem of Neural Architecture Search at its full capacity. Looking at things through the prism of the size of the search space, we can most definitely infer that it is what contributed to the success of the initial two sub-implementations in this section. Having a relatively small search space does not reveal the inadequacies of the Deep Q-Networks, as we noticed with our initial experiments. In these cases, Deep Q-Networks can perform quite well and manage to converge quite fast on the optimal solution. However, enhancing the possible actions' arsenal with layer connections and not just layer additions, in order to try to tackle the problem at our full potential, was the action that revealed the problems of the algorithm. We can probably base the agent's inadequacies on two different things: Firstly, and we have already talked about this before, the way the agent is searching through the search space is definitely not very competent. Our controller makes decisions based on the predicted and the target Q-values. Despite the fact that we have done our best to figure out what the optimal values for each of the parameters were, we still observe our agents underperform. Secondly, we can safely make the assumption that it is quite possible that the way we are representing the states and feeding them to our network is not optimal either. We have made what seemed to be, from a representation correctness standpoint, the best possible choice, considering that we needed to represent the layers and all possible actions in a way that they would be good to feed to an RNN controller. All things considered and taking into account that the flagship of all the differences between the two initial sub-implementations and the third one is the size of the search space, we can conclude that it is the number one reason that our algorithm fails to lead our controller to convergence in environments with larger search space. It is worth reiterating though that our controller does manage to find some exceptionally good architectures.

Having said all of the above, it is important to close the chapter of Reinforcement Learning by saying that we do not focus on improving this current implementation or try a different Reinforcement learning approach. Later down in this thesis, we get into some more detail about what we could have possibly done to improve this implementation. The major purpose of these three sub-implementations, with the third one being the most significant one that definitely stands out as a better formulated problem and a more complete one, was to make some initial steps into the field of Neural Architecture Search using Reinforcement Learning. In the following sections, we follow some completely different approaches.

8 Evolutionary Algorithms

8.1 Introduction

Moving on, we follow a completely different method in order to approach the problem of Neural Architecture Search. Taking a step forward in order to choose a different optimization method from Reinforcement Learning, which we have used so far, we make a transition towards Evolutionary Algorithms [17]. The following implementation is way different to the previous ones from an algorithmic standpoint.

8.2 Evolutionary Algorithm implementation

8.2.1 Introduction

As it has been mentioned, we are utilizing the NASNet search space, tuned in a way to be compliant with the NASBench-101 [15] benchmark limitations. In this implementation, we follow the algorithmic logic presented in this paper [13]. It is important to note that we are merely talking about an evolutionary algorithm here, and not specifically a genetic algorithm, since we only have mutations in our populations, but not any kind of cross-over to the selected parents.

8.2.2 Methodology

It is quite a straight-forward process to work with this relatively simple-to-implement algorithm (the one introduced in the paper cited above). The first step of this algorithm is to create random neural architectures, evaluate them and add them to the end of the population data structure (which is a queue) and the history. This needs to be done for a user-specified amount of times, which is basically the size of the population. We make a more detailed reference to the parameters and the values we have given to them a little bit later. Next, as soon as we have completed the initial step of creating a certain number of architectures, evaluating them and storing them to certain data structures, we are ready to move on to the main part of our evolutionary algorithm. What we need to do is to randomly pick a certain number of architectures from our population, so technically, a sample. From these randomly picked architectures, we pick the best performing architecture, which is something we do know due to the fact that we have already evaluated them right after their construction. Then, the best performing architecture that we have picked, which is the parent, undergoes a certain mutation. The outcome of this mutation applied to the parent is a new, different architecture, which is be the child. Quite certainly, as we have done before, we need to evaluate the child architecture that we have just created by mutating the parent and add it to the right of the population and the history. As a final step, we need to discard a member of the population as we have just added one. The member of the population that is discarded is the oldest one. To put it simply, in every iteration, after we have constructed a population of the desired user-defined size, we add one member to the right of the population and remove one of its left end since, as we have already said, the population is basically a queue. This entire procedure described above is repeated for a certain amount of times, which is another user-defined parameter. As an example, let us suppose that the user-defined population size is 100 and the history size is 500. This actually means that for the first 100 iterations we will focus on creating random neural architectures, evaluating them and adding them to both the population and the history. Then, for the following 400 iterations we will get a sample of a certain size from the population, let us say 25, get the best architecture from that sample, mutate it, evaluate it and add to the population and history. Last but not least, we will dispose of the oldest member of the population. For instance, in the 101st iteration, which is the first time we will pick a sample from the population, we will discard the member that was the initial one in our population.

So far, we have talked about a mutation that the best performing architecture of each iteration's sample undergoes, but we have not given a detailed description of what that might be like. It is known that our implementations are based on the NASNet search space with respect to the limitations enforced by the NASBench-101 benchmark. Therefore, the mutation that the parent architecture undergoes, in order for us to produce a child architecture, is either a layer addition or a connection between two existing layers. What is interesting to mention here, which is something that differentiates the layer addition of this implementation from the others', is that when adding a layer, we may select to place this newly added layer in-between already existing and connected layers, ergo resulting in the disconnection of these two layers and the connection of each one of them with the newly added, intermediate layer. To select whether we add a layer or connect two existing ones, we have defined two ratios, one node addition ratio and one node connection one. Every time the mutation process needs to be executed, a random number is generated, which is utilized to randomly select one of the two possible action types.

8.2.3 Implementation

For the evolutionary algorithm implementation for Neural Architecture Search, as in the previous ones, we used the Python programming language. This time, we do not need to utilize the Keras [96] framework since we are not dealing with any neural network controllers. Nevertheless, using the NORD [16] framework is absolutely necessary in this implementation, too. We are using NORD [16] in order to construct the neural architectures and evaluate them, as in the previous implementations, but also perform the mutations that we have been talking about in this section.

8.2.4 Experimental Setup

In the previous sections, we have been talking about various parameters that we needed to define. It is time we provided more details about what values we experimented with and what were the ones that were prevalent. Firstly, there were various values that were tested for the rate of adding nodes and the one for connecting layers. The ones that proved to be performing well, regardless of the other parameters, were 0.05 for the node addition rate (although the other 4 values that were tested in the experiments did not lead to great differences in the results) and 0.1 or higher for the layer connection rate (the 3 highest values that were tested for this rate were performing noticeably better than the 2 smaller ones). Secondly, regarding the population and sample size, we followed the guidelines of the paper’s experiments. Specifically, we tested all the following combinations for the population and sample size: 100/2, 100/50, 20/20, 100/25, 64/16. As a matter of fact, the findings of the paper were verified, since 100 for the population size and 25 for the sample size were indeed the best performing combination of them all although the differences were not drastic once again. Finally, the number of evolve cycles we chose was 1500. Various values were tested, mostly less than 1500, and we noticed that the algorithm did not have enough time to properly construct good neural architectures. Any amount greater than 1500 was rendered unnecessary since, by that point, most times, our algorithm had managed to find the best possible architecture based on its current limitations and parameter settings (and by that we mean that certain parameter combinations enforced some performance ceiling on our algorithm). On a side note, it is quite interesting to notice that in this evolutionary algorithm implementation we have way fewer parameters to tune and experiment with compared to the Reinforcement Learning one.

8.2.5 Experiments and Results

We conducted experiments for 5 values for the connection and the node addition rates. Concretely, the five values we tried for each of them were 0.01, 0.05, 0.1, 0.15 and 0.2. We need to reiterate that there are also 5 possible combinations for the population and sample size, as instructed by said paper, which are 100/2, 100/25, 100/50, 20/20 and 64/16. The number of evolve cycles we used in our experiments is 1500. For each combination of node addition rates, connection rates and population and sample size combination (5 node rates x 5 connection rates x 5 population and sample size combinations = 125 possible combinations), we conducted a total of 10 experiments. In the following figures, we present the results of these experiments using heatmaps that demonstrate the accuracy of the best neural architecture that has been found, in all 10 experiments, for each of the 125 possible combinations.

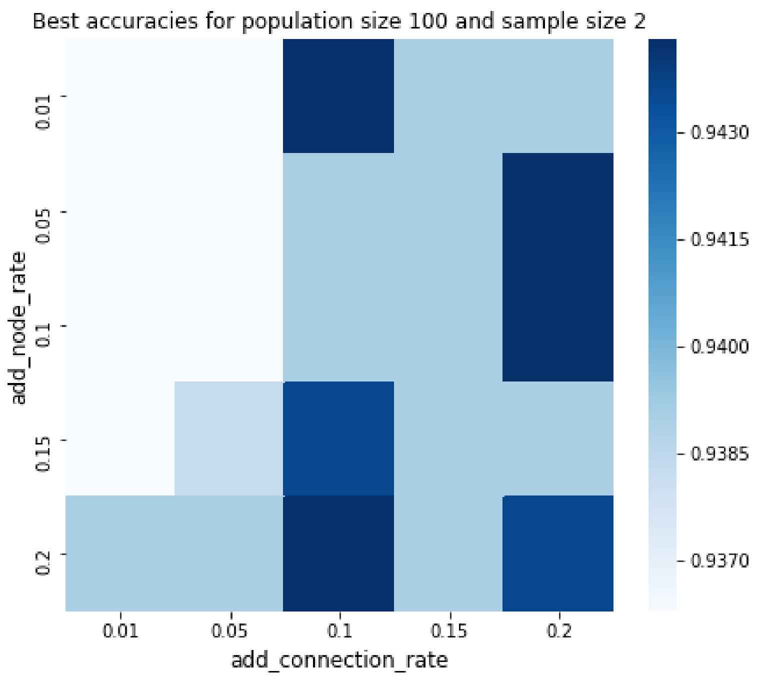


Figure 20: Best accuracies for population size 100 and sample size 2

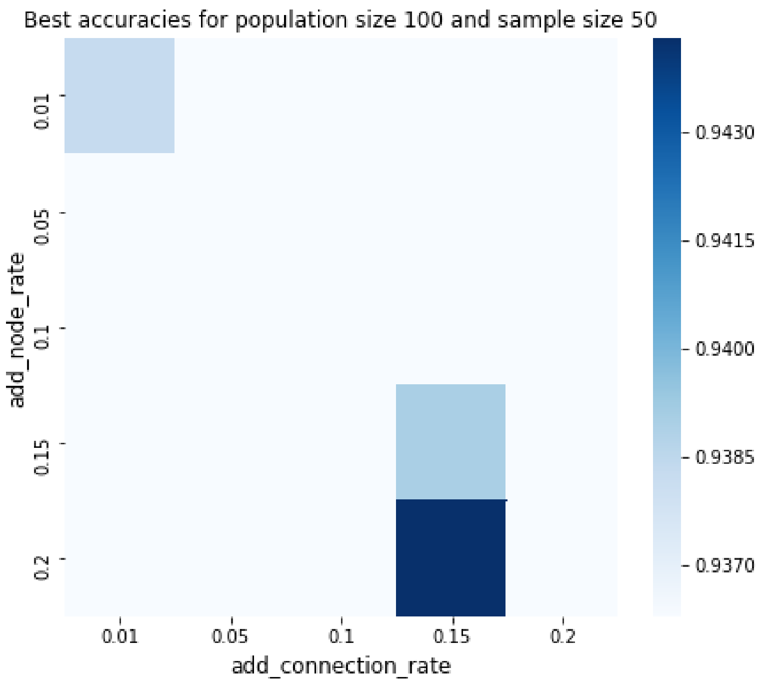


Figure 21: Best accuracies for population size 100 and sample size 50

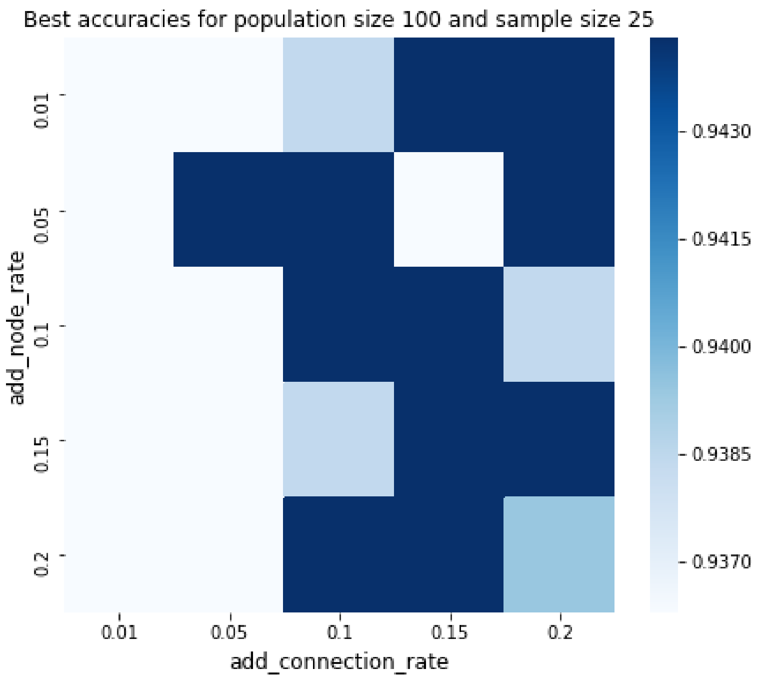


Figure 22: Best accuracies for population size 100 and sample size 25

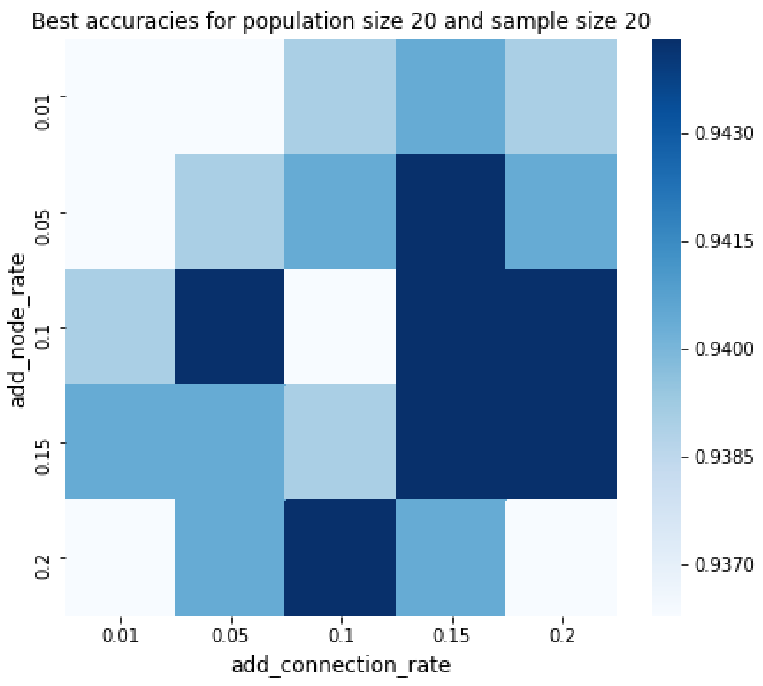


Figure 23: Best accuracies for population size 20 and sample size 20

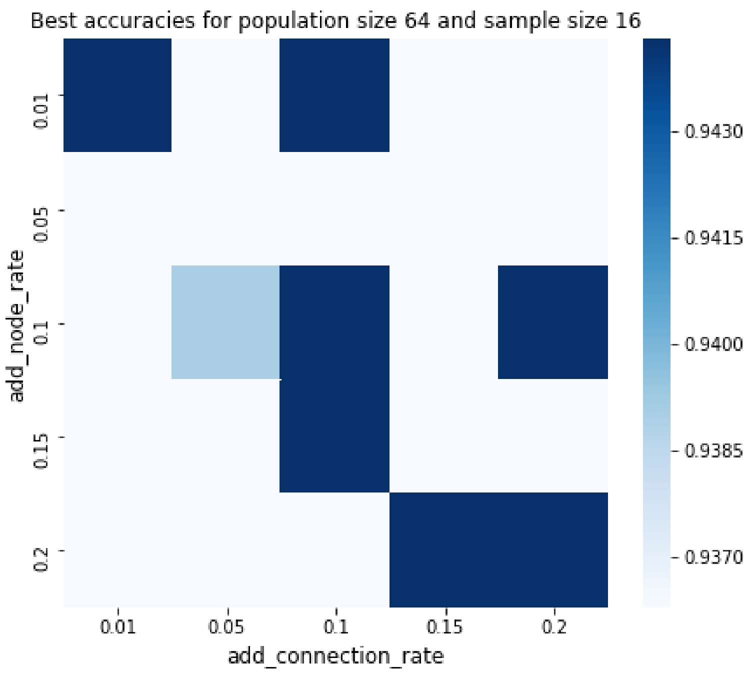


Figure 24: Best accuracies for population size 64 and sample size 16

In these 5 figures, we can see that there is no clear pattern of what values of the node addition rate could be optimal. However, we do notice that the vast majority of the cases in which the highest performing architecture (that this algorithm has found) was found, were for connection rates over 0.1. The best accuracy in our experiments is equal to 94.43%, represented by the darkest blue squares in the heatmaps. The worst of the best accuracies that has been found, which is represented by the totally white squares, is one equal to 93.62%. It is noticeable that 4 of the 5 combinations are doing relatively well, except for the combination of population size of 100 and sample size of 50, which has very few to almost no successful runs at all. In Table 8, we can see some extra details about the best accuracies for the population of 100 and sample of 25 combination.

Mean	Std	Min	25%	50%	75%	Max
0.940189	0.003811	0.936287	0.936287	0.938386	0.944300	0.944300

Table 8: Best accuracy experiment details (population of 100 and sample of 25)

8.3 Conclusion

In summary, in this implementation we tackled the problem with an evolutionary algorithm. Truthfully, the algorithm is quite a straight-forward one, without much complexity in its structure and is presented in quite a simplistic way in the paper [13]. In this current approach, we had to deal with way fewer parameters that needed to be tuned and experimented with, especially compared to the Reinforcement Learning implementations we had analyzed earlier. It is quite important that with a much simpler and quicker (in terms of performance) algorithm we manage to find architectures that evaluate to quite high accuracies. What is worth mentioning here is that in the two different types of implementations we have talked about so far, the Reinforcement Learning and the Evolutionary Algorithm one, we are concerned about different things. When it comes to the Reinforcement Learning implementation where we are dealing with an agent which plays the role of the controller that makes decisions, we care about the convergence of the agent’s network on the best possible architecture, which was not feasible when it came down to a wider search space. On the contrary, in the case of the evolutionary algorithm, we are obviously not dealing with a controller, hence the whole idea of converging on a certain architecture is just non-existent. What matters in this particular case is to have our algorithm meander through the various members of the population, mutate them and eventually manage to produce the best possible neural architectures. This is a considerable differentiator between the two methods, since the convergence criterion constitutes a pathway to inadequacy for the Deep Q-Networks' algorithm, while aiming for the best possible neural architecture renders the evolutionary algorithm relatively successful. It is obvious that the best architecture which was found by our evolutionary algorithm might not evaluate to an accuracy as high as the one found in the Reinforcement Learning experiment, but that would possibly be the case, had we given the algorithm many more generations (or evolve cycles) to work with.

9 Metaheuristic Algorithms using PyGMO

9.1 Introduction

In the final set of experiments we are about to present, we focus on some metaheuristic algorithms and utilize PyGMO [14]. PyGMO (Python Parallel Global Multi-Objective Optimizer) is a scientific library that includes the parallelized implementation of various optimization problems and algorithms. This library is extremely useful for us since it includes the implementation of multiple metaheuristic algorithms, some of which we use and compare by conducting experiments, which is the main scope of this final section. In the following chapters, we examine the requirements that need to be met to successfully utilize PyGMO more thoroughly. For instance, so far in this thesis we have had a pretty standard way to depict the state of our network, or more specifically, its layers and connections. However, in this implementation, we need to comply with some different standards, defined by PyGMO, in order to utilize its already implemented optimization algorithms.

9.2 Metaheuristic global optimization algorithms using PyGMO

9.2.1 Introduction

Our focus in this section is the comparison between 3 metaheuristic global optimization algorithms from the ones already implemented in PyGMO. PyGMO provides us with a large variety of algorithms. The three algorithms we have chosen are: Extended Ant Colony Optimization (GACO) [18], [19], Particle Swarm Optimization (PSO) [20], [21] and Artificial Bee Colony (ABC) [22], [23]. Mostly for computational expedience purposes, we are not going to allow these algorithms to operate for way too long, meaning that we will not sample through a lot of models. This is to insinuate that it might not be quite likely to ever manage to find any of the best performing architectures NASBench-101 [15]. Due to this, we are planning to shift our focus on comparing how fast each of these algorithms has found the best architecture they have managed to find or also even something less relative than that which could be a common ground comparison. For example, we can compare how fast each of these algorithms manages to find a good enough architecture (e.g. one that evaluates to over 94%). In the following sections, we see the way we tackled the problem of Neural Architecture Search, always in the scope of a NASNet search space with respect to the limitations of NASBench-101, in order for it to be compliant with certain PyGMO guidelines that we needed to follow.

9.2.2 Methodology

We begin by analyzing how we formulated our problem, with respect to all its limitations, which is a requirement to use the framework we have mentioned in this section. PyGMO’s input needs to be a formulated optimization problem that needs to be solved. This is what we have practically done so far although we have just not been using certain mathematical notation in order to express the objective function and the equality or inequality constraints of our problem. So, this would make us think what our problem would look like when breaking it down into pieces.

To begin with, it is very clear that we are talking about a maximization problem here. The objective function we need to maximize is essentially the evaluation of our constructed neural network. As for the constraints, it might seem that we have quite many of them, but we manage to enforce only 2 inequality constraints. The first of these 2 inequality constraints is related to the maximum number of 9 edges/connections, while the second one is there to ensure the existence of at least one input for our output layer. We use the latter to reduce the amount of invalid constructed neural architectures. One would wonder what happened with the rest of the constraints or certain limitations we had in the previous implementations. What about the maximum of 5 hidden layers or the certain handling we have been doing to avoid connecting layers to ones that have not really been added yet?

We have not talked about the variables that are part of our optimization problem. We have the freedom to choose the number and data type of our variables, always with respect to certain PyGMO constraints. It has been decided to use a total of 26 variables that are essentially a vector of length equal to 26. The first 5 indices are the ones representing the layer types of the hidden layers of our network. This way, we are indirectly enforcing the maximum of 5 hidden layers. The possible values of these 5 initial indices are: 0 (no-layer), 1 (1x1 convolution), 2 (3x3 convolution) and 3 (3x3 max pooling), just like in the previous implementations. The remaining 21 indices are the ones representing the possible inputs for each layer. We have defined their sequence in a very understandable and comprehensible way, which we will analyze, but first, let us present the adjacency matrix for our neural network. As a reminder, our constructed networks are limited to a maximum of 7 layers, including the input and output ones, ergo a maximum of 5 hidden layers.

	input	**hidden_1**	**hidden_2**	**hidden_3**	**hidden_4**	**hidden_5**	**output**
input	x_{00}	x_{01}	x_{02}	x_{03}	x_{04}	x_{05}	x_{06}
hidden_1	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}
hidden_2	x_{20}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}
hidden_3	x_{30}	x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}
hidden_4	x_{40}	x_{41}	x_{42}	x_{43}	x_{44}	x_{45}	x_{46}
hidden_5	x_{50}	x_{51}	x_{52}	x_{53}	x_{54}	x_{55}	x_{56}
output	x_{60}	x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	x_{66}

Table 9: Adjacency matrix of a seven-layer neural network

By observing the adjacency matrix (Table 9) we have presented right above, we can see that the top triangular matrix (highlighted in light blue colour) is sufficient for our formulated problem, with x_{01} indicating a connection from layer 0 (the input layer), x_{06} indicating a connection from layer 0 to layer 6 (the output layer) and so forth. Picking only the upper triangular matrix of this adjacency matrix, not only are we selecting the subset of connections that we truly care about, but we are also indirectly banishing any invalid connections. Previously we had mentioned that we would have 26 variables, with the first 5 indices representing the 5 hidden layer types and the remaining 21 the connections among the layers. The order we have set for these variables is of key importance. The first of these 21 indices corresponds to x_{01}, the following two correspond to x_{02} and x_{12}, the following three to x_{03}, x_{13} and x_{23} and so on, with the final 6 indices corresponding to all possible inputs of layer 6, which is the output layer. By doing this, we have found quite a smart way to tackle the problem, because not only do we make this data structure easy to iterate over (iteration 1 is related to index 1, iteration 2 is related to the following 2 indices, iteration 3 is related to the following 3 indices etc.), but we also ensure that we are not consistently making invalid connections. The latter is ensured due to the fact that each layer's possible inputs are evaluated only after that layer has been added.

Having said all of the above, we are finally prepared to define our optimization problem:

max: Neural network evaluation (accuracy)

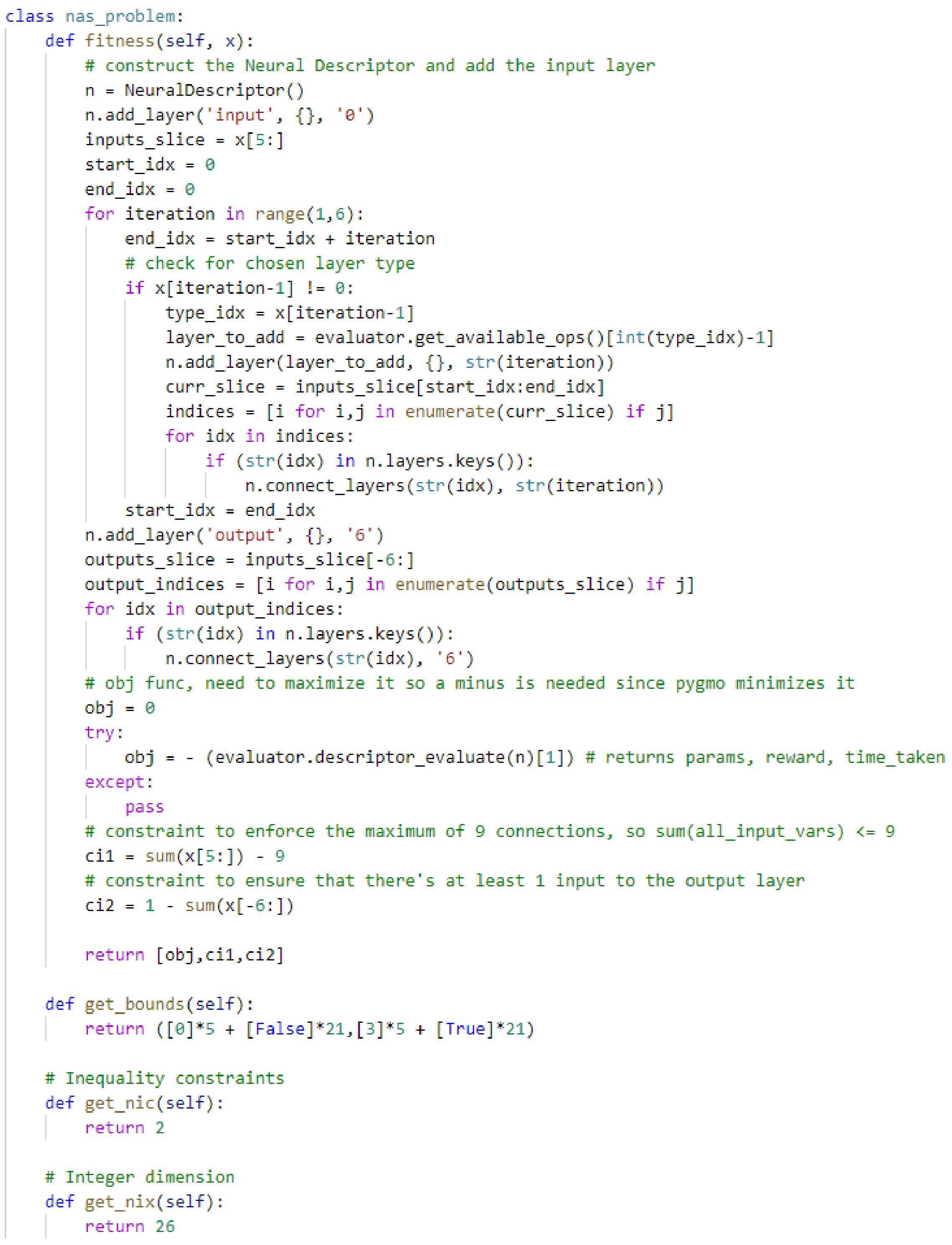
subject to: $\sum_{i=6}^{26} x_i \leq 9$

$\sum_{i=21}^{26} x_i \geq 1$

It is of crucial importance to clarify something, and that is that the way some optimization algorithms have been implemented in PyGMO, might or might not require these constraints. For instance, Extended Ant Colony Optimization (GACO) is an algorithm in PyGMO that can perform under certain constraints, while Artificial Bee Colony (ABC) cannot. Therefore, the problem formulation we have described above is not always going to be constraint inclusive. Additionally, due to certain PyGMO standards, this is not the exact way we have formulated our problem. For example, PyGMO treats every problem as a minimization problem. Therefore, in order to turn this into a maximization one, we have basically set up the minimization of the negative value of the neural architecture's evaluation. Also, there are some additional minor tweaks in the way the inequality constraints look, since PyGMO requires us to express them as equality constraints, which will afterwards be transformed to inequality constraints with the assumption that it needs to be less than or equal to 0. So, for example, if we want to define this $x_1 + x_2 \leq 9$ constraint, we need to define an equality constraint like *constraint_1* $= x_1 + x_2 - 9$, which in practice will be transformed to $x_1 + x_2 - 9 \leq 0$.

We have come to an end when it comes to describing all the steps to formulate our problem and make it PyGMO compliant. Needless to say, in-between those algorithmic steps described above, we are utilizing NORD [16] to construct our NeuralDescriptor, step by step, and eventually evaluate it using a Benchmark Evaluator instance. The prerequisites for the problem formulation have been met. Now, we need to move on to the step of the algorithm parametrization. We mentioned above that we are using three different algorithms in order to compare them: Extended Ant Colony Optimization (GACO), Particle Swarm Optimization (PSO) and Artificial Bee Colony (ABC). PyGMO provides us with the ability to tweak some of the parameters of these algorithms. The number of parameters that can be tuned differs per algorithm, some might have just a couple of parameters that are editable and some might have quadruple that amount. We dive into some more details about the parameters we experimented with for each of these algorithms in section 9.2.4.

Following, we are including the piece of code (Figure 25) that is related to the PyGMO user-defined problem we need to formulate in order to properly use its implemented algorithms:



```python
class nas_problem:
    def fitness(self, x):
        # construct the Neural Descriptor and add the input layer
        n = NeuralDescriptor()
        n.add_layer('input', {}, '0')
        inputs_slice = x[5:]
        start_idx = 0
        end_idx = 0
        for iteration in range(1,6):
            end_idx = start_idx + iteration
            # check for chosen layer type
            if x[iteration-1] != 0:
                type_idx = x[iteration-1]
                layer_to_add = evaluator.get_available_ops()[int(type_idx)-1]
                n.add_layer(layer_to_add, {}, str(iteration))
                curr_slice = inputs_slice[start_idx:end_idx]
                indices = [i for i,j in enumerate(curr_slice) if j]
                for idx in indices:
                    if (str(idx) in n.layers.keys()):
                        n.connect_layers(str(idx), str(iteration))
            start_idx = end_idx
        n.add_layer('output', {}, '6')
        outputs_slice = inputs_slice[-6:]
        output_indices = [i for i,j in enumerate(outputs_slice) if j]
        for idx in output_indices:
            if (str(idx) in n.layers.keys()):
                n.connect_layers(str(idx), '6')
        # obj func, need to maximize it so a minus is needed since pygmo minimizes it
        obj = 0
        try:
            obj = - (evaluator.descriptor_evaluate(n)[1]) # returns params, reward, time_taken
        except:
            pass
        # constraint to enforce the maximum of 9 connections, so sum(all_input_vars) <= 9
        ci1 = sum(x[5:]) - 9
        # constraint to ensure that there's at least 1 input to the output layer
        ci2 = 1 - sum(x[-6:])

        return [obj,ci1,ci2]

    def get_bounds(self):
        return ([0]*5 + [False]*21,[3]*5 + [True]*21)

    # Inequality constraints
    def get_nic(self):
        return 2

    # Integer dimension
    def get_nix(self):
        return 26
```

Figure 25: PyGMO User-Defined Problem code

9.2.3 Implementation

For this implementation, we used the Python programming language yet another time. Quite certainly, the protagonist of this final piece of code is the framework we introduced in this section, PyGMO. Its utilities and already implemented optimization algorithms proved to be crucial for these final experiments of this thesis. It goes without saying that NORD [16] was absolutely necessary for these final experiments, too, since its provision with convenient ways to construct neural architectures and evaluate them in an extremely quick manner is of vital importance.

9.2.4 Experimental Setup

Starting with the Extended Ant Colony Optimization algorithm, we would say that the flagships of this algorithm's parametrization revolve around the speed of convergence and the greediness of the algorithm. Specifically, there are 2 parameters, "q" and "n_gen_mark" that generally affect the speed of convergence. By experimenting with these two, we found that speedy convergence is not functioning well for this problem, using this algorithm. It was found that value 1.0 for parameter "q" was performing the best, while much smaller values like 0.2 or 0.01 seemed to underperform (the smaller the value of "q", the faster the convergence). Similarly, smaller values for "n_gen_mark" seemed to perform better, since they represented slower convergence, too. The values we experimented with, for this parameter, were 5, 7, 17, 37, 50 and 100, and the one we chose for our additional experiments was 7. In addition, when it comes to the algorithm greediness, we experimented with the 'focus' parameter (the higher its value, the greedier the algorithm and more focused in local improvements). We ended up using value 0.0 for this parameter, since higher values (ergo, a greedier approach) like 0.5 or 1.0 showed repetitive signs of underperformance. Finally, the "ker" parameter, which is the kernel size and represents the number of solutions that are stored in the solution archive, was set to 13. The values that we tested for this parameter were 2, 5, 13, 26 and 30. Frankly, this was the parameter whose different values seemed to affect the performance of the algorithm the least. We chose value 13, which, alongside value 2, seemed to be performing the best out of its peers.

Moving on, we evaluate the parametrization of the Particle Swarm Optimization algorithm. The parameters we mostly experiment with are related to the neighbourhood type, the maximum allowed particle velocity, the algorithm variant and the particles' inertia weight or constriction coefficient. To begin with, for the choice of the neighbourhood type, based on our experiments, we mostly vacillated between two alternatives, "1" (gbest) and "2" (lbest). Between these two, "1" (gbest) was performing much better than "2" (lbest), for all possible combinations of the other parameters. Especially for the comparison between these two aforementioned neighbourhood types, we had to tune the parameter "neighb_param", which technically dictates the amount of neighbours to consider in the 'lbest' scenario. We experimented with various amounts in order to set this parameter, with the smaller ones, like 4, proving to be the best, but still not sufficient to outperform the 'gbest' neighbourhood type. As about the "max_vel" parameter, which is the maximum velocity of the particles, the values that seemed to be of higher efficiency were intermediate ones, like 0.5, while higher and lower values, like 0.1 or 0.9, proved to be resulting in lower accuracies. Regarding the "variant" parameter, once again there were two alternatives that were outperforming the rest by quite a noticeable margin. These two options we are referring to are "1. Canonical (with inertia weight)" and "5. Canonical (with constriction fact.)". The differences between these two alternatives were barely noticeable. Due to some signs of slightly better performance, the variant we chose for our experiments is "1". Lastly, we had to tune the "omega" parameter, which is the inertia weight or constraint coefficient, depending on the variant that has been chosen. Based on some of the PyGMO examples on the website, we started by testing this parameter at a value of 0.7298, combined with various other combinations of the other parameters. As a matter of fact, smaller or greater values, like 0.1 or 0.9 did not lead to any noticeable differences. Based on the experiments we have run, this seems to be a parameter that has little effect on the performance of the algorithm. The value that has been finally chosen for it is the one that was initially tested, 0.7298.

For our final algorithm, the Artificial Bee Colony, we would not have much to say since there is only one parameter to be tuned, and that is the "limit" parameter. This parameter merely dictates the number of trials before abandoning a source. The values we experimented with range from 0 to 250. Specifically, the values that seemed to be performing the best out of all were smaller values, like 2, 5, 10, 20, 25 and not much greater ones like 50, 100, 200, 250. The value we chose to conduct most of our experiments with is 20.

In the section of the experimental results, we present some details about the parameter testing we have explained in the previous paragraphs. Also, we examine the results of the comparison between the three algorithms.

9.2.5 Experiments and Results

We begin by presenting the results of the experiments that were conducted for the selection of the best possible parameters per algorithm. For each algorithm, just like in the previous implementations, we present only a subset of the experiments and their respective results (Table 10, Table 11, Table 12), based on the presented efficiency of the parameters and their impact on it. It is important to mention here that all of the experiments were conducted for a population size of 30, for 50 generations, which is something applicable to all three algorithms.

ker	q	n_gen_mark	focus	Average Best Accuracy
5	1.0	17	0.0	94.04%
13	1.0	17	0.0	94.01%
13	0.2	37	1.0	93.93%
13	0.01	100	1.0	93.91%
26	1.0	50	0.0	94.02%
13	1.0	7	0.0	94.08%
2	1.0	7	0.0	94.05%
2	0.2	37	0.0	94.01%

Table 10: Results of the Ant Colony Optimization parameter experimentation

omega	max_vel	variant	neighb_type	neighb_param	Average Best Accuracy
0.7298	0.5	1	1 (gbest)	-	94.19%
0.7298	0.5	1	2 (lbest)	4	94.05%
0.7298	0.1	5	1 (gbest)	-	94.08%
0.7298	0.5	5	2 (lbest)	4	94.08%

0.1	0.9	1	2 (lbest)	4	94.04%
0.9	0.5	5	2 (lbest)	20	94.01%
0.7298	0.9	5	1 (gbest)	-	94.13%
0.7298	0.1	1	2 (lbest)	10	93.98%

Table 11: Results of the Particle Swarm Optimization parameter experimentation

limit	Average Best Accuracy
2	93.78%
5	93.71%
10	93.64%
20	93.87%
25	93.77%
50	93.54%
100	92.91%

Table 12: Results of the Artificial Bee Colony parameter experimentation

These are a few samples that are indicative of the impact of some of the adjustable parameters on the performance of the algorithms for our formulated problem. Do notice that one row in each of the three tables we have just presented is highlighted in light blue colour. These rows indicate the parameter(s) that we have finally chosen (we have already talked about them two sections before the current one) for the experiments that we use to compare all three algorithms.

In the scope of the comparison between the three algorithms, we have conducted experiments with population size equal to 30 and the number of generations being equal to 50. We have conducted a total of 10 experiments per algorithm, using the light blue highlighted parameters from the previous tables. We now present a table (Table 13) in which we include a few metrics that help us understand how all three algorithms are performing compared to each other and the random search. We have calculated and present:

- The highest accuracy found in all experiments of each algorithm
- The highest value of the calculated mean of each experiment
- The mean of all experiments for each algorithm
- The mean of all highest accuracies for each algorithm
- The mean generation that the best accuracy of each experiment was first met on

	Highest Accuracy	Highest Mean (per experiment)	Mean	Mean of Highest Accuracies	Average Generation in which the highest was first met
GACO	94.121%	94.017%	93.731%	94.088%	31.0
PSO	94.431%	94.179%	94.029%	94.186%	19.5
ABC	94.151%	94.018%	87.706%	93.847%	9.0
Random Search	94.061%	92.591%	90.109%	93.964%	18.7

Table 13: GACO, PSO, ABC and Random Search experimental results

It is important to understand that the final metric of this table indicates how fast the highest accuracy of each experiment, not the highest of all the experiments, was found, on average.

By examining the presented results, one comes to the conclusion that Particle Swarm Optimization outperforms every other algorithm, including the Random Search. The accuracies (both the highest and the mean ones) that are met in the experiments of this algorithm are evidently higher than the ones of the rest. In fact, we can see that all 3 algorithms manage to outperform the Random Search. Perhaps there is a bit of hesitancy to say that for the Artificial Bee Colony algorithm, in which we see that although it has managed to construct an architecture that evaluates to an accuracy higher than the best of GACO and Random Search, it has quite a low calculated mean. This is properly depicted in the plots we are about to present, in which we see that there are signs of repetitive instability in the Artificial Bee Colony's performance. It is also interesting to notice that the Artificial Bee Colony algorithm manages to find the best architecture per experiment the fastest out of all of its competitors, which is the only comparison metric in which the Particle Swarm Optimization does not outperform the rest. However, we need to be objective here and understand that this final metric is quite a vague one, from which we cannot infer way too much information.

Up next, we present a series of plots to understand the performance of our algorithms even better. We show two types of comparisons:

- We compare the highest accuracy found per generation
- We compare the highest accuracy that has been found up until that generation

For both categories, we include four different plots. One to compare the three implemented algorithms (GACO, PSO and ABC) and one to compare each algorithm with the Random Search. We need to reiterate that we have conducted 10 experiments for each method. To this end, we present each experiment with a different line.

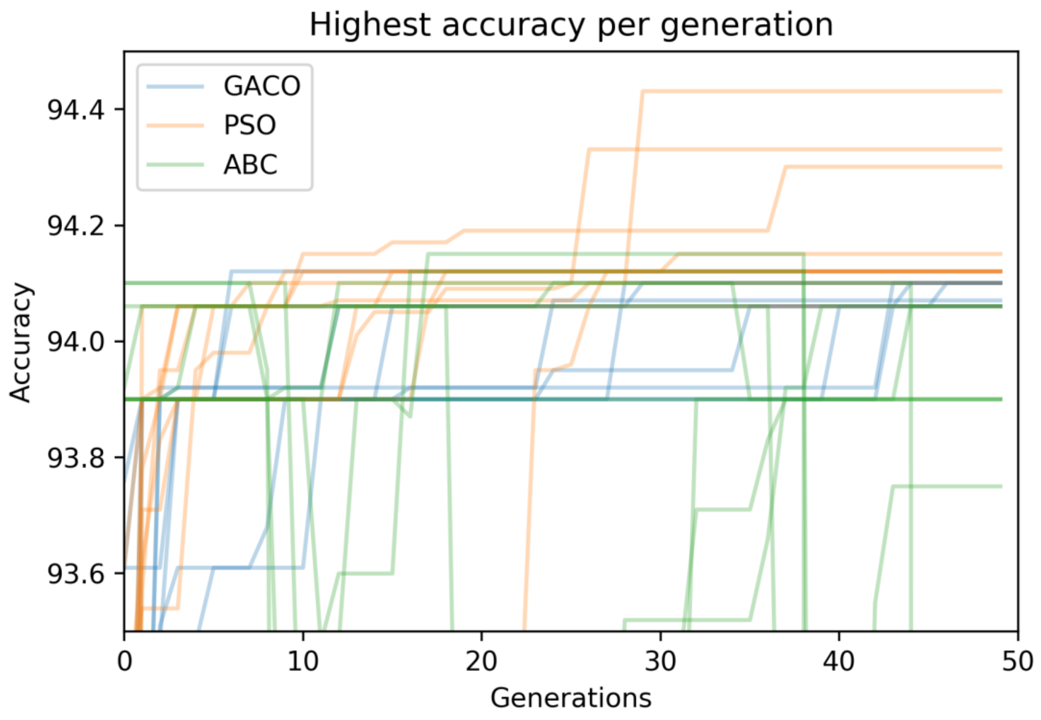


Figure 26: Highest accuracy per generation (GACO, PSO & ABC)

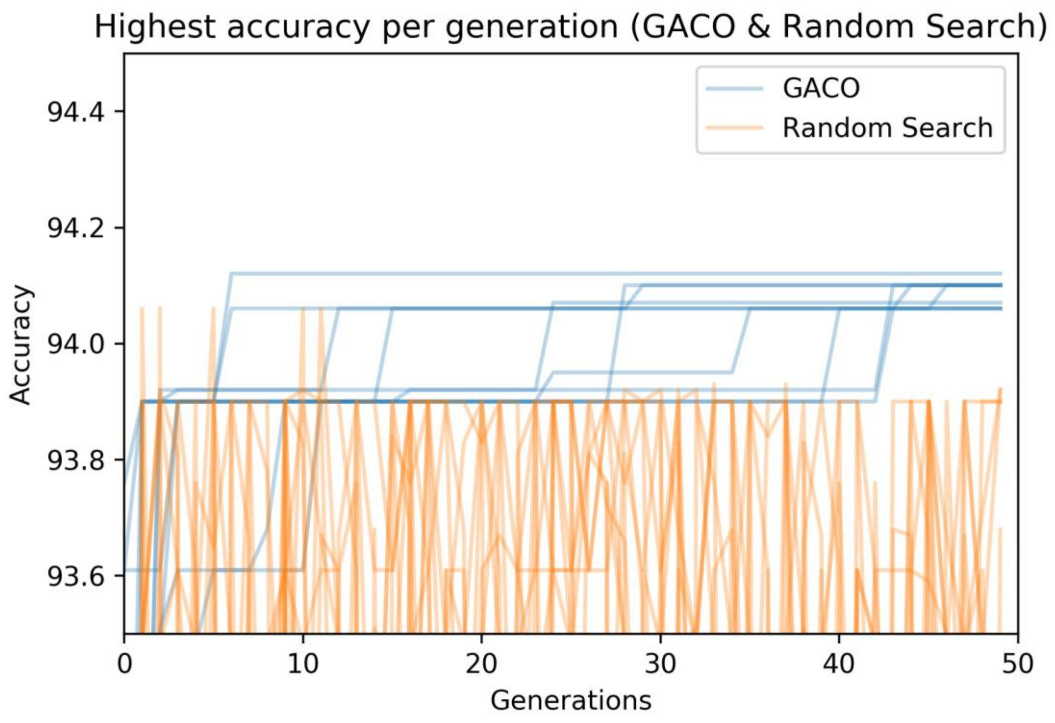


Figure 27: Highest accuracy per generation (GACO & Random Search)

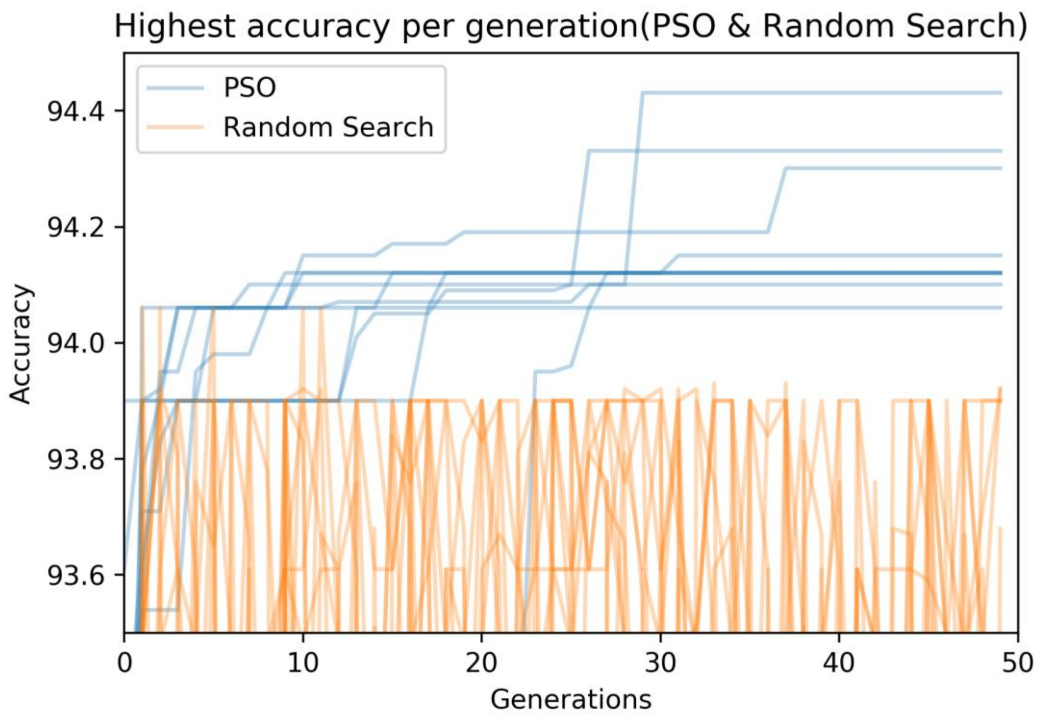


Figure 28: Highest accuracy per generation (PSO & Random Search)

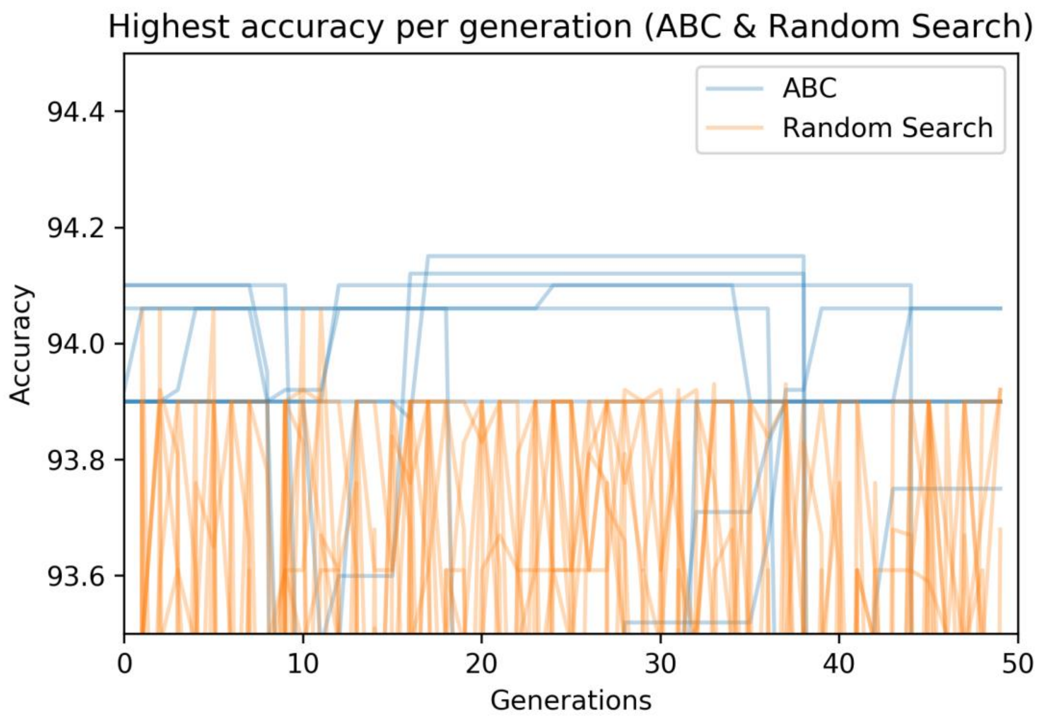


Figure 29: Highest accuracy per generation (ABC & Random Search)

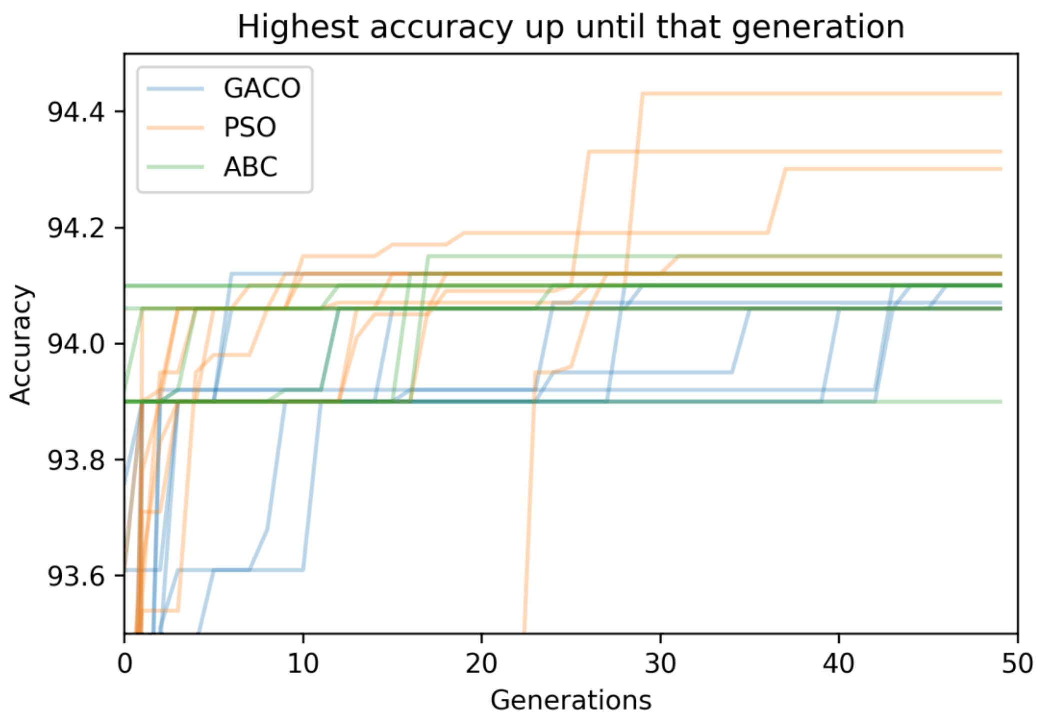


Figure 30: Highest accuracy up until that generation (GACO, PSO & ABC)

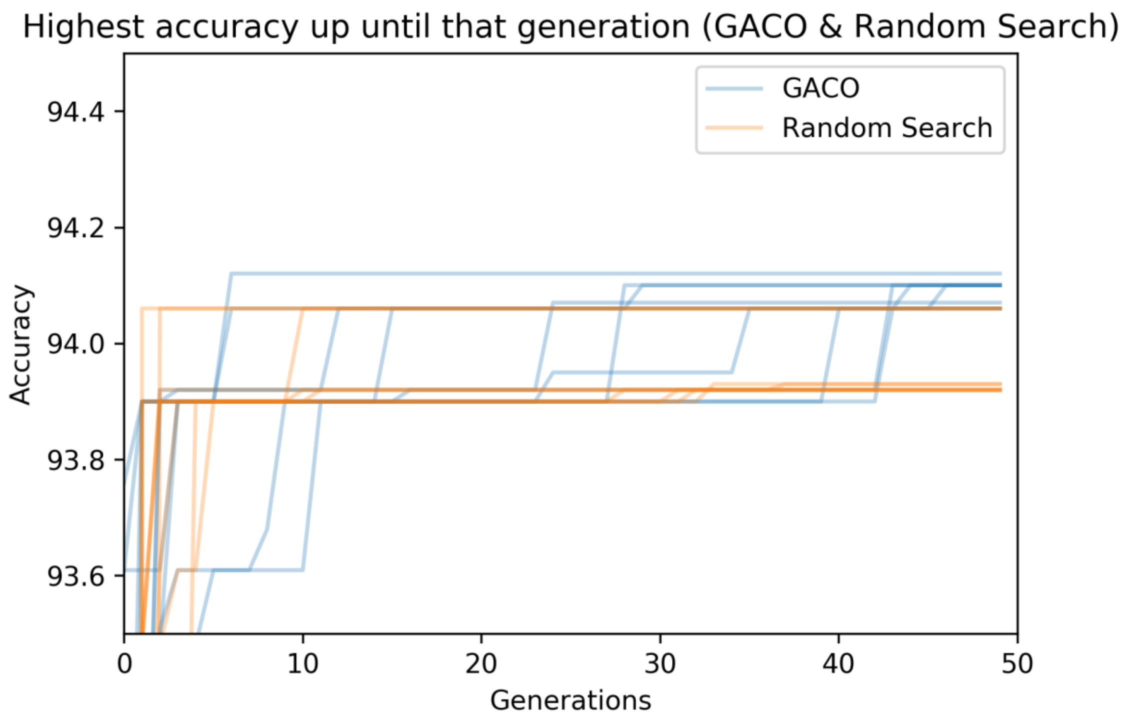


Figure 31: Highest accuracy up until that generation (GACO & Random Search)

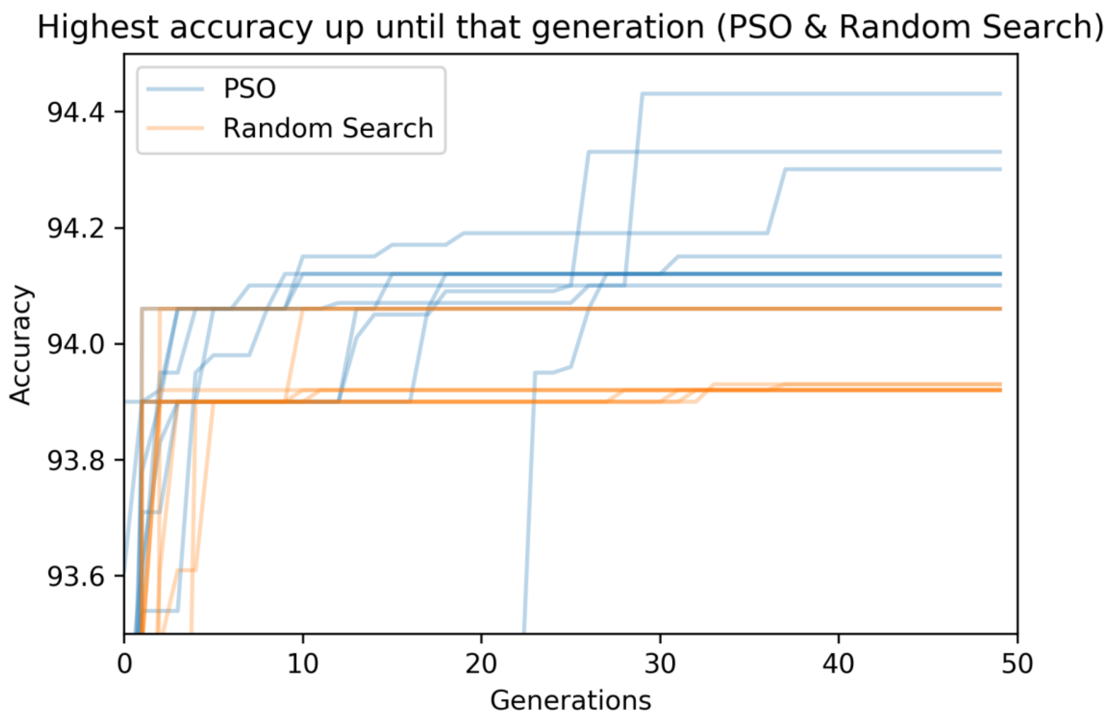


Figure 32: Highest accuracy up until that generation (PSO & Random Search)

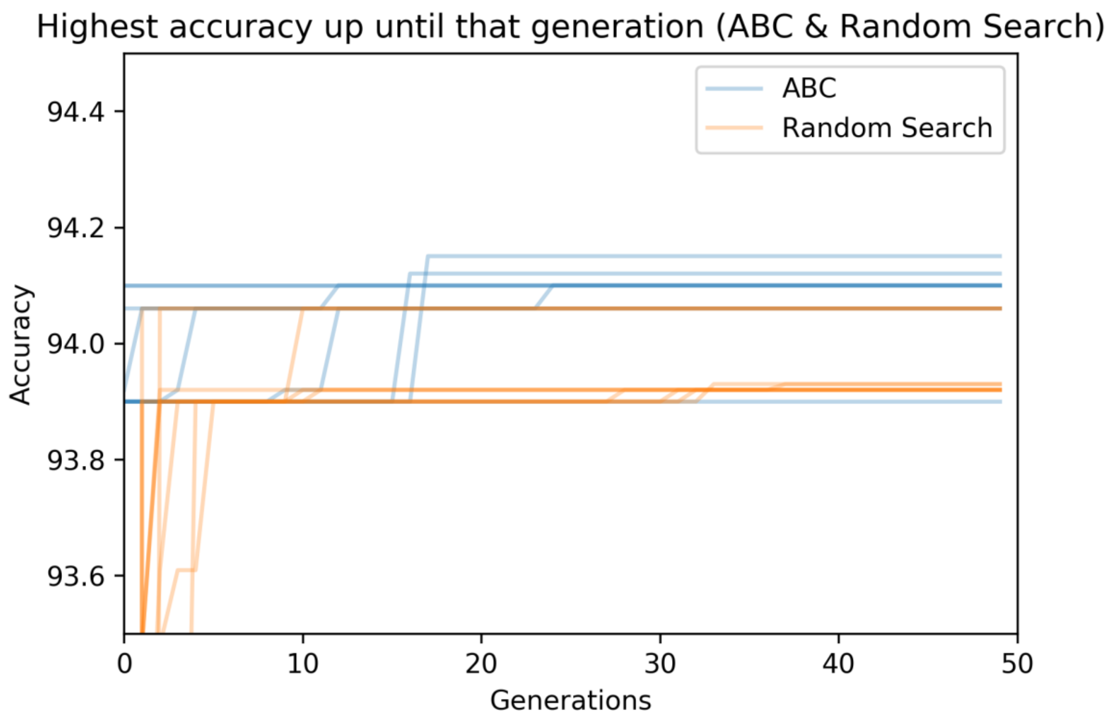


Figure 33: Highest accuracy up until that generation (ABC & Random Search)

As we can see from the 8 plots we have just presented, the conclusions we have already drawn, based on that one table, are correct. Looking at the first four plots we have included, which are related to the highest accuracy that has been found per generation, we clearly see that PSO outperforms the other two algorithms. As it was expected due to its low mean, we can also see that ABC has many of these repetitive instability signs. Generally speaking, all three algorithms seem to be performing better than the Random Search. Arguably, this could be deemed questionable in the case of ABC, since despite the fact that it manages to find some architectures which evaluate to higher accuracies than the Random Search, it does seem to be generating way too many architectures that are of really low performance. If we were to present some graphs that would demonstrate a wider range in the "y" axis, we would be able to see the wide variety of low accuracy architectures that are generated from the ABC algorithm. The major reason we have chosen to present a 93.5 to 94.5 range in our "y" axis is to make all the possible details that differentiate the algorithms distinguishable. The final four plots, which are related to the best accuracy of a generation that has been found up until that generation, lead to the same conclusions regarding the comparison between the three algorithms. The only difference might be that in the comparison between ABC and the

Random Search, ABC is not presented as such an unstable algorithm since the visible instabilities have been eradicated.

9.3 Conclusion

To sum up, we have conducted experiments to compare the three algorithms we have chosen from PyGMO. To compound this, we also compared the performance of the algorithms with the Random Search. It is quite positive that even for such a small amount of examined models, considering that we have a population size of 30, for 50 generations, the chosen algorithms seem to be outperforming the Random Search. This can be said with a spice of hesitancy for the Artificial Bee Colony since despite the fact that it manages to construct architectures that evaluate to higher accuracies than the ones of the Random Search, it seems to be quite unstable, repetitively producing architectures which evaluate to quite low accuracies. Particle Swarm Optimization seems to be the algorithm that performs the best because not only do we see the highest architectures produced by it, but it is also the most consistent one in its performance through the entirety of the generations of all the experiments. It is quite interesting to notice that the highest accuracy from the PyGMO experiments, which was met in the PSO experiments, is really close to the best architecture that has been met in the evolutionary algorithm experiments. We do need to remember that the highest accuracy we have found in all of our experiments was found in the Deep Q-Networks ones and was 94.72%. We have a summarized comparison between all three methods we used in our implementations in one of the final sections.

10 Conclusion

10.1 Conclusions

In this thesis, we implemented and evaluated various approaches to tackle the problem of Neural Architecture Search. Prior to the implementations that we presented in this study, we had provided an introduction to all the key elements which are required to understand the implementations, with a lot of material for further study. In this section, we summarize our findings and then suggest some extra work that can be done in the future in order to improve certain aspects of it. It is crucial to remind here that, for our experiments, we have used a specific benchmark [15] in order to avoid the arduous and time-consuming process of training the networks. In addition to this, we have also approached the problems enforcing the limitations described in NASNet [87].

At first, we began with the introduction of a Reinforcement Learning implementation. We used the algorithm of Deep Q-Learning to tackle the problem. In the implementations we presented, there are a few differentiators. A minor one is the type of the controller (dense-layer controller vs RNN controller). However, the major differentiator, introduced in the 3rd and final Deep Q-Learning implementation, is the wider search space. In this implementation, we widened the search space, aiming to find better neural architectures than the sequential ones we had found in the previous two implementations of Deep Q-Learning. Through these implementations, we discovered the strengths, but also the inadequacies of the Deep Q-Learning algorithm as the simplicity of the algorithm did not allow it to converge. However, it is worth mentioning that in the experiment of the Deep Q-Learning algorithm in a wider search space, we found the highest-accuracy architecture from all the experiments we conducted, which evaluates to 94.72%.

Moving on, we continued our implementations with an evolutionary algorithm approach [13]. This one shifted our focus from Reinforcement Learning algorithms to metaheuristic ones. Despite the fact that it is a relatively simple algorithm, it managed to find a relatively good architecture in just 1500 evolve cycles, which evaluated to approximately 94.43%.

Finally, we utilized a Python framework which includes a set of pre-implemented optimization algorithms, PyGMO [14]. We selected three algorithms, Extended Ant Colony Optimization (GACO), Particle Swarm Optimization (PSO) and Artificial Bee

Colony (ABC). In order to use these implemented algorithms of PyGMO, we formulated the problem of Neural Architecture Search as an optimization problem with respect to certain constraints. The results of this implementation allowed us to provide a detailed comparison between the three algorithms. Our findings showed that, for 1500 examined models, PSO outperformed the other two algorithms. GACO and ABC were relatively close, with some concerns being raised for ABC's performance due to repetitive instabilities. The best architecture found from the prevailing algorithm, which is PSO, evaluates to approximately 94.43%, which is very similar to the outcome of the evolutionary algorithm. What is interesting here is that all three algorithms managed to outperform the Random Search, as we presented in our results in section 9.2.5 .

In summary, the architecture with the highest accuracy was found in the wide-search-space Deep Q-Learning implementation, but with the agent being unable to converge. It is also worth mentioning that the execution of this implementation was much more time-consuming than the metaheuristic algorithm ones. When it comes to the metaheuristics, all the algorithms outperformed the Random Search, with the evolutionary algorithm and PSO being the ones generating the architectures of the highest accuracy out of all 4 metaheuristic algorithms. ACO and ABC had similar performances when it came down to the best architectures found, but ABC showed repetitive signs of instability over the evolution cycles.

10.2 Future Work

In this thesis, we have included implementations for Neural Architecture Search on two fronts: Reinforcement Learning and Metaheuristic Algorithms. There are certainly things that could be done to furtherly enhance the performance of the implementations we have conducted, and this is mostly said for the Reinforcement Learning one.

In particular, when it comes to our Deep Q-Learning implementations, through the wider search space enhancement, we came to the conclusion that the algorithm is relatively weak and its process is kind of simplistic to tackle a problem of such a wide search space. As we have also mentioned in the concluding section of the Deep Q-Learning implementation, our focus was not to improve that particular implementation, but to turn to other approaches, like heuristic algorithms. However, if we were to improve the Deep Q-Learning implementation, our next step would be to introduce the interesting mechanism of attention [106]. Using the functionality of attention in our Deep Q-Network controller, we would be likely to achieve better results in a wider search space due to the proper correlation that would be formed among the layers, allowing the controller to have more leeway to learn.

Despite the improvement the mechanism of attention would ensure, we must not forget that the Deep Q-Learning algorithm is a relatively weak Reinforcement Learning method. Thus, the utilization of more powerful Reinforcement Learning methods, such as proximal policy [107] ones, would be likely to show very noticeable improvements. A proximal policy algorithm would be a very important asset in our arsenal since it has been used for very demanding and difficult Reinforcement Learning tasks, some of which we mentioned in the introduction, such as learning how to defeat the best human players in the world in certain video games [45].

A different evaluation of all of the implemented algorithms would also give us a more objective view of their performance. Concretely, for our experiments, we have been using a selected benchmark [15] from which we query the performance of the constructed networks. The limitations of the NASBench-101 combined with the limitations from the NASNet-like search space that we are using [87], resulting in our networks being confined to some certain actions and architectures. Conducting similar experiments using different datasets/benchmarks and different search spaces would provide us with more concrete data about each algorithm’s performance, alongside the variety of selected parameters for each one of them.

References

[1] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, no. 6, pp. 84–90, Jun. 2017, doi: 10.1145/3065386.

[3] R. Collobert and J. Weston, “A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning,” Accessed: Dec. 27, 2020. [Online]. Available: http://wordnet.princeton.edu.

[4] Y. Goldberg, “Neural Network Methods for Natural Language Processing,” *Synth. Lect. Hum. Lang. Technol.*, vol. 10, no. 1, pp. 1–311, Apr. 2017, doi: 10.2200/S00762ED1V01Y201703HLT037.

[5] Y. Assael, T. Sommerschield, and J. Prag, “Restoring ancient text using deep learning: A case study on Greek epigraphy,” *EMNLP-IJCNLP 2019 - 2019 Conf. Empir. Methods Nat. Lang. Process. 9th Int. Jt. Conf. Nat. Lang. Process. Proc. Conf.*, no. Figure 1, pp. 6368–6375, 2020, doi: 10.18653/v1/d19-1668.

[6] L. Deng, G. Hinton, and B. Kingsbury, “New types of deep neural network learning for speech recognition and related applications: An overview,” in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, Oct. 2013, pp. 8599–8603, doi: 10.1109/ICASSP.2013.6639344.

[7] W. Xiong *et al.*, “Achieving Human Parity in Conversational Speech Recognition,” *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 25, no. 12, pp. 2410–2423, Oct. 2016, Accessed: Dec. 27, 2020. [Online]. Available: http://arxiv.org/abs/1610.05256.

[8] C. Badue *et al.*, “Self-driving cars: A survey,” *Expert Systems with Applications*, vol. 165. Elsevier Ltd, p. 113816, Mar. 01, 2021, doi: 10.1016/j.eswa.2020.113816.

[9] M. Bojarski *et al.*, “End to End Learning for Self-Driving Cars,” Apr. 2016, Accessed: Dec. 27, 2020. [Online]. Available: http://arxiv.org/abs/1604.07316.

[10] “The Fourth Industrial Revolution | Foreign Affairs.” https://www.foreignaffairs.com/articles/2015-12-12/fourth-industrial-revolution

(accessed Dec. 27, 2020).

[11] G. Kyriakides and K. Margaritis, “An Introduction to Neural Architecture Search for Convolutional Networks,” pp. 1–17, 2020, [Online]. Available: http://arxiv.org/abs/2005.11074.

[12] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” *30th AAAI Conf. Artif. Intell. AAAI 2016*, pp. 2094–2100, Sep. 2015, Accessed: Dec. 26, 2020. [Online]. Available: http://arxiv.org/abs/1509.06461.

[13] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, “Regularized Evolution for Image Classifier Architecture Search,” *Proc. AAAI Conf. Artif. Intell.*, vol. 33, pp. 4780–4789, 2019, doi: 10.1609/aaai.v33i01.33014780.

[14] F. Biscani and D. Izzo, “A parallel global multiobjective framework for optimization: pagmo,” *J. Open Source Softw.*, vol. 5, no. 53, p. 2338, Sep. 2020, doi: 10.21105/joss.02338.

[15] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, “NAS-BENCH-101: Towards reproducible neural architecture search,” *36th Int. Conf. Mach. Learn. ICML 2019*, vol. 2019-June, pp. 12334–12348, 2019.

[16] G. Kyriakides and K. Margaritis, “NORD: A python framework for Neural Architecture Search,” *Softw. Impacts*, vol. 6, p. 100042, Nov. 2020, doi: 10.1016/j.simpa.2020.100042.

[17] T. Bäck and H.-P. Schwefel, “An Overview of Evolutionary Algorithms for Parameter Optimization,” *Evol. Comput.*, vol. 1, no. 1, pp. 1–23, Mar. 1993, doi: 10.1162/evco.1993.1.1.1.

[18] M. Schlüter, J. A. Egea, and J. R. Banga, “Extended ant colony optimization for non-convex mixed integer nonlinear programming,” *Comput. Oper. Res.*, vol. 36, no. 7, pp. 2217–2229, Jul. 2009, doi: 10.1016/j.cor.2008.08.015.

[19] M. Dorigo, V. Maniezzo, and A. Colorni, “Ant system: Optimization by a colony of cooperating agents,” *IEEE Trans. Syst. Man, Cybern. Part B Cybern.*, vol. 26, no. 1, pp. 29–41, 1996, doi: 10.1109/3477.484436.

[20] D. Wang, D. Tan, and L. Liu, “Particle swarm optimization algorithm: an overview,” *Soft Comput.*, vol. 22, no. 2, pp. 387–408, Jan. 2018, doi: 10.1007/s00500-016-2474-6.

[21] G. Venter and J. Sobieszczanski-Sobieski, “Particle swarm optimization,” *AIAA J.*,

vol. 41, no. 8, pp. 1583–1589, May 2003, doi: 10.2514/2.2111.

[22] D. Karaboga and B. Akay, “A comparative study of Artificial Bee Colony algorithm,” *Appl. Math. Comput.*, vol. 214, no. 1, pp. 108–132, Aug. 2009, doi: 10.1016/j.amc.2009.03.090.

[23] D. Karaboga and C. Ozturk, “Fuzzy clustering with artificial bee colony algorithm,” *Sci. Res. Essays*, vol. 5, no. 14, pp. 1899–1902, 2010, doi: 10.4249/scholarpedia.6915.

[24] K. Hinkelmann, “Neural Networks.” University of Applied Sciences Northwestern Switzerland.

[25] P. Sibi, S. A. Jones, and P. Siddarth, “ANALYSIS OF DIFFERENT ACTIVATION FUNCTIONS USING BACK PROPAGATION NEURAL NETWORKS,” *J. Theor. Appl. Inf. Technol.*, vol. 31, no. 3, 2013, Accessed: Dec. 24, 2020. [Online]. Available: www.jatit.org.

[26] B. Xu, N. Wang, H. Kong, T. Chen, and M. Li, “Empirical Evaluation of Rectified Activations in Convolution Network.” Accessed: Dec. 24, 2020. [Online]. Available: https://github.com/.

[27] K. Hara, D. Saito, and H. Shouno, “Analysis of function of rectified linear unit used in deep learning,” *Proc. Int. Jt. Conf. Neural Networks*, vol. 2015-Septe, 2015, doi: 10.1109/IJCNN.2015.7280578.

[28] J. Feng and S. Lu, “Performance Analysis of Various Activation Functions in Artificial Neural Networks,” *J. Phys. Conf. Ser.*, vol. 1237, no. 2, pp. 111–122, 2019, doi: 10.1088/1742-6596/1237/2/022030.

[29] B. Zamanlooy and M. Mirhassani, “Efficient VLSI implementation of neural networks with hyperbolic tangent activation function,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 22, no. 1, pp. 39–48, 2014, doi: 10.1109/TVLSI.2012.2232321.

[30] Y. Bengio and Y. LeCun, “Convolutional Networks for Images, Speech, and Time-Series Oracle Performance for Visual Captioning View project MoDeep View project,” 1997. Accessed: Dec. 27, 2020. [Online]. Available: https://www.researchgate.net/publication/2453996.

[31] Y. LeCun *et al.*, “Backpropagation Applied to Handwritten Zip Code Recognition,” *Neural Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989, doi: 10.1162/neco.1989.1.4.541.

[32] M. Hashemi, “Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation,” *J. Big Data*, vol. 6, no. 1, pp. 1–13, Dec. 2019, doi: 10.1186/s40537-019-0263-7.

[33] J. Nagi *et al.*, “Max-pooling convolutional neural networks for vision-based hand gesture recognition,” in *2011 IEEE International Conference on Signal and Image Processing Applications, ICSIPA 2011*, 2011, pp. 342–347, doi: 10.1109/ICSIPA.2011.6144164.

[34] D. Yu, H. Wang, P. Chen, and Z. Wei, “Mixed pooling for convolutional neural networks,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Oct. 2014, vol. 8818, pp. 364–375, doi: 10.1007/978-3-319-11740-9_34.

[35] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735.

[36] B. Bakker, “Reinforcement Learning with long short-term memory.”

[37] “Long short-term memory - Wikipedia.” https://en.wikipedia.org/wiki/Long_short-term_memory (accessed Dec. 28, 2020).

[38] A. Sherstinsky, “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network,” *Phys. D Nonlinear Phenom.*, vol. 404, p. 132306, Mar. 2020, doi: 10.1016/j.physd.2019.132306.

[39] T. Hastie, R. Tibshirani, and J. Friedman, “Overview of Supervised Learning,” Springer, New York, NY, 2009, pp. 9–41.

[40] Z. Ghahramani, “Unsupervised learning,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3176, pp. 72–112, 2004, doi: 10.1007/978-3-540-28650-9_5.

[41] X. (Jerry) Zhu, “Semi-Supervised Learning Literature Survey,” 2005, Accessed: Dec. 28, 2020. [Online]. Available: https://minds.wisconsin.edu/handle/1793/60444.

[42] X. Goldberg, “Introduction to semi-supervised learning,” *Synth. Lect. Artif. Intell. Mach. Learn.*, vol. 6, pp. 1–116, Jun. 2009, doi: 10.2200/S00196ED1V01Y200906AIM006.

[43] D. Silver *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv*. arXiv, Dec. 05, 2017, Accessed: Dec. 28, 2020. [Online]. Available: https://arxiv.org/abs/1712.01815v1.

[44] D. Silver *et al.*, “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017, doi: 10.1038/nature24270.

[45] OpenAI *et al.*, “Dota 2 with Large Scale Deep Reinforcement Learning,” *arXiv*, Dec. 2019, Accessed: Dec. 28, 2020. [Online]. Available: http://arxiv.org/abs/1912.06680.

[46] V. Mnih *et al.*, “Playing Atari with Deep Reinforcement Learning,” Dec. 2013, Accessed: Dec. 28, 2020. [Online]. Available: http://arxiv.org/abs/1312.5602.

[47] J. Li, W. Monroe, A. Ritter, M. Galley, J. Gao, and D. Jurafsky, “Deep Reinforcement Learning for Dialogue Generation,” *EMNLP 2016 - Conf. Empir. Methods Nat. Lang. Process. Proc.*, pp. 1192–1202, Jun. 2016, Accessed: Dec. 28, 2020. [Online]. Available: http://arxiv.org/abs/1606.01541.

[48] R. S. Sutton and A. G. Barto, “Reinforcement Learning: An Introduction,” 1998.

[49] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, May 1996, doi: 10.1613/jair.301.

[50] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, “An Introduction to Deep Reinforcement Learning,” *Found. Trends Mach. Learn.*, vol. 11, no. 3–4, pp. 219–354, Nov. 2018, doi: 10.1561/2200000071.

[51] Y. Li, “Deep Reinforcement Learning: An Overview,” Jan. 2017, Accessed: Dec. 28, 2020. [Online]. Available: http://arxiv.org/abs/1701.07274.

[52] J. Y. Audibert, R. Munos, and C. Szepesvári, “Exploration-exploitation tradeoff using variance estimates in multi-armed bandits,” *Theor. Comput. Sci.*, vol. 410, no. 19, pp. 1876–1902, Apr. 2009, doi: 10.1016/j.tcs.2009.01.016.

[53] M. Wunder, M. Littman, and M. Babes, “Classes of Multiagent Q-learning Dynamics with-greedy Exploration.”

[54] “Deep Q-Learning | An Introduction To Deep Reinforcement Learning.” https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ (accessed Dec. 28, 2020).

[55] S. Adam, L. Buşoniu, and R. Babuška, “Experience replay for real-time reinforcement learning control,” *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.*, vol. 42, no. 2, pp. 201–212, Mar. 2012, doi: 10.1109/TSMCC.2011.2106494.

[56] “Gym, FrozenLake8x8.” https://gym.openai.com/envs/FrozenLake8x8-v0/ (accessed Dec. 28, 2020).

[57] G. Brockman *et al.*, “OpenAI Gym,” Jun. 2016, Accessed: Dec. 28, 2020.

[Online]. Available: http://arxiv.org/abs/1606.01540.

[58] "Heuristic algorithms - optimization." https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms (accessed Dec. 29, 2020).

[59] N. Kokash, "An introduction to heuristic algorithms."

[60] R. C. Ebenhart, Y. Shi, and J. Kennedy, *Swarm Intelligence*. Elsevier, 2001.

[61] G. Jones, "Genetic and Evolutionary Algorithms."

[62] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: JAddison-Wesley Longman Publishing Co., Inc., 1989.

[63] M. Mitchell, *An introduction to genetic algorithms*. MIT Press, 1998.

[64] F. Rothlauf, "Representations for Genetic and Evolutionary Algorithms," in *Representations for Genetic and Evolutionary Algorithms,* Springer Berlin Heidelberg, 2006, pp. 9–32.

[65] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, 1996.

[66] F. J. Lobo, C. F. Lima, and Z. Michalewicz, *Parameter setting in evolutionary algorithms (Vol 54)*. Springer Science & Business Media, 2007.

[67] "Evolutionary algorithm - Wikipedia." https://en.wikipedia.org/wiki/Evolutionary_algorithm (accessed Dec. 29, 2020).

[68] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Comput. Intell. Mag.*, vol. 1, no. 4, pp. 28–39, Nov. 2006, doi: 10.1109/MCI.2006.329691.

[69] M. Dorigo and C. Blum, "Ant colony optimization theory: A survey," *Theor. Comput. Sci.*, vol. 344, pp. 243–278, 2005, doi: 10.1016/j.tcs.2005.05.020.

[70] C. Blum, "Ant colony optimization: Introduction and recent trends," *Physics of Life Reviews,* vol. 2, no. 4. Elsevier, pp. 353–373, Dec. 01, 2005, doi: 10.1016/j.plrev.2005.10.001.

[71] "Extended Ant Colony Optimization (gaco) — pagmo 2.16.1 documentation." https://esa.github.io/pagmo2/docs/cpp/algorithms/gaco.html#_CPPv4N5pagmo4gacoE (accessed Dec. 30, 2020).

[72] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks,* vol. 4, pp. 1942–1948, doi: 10.1109/ICNN.1995.488968.

[73] Y. Shi and R. C. Eberhart, “Empirical study of particle swarm optimization,” in *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999*, 1999, vol. 3, pp. 1945–1950, doi: 10.1109/CEC.1999.785511.

[74] R. Eberhart and J. Kennedy, “New optimizer using particle swarm theory,” in *Proceedings of the International Symposium on Micro Machine and Human Science*, 1995, pp. 39–43, doi: 10.1109/mhs.1995.494215.

[75] R. C. Eberhart and Y. Shi, “Particle swarm optimization: Developments, applications and resources,” in *Proceedings of the IEEE Conference on Evolutionary Computation, ICEC*, 2001, vol. 1, pp. 81–86, doi: 10.1109/cec.2001.934374.

[76] R. Poli, “Analysis of the Publications on the Applications of Particle Swarm Optimisation,” *J. Artif. Evol. Appl.*, vol. 685175, 2008, doi: 10.1155/2008/685175.

[77] “Particle Swarm Optimization (PSO) — pagmo 2.16.1 documentation.” https://esa.github.io/pagmo2/docs/cpp/algorithms/pso.html#_CPPv4NK5pagmo3pso7get_logEv (accessed Dec. 30, 2020).

[78] D. Karaboga, “AN IDEA BASED ON HONEY BEE SWARM FOR NUMERICAL OPTIMIZATION,” 2005.

[79] D. Karaboga and B. Basturk, “A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm,” *J. Glob. Optim.*, vol. 39, no. 3, pp. 459–471, Nov. 2007, doi: 10.1007/s10898-007-9149-x.

[80] S.-H. Liu, M. Mernik, D. Karaboga, and M. M. Matejčrepinšek, “On clarifying misconceptions when comparing variants of the Artificial Bee Colony Algorithm by offering a new implementation,” doi: 10.1016/j.ins.2014.08.040.

[81] D. Karaboga and B. Basturk, “On the performance of artificial bee colony (ABC) algorithm,” *Appl. Soft Comput. J.*, vol. 8, no. 1, pp. 687–697, Jan. 2008, doi: 10.1016/j.asoc.2007.05.007.

[82] D. Karaboga, B. Gorkemli, C. Ozturk, and N. Karaboga, “A comprehensive survey: Artificial bee colony (ABC) algorithm and applications,” *Artif. Intell. Rev.*, vol. 42, no. 1, pp. 21–57, Mar. 2014, doi: 10.1007/s10462-012-9328-0.

[83] “Artificial Bee Colony — pagmo 2.16.1 documentation.” https://esa.github.io/pagmo2/docs/cpp/algorithms/bee_colony.html#_CPPv4N5pagmo10bee_colonyE (accessed Dec. 30, 2020).

[84] T. Elsken, J. H. Metzen, and F. Hutter, “Neural Architecture Search: A Survey,”

2019. Accessed: Dec. 24, 2020. [Online]. Available: http://jmlr.org/papers/v20/18-598.html.

[85] “Literature on Neural Architecture Search.” https://www.automl.org/automl/literature-on-neural-architecture-search/ (accessed Jan. 23, 2021).

[86] T. Elsken, J. H. Metzen, and F. Hutter, “Neural architecture search: A survey,” *J. Mach. Learn. Res.*, vol. 20, pp. 1–21, 2019.

[87] R. Miikkulainen *et al.*, “Evolving deep neural networks,” *Artif. Intell. Age Neural Networks Brain Comput.*, pp. 293–312, 2018, doi: 10.1016/B978-0-12-815480-9.00015-3.

[88] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning Transferable Architectures for Scalable Image Recognition,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 8697–8710, 2018, doi: 10.1109/CVPR.2018.00907.

[89] B. Zoph and Q. V. Le, “Neural Architecture Search with Reinforcement Learning,” *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, Nov. 2016, Accessed: Jan. 01, 2021. [Online]. Available: http://arxiv.org/abs/1611.01578.

[90] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Dec. 2016, vol. 2016-December, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[91] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks,” *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 2261–2269, Aug. 2016, Accessed: Jan. 01, 2021. [Online]. Available: http://arxiv.org/abs/1608.06993.

[92] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, “Efficient Neural Architecture Search via parameter Sharing,” *35th Int. Conf. Mach. Learn. ICML 2018*, vol. 9, pp. 6522–6531, 2018.

[93] C. Liu *et al.*, “Progressive Neural Architecture Search,” 2018. Accessed: Jan. 01, 2021. [Online]. Available: http://github.com/tensorflow/.

[94] H. Liu, K. Simonyan, and Y. Yang, “DARTS: Differentiable architecture search,” *7th Int. Conf. Learn. Represent. ICLR 2019*, pp. 1–13, 2019.

[95] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evol. Comput.*, vol. 10, no. 2, pp. 99–127, Mar. 2002, doi: 10.1162/106365602320169811.

[96] D. El, M. Pelt, and J. A. Sethian, “A mixed-scale dense convolutional neural network for image analysis,” vol. 115, no. 2, pp. 254–259, 2018, doi: 10.1073/pnas.1715832114.

[97] F. Chollet, “Keras.” GitHub, 2015, [Online]. Available: https://github.com/keras-team/keras.

[98] B. Zamanlooy and M. Mirhassani, “Efficient VLSI implementation of neural networks with hyperbolic tangent activation function,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 22, no. 1, pp. 39–48, Jan. 2014, doi: 10.1109/TVLSI.2012.2232321.

[99] Ł. Kaiser, G. Brain, A. N. Gomez, and F. Chollet, “Depthwise Separable Convolutions for Neural Machine Translation.” Accessed: Dec. 24, 2020. [Online]. Available: https://github.com/tensorflow/tensor2tensor.

[100] J. Jin, A. Dundar, and E. Culurciello, “Flattened Convolutional Neural Networks for Feedforward Acceleration,” *3rd Int. Conf. Learn. Represent. ICLR 2015 - Work. Track Proc.*, Dec. 2014, Accessed: Dec. 25, 2020. [Online]. Available: http://arxiv.org/abs/1412.5474.

[101] B. Zoph and Q. V Le Google Brain, “NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING.”

[102] J. Chung, C. Gulcehre, and K. Cho, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.”

[103] S. Bock, J. Goppold, and M. Weiß, “An improvement of the convergence proof of the ADAM-Optimizer.”

[104] Z. Zhang, “Improved Adam Optimizer for Deep Neural Networks,” Jan. 2019, doi: 10.1109/IWQoS.2018.8624183.

[105] S. Vani and T. V. M. Rao, “An experimental approach towards the performance assessment of various optimizers on convolutional neural network,” in *Proceedings of the International Conference on Trends in Electronics and Informatics, ICOEI 2019*, Apr. 2019, pp. 331–336, doi: 10.1109/ICOEI.2019.8862686.

[106] S. ichi Amari, “Backpropagation and stochastic gradient descent method,”

Neurocomputing, vol. 5, no. 4–5, pp. 185–196, Jun. 1993, doi: 10.1016/0925-2312(93)90006-O.

[107] A. Vaswani *et al.*, “Attention is all you need,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-Decem, no. Nips, pp. 5999–6009, 2017.

[108] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” pp. 1–12, 2017, [Online]. Available: http://arxiv.org/abs/1707.06347.