Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Improving Neural Architecture Search with Reinforcement Learning

Author:
Maurizio Zen

Supervisor:
Dr. Jonathan Passerat-Palmbach

Second Marker:
Dr. Ben Glocker

June 19, 2019

Abstract

Neural Networks are powerful models that provide great performance on numerous image classification tasks. While some of the best architectures are still manually designed, in recent years researchers have focused on developing methods to automatically search for network architectures. We investigate the topic of neural architecture search with reinforcement learning, a method in which a recurrent network, the controller, learns to sample better convolutional architectures. We highlight different methods that influence the sampling decisions of the controller, and present a novel approach that improves the search strategy employed by the controller.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Jonathan Passerat-Palmbach, for his constant support, guidance and invaluable feedback. I would also like to thank Dr. Amir Alansary for his great suggestions and help.

I send my gratitude to my former Mathematics teachers, Mr. Lucian Ruba and Mr. Alexandru Blaga, for inspiring me academically and personally.

Finally, I would not be the person I am today without my family and closest friends. I owe them everything and hold them dear.

Contents

Chapter 1

Introduction

1.1 Problem Description

With the increasing popularity of machine learning models as solutions to complicated problems, engineers face the challenge of having to optimise their models. The problems addressed are usually complex in nature and involve high-dimensional inputs, making it hard for humans to visualise the data and come up with a solution. Since its earliest days as a discipline, machine learning has made use of optimisation formulations and algorithms. Similarly, machine learning has contributed to optimisation by driving the development of new optimisation approaches that address the challenges presented by machine learning applications. Due to their theoretical properties and wide area of applicability, optimisation approaches have enjoyed prominence in machine learning [1]. After a machine learning model is agreed upon, engineers need to take model-specific decisions. As an analogy, consider the task of building a house. One needs to know the surface allocated for the building, the number of rooms, the number of people planning to move in etc. Similarly, after choosing a machine learning model, engineers need to configure multiple initial variables which will have a direct impact on the model. These variables are called hyper-parameters, they are specific to each model and set before training begins, as opposed to "simple" parameters which are learned during training time. Hyper-parameters are essential because they control the behaviour of the training algorithm and have a significant influence on the performance of the model being trained. Some of the most common hyper-parameters include: learning rate, number of epochs, hidden layers, activation functions. Taking neural networks as an example, setting a high value for the learning rate leads to overfitting (the model has high *variance*), whereas a low value for the learning rate leads to underfitting which makes the model inadequate to recognise patterns (high *bias*).

We tackle the problem of *Hyper-parameter Optimisation/Tuning*, which is the process of searching through the parameter space to find an optimal set of parameters that minimises a given loss function. This project is an investigation of automated hyper-parameter optimisation methods, with a particular emphasis on Reinforcement Learning (RL) methods applied to Convolutional Neural Networks (CNNs) (Section 2.4) for which little literature is available. RL is a particular learning paradigm that trains intelligent agents to solve a specific task using a set of actions. Such approaches have shown to surpass human level on different platforms,

such as in Atari [2] and Go [3] games. This paradigm has been recently applied to the problem of hyper-parameter optimisation [4, 5]. The RL agents are capable of learning new architectures that achieve higher accuracy compared to the typical human-designed architectures.

1.2 Motivation and Current Applications

The increase in computational power together with the ability to store and analyse a larger amount of data have led to the design of complex models for solving sophisticated problems. Consider the case of popular mixed integer programming solvers for scheduling and planning which come with a large number of free parameters that have to be tuned manually [6, 7]. It is vital to find the optimal value of each parameter for the model to be trained. However, dependencies between parameters are not to be ignored since good values for one parameter can be overshadowed by poor performance of another parameter. Therefore, it is crucial to consider joint optimisation as a solution to our hyper-parameter tuning problem.

Hyper-parameter search is often performed manually, using rules-of-thumb or by trying out sets of hyper-parameters on a predefined grid. Such approaches are characterised by lack of reproducibility and become impractical as the number of hyper-parameters grows. Standard methods, *e.g.* Grid Search and Random Search, fail to scale well to larger hyper-parameter spaces and training times. Consequently, the idea of automated hyper-parameter tuning is receiving increased attention in machine learning [8]. Furthermore, manual search done by experts has been outperformed by automated methods on several problems [9]. Hyper-parameter tuning and neural architecture search, when compared to hand crafted architectures, can lead to simpler models that produce similar accuracies with less memory and FLOPs [10]. Automated algorithms for hyper-parameter tuning bring benefits to a wide area of commercial applications such as image recognition, medical analysis tools, game engines, robotic agents and design problems. All these tasks come with extensive parameter spaces to explore which makes automation desirable [7].

1.3 Contributions

- An exploration of the manually designed neural architectures used in image classification tasks and of two closely connected subfields of Automated Machine Learning (**AutoML**): hyper-parameter optimisation approaches and **neural architecture search** (Chapter 3);

- An investigation of Efficient Neural Architecture Search (ENAS) [5] technique, demonstrating the usefulness of the concept of **weight sharing** across all child models in the master architecture, as well as showing the **lack of reproducibility** of the results reported by the authors of ENAS (Chapter 4);

- An assessment of different **search strategies** followed by the controller RNN, culminating with the finding that the ENAS controller *does not do significantly better than* a Random Search with Weight Sharing strategy, thus contradicting the claims of the authors of ENAS. Evaluating a new **hybrid search strategy** that combines the sampling decisions of the controller RNN with Random Search (Chapter 5);
- Enhancing the search strategy of the controller RNN by incorporating a *novel approach* based on the reward received by the controller. This leads to the **sampling of improved neural networks** that outperform those found by the ENAS controller (Chapter 6).

Chapter 2

Background - Machine Learning Building Blocks

2.1 Overview

At the heart of computer vision lies the task of image classification that, despite its simplicity, has numerous applications in real-life such as autonomous driving, face detection, automated image organisation.

The success in image classification is mostly owed to deep Convolutional Neural Networks (CNNs), which extract hierarchical features from the data. Furthermore, CNNs have been also applied to Natural Language Processing (NLP) tasks, achieving remarkable results in semantic parsing and text classification [11, 12]. With the rising success of CNNs came an increasing demand for neural network engineering which is regarded as the bottleneck in addressing classification tasks. Consequently, researchers have been attempting to automatise such tasks. The automatic process of finding effective neural networks is referred to as Neural Architecture Search.

To have a better overall picture of Neural Architecture Search, we offer a concise introduction into the topic of Deep Learning, followed by an overview of various types of neural network layers.

2.2 Deep Learning Basics

In the context of classification tasks, deep learning algorithms train a neural network to approximate the true function f^* using a set of observations (x, y), where x is the input and y is the label. In other words, a neural network builds a mapping $y = f^*(x; \theta)$ and optimises the parameters θ so as to generate a good approximation of the true function. The typical deep learning pipeline consists of a neural network, a loss function and an optimisation method.

The neural network can be considered as a directed acyclic graph that describes the connectivity between constituent structures called layers. The layers between

the input and last layer have an indirect impact on the output and are consequently named hidden layers. State of the art neural networks consist of tens or even hundreds of hidden layers that account for millions of parameters to tune.

The loss function measures the discrepancy between the prediction issued by the neural network and the true label. For classification tasks, it is usually expressed as the cross-entropy loss:

$$L(x_i, y_i) = y_i \cdot log(f^*(x_i)) + (1 - y_i) \cdot log(1 - f^*(x_i))$$

While the loss determines how accurate a prediction is, the optimisation method offers the possibility to modify the parameters of the neural network with the aim of increasing the network's performance. Current optimisation methods, such as Adam[13], Adagrad[14], Adadelta[15], SGD[16], come with their own sets of parameters to be tuned, *e.g.* learning rate and momentum. While the available optimisation methods are basically limited, the search space for possible network architectures is huge. Thus, an exhaustive search to find the optimal set of parameters is naturally infeasible due to time constraints and computational resources.

2.3 Neural Network Layers

In the previous section, we stated that a neural network is comprised of various types of layers. It is critical to have a good grasp of these types of layers and their parameters, in order to have a strong understanding of designing neural architecture search spaces. Here, we offer a description of the format and hyper-parameters associated with the different types of layers, summarised in Table 2.1.

Layer Type	Hyper-parameters	Input Shape	Output Shape	Parameters
Convolution	K, filter size F, number of filters S, stride P, padding	$(H,\ W,\ C)$	$(H',\ W',\ F)$ $H' = \frac{H-K+2P}{S} + 1$ $W' = \frac{W-K+2P}{S} + 1$	$K^2CF + F$
Batch Normalisation		1. $(H,\ W,\ C)$ 2. H	1. $(H,\ W,\ C)$ 2. H	1. $2C$ 2. $2H$
Activation	f, activation function	1. $(H,\ W,\ C)$ 2. H	1. $(H,\ W,\ C)$ 2. H	1. 0 2. 0
Pooling	K, filter size F, number of filters S, stride r, reduction function	$(H,\ W,\ C)$	$(H',\ W',\ F)$ $H' = \frac{H-K+2P}{S} + 1$ $W' = \frac{W-K+2P}{S} + 1$	0
Global Pooling	r, reduction function	$(H,\ W,\ C)$	C	0
Dropout	p, dropout rate	1. $(H,\ W,\ C)$ 2. H	1. $(H,\ W,\ C)$ 2. H	1. 0 2. 0
Fully Connected	U, output units	1. $(H,\ W,\ C)$ 2. H	1. U 2. U	1. $UCHW + U$ 2 $UH + U$

Table 2.1: Summary of the frequent types of layers in a CNN and their (hyper)-parameters.

The network starts with the *Input* layer. In the context of image classification it is usually a tensor of size (N,H,W,C), where N is the number of images, and H, W are the height and width of each image, while C is the number of channels of the image (*e.g.* $C = 3$ for RGB channel).

The essential building block of a CNN is the *Convolution* layer, as it extracts meaningful features from the data. It works by sliding multiple filters over the input tensor, performing a dot product between a section of the image and the filter, which generates multiple feature maps. The concatenation of these feature maps forms a tensor that serves as an input for the next layer. In addition, *Activation* layers are used for the purpose of extracting more representative features from the data. These activation functions introduce a non-linear representation of the data. Without using these non-linearities, the output of the network would merely be a linear combination of the input.

Researchers show that training deep neural networks can be accelerated using a *Batch Normalisation* layer after each convolution layer [17]. This type of layer is employed to reduce the covariate shift by using two trainable parameters that learn to decide whether the input tensor for the next layer should be normalised.

Typically inserted between consecutive convolutional layers, the *Pooling* layer shares some commonality with the convolution layer in the sense that it also slides a filter over the input tensor. What makes a pooling layer different than a convolution layer is the operation it performs, namely, it uses a reduction function over a region of the input tensor called receptive field. Employing dimensionality reduction is the main aim for using pooling layers. It is also worth noting that pooling layers have no trainable parameters [18].

The final classification usually takes place inside the *Fully Connected* layers, which compute a weighted sum over the extracted features. These layers can be thought of as convolutions with a filter shape equal to the input tensor shape. It is usually the case that the fully connected layers contain the largest number of learnable parameters in the network. This can lead to over-parametrisation which, in turn, makes the network learn the noise from the input. In this case, the neural network fails to generalise to unseen data. The phenomenon described here is called overfitting [19]. Moreover, a *Dropout* layer is sometimes used in an attempt to fix overfitting by disabling neurons during training.

One way to address the issue of over-parametrisation is to use a *Global Pooling* layer before the first fully connected layer. Akin to a pooling layer presented above, the global pooling layer uses a filter of shape equal to the one of the input tensor.

2.4 Convolutional Neural Networks

CNNs are a class of feed-forward neural networks. An example of the structure of a CNN and the connections between the layers presented above is depicted in

Figure 2.1. We choose to apply hyper-parameter tuning on CNNs because of their wide area of application, especially when it comes to computer vision applications such as facial recognition or image classification [20]. Furthermore, CNNs require a multitude of hyper-parameters to tune which make them the preferred model to work on for our research.

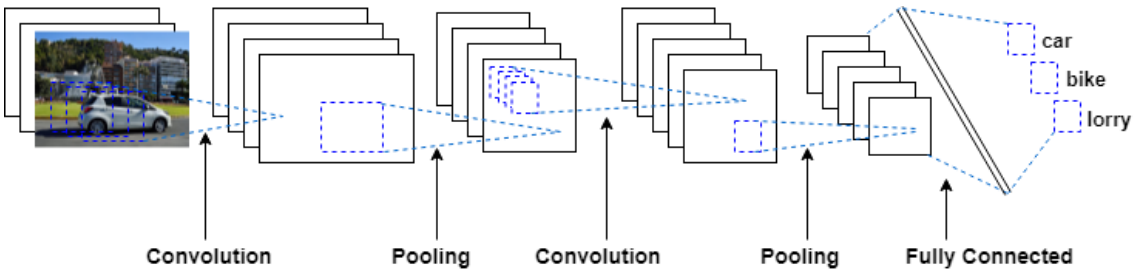


Figure 2.1: Structure of Convolutional Neural Networks.

The problem of hyper-parameter tuning partly owes its high degree of difficulty to the various types of parameters. Usually, neural networks have discrete, continuous and conditional variables (activated only if other variables have specific values) that act as parameters. Consider CNNs as an example, where the number of units in a fully connected layer is discrete, momentum is a continuous variable, and the dropout rate on a fully connected layer is conditioned on whether the "input" dropout, which is a Boolean variable, is set to True [7]. In general, hyper-parameters can be classified into two main categories:

- *Architectural*: number and size of layers
- *Training*: number of epochs, batch size, learning rate, dropout, weight initialisation

There is no consensus in the literature on the number of hyper-parameters to optimise which varies from paper to paper. Furthermore, the hyper-parameters that are optimised also vary: some experiments tune the learning rate, number of convolutional and fully connected layers, number of filters per convolutional layer and their size, number of units per fully connected layer, batch size, and regularisation parameters [21]; other works set the values for the learning rate and batch size [22].

Chapter 3

AutoML Techniques for Hyper-parameter Tuning and Neural Architecture Search

3.1 Overview

We start this chapter by emphasising the complexity of crafting neural networks, as well as underline specific concepts behind popular architectures. Currently, the majority of state-of-the-art (SOTA) neural networks are manually designed by engineers. We then present various hyper-parameter optimisation methods: Grid Search, Random Search, Bayesian Optimisation, Genetic Algorithms and Reinforcement Learning. Afterwards, we concentrate on neural architecture search, a subfield of Automated Machine Learning (AutoML) that is closely connected to hyper-parameter tuning. We focus in particular on performing neural architecture search using Reinforcement Learning.

3.2 Manually Designed Architectures

A breakthrough in the field of Deep Learning was achieved by *AlexNet*[23], when it won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [24] in 2012. It yielded a top-5 error of 15.3% on the challenging ImageNet dataset, a score that was significantly beating other methods based on traditional machine learning. Following that success in 2012, a variety of families of neural networks have been proposed. Some of the most competitive which we are going to cover in this section are: VGGNet(16) [25], GoogLeNet [26], Inception-V3 [27], ResNet [28], SqueezeNet [29], MobileNet [30], DenseNet [31], Xception [32]. A summary of these manually crafted architectures is shown in Table 3.1. It is important to highlight that there are several variations of these architectures even though they maintain a structure which is similar to the one of the architectures these variations originate from. In the original papers [25, 26, 27, 28, 31, 32] more variations were discussed, however for greater clarity we choose to include only the best performing architectures in our table. Moreover, the modules included in some of the architectures are vital for the sake of accuracy and training time. Due to their importance, we offer an insight into the structure of these modules in the following paragraphs.

Network	Input Shape	Weights (M)	Depth	ImageNet accuracy	CIFAR-10 accuracy
AlexNet	224 x 224	62.5	8	84.70	-
VGGNet(16)	224 x 224	138	16	92.70	-
GoogLeNet	224 x 224	4	22	93.33	-
ResNet-32	32 x 32	0.46	32	-	92.49
ResNet-50	224 x 224	25.6	50	94.75	-
Inception-V3	224 x 224	24	42	94.40	-
SqueezeNet	224 x 224	0.12	18	80.30	-
MobileNet	224 x 224	2.59	55	88.20	-
DenseNet-BC12	32 x 32	0.8	100	-	95.49
Xception	224 x 224	22	36	94.50	-

Table 3.1: Summary of manually crafted neural network architectures with their respective structure and performance. For the ImageNet dataset, we report the Top-5 accuracy.

AlexNet has a basic CNN construction which is depicted in Figure 3.1a. It is formed of eight layers: the first five are convolution layers, while the last three are fully connected layers. Additionally, some of the convolution layers are also followed by max pooling layers and dropout is used on the first two fully connected layers. AlexNet employs a Rectified Linear Unit (ReLU) as an activation function, which improves the training performance over other ubiquitous activation functions like tanh or sigmoid. Results from the original paper [23] show that the high performance of the network mostly relies on its depth. This, in turn, required the use of GPUs in order to make training feasible. Although it has a reasonably simple configuration, AlexNet has a surprisingly large number of parameters (approximately 62 million) compared to newer architectures. As a consequence, the authors trained the network on two GPUs for six days.

Influenced by AlexNet, *VGGNet(16)* employs a deeper architecture shown in Figure 3.1b with 13 convolution layers, some of them followed by a pooling layer, and 3 fully connected layers. Despite its appealing configuration, the capacity of this network is enormous with roughly 120 million parameters. The authors trained the network for three weeks on four GPUs, managing to obtain a 7.3% top-5 error at the cost of doubling the number of parameters of AlexNet. Thus, VGGNet(16) was ranked second in the 2014 ILSVRC computer vision competition [24].

The winner of the aforementioned competition was *GoogLeNet*. Also known as Inception-V1, the network scored a 6.6% top-5 error rate which is on par with human level performance. Primarily inspired by the original LeNet architecture [33], the novelty of GoogLeNet comes from its implementation of the Inception module which is shown in Figure 3.2. The main idea behind the Inception module is the use of multiple small convolutions with the aim of strongly reducing the number of parameters. The GoogLeNet architecture consists of a 22-layer neural network, and despite the large number of layers, the number of parameters is just 4 million.

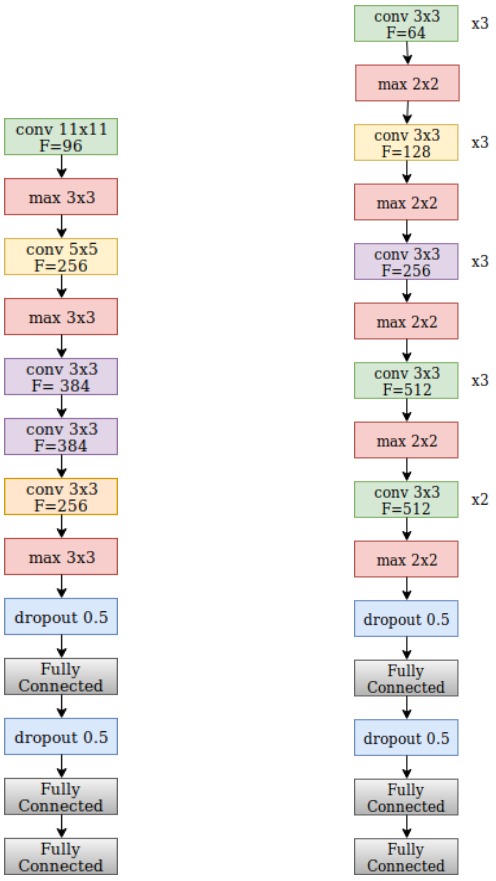


Figure 3.1: Structure of (a) AlexNet, and (b) VGGNet(16).

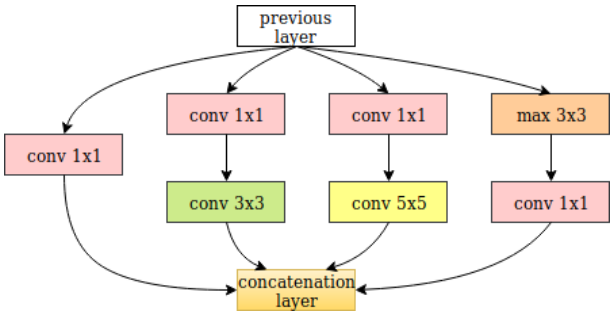


Figure 3.2: Structure of the Inception Module present in GoogLeNet. Smaller convolutions are employed in order to reduce the number of parameters in the network.

The *ResNet* architecture is the winner of the 2015 ILSVRC competition, achieving a top-5 error rate of 3.57% which surpasses human-level performance on ImageNet [24]. Its name stands for Residual Neural Networks and the major traits of it are the use of skip, residual connections and introduction of heavy batch normalisation layers. The advantage of skip connections is that they reduce the impact of the vanishing gradient issue, as the network has fewer layers to propagate through. The main conceptual motifs that drive ResNet are shown in Figure 3.3. Repeating these motifs, the authors of ResNet have trained a neural network composed of 152 layers which outperforms and is less complex in terms of the number of parameters than VGGNet. Without skip connections, the accuracy of deep neural networks has a tendency to saturate relatively early, and then it degrades quickly. This problem is also known as the degradation problem in deep networks. Empirically, the authors have shown that skip connections play an essential role not only by reaching a better local optimum, but also allowing for faster convergence. Furthermore, neural networks without residual connections explore more of the feature space which makes them particularly sensitive to noise in data.

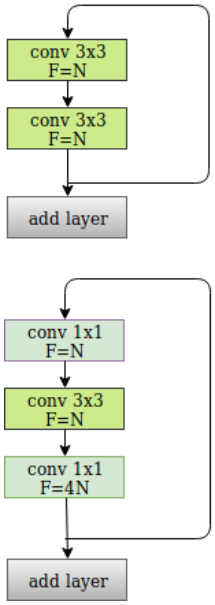


Figure 3.3: The two main motifs of ResNet.

Reaching second place in the 2015 ILSVRC computer vision competition [24], *Inception-V3* replaces the expensive 5x5 convolution in GoogLeNet with a couple of 3x3 convolutions, while still achieving the same accuracy. The sequence of 3x3 convolutions is less demanding to perform, and, moreover, this trick reduces the amount of parameters by about 28%. This replacement can be visualised graphically by comparing the Inception module in GoogLeNet (Figure 3.2) with the top motif shown in Figure 3.7. Working with the same idea, a 3x3 convolution can be factorised into two asymmetric convolutions: a 1x3 convolution followed by a 3x1 convolution. This further reduces the number of parameters by 33%. These architectural differences are highlighted by the comparison between the top and middle motifs in Figure 3.7. Furthermore, the third motif in Figure 3.7 is used for its capability of building representations of the data in higher dimensions. Additionally, batch normalisation

is performed after the convolutions making the network less susceptible to learn the noise in the data.

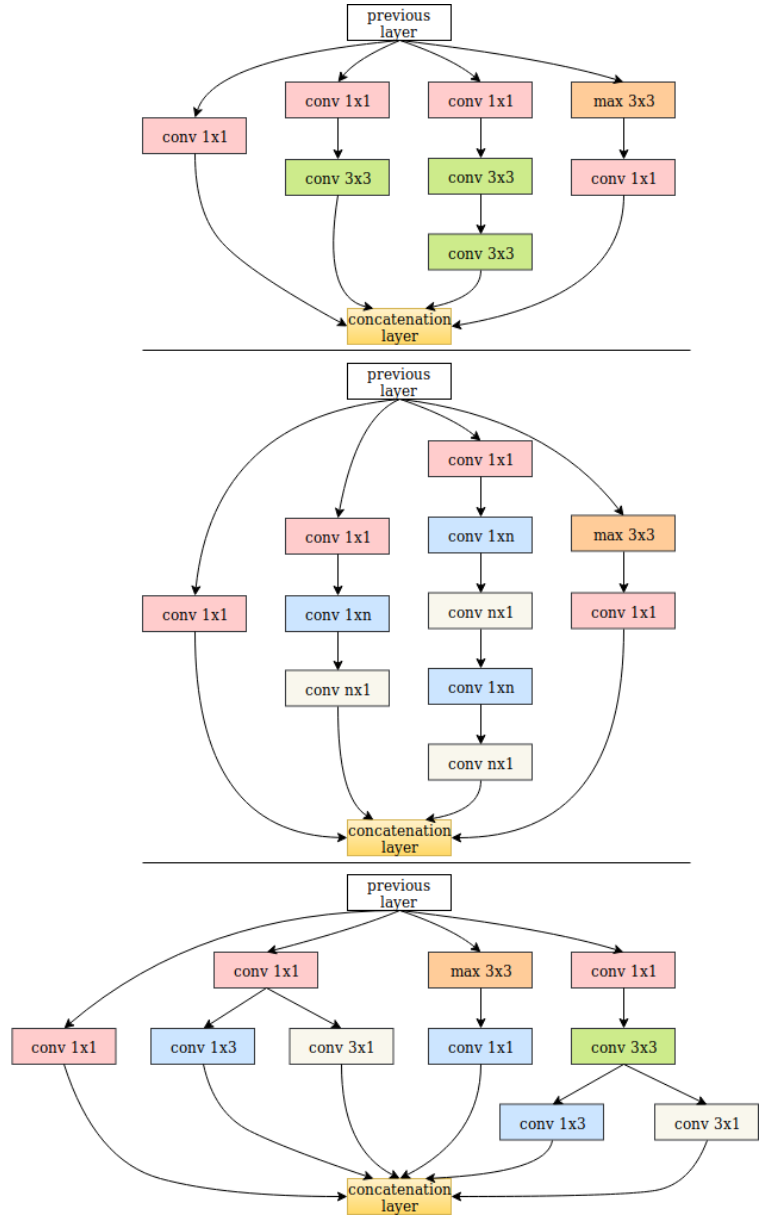


Figure 3.4: Structure of Inception Modules. The top motif replaces the expensive 5x5 convolutions with two consecutive 3x3 convolutions. The middle motif further factorises the 3x3 convolutions into a 1x3 convolution followed by a 3x1 convolution. The third motif incorporates the other two motifs and excels in building higher dimensional representations of the data.

Based on the module shown in Figure 3.5, *SqueezeNets* yield an accuracy similar to AlexNet, but are up to 3 times faster and require fewer parameters by the order of hundreds. The main trait of SqueezeNets is that they use a squeezing layer which substantially reduces the depth of the information going through the network, coupled with an expansion layer that performs the opposite operation. This technique is ubiquitous in modern architectures since it allows for a reduction of the number of parameters, while keeping the accuracy high.

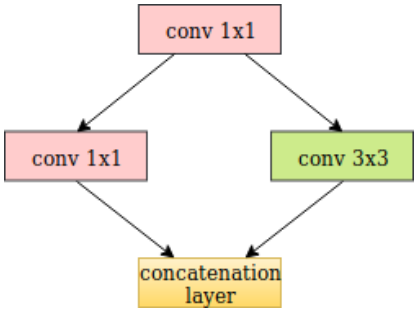


Figure 3.5: Structure of the SqueezeNet Module. It is used in smaller neural networks with few parameters that can easily fit into the computer memory.

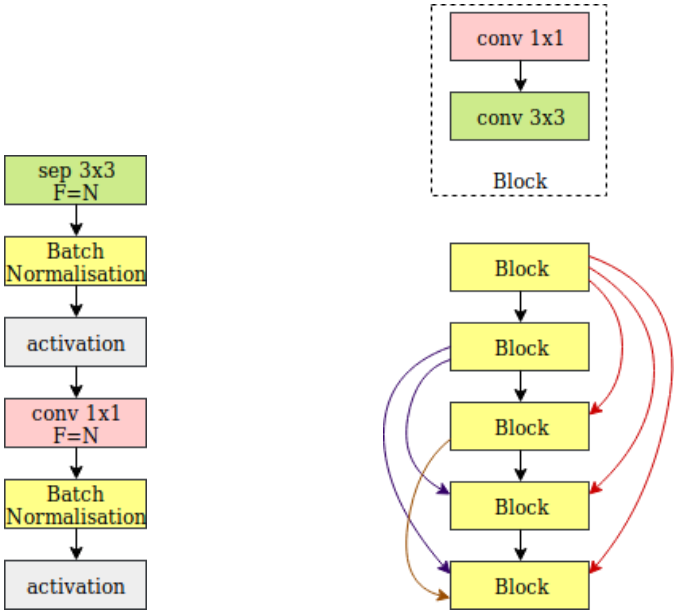
MobileNets provide a good trade-off between accuracy and training cost. Designed for applicability on low budget devices, MobileNets replace the more expensive 3x3 convolutions with stacks of 3x3 depthwise separable convolutions and 1x1 pointwise convolutions. The factorisation of convolutions, depicted in Figure 3.6a, has a good impact on reducing the training time.

Based on convolutions, *DenseNets* provide a better accuracy because of their dense connections, also known as skip connections, between layers present at different depth levels in the neural network as shown in Figure 3.6b. The greatest advantage of the skip connections used in the DenseNet module is that they elude the vanishing gradient issues. This, in turn, favours the reuse of learned features and reduces the number of parameters in the network. Another specificity of DenseNets is the use of concatenations to merge different branches used in skip connections, as opposed to ResNet which uses an addition operation. As a consequence of skip connections and concatenation of merging branches, the network does not have to learn again from scratch the previously learned features.

Xception is an adaptation of the Inception-V3 that includes a distinctive setup of depthwise separable convolutions depicted in Figure 3.7. Similarly to DenseNet and ResNet, Xception also uses skip connections which offer a better flow of information. Furthermore, it is important to note that Xception achieves better accuracy than Inception-V3 while having fewer parameters than its counterpart. This highlights the significant impact of combining skip, residual connections with depthwise separable convolutions.

3.3 Hyper-parameter Optimisation Approaches

We start by briefly describing standard methods, namely Grid Search and Random Search, continuing with the main areas of research for hyper-parameter tuning: Bayesian Optimisation (BO), Genetic Algorithms (GA) and Reinforcement Learning (RL), with the latter constituting our main focus. We can classify hyper-parameter optimisation methods in two broad categories: model-based and model-free, as shown in Figure 3.8. In a model-based scenario, there are cost-sensitive, *commercial* methods: *e.g.* Freeze-thaw (Google Vizier) [34], Fabolas [35], as well as standard, typical methods: Bayesian Optimisation. A similar classification can



(a) MobileNet (b) DenseNet

Figure 3.6: Structure of modules used in (a) MobileNet, and (b) DenseNet.

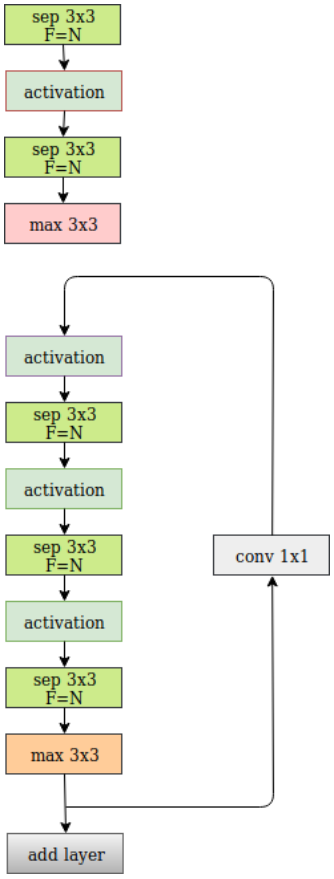


Figure 3.7: Structure of the Xception Modules. The use of depthwise separable convolutions differentiates this module from the ones we previously introduced.

be done in the context of model-free hyper-parameter optimisation methods. There are cost-sensitive methods, *e.g.* Hyperband [36], and standard methods: Random Search, GA and RL.

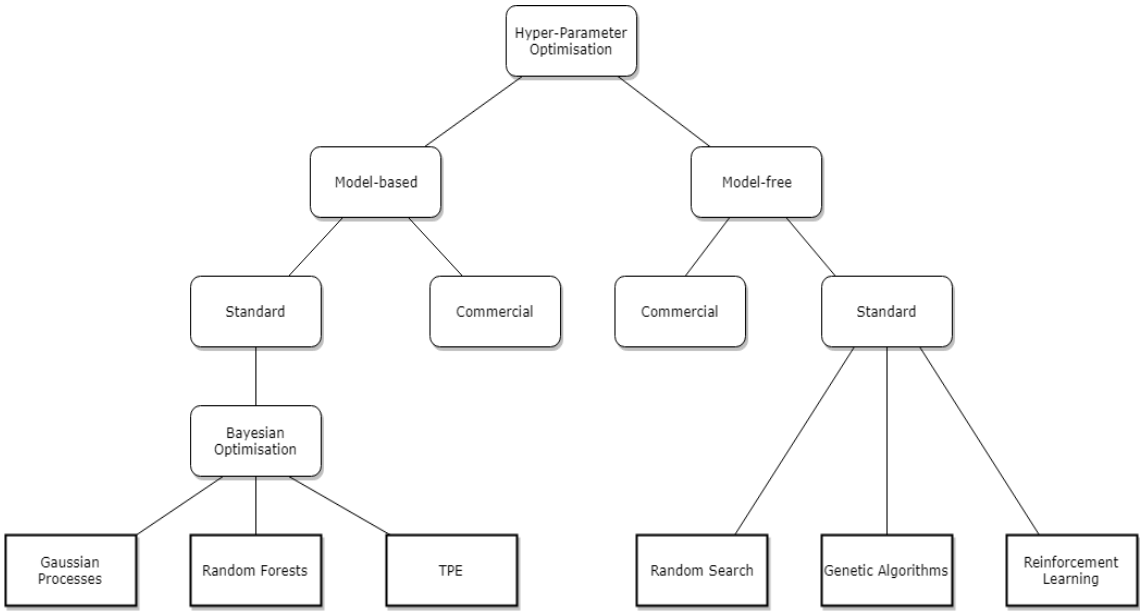


Figure 3.8: Taxonomy of hyper-parameter optimisation methods.

3.3.1 Standard Methods

Grid Search

One of the simplest methods for hyper-parameter optimisation is *Grid Search*. The user defines a range of candidate values to explore, and grid search evaluates the Cartesian product of these ranges. The idea is to exhaustively try out all possible combinations of hyper-parameter values, and for each combination we train a different model. In the end, we keep the model that offers the best performance. While Grid Search is guaranteed by its nature to converge, the main drawback is that the time complexity of this algorithm is exponential with the number of hyper-parameters. Therefore, Grid Search can be infeasible when tuning more than a few hyper-parameters [37].

Random Search

Another standard method for hyper-parameter optimisation is *Random Search*. As opposed to searching the entire grid, *i.e.* the Cartesian product mentioned above, Random Search evaluates random samples of the points on the grid. This method yields approximately the same results as Grid Search, but needs far less computational time. More specifically, it has been showed that if at least 5% of the possible combinations on the grid give a sufficiently good result, *i.e.* close to optimum, then random search finds such a solution after 60 trials with a 95% probability [37]. This is a substantial improvement when compared to classical Grid or Manual Search.

For instance, consider the case when we run 9 different experiments to optimise the learning rate and regularisation term (Figure 3.9). When using Grid Search, we are only testing 3 values for the learning rate and 3 values for the regularisation term. However, using Random Search 9 different possible values for each parameter are explored. Both Grid and Random Search are totally uninformed by past evaluations, which means that they spend time evaluating unreasonable combinations of hyper-parameters.

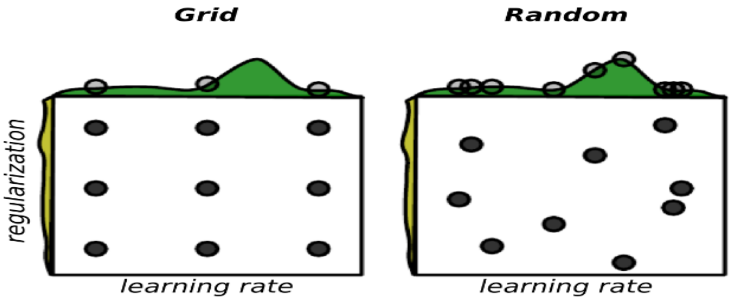


Figure 3.9: Grid and Random Search approaches for optimising learning rate and regulariser [37].

3.3.2 Bayesian Optimisation

Overview

In contrast to standard methods, Bayesian optimisation methods keep track of past evaluation results to determine the next combination of hyper-parameters to evaluate. Let us suppose that our objective function, which is usually the cross-validation error, is expensive to compute and we want to limit calls to it. To overcome this disadvantage, a probabilistic model is built by mapping hyper-parameters to a probability of a score on the objective function. In literature, this probabilistic model is called surrogate model which approximates the true function based on past observations [38]. A Gaussian Process (GP) is attached to the observed data and the GP mean is used to approximate the true objective function within a certain confidence interval (Figure 3.10). The next choice of hyper-parameters to evaluate is based on previous observations by optimising an acquisition function. This function is a trade-off between exploitation of known optima and exploration of previously unknown values in the hyper-parameter space, with different acquisition functions having a different impact on this trade-off (Figure 3.11). The whole process of updating the surrogate model, analysing the acquisition function to choose the next set of hyper-parameters, and making a new observation based on previous ones (Figure 3.12) is repeated until the number of iterations is reached or the method converges. Iterating through such an algorithm (Algorithm 1), Bayesian optimisation methods perform an efficient search of the hyper-parameter space to determine the global optima [39]. Two major decisions have to be made when using Bayesian optimisation. The first one is selecting a prior over functions based on the assumptions made about the function that is being optimised. The second decision to be made is the choice of acquisition function.

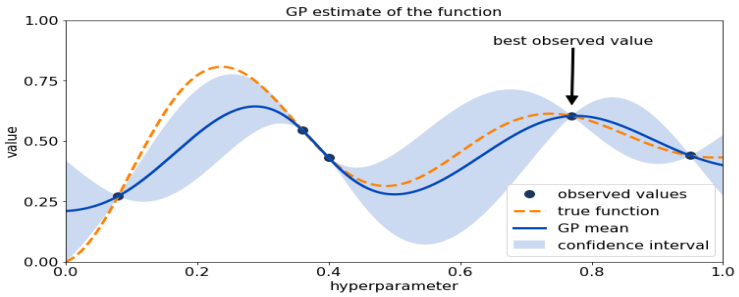


Figure 3.10: GP approximation of the true objective function [40].

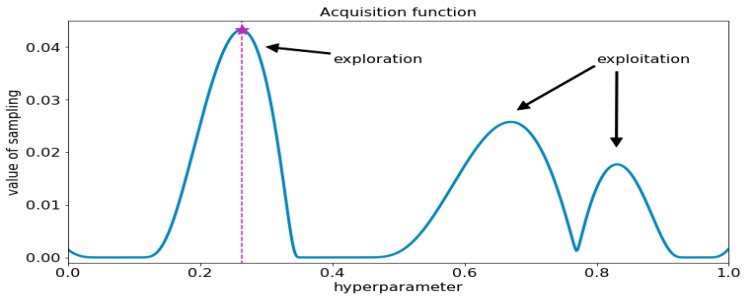


Figure 3.11: Acquisition function and trade-off between exploration and exploitation [40].

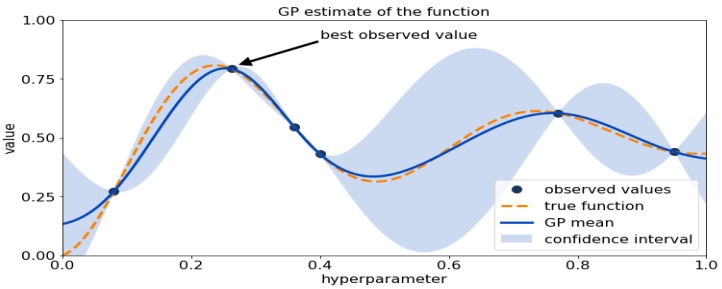


Figure 3.12: Performing a new observation based on past observations [40].

Algorithm 1 Pseudocode for Bayesian Optimisation Algorithm

procedure SOLVE(problem, n, k)
    $X \leftarrow random(n)$
    $Y \leftarrow problem.evaluate(X)$
    $model = GP(X, Y)$
    $model.update()$
    while not $converged()$ **do**:
        $acq \leftarrow getAcquisition(k)$
        $x_{new} \leftarrow acq.optimise()$
        $y_{new} \leftarrow problem.evaluate(x_{new})$
        $model.update(x_{new}, y_{new})$
    return $model.best$

Publicly Available Tools

Spearmint [41] is a software package which employs algorithms outlined by (Snoek et al., 2012) [42]. The four proposed algorithms have the following similarities. A Gaussian Process (GP) prior is chosen because of its adaptability and mathematical convenience. As far as the acquisition function is concerned, although several options are available in Spearmint, the preferred one is Expected Improvement (EI), which is also one of the most widely used acquisition functions in literature [43].

The first algorithm is *GP EI MCMC* which works by integrating out GP hyper-parameters. This integration is performed using a type of Markov chain Monte Carlo (MCMC) algorithms called slice sampling. The second algorithm is *Nx GP EI MCMC* which is an N times parallelised version of GP EI MCMC that aims to determine the next evaluation point while the current evaluation is still running. The third one is GP EI Opt which optimises the GP hyper-parameters. The last algorithm, *GP EI per Second*, introduces the concept of a duration function and employs a GP machinery to model this function so as to improve the performance in terms of wallclock time. The performance of these algorithms was tested on the CIFAR-10 dataset using CNNs. A number of 9 hyper-parameters of a three-layer architecture were tuned: the learning rate, number of epochs, four weight costs: one for each layer and one for the softmax output weights, as well as the width, scale and power of the response normalisation on the pooling layers of the network. A 14.98% error rate was reported for the hyper-parameters determined by GP EI MCMC [42].

Advantages and Disadvantages

The main advantage of Bayesian Optimisation methods over manual search and Grid and Random Search lies in the fact that we are making informed, intelligent decisions about the new set of hyper-parameters to evaluate while considering trade-offs between exploration and exploitation. The efficiency in finding the optimal parameters of these methods is particularly noticeable for non-convex functions, taking less training time and computational resources [36]. However, since each step builds on top of the other, such algorithms are hard to parallelise. While it has been showed that Bayesian Optimisation methods outperform Random Search on some benchmark tasks, when it comes to high-dimensional problems, Bayesian Optimisation methods yield similar performance to Random Search which is manifestly conceptually simpler [44].

3.3.3 Genetic Algorithms

Overview

An alternative to Bayesian Optimisation is a class of algorithms called Genetic Algorithms (GAs). Inspired by the biological process of natural selection, GAs are based on the idea that the fittest individuals in a population reproduce and pass their traits to the next generation. In the context of hyper-parameter optimisation, the individuals represent a set of hyper-parameters. To begin with, individuals need to be encoded *e.g.* as binary arrays. The generic structure of a GA is shown in Figure 3.13. The process begins with the initialisation phase in which a random set of hyper-parameters is generated. In the second stage of the algorithm, the set

of hyper-parameters is evaluated on the test set using the classification error which plays the role of the fitness function. Based on these results and a fitness threshold, a number of individuals are probabilistically selected. Then the Genetic Operators phase starts. The two most common operators are crossover and mutation. Crossover constructs new potential solutions, while the mutation operator ensures diversity. This leads to a new population of individuals which is again evaluated in the second step of the process. Iterations of GA algorithms are carried out until stopping criteria, *e.g.* training time is up, are satisfied.

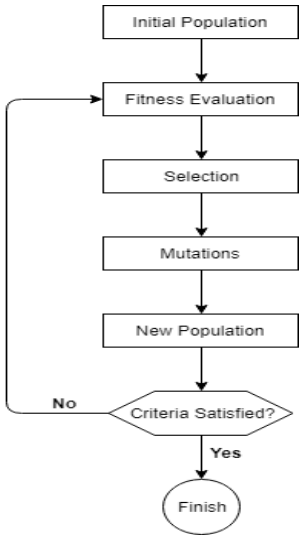


Figure 3.13: Workflow of activities in Genetic Algorithms.

Large-Scale Evolution of Image Classifiers

(Real et al., 2017) [45] propose a solution to the problem of hyper-parameter optimisation involving massive parallel computations in which the individuals represent architectures that are encoded as a simplified graph referred to as a DNA. This simplified graph is only transformed to a full neural network graph for training and evaluation. In the initialisation phase, random individuals are created using single layered networks without convolutions. Fitness testing is done using a method called tournament selection, in which the weaker individual is killed. Mutations are the prevalent genetic operators and they come in several variations such as insert/remove convolution, alter stride, reset weights etc. The population is set to 1000 individuals which is very computationally demanding since each individual has to be trained and undergoes fitness evaluation. For the CIFAR-10 dataset, this algorithm achieves a classification error of 5.4%, which is better than Bayesian Optimisation as reported in their paper, albeit computationally expensive.

Advantages and Disadvantages

The main asset of genetic algorithms is that they are open to parallelisation. Therefore, given enough computational power, GAs can outperform Bayesian Optimisation as described in the previous section. However, if one does not have a highly competitive hardware, GAs may not be ideal for hyper-parameter tuning. These algorithms require large populations to be trained. While smaller samples can be

used in experiments, it may happen that the algorithms converge to a local optimum and get stuck there.

3.3.4 Reinforcement Learning

Overview

RL is an area of machine learning concerned with the study of optimal decision making. The main idea behind RL is how software agents decide to take actions in an environment in order to maximise some cumulative reward. An agent needs to find the best actions to take or how to behave in an environment so as to carry out a task in the best possible way. The environment sends observations to the agent in the form of a reward signal for its actions and information about the new state. The reward gives notice to the agent about how good or bad the action taken was.

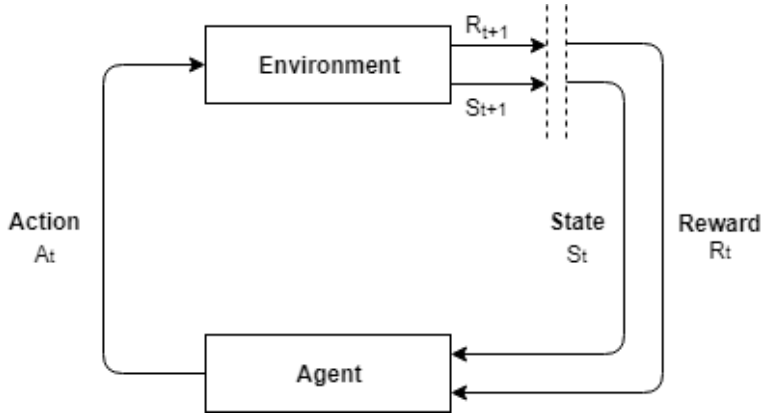


Figure 3.14: The agent-environment interaction.

RL can be seen as a paradigm which is situated at the boundary between supervised and unsupervised learning. It is not completely supervised because it does not have a set of labelled data for training, yet it cannot be considered unsupervised either due to cumulative reward which our agent needs to maximise. A particularly important aspect in RL is time; the learning process is done sequentially with delayed feedback from the environment. Furthermore, the feedback might be delayed over a few time steps, or even to the point that the agent receives the feedback only after successfully completing a task, *e.g.* if the agent's goal is to escape a maze, the feedback may well be at the end. No supervisor is involved in the process of learning, it is just the reward signal that informs the agent how well it performs.

The interaction between the agent and the environment (Figure 3.14) is performed over a sequence of discrete time steps. At time step t, the agent receives some notion about the state S_t of the environment, and using that information performs an action A_t. At time step $t+1$, the agent receives a reward R_t which is usually a real number, and finds itself in the new state S_{t+1}. The set of sequences of the form state-action-reward is called history. A naive attempt to pick the best action to take for some time step t would be based entirely on the history. However, this kind of approach is prone to failure in the real-world problems because of the vast history. Instead, the state, which encapsulates the information acquired thus far, is

used to make the next decision. The environment state is a private representation of the environment based on which the next state and reward are issued. As far as the representation of the agent state is concerned, there are two possible methods. The first one is the Bayesian approach using Markov Decision Processes (MDP) for a fully observable environments, and Partially Observable MDP for partially observable environments. The second approach to represent the agent state is by using Recurrent Neural Networks (RNNs):

$$S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$$

Here, the current agent state S_t^a depends on a linear combination of the previous state S_{t-1}^a multiplied by some weight W_s and the current observation O_t multiplied by some weight W_o.

There are three main components of an RL agent: a policy, a value function, and a model. The way the agent's action selection happens is through a policy $\pi(a|s)$, which usually gives the probability of taking action a when in state s, i.e. Stochastic Policy:

$$\pi(a|s) = P(A_t = a|S_t = s)$$

The value function depends on the policy and it informs the agent what reward to expect if taking action a in state s. More specifically, the value function returns the expected sum of future rewards starting from state s:

$$V^\pi(s) = E[\sum_{t=1}^{T} \gamma^{t-1} R_t]$$

where γ is a discount factor between 0 and 1, which has the role of balancing between current and future rewards. For instance, a high discount factor means that the agent prioritises rewards in the future.

For the task of hyper-parameter optimisation we are interested in Model-Free Agents. One of the most popular model-free algorithms for RL is Q-learning. This algorithm finds the optimal action-selection policy using Q-values stored in a Q-table. A Q-value gives the expected reward given the undergoing of action a_t at state s_t and following the policy π afterwards. In order to calculate Q-values we use:

$$Q^\pi(s_t, a_t) = E[\sum_{t=1}^{T} \gamma^{t-1} R_t|s_t, a_t]$$

and the update equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \cdot maxQ(s_{t+1}, a') - Q(s_t, a_t)]$$

where α is the learning rate, R_{t+1} is the reward for taking action a_t at state s_t, γ is the discount factor and $maxQ(s_{t+1}, a')$ is the maximum expected future reward given the new state s_{t+1} and all possible actions a' at that state. When picking a new action it is important to consider the trade-off between exploration and exploitation. We usually specify an exploration rate which influences the randomness of the actions taken. However, using Q-tables is ineffective, and instead, an approximation

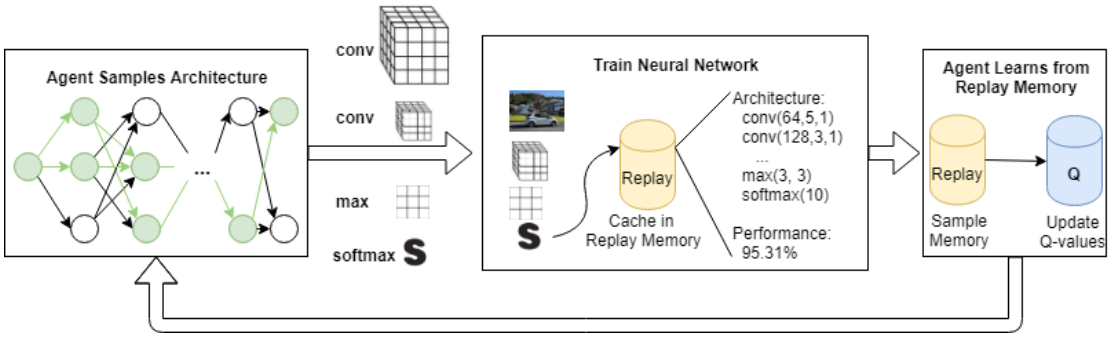


Figure 3.15: Designing CNN architectures using RL algorithms (*e.g.* Q-learning).

for the Q-values can be obtained using Neural Networks.

Based on the Q-learning algorithm, CNN architectures can be learned. The agent samples a CNN architecture based on the agent's past results and conditioned on a predefined policy. This CNN architecture is then trained on a specific task and its performance is evaluated using *e.g.* the validation accuracy. All sampled CNNs are stored in the agent's memory. Consequently, through Q-learning the agent makes use of the past configurations to learn about the space of CNN architectures [22].

Recall that we identified two main categories of hyper-parameters in Section 2.4. The majority of RL approaches only tune architectural hyper-parameters, while training hyper-parameters like learning rate are selected manually [21]. For instance, (Baker et al., 2016) [22] choose to set a fixed value for learning rate and batch size, while architectural hyper-parameters are tuned. For the latter, an agent is trained to maximise the validation accuracy. The topology of the CNN is defined to be formed of convolutional, pooling, fully connected, global average pooling and softmax layers, where each layer comes with its own set of hyper-parameters to tune. The Q-learning algorithm used presents two fixed hyper-parameters: the learning rate and discount factor which are set to 0.01 and 1, respectively. To address the exploration/exploitation trade-off issue, the algorithm also uses an ϵ-greedy strategy which is similar to the exploitation rate from the generic Q-learning algorithm discussed previously. CNNs were trained for 20 epochs with a fixed batch size and learning rate. Experiments show a 6.92% error rate when using the most performant model. This approach proved to be expensive in terms of both time and resources needing at least 8 days to complete, as well as 10 GPUs.

A less expensive method, *Efficient Neural Architecture Search (ENAS)* is proposed by (Pham et al., 2018) [5]. A controller searches for an optimal subgraph within a larger computational graph to find different neural network topologies. The model corresponding to the discovered subgraph is trained so as to optimise a certain loss function. ENAS brings a major improvement to NAS [4] by forcing all child models to share weights so as to train each model to convergence. While significantly less computationally expensive than NAS, ENAS finds an architecture that achieves a 2.89% test error, highly competitive when compared to the 2.65% achieved by NASNet.

These methods show some promising results and since RL approaches to hyper-

parameter optimisation yield good results, it is worth experimenting more with this method. We conclude this section with a summary of Bayesian, Genetic and Reinforcement Learning Algorithms in the context of hyper-parameter tuning presented in Table 3.2.

	Bayesian Algorithms	**Genetic** Algorithms	**Reinforcement Learning** Algorithms
Type	Sequential model-based optimisation	Stochastic optimisation and metaheuristics	Sequential decision making
Optimisation based on	Surrogate model	Fitness of candidates	Reward signal
Exploration	Explore uncertain regions	Mutation and crossover	Based on the decision making policy(*e.g.* ϵ-Greedy)
Exploitation	Exploit low mean regions	Elitism and selection pressure	

Table 3.2: Summary of three widely used families of algorithms for navigating through the search space of CNN.

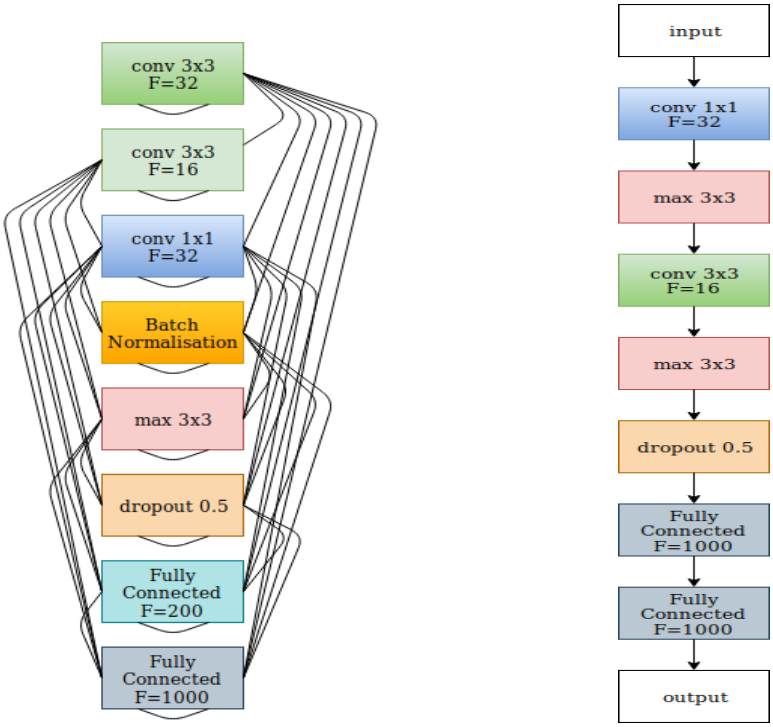
3.4 Primary Bottleneck in Neural Architecture Search

The remarkable results achieved by hierarchical feature extractors in the manually crafted architectures presented in Section 3.2 have generated a high interest in architecture engineering. Designing such complex networks and modules manually can be a laborious and time-consuming task. This has led to an increasing demand for automation of network architecture search.

We discuss the current state of research in automation of architecture search [46] from three perspectives: the search space and strategy, and the performance estimation strategy. In the context of neural architecture search, a state is defined as a neural network architecture. The search space is comprised of all the possible states that are reachable during the architecture search, while the search strategy provides a way of exploring the search space, in particular how the searching of architectures starts and what actions are required to move to a new state in the search space. Additionally, every state that is reached is evaluated using a performance estimation strategy which usually means that a particular configuration indicated by the current state is trained on a dataset and then the accuracy of the resulting architecture is evaluated on unseen data. Since the size of the dataset can be very large, the performance estimation phase can take a significant amount of time. Trying out the same architecture search on various datasets makes this phase more expensive in terms of both resources and time. Thus, the primary bottleneck of neural architecture search is accelerating the performance estimation strategy. To address this, a trade-off between accuracy and time is required.

From the perspective of the *Search Space*, we notice two different, broad types of networks: chain-structured (e.g. AlexNet - Figure 3.1a, and VGGNet(16) - Figure 3.1b and multi-branch networks such as DenseNet (Figure 3.6b) and ResNet (Figure 3.3).

Considering chain-structured architectures, the search space is determined by the type of each layer and the hyper-parameters associated with each layer, as well as



(a) Chain-Structure Search Space

(b) Sampled Network

Figure 3.16: Example of a Chain-Structure Search Space and a possible sampling of a neural network. Note that in this search space we have more than 2.3 million configurations for possible architectures.

the depth of the network. Usually, the types of layers that are part of the search space consist of Convolution, Dropout, Pooling, Fully Connected etc. We show an example in Figure 3.16 of a chain-structured search space from which we sample a neural network. In this example we fix the number of layers so that an architecture can have up to 7 hidden layers. Our search space consists of 8 types of layers, which means that the total number of possible network configurations is enormous, with more than 2 million configurations. Using grid search, we could generate every possible combination of layers, however this is unfeasible since training each sampled network would sum up to years of training time.

Multi-branch networks allow for greater flexibility when sampling a candidate architecture which implies that the search space grows exponentially. Most of the modules presented in Section 3.2 are particular instances of multi-branch networks. Such architectures tend to outperform the simpler chain-structured networks.

Guided by the manually engineered modules, the idea of combining automatically designed modules to form complex architectures emerged. This idea is based on the concept of a fixed master architecture from which modules called cells are combined to give rise to performant architectures [5, 22]. We note that such cells are less complicated than the whole network and thus this approach offers the possibility of reducing the search space significantly which, in turn, reduces the time needed for training. However, such a cell-based approach requires non-trivial rules on how to

combine the modules between them. This means that the architecture search phase can concentrate merely on finding the cells without properly combining them into a whole network and in such a case the complexity of the search space remains high.

The second characteristic of an automated neural architecture search is the *Search Strategy* which offers a way to explore and exploit the search space. While multiple strategies have been attempted, only some of them provide an accuracy on-par with the manually engineered neural networks. These successful search include Random Search, Bayesian Optimisation, Genetic Algorithms, and Reinforcement Learning, all of which were introduced in Section 3.3.

RL has gained a particular interest especially after the work of (Zoph et al., 2016) [4] achieved SOTA results on both a text classification dataset (Penn Treebank [47]) and an image classification dataset (CIFAR-10 [48]). However, this work can be identified as proof of concept due to the fact that the large computational resources involved are not accessible to the average Machine Learning practitioner. The experiments performed by Zoph et al. required 800 GPUs running for 4 weeks to find an architecture achieving nearly on-par with SOTA results on CIFAR-10. Afterwards, efforts have been made [5, 49, 22] to maintain high accuracy levels while at the same time diminishing the cost of computations. The core of an RL problem is how an agent decides what actions to take in an environment in order to maximise some cumulative reward. In this context, the environment represents the search space while the actions correspond to sampling different configurations from the search space. The most interesting part is the agent. The agent can be an RNN as employed by NAS [4] that samples strings which represent encodings of CNNs. Then, the convolutional networks are trained and the accuracies are fed as reward signals to the RNN. In another approach using RL, (Baker et al., 2016) [22] build the network layer by layer in a sequential fashion while using Q-learning to guide through the search space.

Another successful method used for finding novel architectures is Bayesian Optimisation. Some of the discovered neural networks achieve SOTA accuracies on CIFAR-10 [50, 51]. An alternative search strategy to Reinforcement Learning and Bayesian Optimisation is the family of evolutionary, genetic algorithms. Starting from an initial population of random neural networks called the first generation, an evolutionary algorithm is an iterative process. A network is sampled from each generation, then it is mutated and evaluated using a fitness function which is usually the accuracy on the validation set. The mutation is an operation that changes the configuration of the network by, e.g. removing or adding a layer, and after performing the mutation, the network is added back to the population. Genetic algorithms are not only applied in neural architecture search, but also when it comes to updating the network weights. In the latter context, given the large number of weights evolutionary algorithms do not match the performance of gradient-based approaches, however there are works which demonstrate the capability of using evolutionary algorithms to combine architecture search with learning the weights, and similarly, this method can also be applied in the RL area of research [52, 53].

One way to guide a search strategy is through *Performance Estimation*. This means

that when a neural network is sampled, the performance of this architecture needs to be evaluated in order to efficiently navigate through the search space. It is important to realise that the search space contains a large number of possible configurations and the cost of fully evaluating each sampled architecture from the search space requires hundreds or even thousands of GPU days [49]. Various methods have been employed in an attempt to reduce the cost of computation and speed up the performance evaluation phase. One such method involves training for shorter periods of time [49]. A second approach trains the sampled architectures only on a subset of the data [35]. Other methods perform training with a scaled version of the neural network [54], as well as lowering the resolution of the images [55]. While all these methods reduce the computational resources needed, they only provide approximations of the true evaluation phase, and these approximation do not preserve the proper hierarchy between networks and thus highly affect the outcome of the search [56].

3.5 Neural Architecture Search

Deep Learning has led to the development of novel neural architectures for tasks such as image recognition. Most of these architecture are developed manually by humans. However, human expertise is an expensive, time-consuming and error-prone process. To address this, a growing interest in automated architecture search has emerged. The automation of architecture engineering bears the name of *Neural Architecture Search*. We note that neural architecture search is not equivalent to, but rather a subfield of AutoML [57] with a substantial overlap with hyper-parameter tuning.

Neural Architecture Search with RL (NAS) [4] uses a predefined RNN as a controller. The controller is responsible for generating new architectural hyper-parameters of CNNs, and is trained using RL. After the controller predicts a set of hyper-parameters, a neural network with the specified configuration is built, *i.e.* a child model, and trained to convergence on a dataset like, for example, CIFAR-10. The validation accuracy of the child model is then used as a reward signal R to train the controller RNN, whose parameters θ are updated using an RL policy gradient method described in the next paragraph. Also, an upper threshold on the number of layers in the CNN is used to stop the process of generating new architectures.

The parameters predicted by the controller can be thought of as a series of actions $a_{1:T}$ to create an architecture for a child model. The controller aims to maximise its expected reward, denoted by $J(\theta)$:

$$J(\theta) = E_{P(a_{1:T};\theta)}[R]$$

As the reward signal R obtained from the validation accuracy on the child model is not differentiable, a policy gradient method is applied to update the parameters of the controller, θ, using the Reinforce Rule [58]:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{T} E_{P(a_{1:T};\theta)}[\nabla_\theta log P(a_t|a_{(t-1):1};\theta)R]$$

which is approximated by:

$$\nabla_\theta J(\theta) \approx \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_\theta log P(a_t | a_{(t-1):1}; \theta) R_k$$

where T is the number of hyper-parameters predicted by the controller to generate a neural network architecture, i.e. child model; m is the number of architectures sampled by the controller RNN so far at the current time step; and R_k is the validation accuracy from the k^{th} child model. Even though the above approximation is an unbiased estimate of the gradient in the Reinforce Rule, it still suffers from high variance which can be reduced by subtracting the exponential moving average of the past rewards, b, from the current reward R_k:

$$\nabla_\theta J(\theta) \approx \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_\theta log P(a_t | a_{(t-1):1}; \theta)(R_k - b)$$

It is worth noting that the parameters of the controller, θ, are updated only after training a child model to convergence. Since training a child model is time-consuming, training the controller RNN can be accelerated using parallelisation and asynchronous updates. A parameter-server scheme is used to store the shared parameters of the several controller replicas. Each replica samples a different set of hyper-parameters to design different child models which are trained in parallel. Then, each replica collects its corresponding gradients which are sent to the parameter-server. The parameter-server then updates the weights across all the replicas.

The controller RNN, which consists of a two-layer LSTM, is trained using the Adam optimiser [13] with the learning rate set to 0.0006. The weights of the controller are initialised uniformly between -0.08 and 0.08. Using the distributed training method described above, 800 child models are trained in parallel. The child models are trained using Momentum Optimiser with the learning rate set to 0.1, weight decay of 0.0001 and momentum of 0.9 with Nesterov Momentum. Each child architecture is trained on CIFAR-10 from scratch, and then is evaluated on a held-out validation set. The resulting accuracy is then used to update the weights of the controller RNN.

As far as the setup for the experiments on CIFAR-10 is concerned, the hyper-parameter search space includes convolutional architectures, with Rectified Linear Units (ReLU) as non-linearities, batch normalisation and skip connections. Skip connections [59], as the name suggests, allow for a wider search space since at any particular layer N, there are 2^{N-1} possible combinations of the previous $N-1$ layers to serve as inputs to layer N. The intuition behind this type of skip connection is that they have uninterrupted gradient flow from the first layer to the last layer, which tackles the vanishing gradient problem. Apart from the aforementioned parameters, for each layer the controller selects a filter height, filter width and number of filters. Also, we observe that the learning rate is not among the predicted hyper-parameters, and that the child models generated by the controller are restricted to only convolutional layers.

With this setup, NAS discovers a CNN with 15 layers achieving a 5.50% error rate

on the test set. An important aspect to underline is the role of skip connections. Connecting each layer to all the layers that precede it results in a small performance drop, yielding a 5.56% error rate. Conversely, removing all skip connections has a significant impact on the performance of the architecture, with the test error dropping to 7.97%.

A second set of experiments on CIFAR-10 also includes the stride width and stride height for each layer, along with the hyper-parameters predicted in the previous setup, thus increasing the search space at the cost of computational challenges. In this setup, NAS discovers a 20-layer CNN which achieves 6.01% error rate on the test set, similar to the previously suggested architecture. In a third scenario, the controller can also decide to include a pooling layer for layers 13 and 24, respectively, of the generated architectures. Moreover, a number of 40 filters are added to each layer of the child models. With this new setup, the controller predicts a CNN that achieves 3.65% test error rate.

In spite of the strong empirical performance of NAS, this method is extremely time-consuming and computationally expensive, using 450 GPUs for 3-4 days for a single experiment, *i.e.* an average of 38,000 GPU hours. Training thousands of models is nearly impossible for the usual machine learning practitioner. An explanation for the computational bottleneck of NAS is training every child model sampled by the controller RNN to convergence and throwing away all the trained weights after updating the controller. One way to overcome this disadvantage is to share the weights among the child models so that training each child model from scratch is not required anymore which, in turn, leads to a significant reduction in both time and resources allotted for training. Thus, rather than training thousands of child models from scratch, it is possible to train a single large neural network capable of emulating any architecture within the defined search space.

Chapter 4

Investigation of the Efficient Neural Architecture Search Approach

This project is an investigation of automated hyper-parameter optimisation focused on Reinforcement Learning (RL) methods applied to Convolutional Neural Networks (CNNs). It aims to identify strengths and weaknesses of RL algorithms for the problem of hyper-parameter tuning. Also, it is worth mentioning that we explore uncharted territory since very little literature is available for this kind of approach to our problem. Hence, there is no guarantee that an RL algorithm will outperform existing methods like Bayesian Optimisation or Genetic Algorithms. With his in mind, we search for solid neural networks architectures trained on the CIFAR-10 dataset.

Two of the main aspects to be taken into consideration when it comes to the stopping criteria for the automated hyper-parameter tuning problem are:

- *Performance*: One would like to pass a threshold in terms of performance, *i.e.* achieve a certain classification accuracy percentage, or even beat state-of-the-art algorithms, without necessarily considering time and hardware constraints;

- *Time*: One has a limited time to allocate for training the model. Experiments are done in a manner that is optimised for time.

We opt for the CIFAR-10 dataset [48] as the input for all our experiments. CIFAR-10 is among the most popular datasets for machine learning research, especially for image classification. This dataset is made up of 60000 colour images of size 32x32 and is split equally in 10 classes. The lowest error rates for CIFAR-10 are achieved using large neural networks, particularly CNNs, which require a multitude of hyper-parameters to optimise [4]. Therefore, we argue that this dataset is highly suitable for the task of neural architecture search. Another well-known dataset is MNIST, however it is not difficult to achieve above 90% which in turn makes comparison between various optimisation methods hard to evaluate. Furthermore, any slight improvement in performance is overshadowed by the amount of resources needed for this.

4.1 Overview

Similar to the NAS method presented in Section 3.5, Efficient Neural Architecture Search via Parameter Sharing (ENAS) [5] uses an RNN controller to sample CNN architectures. The key observation is that, as opposed to NAS which trains all child models from scratch, ENAS uses weight sharing among child architectures to amortise the cost of training. The idea of weight sharing is inspired by previous work [49] on transfer learning which shows that weights learned for a particular model on a specific task can be used, with little changes, for different models on different tasks.

We can represent the search space of ENAS using a Directed Acyclic Graph (DAG) whose nodes represent the local computations that can be performed at a particular layer, while the edges represent the flow of information, *i.e.* what previous nodes are connected to the current node. The local computations for a node, *i.e.* layer, are the following 6 operations:

- convolution with filter size 3x3 (conv 3x3);
- convolution with filter size 5x5 (conv 5x5);
- depthwise separable convolution with filter size 3x3 (sep 3x3);
- depthwise separable convolution with filter size 5x5 (sep 5x5);
- max pooling with kernel size 3x3 (max 3x3);
- average pooling with kernel size 3x3 (avg 3x3).

Both the computation operations and the flow of information for every node, *i.e.* *layer*, are sampled by the controller RNN. What this means is that a child model generated by the controller RNN is just one possible configuration of local operations and connections between nodes (layers), which is basically a subgraph of the larger DAG. The activation function at each layer is ReLU and every convolution is followed by batch normalisation.

Figure 4.1 demonstrates how the controller RNN samples a child model from 5 nodes in the DAG. At each node, the controller RNN decides on what previous nodes to connect to, as well as a computational operation:

- At node 1, *i.e.* layer 1, the controller does the following sampling: it can only connect the input to node 1, and it chooses a conv 3x3 operation;
- At node 2, *i.e.* layer 2, the controller samples previous index 1 and a max 3x3 operation;
- At node 3, *i.e.* layer 3, the controller samples previous index 2 and a sep 5x5 operation;
- At node 4, *i.e.* layer 4, the controller samples previous indices 2, 3, and a conv 3x3 operation;
- At node 5, *i.e.* layer 5, the controller samples previous index 1 and a conv 5x5 operation.

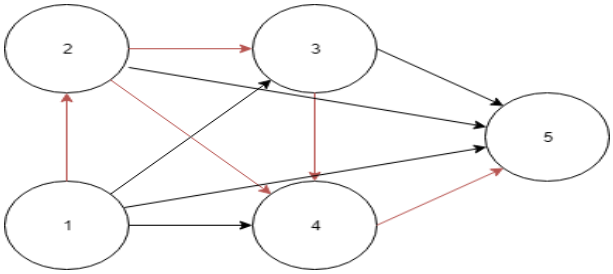


Figure 4.1: The DAG depicts the entire search space which contains 5 nodes. The red arrows denote the configuration of a model sampled by the controller RNN. The black arrows denote other possible paths between the nodes that are available for sampling to the controller.

The output of the controller, namely the child model, is shown in Figure 4.2. In our example, for every pair of nodes (i, j) with j $<$ i, there are 6 weight matrices $\boldsymbol{W}_{i,j}^{op}$ corresponding to the different computational operations. In other words, each operation at each layer has a different set of parameters.

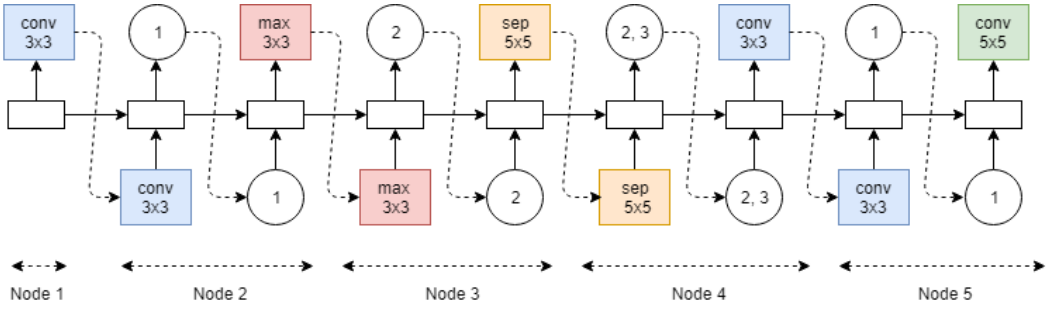
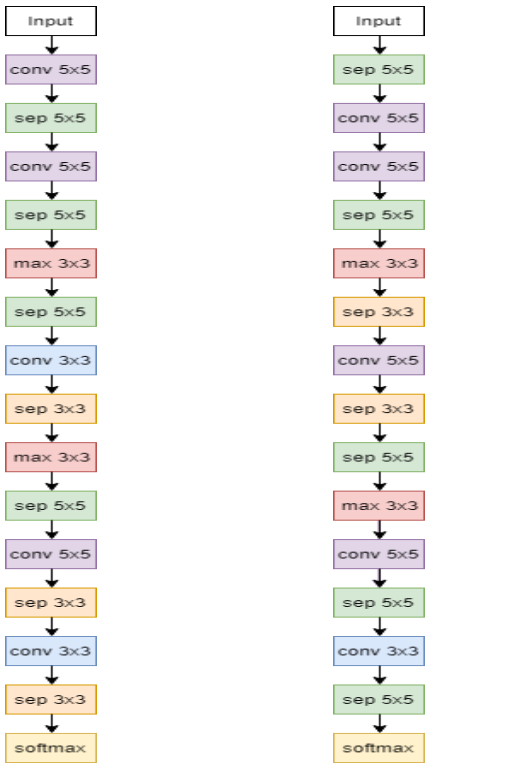


Figure 4.2: The child model CNN generated by the controller RNN. The figure shows the layer type and connections for each layer.

We now demonstrate the concept of weight sharing. If a computational operation between two nodes has been done before (*i.e.* was trained before in another child model), then the weights from the convolutional filters and 1×1 convolutions (to maintain number of channel outputs) will be reused. This can be easily understood with another example. Starting from the setup in our previous example, assume that now at node 3 the controller samples previous index 1 and a sep 5x5 operation; and at node 5, the controller samples previous index 1 and a avg 3x3 operation. Then, when training this new child model, we can reuse the weights corresponding to layers 1, 2 and 4: $\boldsymbol{W}_{\mathbf{1,0}}^{\boldsymbol{conv3x3}}$, $\boldsymbol{W}_{\mathbf{2,1}}^{\boldsymbol{max3x3}}$, and $\boldsymbol{W}_{\mathbf{4,3}}^{\boldsymbol{conv3x3}}$. Thus, only the newly established connections need to be trained. The main purpose behind weight sharing is that the controller RNN does not need to train every child model from scratch and thus the training phase is shortened. It is important to highlight that reducing training costs by a large margin, approximately 1000 times in terms of GPU-hours compared to NAS, allows the average practitioner to perform Neural Architecture Search using RL.

4.2 Experiments and Results

Previously, reproducing NAS results was unfeasible due to the large amount of computation that came with these methods [4]. However, more recent works [5, 60] have reduced the associated costs significantly and as such offer the possibility to attempt the reproduction of the results reported in the respective papers. When experimenting with ENAS [5], we found that the maximum resident set size of the process during its lifetime was approximately 18 GB. A key aspect to underline is the *lack of reproducibility* of the experiments ran by the authors of ENAS to find performant CNNs. Using the same setup as indicated in the original paper [5], we managed to achieve an error rate of 4.93%. In comparison, the authors reported a 3.85% error rate. Other researchers [61] have highlighted the issue caused by the inability to reproduce the results of the experiments performed by the authors of ENAS. Usually, a specific random seed is needed to have deterministic results for such experiments. In the original implementation of ENAS in TensorFlow, no such random seed is provided as an argument. We run the same experiment several times, *i.e.* using the same search space and parameters as input. The learning rate for the controller RNN is set to 0.001. We constrain the controller RNN to build child models that contain a fixed number of hidden layers, namely 14 hidden layers. We set the upper threshold for the number of epochs allotted for training to 310.



(a) Own Result

(b) Original Result

Figure 4.3: CNNs discovered by the controller RNN.

Our best performing child neural network sampled by the controller RNN, depicted in Figure 4.3a, yielded a test accuracy of 95.07% which is just short of the 96.15%

Method	Time (days)	GPUs	Parameters (M)	Test Accuracy (%)
Macro NAS + Q-Learning[22]	10	10	11.2	93.07
Net Transformation[60]	2	5	19.7	94.30
SMASH[62]	1.5	1	16.0	95.96
NAS[4]	28	800	7.1	95.53
NAS + more filters[4]	28	800	37.4	**96.35**
ENAS[5]	**0.7**	1	4.6	96.15
ENAS†	**0.7**	1	4.6	95.07
Hierarchical NAS[63]	1.5	200	61.3	96.37
Progressive NAS[64]	1.5	100	3.2	96.37
NASNet-A[49]	4	450	3.3	96.59
NASNet-A + CutOut[49]	4	450	3.3	**97.35**

Table 4.1: Top-1 accuracy for ENAS and other approaches on the CIFAR-10 benchmark. The first block contains techniques that search for the entire CNN. The second block presents approaches that search for convolutional cells which are combined to design the final neural network. For *ENAS*† we report the results achieved by running our own experiments.

result reported by the authors of ENAS. Our resulting CNN is on par with the state-of-the-art architectures, while requiring far less resources since we only use one GPU. The ENAS method excels when we consider the short period of time of less than 17 hours needed to find such a performant convolutional architecture. We present the performances achieved by various searching methods on CIFAR-10, as well as the resources and time required, in Table 4.1. It is worth comparing and contrasting the network obtained by the authors, shown in Figure 4.3b, and our neural network. We notice that for the first few layers the controller RNN prefers convolutions with larger kernels and samples the 5x5 convolution more frequently than the 3x3 convolution. This tendency occurs in all our experiments on CIFAR-10. However, stacking multiple smaller 3x3 convolution layers, as opposed to using a single larger 5x5 convolution layers, makes the network lighter and provides better accuracy for the simple reason that it results in more layers and a deeper network. In our experiments, the number of hidden layers is fixed for the child models to 14, so the RNN controller is forced at the first few layers to predominantly pick the larger 5x5 convolutions. Moreover, we observe the prevalence of depthwise separable convolution layers which reduce the computation costs by up to 7 times. Additionally, replacing the depthwise separable convolution layers with normal convolution layers decreases the test accuracy by 1.5%, while randomly modifying the skip connections between layers further decreases the test accuracy by 2.4%.

4.3 Summary

In this chapter we presented an investigation of the ENAS approach. After analysing the search space of ENAS and how the controller RNN samples architectural con-

figurations from the DAG, we explained the concept of weight sharing which is vital for significantly reducing the time required, namely less than 17 hours, to find a performant CNN architecture.

Moreover, we highlighted the lack of reproducibility of the results reported by the authors of ENAS. Based on the setup used by the authors, we discovered a child model that yielded a test accuracy of 95.07% on CIFAR-10. We underlined the tendencies encountered in the sampling decisions of the controller RNN and discussed how modifying the configuration of our best performing CNN affects its performance.

Chapter 5

Evaluation of the Controller RNN

5.1 Overview

The controller RNN is responsible for designing CNNs, namely what operations to employ at each layer and what skip connections to use. In this chapter, we explore and compare between different search strategies that can be followed by the controller RNN. A key point to highlight is that ENAS uses an ϵ-greedy algorithm to search for an optimal network configuration. Adjusting the values of the parameter ϵ, we can compare between using a strategy that allows the controller RNN to make its own sampling decisions and a controller RNN that employs a random search with weight sharing strategy, as well as a combination of these two strategies. Since randomness plays a role in all approaches, it is worth examining the training, validation and test accuracies of the controller RNN plotted against the number of epochs in order to evaluate these search strategies. We need to emphasise that the controller is guided by the validation accuracy of the sampled child models. This means that we expect a high level of similarity between the graphs depicting the training and validation accuracies for each of the three strategies.

5.2 Random Search with Weight Sharing

We first discuss the results for the experiment in which the controller RNN employs a Random Search with Weight Sharing strategy, in other words we set the ϵ parameter to 1. Analysing the graphs in Figure 5.1, we observe that we can partition the graph in four main areas of the search space. We recall that after a child model, *i.e.* a sampled CNN, is trained to convergence, the controller RNN samples an architectural configuration from another region of the search space. We show an in-depth analysis of the results for training accuracy of the controller, however the same applies for both validation and test accuracies:

- The first area is exploited in the first 25 epochs, the training accuracy of the controller RNN increases nearly monotonically starting from 58% and converging to 79%;
- Another region of the search space expands from epoch 25 to epoch 55. Here the training accuracy grows from 61% to 81%, where it converges;

- The third phase spans from epoch 55 to epoch 150. We notice that in this particular region the controller RNN initially converges to a training accuracy of approximately 71%. This underlines the high probability of a local optimum in the third area of the search space. Afterwards, in the last third of this phase, the accuracy quickly converges to another better-performing local optimum which yields a roughly 83% result;
- The final phase spans from epoch 150 to epoch 310 which is the limit for the number of epochs we allow for training. It is probably the most remarkable phase since the training accuracy converges linearly to the best performing local optimum. Starting from an initial training accuracy of 65%, we observe that the accuracy grows smoothly to an accuracy of approximately 85%.

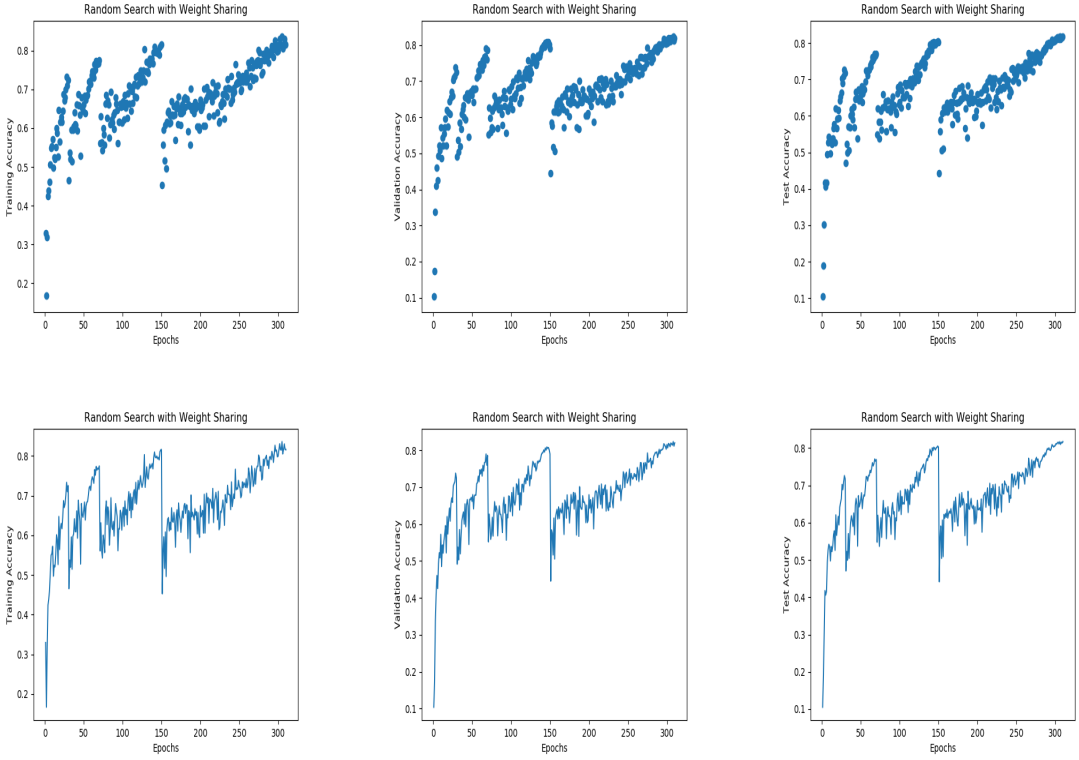


Figure 5.1: The controller RNN employs a Random Search with Weight Sharing Strategy. The plots show the training, validation and, respectively, test accuracy of the controller RNN plotted against the number of epochs.

We reproduced this experiment 3 times and observed that the training, validation and, respectively, test accuracies follow the pattern described above. Thus, we infer that the results presented in the previous paragraph are not under the influence of chance, but rather it demonstrates the power of Random Search with Weight Sharing. Another aspect worth mentioning is that the controller RNN is not affected by a few outliers in the search space that yielded significantly lower accuracies. It is important to recall that ENAS receives as an input a master architecture which contains numerous possible configurations. The master architecture directly influences the accuracies for the controller RNN since it already provides a search space designed in a way such that it contains good-performing configurations.

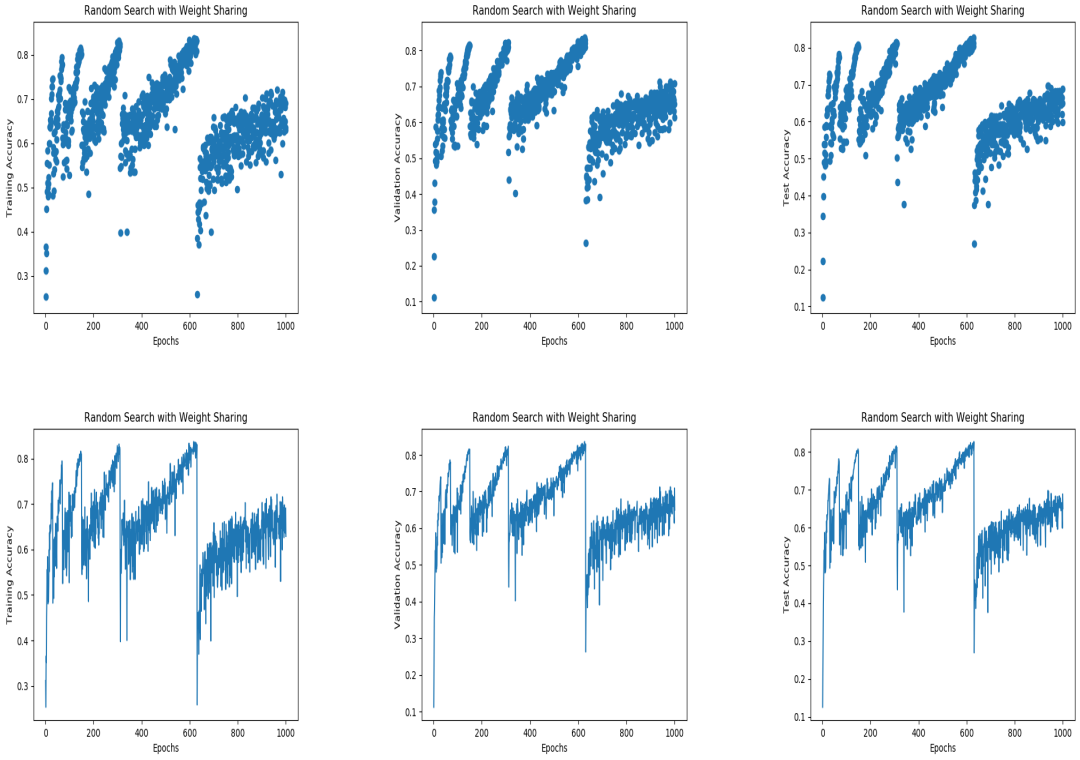


Figure 5.2: Random Search with Weight Sharing. Increasing the number of epochs for training to 1000 does not yield better performance when compared to results from the previous experiments with three times fewer epochs shown in Figure 5.1.

We also performed another experiment in which we increase the number of epochs to 1000. This idea was inspired by the results obtained in the fourth region of the search space. Analysing the plots in Figure 5.1 we cannot guarantee that we reached a plateau which means that the accuracies could grow even more. Consequently it is natural to allow more epochs for finding child models. The results for this experiment are shown in Figure 5.2. We note that, indeed, the results from the experiment in which we set the number of epochs to 1000 offer an insignificant improvement while increasing the computational costs three times. Thus, we infer that allowing more epochs for searching and the larger costs associated with it are not justified by the small performance improvement. Also the random controller goes on to explore another region of the search space, but with less satisfactory results.

5.3 Full Controller RNN

In our second set of experiments, we employ a search strategy that uses only the controller RNN to make sampling decisions by setting the ϵ parameter to 0. The learning rate of the controller RNN is set to 0.001. We allow 310 epochs for training, and constrain the controller RNN to sample CNNs with exactly 14 hidden layers, as per our previous experiment in which we performed a Random Search with Weight Sharing strategy. We start by discussing the plots for training, validation and test accuracies that are shown in Figure 5.3. We proceed in the same manner as we did with Random Search with Weight Sharing: we focus our discussion on the training accuracy which is influenced by the validation accuracy of the sampled child models. Naturally, the validation and test accuracies follow a very similar pattern. After training a child model to convergence, the controller RNN samples another child model from a different area of the search space taking into account its past sampling decisions. We again identify 4 separate regions of the search space that are exploited by the controller RNN:

- The first region is exploited for the first 30 epochs. We note how initially the controller RNN seems to get stuck in a local optimum yielding a training accuracy of around 65%. Afterwards, the training accuracy slowly converges to a better local optimum of 79%. As expected, this behaviour reflects in the plots for validation accuracy and training accuracy, as well;

- In the second region of the search space spanning from epoch 30 to epochs 75 the training accuracy grows almost linearly from 60% and converges to approximately 84% which corresponds to the local optimum for this particular area of the search space;

- After converging to 84%, the controller RNN samples another CNN architecture based on the results from the previous two exploitation phases. This can be observed by looking at the starting point of the third region of the search space which gives an initial training accuracy of 63%. The training accuracy converges to 86%, after which the controller RNN samples another child model;

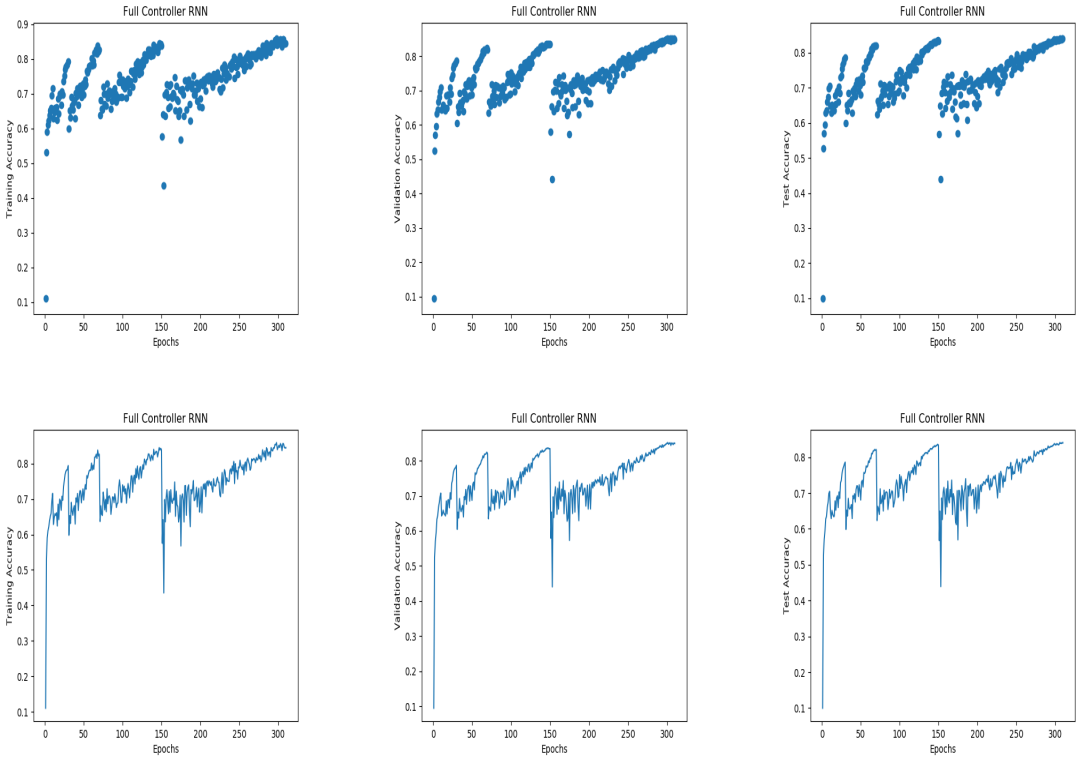


Figure 5.3: Sampling decisions are solely based on the actions of the controller RNN. Note the similar performances of this method and the Random Search with Weight Sharing approach.

- The fourth region of the search space is exploited from epoch 150 until reaching the threshold of 310 epochs. Here, the training accuracy of the controller RNN plateaus at 87%.

As we did in our experiments with Random Search with Weight Sharing, we try to extend the number of epochs allotted for training in order to confirm that our search strategy has reached its maximal potential. The plots in Figure 5.4 show the results for the experiment in which we increased the number of epochs to 1000. Compared to the results of the previous experiment (Figure 5.3), we observe that for the first four regions of the search space that are exploited the differences in performance are non-essential given that by increasing the number of epochs we also increased the costs. This serves as a confirmation that the performances obtained when allowing 310 epochs for training are very solid.

5.4 Full Controller RNN and Random Search with Weight Sharing Comparison

In Section 5.2 and 5.3, we performed an independent analysis of two search strategies, namely Random Search with Weight Sharing and a search strategy fully guided by the controller RNN. Investigating the validation and test accuracies given by the

two strategies, we notice that a search strategy solely based on the controller RNN to make sampling decisions does not perform significantly better than a Random Search with Weight Sharing strategy. This observation contradicts the claims of the authors of ENAS.

To illustrate our remark we provide the following argument. During the explo-

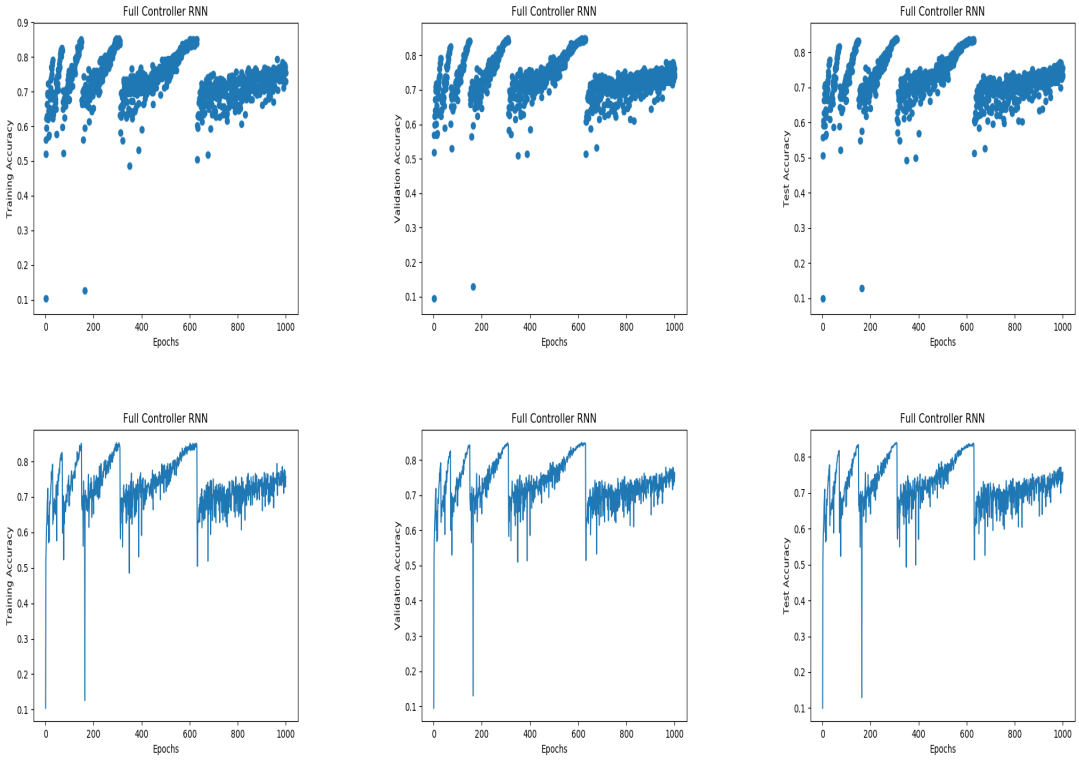


Figure 5.4: Sampling decisions are solely based on the actions of the controller RNN. Increasing the number of epochs to 1000 does not offer significant performance improvement when comparing with the results from the experiment where we set the number of epochs to 310 (Figure 5.3).

ration phase the controller RNN samples architectures in a manner which is *biased*. That is, given its past experience the controller RNN focuses on a particular region of the search space in which specific activations and skip connections are more frequent. This can guide the controller RNN into exploiting an area of the search space that does not contain performant architectural configurations. Thus, the performance of the CNNs sampled by the controller RNN heavily relies on the search space, *i.e.* master architecture, the controller RNN has at its disposal. This search space needs to be configured manually by the user and this usually constitutes one of the drawbacks of Neural Architecture Search methods. Intuitively, the bias in the sampling performed by the controller RNN can be explained as follows: the training stage in ENAS switches between updating the shared weights in the master architecture, *i.e.* the Directed Acyclic Graph, and updating the parameters of the controller RNN, thus the controller RNN is biased towards sampling the same architecture for the child model again since the shared weights of the corresponding model have already been improved. Thus, sharing the weights of the child models

is an artifact that leads to biased decisions of the controller RNN, but this is the trade-off we are willing to take in order to significantly shorten the training time and make this approach available to the ordinary ML practitioners. Furthermore, the argument goes to show the principal characteristic of the former search strategy: biased exploration. Making informed sampling decisions is the main trait of employing a search strategy that uses RL to guide the controller RNN through the search space and exploit particular areas of the space. Even though the two approaches, namely Random Search with Weight Sharing strategy and a Full Controller RNN strategy, both benefit from employing a common weight sharing scheme, they still constitute different search methods. However, Figure 5.1 and 5.3 show that both methods perform similarly. Thus, we argue against the initial claims of the authors of ENAS and show that the controller RNN does not do significantly better than Random Search with Weight Sharing.

5.5 Hybrid Search Strategy

The previous two experiments set the stage for our third approach in which we propose a hybrid search strategy in which the controller RNN navigates through the search space switching between sampling child models using the actions issued by itself and sampling using Random Search with Weight Sharing. We use the following configuration for this experiment. The learning rate is set to 0.001, and we use an ϵ-greedy strategy where we set the parameter ϵ to 0.5. To offer a fair ground for comparison, we constrain the controller RNN to sample from the same search space we used in the Random Search with Weight Sharing and sampling solely based on the controller RNN approaches. Furthermore, we make the controller RNN sample convolutional networks with exactly 14 hidden layers. We investigate the training, validation and test accuracies of the controller RNN which are shown in Figure 5.5. Again, we focus on the training accuracy since the other two follow a similar pattern. Analysing the graph, we note that the search space is partitioned into four regions:

- The first region of the search space that is exploited lasts until epoch 30. The controller RNN samples a sub-optimal configuration. This is reflected in the low training accuracy it achieves, reaching only about 68% accuracy at epoch 30 which shows that the controller RNN has sampled in a region with poorly performing architectures;

- The controller RNN samples another child model from another area of the search space spanning from epoch 30 to epoch 75. Although at first the controller RNN gets stuck in a local optimum yielding an accuracy of 62%, it then converges to an improved local optimum of 74%;

- Afterwards, the controller RNN samples from the third area of the search space which lasts from epoch 75 until epoch 150. Note that now the controller RNN sampled a region which contains good architectures since the training accuracy seems to grow quite fast to 79% where it looks to converge. However, there still seems to be room for improvement which is a plausible reason for

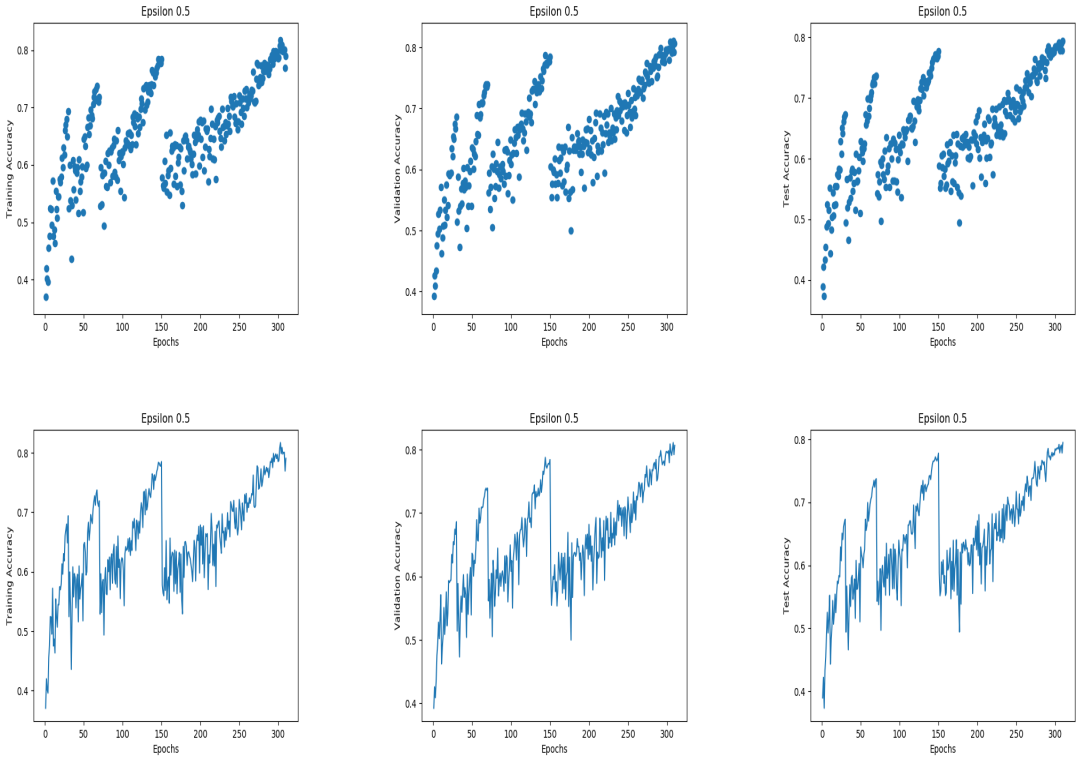


Figure 5.5: The controller RNN employs an ϵ-greedy Search Strategy, where the parameter ϵ is set to 0.5, switching between the two previous strategies.

us to run another experiment with more epochs. We discuss this idea later in this section;

- Finally, the last area of the search space is exploited from epoch 150 until reaching the threshold of 310 epochs where the training terminates. The controller RNN now samples a good performing architecture. This is reflected in the training accuracy the controller RNN achieves, namely 84%, since training the controller is guided by the validation accuracy of the child models. Again, we suspect that the training accuracy could grow even more if training was allowed for a larger number of epochs. We investigate this option in our next experiment.

As mentioned above, we have a good reason to perform another experiment in which we increase the number of epochs to 1000 (Figure 5.6) since the graphs corresponding to our last experiment (Figure 5.5) do not show that the accuracy plateaus especially in the third and fourth regions exploited by the controller RNN. However, we only observe a very slight increase in the accuracies which is clearly not satisfactory taking into account the large amount of computational costs involved. We also note that a fifth area of the search space is exploited, but the performance of architectural configurations from this region does not match the results from the fourth area of the search space.

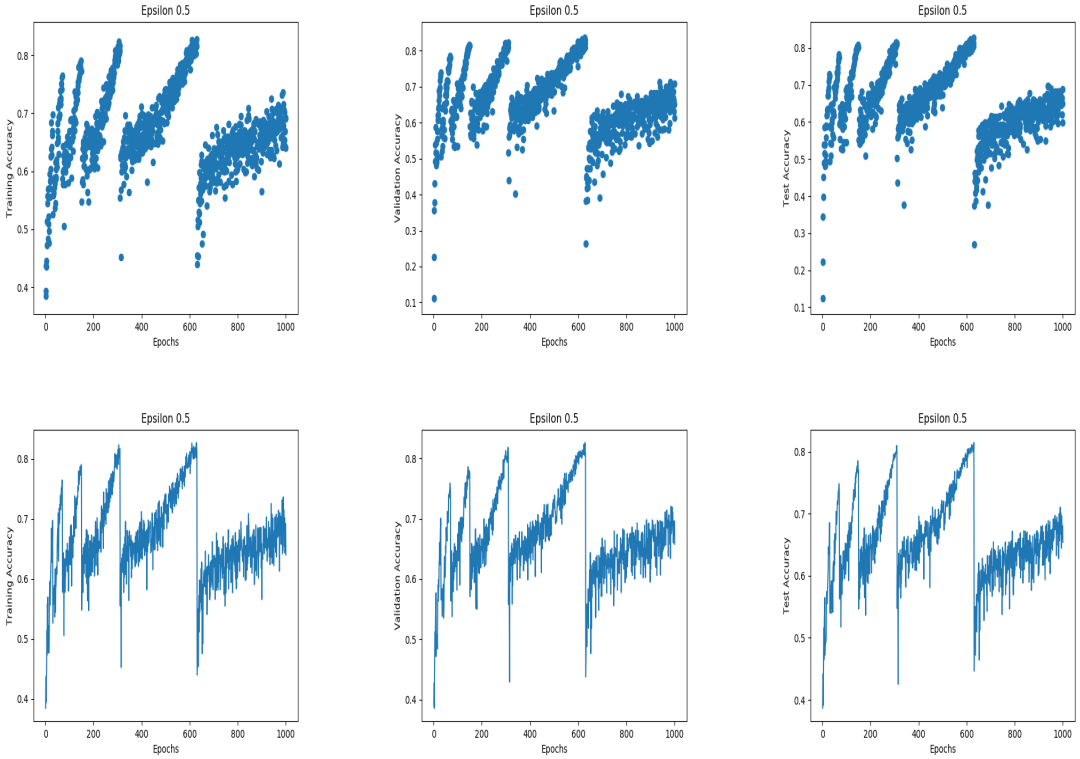


Figure 5.6: The controller RNN employs an ϵ-greedy Search Strategy, where the parameter ϵ is set to 0.5. Extending the number of epochs for training the controller RNN to 1000 insignificantly improves the performance considering the high increase in computational costs.

5.6 Summary

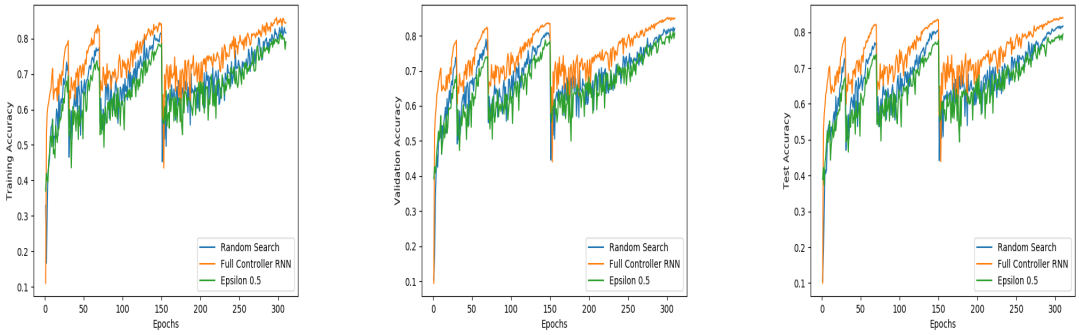


Figure 5.7: Comparison between three different search strategies in terms of training, validation and, respectively, test accuracy. The number of epochs is set to 310. The Full Controller RNN strategy performs better than the other two approaches, but not by significant margin.

In this section we evaluated the search strategy of the controller RNN and compared it to Random Search with Weight Sharing. We showed that the best child model sampled by the controller RNN is slightly better than the child model sampled by employing a Random Search with Weight Sharing strategy. However, we argue the Full Controller RNN strategy is not significantly better than Random Search with Weight Sharing, thus contradicting the claims of the authors of ENAS. Furthermore, we evaluated a hybrid method that combines the two aforementioned search strategies. A summary of the experiments discussed in this section is presented in Table 5.1. The Full Controller RNN approach slightly outperforms the other two methods, as shown in Figure 5.7 where we compare the three different search strategies. We have also performed more experiments with various search strategies in which the controller RNN switches between a Random Search with Weight Sharing strategy and a search strategy fully guided by the sampling decisions of the controller RNN according to the ϵ parameter. We refer the reader to the Appendix A for further details regarding these experiments.

Search Strategy (epochs)	Average Test Accuracy (%)	Best CNN Child Model Test Accuracy (%)
Random Search with Weight Sharing (310)	82.93	94.91
Full Controller RNN (310)	**85.07**	**95.07**
ϵ=0.5 (310)	80.73	93.68
Random Search with Weight Sharing (1000)	83.12	94.96
Full Controller RNN (1000)	**85.78**	**95.11**
ϵ=0.5 (1000)	82.21	93.75

Table 5.1: Summary of the experiments discussed in this section. For each of the search strategies the controller employs we report the average test accuracy of the controller RNN and the test accuracy of the best performing child model sampled by the controller RNN.

Chapter 6

Enhancing the Controller RNN

6.1 Extracting the Controller RNN

The structure and connectivity of a neural network can be expressed usually by a string of variable-length. Such a string can be generated by a Recurrent Neural Network which in the context of ENAS is the controller RNN. Training the architecture, *i.e.* child model, indicated by the variable-length string on the CIFAR-10 dataset leads to a validation accuracy that is used as a reward signal for the controller. The policy gradient used to update the weights of the controller is analogous to the Reinforce Rule [58] employed by the Neural Architecture Search with RL method [4] presented in Section 3.5. Thus, the controller RNN is trained to improve its sampling decisions by giving higher probabilities to architectures that yield a good validation accuracy. An essential trait of ENAS is the strong connection between the controller RNN and the sampled child models, as shown in Figure 6.1. In this section, we focus on extracting the controller and investigating its characteristics.

The controller RNN is a 2-layer LSTM with 100 hidden units. At the beginning, the learning rate is set to 0.0010. The weights of the controller RNN are initialised uniformly in the interval [-0.01, 0.01] and are trained using the Adam optimiser [13] for which the gradient is computed by employing the Reinforce Rule. The reward fed to the controller RNN is computed on the validation set so as to force the controller RNN to sample CNNs that have the capability to generalise on unseen data, as opposed to using the training accuracy of the child models as the reward signal which would cause the controller RNN to sample architectures that overfit the training data.

In the most basic case, the controller is capable of sampling *feed-forward* neural networks with only convolutional and pooling layers. In this case, the controller has to predict for each layer the number of filters, the filter height, the filter width, as well as stride height and stride width. At the starting point, the input for the controller RNN is an empty embedding. Afterwards, the controller RNN samples decisions using softmax classifiers, where the decision sampled in the previous step serves as input embedding for the current step. This process is repeated for each layer.

In order to increase the complexity of the search space, the controller RNN can

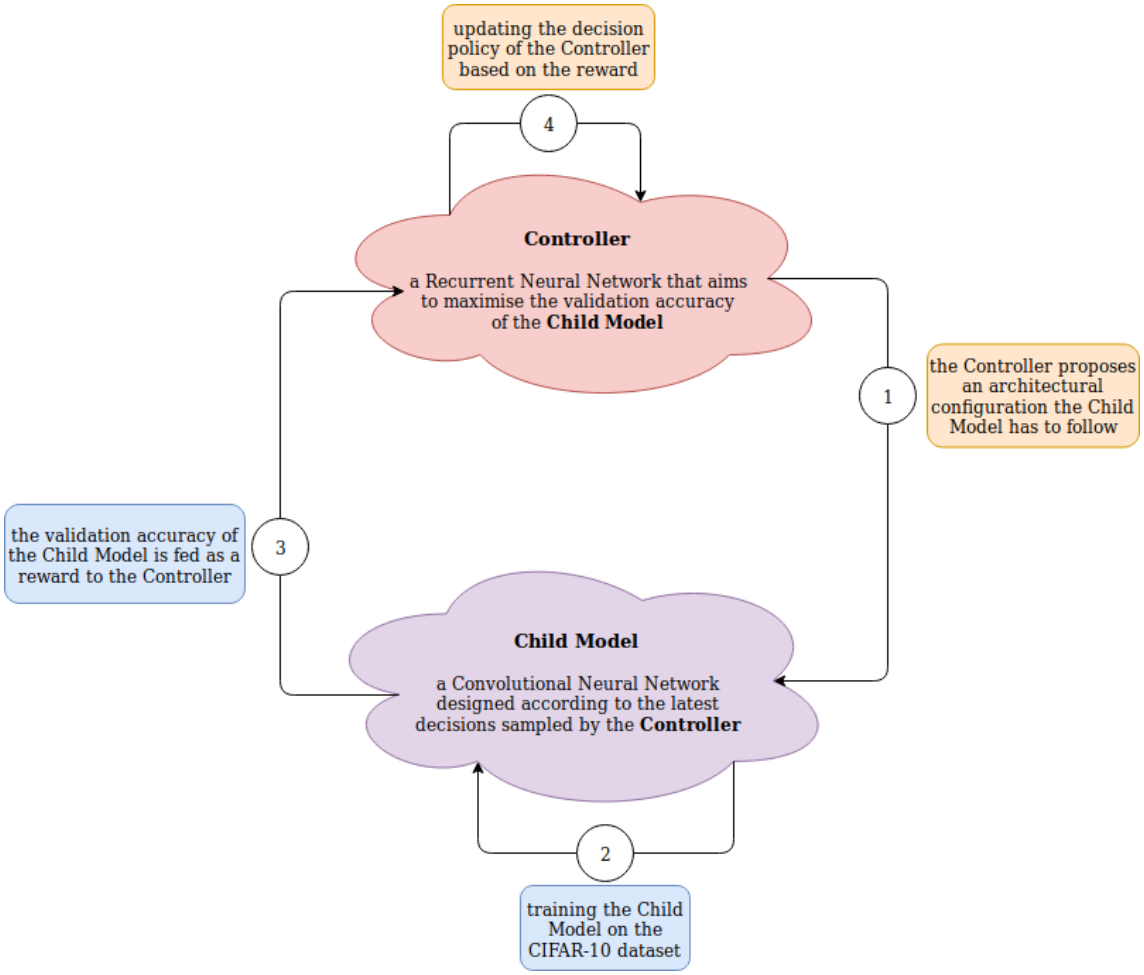


Figure 6.1: Interaction between the controller RNN and the child model

also propose skip connections which are part of modern manually designed architectures such as ResNet. To allow the controller RNN to sample such connections, at each layer we need to establish an anchor point which encompasses the content-based sigmoid of each of the previous layers that need to be connected. A similar approach is employed in Neural Architecture Search with RL (NAS) [4]. Each content-based sigmoid determines the probability that an earlier layer in the child model serves as input to the current layer together with preceding layer. It is a function that encapsulates the current state of the controller RNN and the states of the previous anchor points. Mathematically, this can be expressed as:

$$P(\text{layer } A \text{ is an input to layer } B) = sigmoid(v^T \cdot tanh(W_A \cdot s_A + W_B \cdot s_B))$$

where A ranges from 0 to B - 1 and s_A is the state of the controller RNN at the anchor point belonging to the A-th layer, while s_B is the state of the controller RNN at the anchor point of the B-th layer. The equation mentioned above constitutes the main concept behind how the controller RNN samples skip connections. The anchor point from a preceding layer, *i.e.* s_A, is multiplied by the corresponding weight matrix which is a learnable parameter. Then, this product gets added with the product of the anchor point at the current layer, *i.e.* s_B, and the corresponding weight matrix. The resulting sum is then used as an argument to a sigmoid (logit) function that maps the sum in the interval (0, 1), thus giving the probability of a previous layer to be connected to the current layer using skip connections. This process is repeated for all preceding layers. Afterwards, samples are drawn from a multinomial distribution to decide which of the previous layers are going to be connected to the current layer. Figure 6.2 shows how the controller RNN uses anchor points to generate skip connections. If a layer has multiple input layers due to skip connections, then the input layers are concatenated along the channel dimension. In the case that the input layers to be joined are incompatible, namely they have different sizes, then the smaller layers are padded with zeros so as to force the concatenated input layers to have the same sizes.

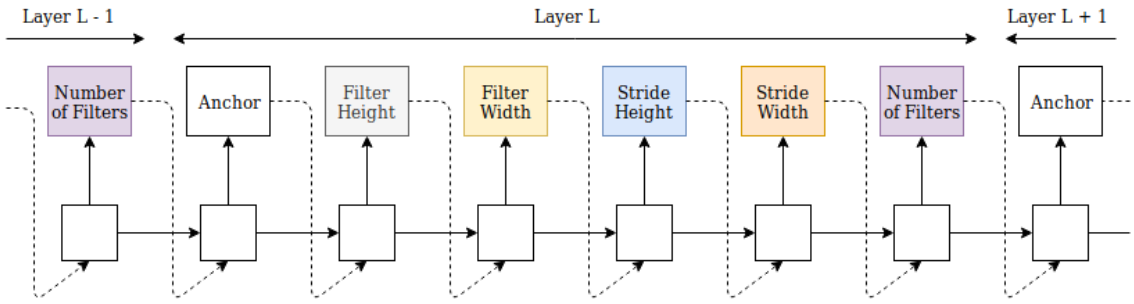


Figure 6.2: The controller RNN uses anchor points to generate skip connections.

It is important to understand how the controller RNN builds a CNN child model that is going to be trained. In the paragraph above we investigated how the controller RNN samples skip connections. The other task of the controller RNN is to decide what operation to use for each layer. This is done sequentially, layer by layer. Recall that we forced the controller RNN to build child models consisting of exactly 14 hidden layers and that the search space is determined by the master

architecture, *i.e.* the Directed Acyclic Graph (DAG). For each layer the controller RNN samples a node from the master architecture. A softmax classifier is used to assign each node the probability of being selected as the operation to be performed at the current layer. Then, the controller RNN draws a sample from the resulting multinomial distribution and, thus, records the operation to use for the current layer.

Up to this point for the two tasks it has to carry out, the controller RNN samples actions from multinomial distributions. However, we have to emphasise the importance of training the controller RNN using the validation accuracy from the selected child models. In ENAS, the controller RNN models the selection of a new architecture a_t based on the following equation:

$$P(a_t) \approx P(a_t|a_1, ..., a_{t-1})$$

where a_1, ..., a_{t-1} represent convolutional architectures previously sampled by the controller RNN. We observe that this assumption is too relaxed in the sense that it makes the controller RNN take into account architectures from the first sampling trials which are naturally poor in terms of performance since the controller has not been trained properly at the time of making these early sampling decisions. Although the controller RNN eventually manages to learn to sample better and better CNNs, the sub-performing architectures sampled initially still negatively impact the controller RNN especially in the early stages of training. In Section 6.2 we propose a solution to tackle this problem.

To demonstrate that the controller RNN is indeed improving its sampling decisions we set out two experiments. In the first experiment we extract the controller RNN and decouple it from the child model. We replace the reward signal fed to the controller RNN, more specifically, instead of using the validation accuracy of the child models, we replace it with a small constant value. Thus, the weights of the controller RNN do not get updated properly which in turn leads to poor decision making. To demonstrate this, we run the first experiment for 20 epoch. The training, validation and test accuracies of the controller RNN are shown in Figure 6.3. Analysing these plots, we observe how he controller RNN is not capable of learning anything meaningful since in the case of the training accuracy it merely scatters in the interval [0.10, 0.18]. Naturally, the same phenomenon occurs in the case of the validation and test accuracies with the results being even worse. The resulting child model has 9 out of the 14 layers set to the identity operation and no skip connections are employed. This further shows that in this setup the controller RNN is not able to learn to sample performant architectures. We highlight the importance of the reward by comparing between the results from Figure 6.3 and the results in Figure 6.4 where the reward signal is based on the validation accuracy of the child model. Clearly, in this second experiment we observe how the accuracies grow in each of the regions of the search space that are exploited. This indicates that the controller RNN clearly learns to sample better performing architectures over time.

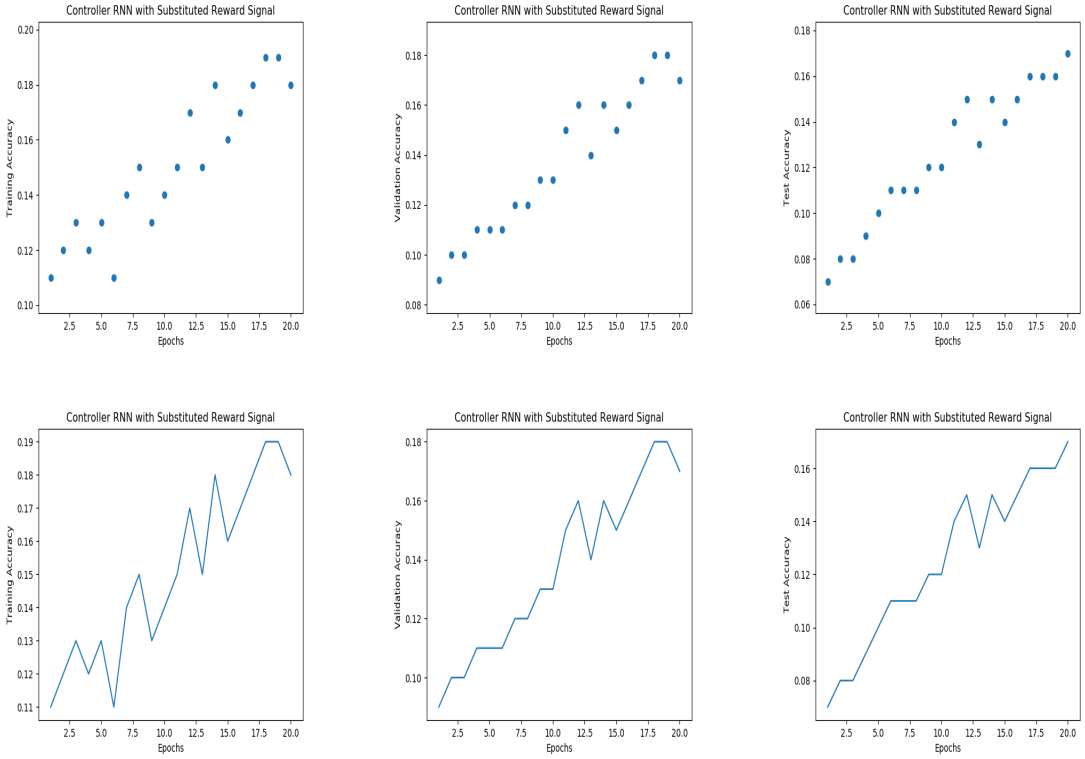


Figure 6.3: Substituting the reward signal with a small constant value affects the sampling decisions of the controller RNN. It shows that the controller RNN is dependent on a meaningful reward signal to improve its searching over time. Note the very small accuracies reached in each of the plots. The training accuracy is always less than 19%. Conversely, the experiment also demonstrates that in the general case the controller RNN learns based on previous experience.

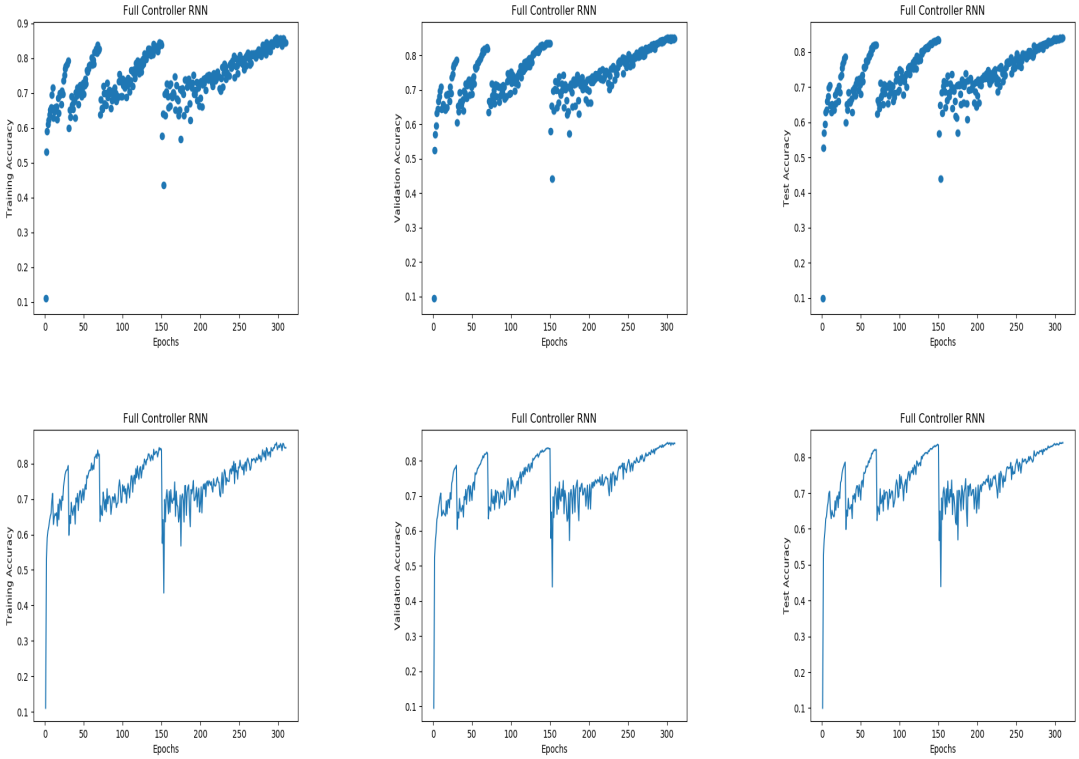


Figure 6.4: The choice of reward signal is crucial for the controller RNN to learn to sample better child models. The growing accuracy for each of the 4 regions of the search space that are exploited indicate that the controller RNN improves its decision making with time.

6.2 A Novel Approach: ENAS with Early Stopping

6.2.1 Overview

The core mechanism behind the controller RNN is the reward signal which helps the controller navigate through the search space. Building on the investigation presented in Section 6.1, we establish that it is highly relevant to encourage the controller RNN to sample better performing architectures from the initial stages of training. In this section, we propose a novel approach to improve the sampling decisions of the controller RNN. We call our method *ENAS with Early Stopping*.

Conceptually, when the controller RNN explores a new region of the search space we want to predict whether this region contains good performing configurations for CNNs. After the controller RNN decides on a particular configuration, it uses the resulting validation accuracy of the sampled child model to decide whether this particular region of the search space is worth exploiting further. If the validation accuracy, which is fed as reward to the controller, is low we want to stop training the child model and, thus, the controller RNN should choose to sample another region of the search space. We recall that the controller RNN keeps track of all sampled architectures, and the low reward it receives from underperforming child models forces the controller RNN to sample a different architecture from a different area of the search space. We repeat this process until the controller RNN finds a satisfactory configuration of the child model.

The main advantage of our novel approach is that it allows the controller RNN to learn to sample better child models even from the early stages of training. Since the decisions of the controller RNN are strongly influenced by its past samplings, the controller ends up discovering superior CNNs.

In the early stages of training, we want to encourage the controller RNN to explore the search space more efficiently. We do this by establishing a lower bound on the validation accuracy of the sampled child models. Specifically, if the validation accuracy of the child model is less than the lower bound we established, then the controller RNN samples from another area of the search space taking into account the past decisions. We highlight that at the beginning the controller RNN has no knowledge of the search space. We could address this issue by injecting some prior knowledge so as to influence the initial sampling decisions of the controller RNN, however this would lead to highly biased exploration. Thus, we propose two modified search strategies: *ENAS with Early Stopping v1* and *ENAS with Early Stopping v2*. In both approaches, the search strategy is fully guided by the controller RNN, presented in Section 6.1, combined with different early stopping techniques. Furthermore, since initially no prior knowledge is provided to the controller RNN, we enforce the validation accuracy threshold only after the controller samples the first child model. Then, the controller RNN samples different architectures until finding a configuration that produces a validation accuracy not less than the threshold. As training progresses, the controller RNN enhances its decision making and designs improved child models.

6.2.2 ENAS with Early Stopping v1

In this approach we set the lower bound for the validation accuracy of a sampled child model to 45%. Throughout training the controller RNN we do not modify this threshold. We chose to set the threshold to 45% for empirical reasons: we attempted to increase the threshold, but this led to a very slow search strategy. We note that for each region of the search space the controller samples from, receiving a modest reward in the early stages of the exploitation phase, yet still greater than 45%, does not necessarily mean that a region contains only subperforming configurations. This is the reason why in this approach we allow the controller RNN to further exploit such regions.

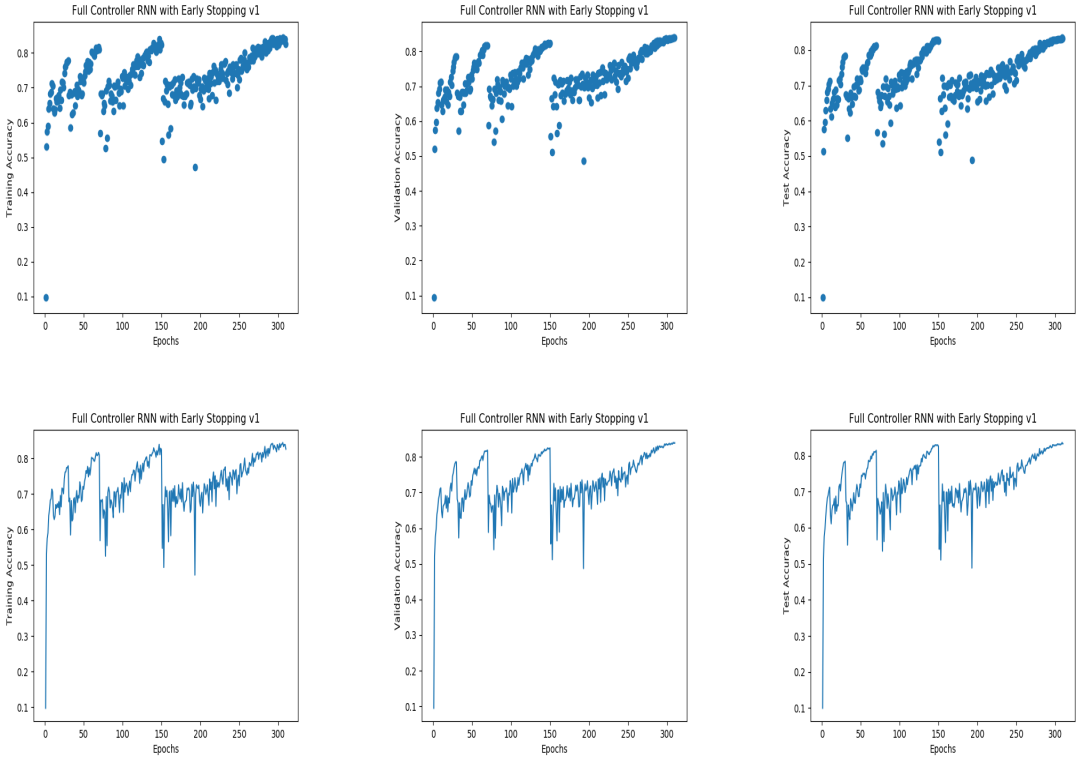


Figure 6.5: *ENAS with Early Stopping v1.* We set the lower bound for the validation accuracy of a child model to 45% to avoid sampling poorly performing configurations. While the results are promising, we underline that when the controller RNN samples from a new region of the search space there is a significant drop in the accuracies. A reason behind this phenomenon is that establishing a threshold and not modifying it as training progresses is not a strong enough early stopping policy.

We experiment with this new technique using the same setup for the controller as in the *Full Controller RNN* search strategy presented in Section 5.3. As per our previous experiments discussed in Chapter 5, we expect a strong connection between the plots for training, validation and, respectively, test accuracy. Analysing the plots in Figure 6.5, we observe that the search space is split into four regions. We focus on the training accuracy, however a similar pattern emerges in the other two graph corresponding to validation and, respectively, test accuracy:

- The first region is exploited until epoch 35. By employing our early stopping technique, the controller RNN avoids getting trapped in a local optimum corresponding to roughly 66% training accuracy, and quickly converges to 79%. This goes to show that indeed the controller is encouraged to sample better configuration even from the early stages of training.

- The second region of the search space spans from epoch 35 to epoch 75. Here, the training accuracy of the controller grows almost linearly to 82% where it converges to a local optimum.

- Then, the controller RNN exploits a new area of the search space until epoch 200. The training accuracy grows to 84% reaching a local optimum at this stage, after which the controller samples another area of the search space.

- The fourth region of the search space spans from epoch 200 until the end of training. Here, the controller RNN reaches a training accuracy of 86% which corresponds to a better local optimum of the search space when compared to the training accuracies achieved by the controller on the previously sampled regions. This is another confirmation that the controller RNN samples better configurations as training progresses.

It is important to observe that the accuracies never drop below 50% since the improved controller RNN is able to avoid sampling areas of the search space that contain less competitive configurations for CNNs. However, we note that when the controller RNN samples from a new region, the accuracies drastically drop in the very early stages of exploiting the new region. This fact can be visualised by investigating the scatter plots in Figure 6.5, specifically we notice that at the beginning of each new region of the search space there is a small number of points corresponding to lower accuracies than expected. This can occur because our early stopping strategy is not strict enough.

6.2.3 ENAS with Early Stopping v2

In the previous strategy we did not modify the reward threshold which was kept at 45%. Based on the results from the previous experiment we speculate that *ENAS with Early Stopping v1* is not strong enough. Consequently, we propose a stricter early stopping policy in which we decide to adjust the reward threshold as training progresses. Initially we set the threshold to 45% and gradually increase it by 1% every 15 epochs, giving a final threshold of 65%.

The experiment with this novel approach is depicted in Figure 6.6. Due to the similar patterns followed by the training, validation and test accuracies of the controller RNN, we analyse the results for the training accuracy. We can partition the graph in four separate regions of the search space exploited by the controller RNN:

- In the first region extends up to epoch 35. We emphasise the high initial training accuracy of the controller which is approximately 60%. Then, the training accuracy grows rapidly to 80% which corresponds to a local optimum in this particular area of the search space;

- The controller RNN samples a child model from another area of the search space. The aforementioned observation regarding the high initial accuracy holds in this instance as well, attesting the power of this novel approach with early stopping. The second area of the search space exploited by the controller spans from epoch 35 to epoch 75. In this region the controller RNN converges to a training accuracy of 85%;

- Based on the previously sampled child models, the controller decides on the third area of the search space. Indeed, this region contains good architectural configurations, a fact that is confirmed by the high accuracy achieved by the controller. It converges to a training accuracy of 87%;

- The fourth region of the search space spans from epoch 200 until the upper limit for the number of epochs allowed for searching a suitable CNN which is set to 310. The controller RNN demonstrates that it has honed its decision making abilities by converging to 88% training accuracy, the highest accuracy in all of the four regions exploited.

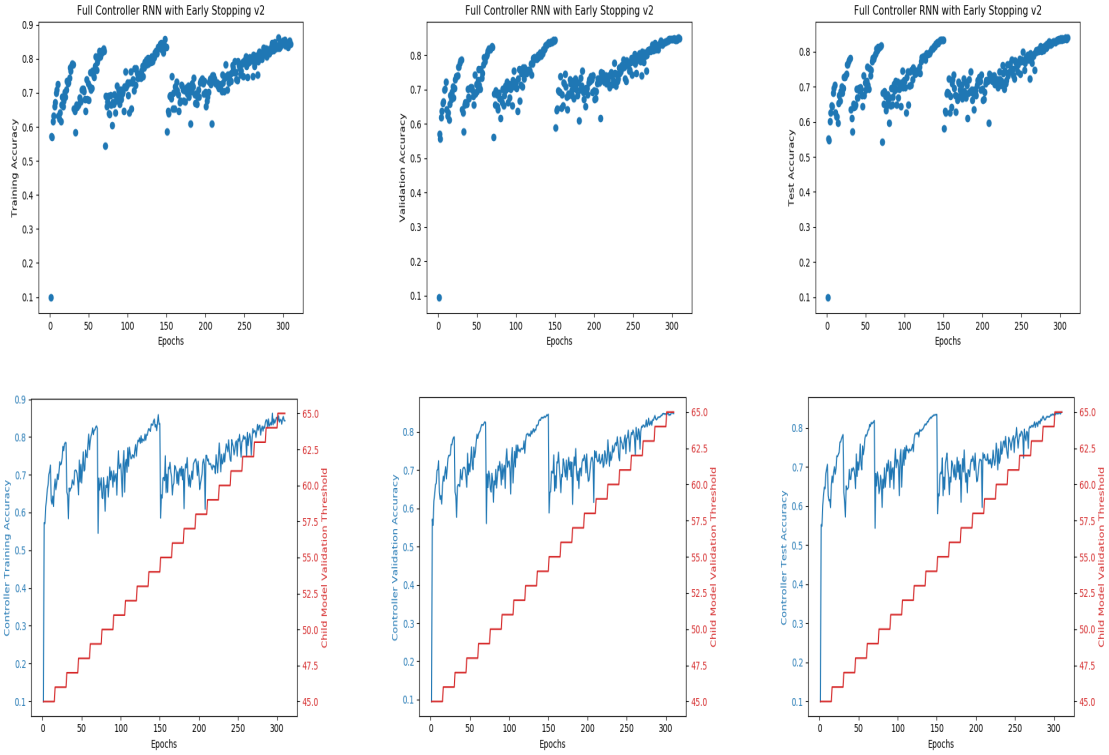


Figure 6.6: *ENAS with Early Stopping v2.* Building on top of the first technique, we initially set the lower bound for the validation accuracy of the sampled child models to 45%. As training progresses, we gradually alter this threshold, increasing it by 1% every 15 epochs. This provides the controller RNN with the ability to avoid sampling architectures from areas of the search space with underperforming configurations.

We notice that using this approach the accuracies never drop below 55% which is a great improvement when compared to the previous early stopping strategy. The

best performing child model sampled by the controller RNN using our refined early stopping approach yielded a test accuracy of 95.16%, which outperforms the 95.07% test accuracy obtained from the best performing child model sampled using solely the Full Controller strategy.

6.3 Summary

We started this chapter by doing an in-depth investigation of the controller RNN employed by ENAS. The controller RNN is responsible for designing CNNs: for each layer in a child model, the controller RNN samples what operation to employ, as well as what skip connections to use. We performed experiments that show how the controller RNN learns to sample better architectural configurations and highlighted the significance of the reward signal.

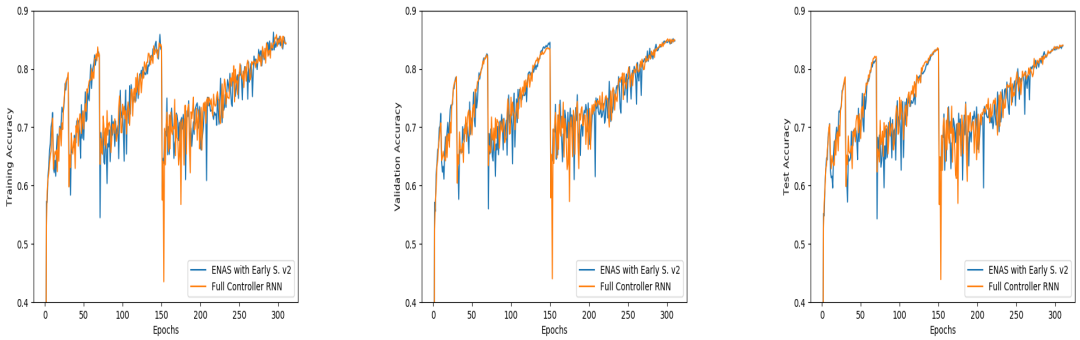


Figure 6.7: Comparing between two search strategies: ENAS with Early Stopping v2 and Full Controller RNN.

Understanding the subtleties behind the controller RNN and the reward signal are crucial in the development of our early stopping approaches. In *ENAS with Early Stopping v1* we set a constant lower bound for the validation accuracy of the sampled child models. Empirically, we showed that this constraint was not strong enough to encourage the controller RNN to sample better architectural configurations from the early stages of training. Consequently, we devised an enhanced early stopping strategy, *ENAS with Early Stopping v2* in which we gradually increase the lower bound for the reward as training progresses. The results for training, validation and test accuracies are better than the ones belonging to the Full Controller RNN method, as depicted in Figure 6.7. Using the controller in combination with this novel technique led to the sampling of a child model that outperformed the CNN sampled by the Full Controller search strategy. We report the performances of the early stopping methods in Table 6.1.

Search Strategy (epochs)	Average Test Accuracy (%)	Best CNN Child Model Test Accuracy (%)
ENAS with Early Stopping v1 (310)	83.34	94.96
ENAS† (310)	85.07	95.07
ENAS with Early Stopping v2 (310)	**85.81**	**95.16**
ENAS[5] (310)	N/A	96.15

Table 6.1: Summary of the experiments for our early stopping methods. For ENAS† we report the results achieved by running our own experiments. For each of the search strategies the controller employs we report the average test accuracy of the controller RNN and the test accuracy of the best performing child model sampled by the controller RNN.

Chapter 7

Conclusions and Future Work

In this chapter we reiterate the main points discussed throughout the previous chapters, then present ideas for future work.

We started by highlighting some of the best manually designed architectures and the main motifs behind them, while at the same time emphasising the complexity of crafting such networks and the need for automation. Therefore, we investigated several hyper-parameter optimisation approaches: Grid and Random Search, Bayesian Optimisation, Genetic Algorithms and Reinforcement Learning (RL). We chose Neural Architecture Search (NAS) with RL as our main focus of discussion for the next chapters. In spite of the strong empirical performance of NAS, this method is extremely time-consuming and computationally expensive, using 450 GPUs for 3-4 days for a single experiment.

An explanation for the computational bottleneck of NAS is training every child model sampled by the controller RNN to convergence and throwing away all the trained weights after updating the controller. The ENAS approach overcomes this disadvantage by sharing the weights among the child models so that training each child model from scratch is not required anymore. Needing only a single GPU and less than 17 hours to find a performant CNN, this method can also be used by the average practitioner. This is the reason we chose to further explore and enhance this approach in the rest of our work.

The controller RNN is responsible for designing CNNs, namely what operations to employ at each layer and what skip connections to use. After analysing the search space of ENAS and how the controller RNN samples architectural configurations from the master architecture, *i.e.* Directed Acyclic Graph, we explained the concept of weight sharing and underlined the tendencies encountered in the sampling decisions of the controller RNN. We also highlighted the lack of reproducibility of the results reported by the authors of ENAS. Our best performing CNN yielded a test accuracy of 95.07% on the CIFAR-10 dataset. Furthermore, we discussed how modifying the configuration of our best performing CNN affects its performance.

We then evaluated different search strategies of the controller. We showed that the best child model sampled using a search strategy fully guided by the controller RNN is slightly better than the child model sampled by employing a Random Search

with Weight Sharing strategy. However, we argued the Full Controller RNN strategy is not significantly better than Random Search with Weight Sharing, thus contradicting the claims of the authors of ENAS. We also experimented with a new, hybrid search method that combines the two aforementioned search strategies.

We performed experiments that show how the controller RNN learns to sample better architectural configurations and highlighted the significance of the reward signal. We established that it is highly relevant to encourage the controller to sample better architectures even in the early stages of training. Consequently, we developed a novel approach, *ENAS with Early Stopping*, in which we enforce a reward threshold with the aim of improving the sampling decisions of the controller RNN. Our best performing CNN sampled using this approach achieved a 95.16% test accuracy on CIFAR-10, thus outperforming the child model sampled by the ENAS controller.

As far as ideas for future work are concerned we propose extending the search space available to the controller RNN by adding more operations, *e.g.* Dropout layer, this can lead to even better child models. Another suggestion worth exploring is replacing the 5x5 convolutions with consecutive 3x3 convolutions which may help reduce the number of parameters in the network.

Appendix A

Hybrid Search Strategies

Here we list the experiments we performed for different hybrid methods we investigated. The higher the value for the ϵ parameter, the more the controller RNN uses the Random Search with Weight Sharing strategy. Notice that the controller RNN learns better, *i.e.* yields higher training, validation and test accuracies, when the sampling is done preponderantly based on the decisions issued by the controller RNN. However, this does not mean that the best performing Convolutional Neural Network sampled by such strategies are better than the architectures sampled using a search strategy that has more randomness in its sampling.

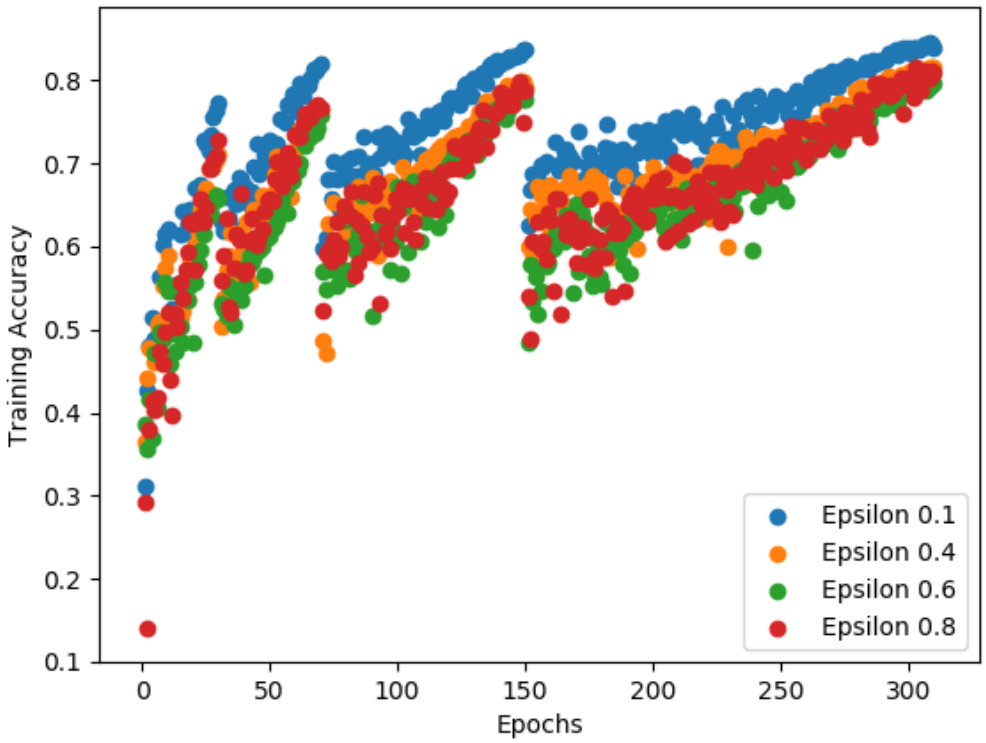


Figure A.1: Scatter plot for different ϵ-greedy search strategies employed by the controller RNN, where the parameter ϵ is set to 0.1, 0.4, 0.6, 0.8.

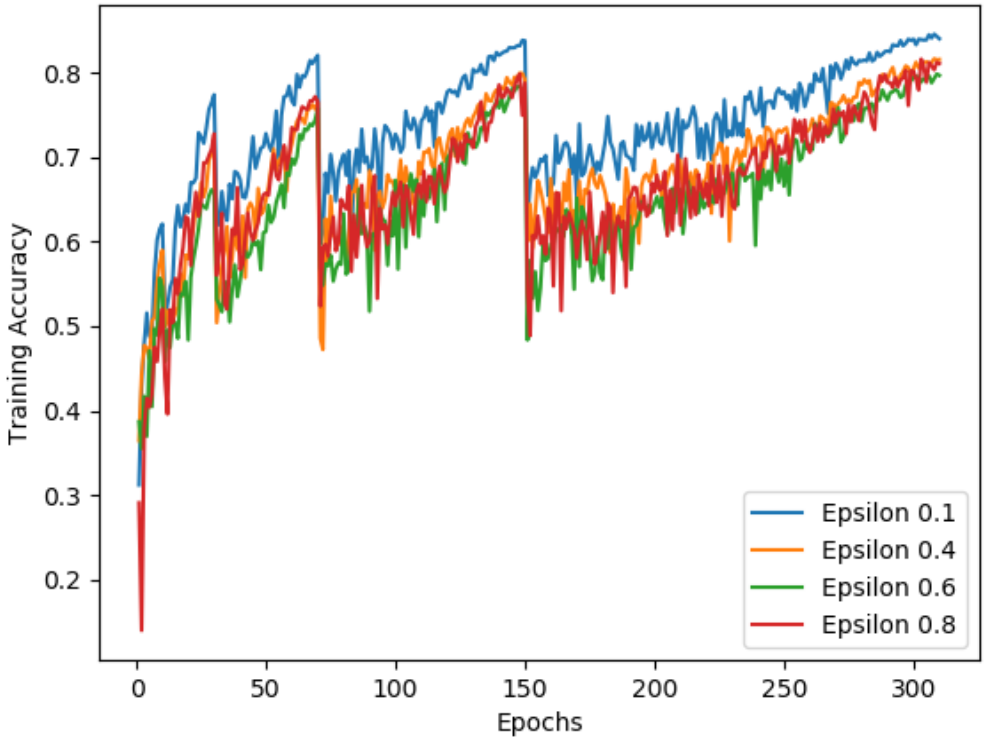


Figure A.2: Line plot for different ϵ-greedy search strategies employed by the controller RNN, where the parameter ϵ is set to 0.1, 0.4, 0.6, 0.8.

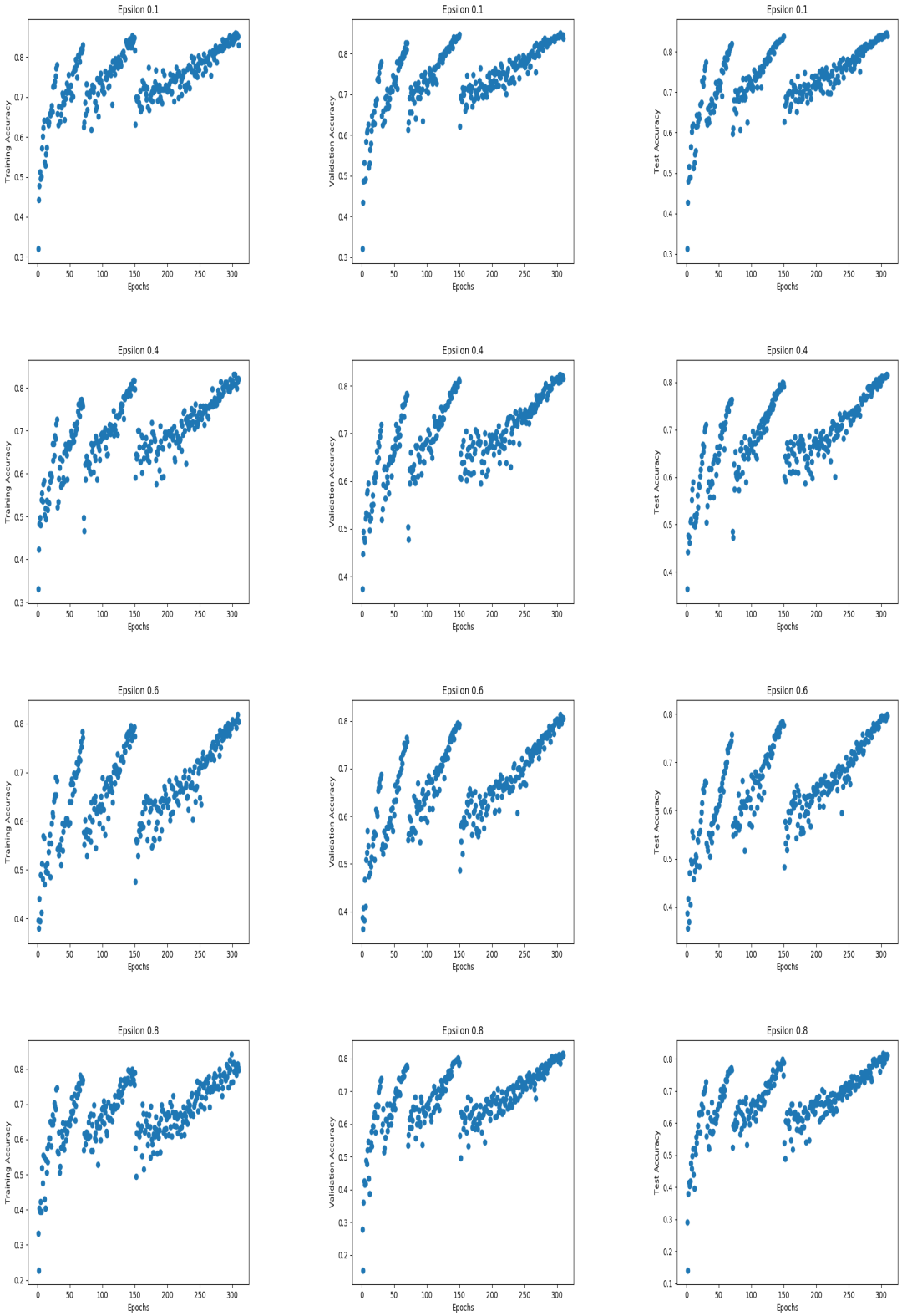


Figure A.3: Scatter plots for different ϵ-greedy search strategies employed by the controller RNN, where the parameter ϵ is set to (from top row to bottom row) 0.1, 0.4, 0.6, 0.8.

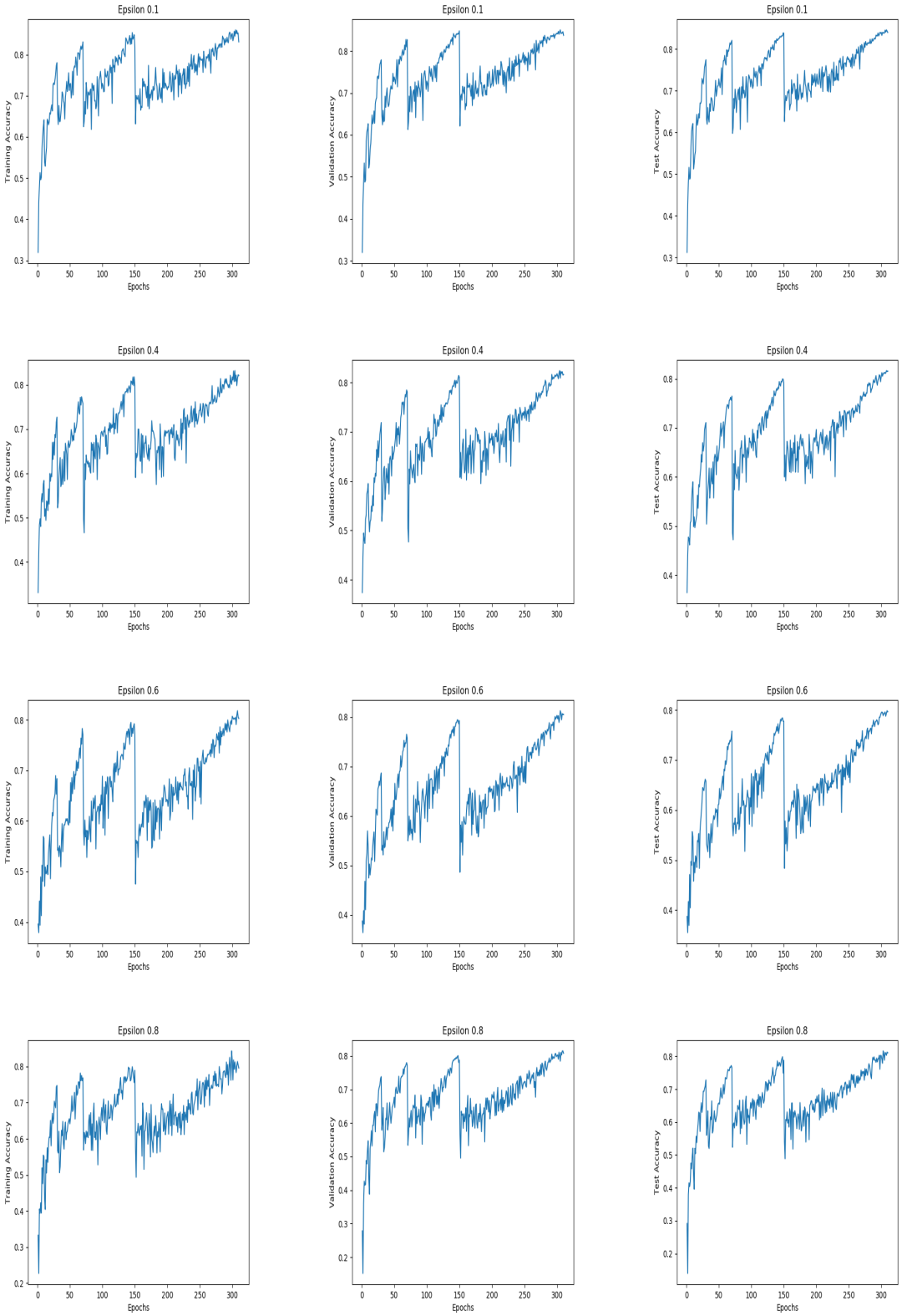


Figure A.4: Line plots for different ϵ-greedy search strategies employed by the controller RNN, where the parameter ϵ is set to (from top row to bottom row) 0.1, 0.4, 0.6, 0.8.

Bibliography

[1] Suvrit Sra, Sebastian Nowozin, and Stephen J. Wright. *Optimization for Machine Learning*, volume 2, pages 1–3. MIT Press, 2012.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[4] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. *arXiv e-prints*, art. arXiv:1611.01578, November 2016.

[5] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient Neural Architecture Search via Parameter Sharing. *arXiv e-prints*, art. arXiv:1802.03268, February 2018.

[6] IBM ILOG CPLEX optimization studio. URL https://www.ibm.com/products/ilog-cplex-optimization-studio?mhq=cplex&mhsrc=ibmsearch_a. Accessed: 16/01/2019.

[7] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, Jan 2016. ISSN 0018-9219. doi: 10.1109/JPROC.2015.2494218.

[8] Marc Claesen and Bart De Moor. Hyperparameter Search in Machine Learning. *arXiv e-prints*, art. arXiv:1502.02127, February 2015.

[9] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

[10] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable Architecture Search. *arXiv e-prints*, art. arXiv:1806.09055, Jun 2018.

[11] Yoon Kim. Convolutional Neural Networks for Sentence Classification. *arXiv e-prints*, art. arXiv:1408.5882, Aug 2014.

[12] Wen-tau Yih, Xiaodong He, and Christopher Meek. Semantic parsing for single-relation question answering. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 643–648, Baltimore, Maryland, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/P14-2105.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, art. arXiv:1412.6980, Dec 2014.

[14] Rachel Ward, Xiaoxia Wu, and Leon Bottou. AdaGrad stepsizes: Sharp convergence over nonconvex landscapes, from any initialization. *arXiv e-prints*, art. arXiv:1806.01811, Jun 2018.

[15] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv e-prints*, art. arXiv:1212.5701, Dec 2012.

[16] Difan Zou, Yuan Cao, Dongruo Zhou, and Quanquan Gu. Stochastic Gradient Descent Optimizes Over-parameterized Deep ReLU Networks. *arXiv e-prints*, art. arXiv:1811.08888, Nov 2018.

[17] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, art. arXiv:1502.03167, Feb 2015.

[18] Dan Cireşan, Ueli Meier, and Juergen Schmidhuber. Multi-column Deep Neural Networks for Image Classification. *arXiv e-prints*, art. arXiv:1202.2745, Feb 2012.

[19] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network. *arXiv e-prints*, art. arXiv:1312.4400, Dec 2013.

[20] Ashwin Bhandare, Maithili Bhide, Pranav Gokhale, and Rohan Chandavarkar. Applications of convolutional neural networks. *International Journal of Computer Science and Information Technologies, Vol. 7 (5)*, pages 2–6, 2016.

[21] Tobias Hinz, Nicolás Navarro-Guerrero, Sven Magg, and Stefan Wermter. Speeding up the Hyperparameter Optimization of Deep Convolutional Neural Networks. *arXiv e-prints*, art. arXiv:1807.07362, July 2018.

[22] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing Neural Network Architectures using Reinforcement Learning. *arXiv e-prints*, art. arXiv:1611.02167, November 2016.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[24] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[25] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-prints*, art. arXiv:1409.1556, Sep 2014.

[26] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[27] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. *arXiv e-prints*, art. arXiv:1409.4842, Sep 2014.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, art. arXiv:1512.03385, Dec 2015.

[29] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv e-prints*, art. arXiv:1602.07360, Feb 2016.

[30] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints*, art. arXiv:1704.04861, Apr 2017.

[31] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks. *arXiv e-prints*, art. arXiv:1608.06993, Aug 2016.

[32] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *arXiv e-prints*, art. arXiv:1610.02357, Oct 2016.

[33] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86 (11):2278–2324, 1998.

[34] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1487–1495, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4887-4.

[35] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. *arXiv e-prints*, art. arXiv:1605.07079, May 2016.

[36] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv e-prints*, art. arXiv:1603.06560, March 2016.

[37] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[38] Ian Dewancker, Michael McCourt, and Scott Clark. Bayesian optimization primer. URL https://app.sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf. Accessed: 19/01/2019.

[39] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *arXiv e-prints*, art. arXiv:1012.2599, December 2010.

[40] Meghana Ravikumar. Let's talk bayesian optimisation. URL https://mlconf.com/lets-talk-bayesian-optimization/, 2018. Accessed: 22/01/2019.

[41] Jasper Snoek. Spearmint. URL https://github.com/JasperSnoek/spearmint. Accessed: 22/01/2019.

[42] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[43] Chao Qin, Diego Klabjan, and Daniel Russo. Improving the expected improvement algorithm. In *Advances in Neural Information Processing Systems*, pages 5381–5391, 2017.

[44] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, Nando De Freitas, et al. Bayesian optimization in high dimensions via random embeddings. pages 4–6.

[45] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-Scale Evolution of Image Classifiers. *arXiv e-prints*, art. arXiv:1703.01041, March 2017.

[46] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural Architecture Search: A Survey. *arXiv e-prints*, art. arXiv:1808.05377, Aug 2018.

[47] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, HLT '94, pages 114–119, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics. ISBN 1-55860-357-3. Accessed: 17/01/2019.

[48] CIFAR-10 and CIFAR-100 datasets. URL https://www.cs.toronto.edu/~kriz/cifar.html. Accessed: 17/01/2019.

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

[50] J. Bergstra, D. Yamins, and D. D. Cox. Making a Science of Model Search. *arXiv e-prints*, art. arXiv:1209.5111, Sep 2012.

[51] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3460–3468. AAAI Press, 2015. ISBN 978-1-57735-738-4.

[52] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *arXiv e-prints*, art. arXiv:1703.03864, Mar 2017.

[53] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv e-prints*, art. arXiv:1712.06567, Dec 2017.

[54] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. *arXiv e-prints*, art. arXiv:1802.01548, Feb 2018.

[55] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. Back to Basics: Benchmarking Canonical Evolution Strategies for Playing Atari. *arXiv e-prints*, art. arXiv:1802.08842, Feb 2018.

[56] Arber Zela, Aaron Klein, Stefan Falkner, and Frank Hutter. Towards Automated Deep Learning: Efficient Joint Neural Architecture and Hyperparameter Search. *arXiv e-prints*, art. arXiv:1807.06906, Jul 2018.

[57] Google. Cloud automl. URL https://cloud.google.com/automl/. Accessed: 26/04/2019.

[58] Ronald J. Williams. *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, volume 8, pages 229–256. Kluwer Academic Publishers Hingham, 1992.

[59] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014.

[60] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-Level Network Transformation for Efficient Architecture Search. *arXiv e-prints*, art. arXiv:1806.02639, Jun 2018.

[61] Liam Li and Ameet Talwalkar. Random Search and Reproducibility for Neural Architecture Search. *arXiv e-prints*, art. arXiv:1902.07638, Feb 2019.

[62] Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. SMASH: One-Shot Model Architecture Search through HyperNetworks. *arXiv e-prints*, art. arXiv:1708.05344, Aug 2017.

[63] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernand o, and Koray Kavukcuoglu. Hierarchical Representations for Efficient Architecture Search. *arXiv e-prints*, art. arXiv:1711.00436, Nov 2017.

[64] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive Neural Architecture Search. *arXiv e-prints*, art. arXiv:1712.00559, Dec 2017.