MEMO-F524 - Master thesis

Convolutional Neural Architecture Search : Evaluating Hillclimbing and Knowledge Transfer Approaches

Author:

Anthony ZHOU

Supervisors:

Hugues BERSINI

Antonio GARCIA DIAZ

Academic year 2022-2023

Acknowledgements

I would first like to thank my thesis promotor Prof. Hugues Bersini for being my promotor and guiding me for through this thesis. Besides my promoter, I would like to thank my supervisor Antonio Garcia Diaz for his time, support and guidance during the time of my research.

I would like to express my sincere gratitude to Pierre Defraene, Antoine Lemahieu, Tristan Philips and Mi Zhou as proofreaders of this thesis. Their valuable comments and feedback allowed to increase the quality of this work, by ensuring that it is error-free, clear, easy to understand and self-contained.

Finally, I would like to show my appreciation to my family for their constant support and encouragement during my academic journey and throughout the process of researching and writing of this thesis.

Abstract

This master thesis explores the Neural Architecture Search by Hillclimbing (NASH) algorithm for automatically designing architectures for Convolutional Neural Network (CNN)s. The NASH algorithm is a combination of the hillclimbing strategy, network morphisms, and training via an optimization algorithm that improves the hillclimbing process. The aim of the experiments was to find the best combination of hyperparameters, study the effect of incorporating dropout and early stopping techniques on the performance and computational cost of the models and study the effect of the network morphisms in our methods. The experiments were conducted on three different datasets, namely CIFAR-10, CIFAR-100, and SVHN. The results showed that the NASH algorithm is a robust algorithm with consistent and reliable performance. However, changing the learning rate scheduler proved to have a slight increase on the overall performance of the models. Incorporating both dropout and early stopping techniques led to a minor decrease in the overall performance of the models, but this reduction in performance was accompanied by a considerable decrease in the execution time. Furthermore, using a weight reset on the combination of dropout and early stopping techniques allows to reach better performances while keeping a low execution time. The results of our study showed that, while our models did not perform as well as the state-of-the-art on CIFAR-10 and CIFAR-100, they were less complex and achieved comparable results to our base paper in the same amount of time. On the other hand, for the SVHN dataset, we were able to reach performances that were comparable to the state-of-the-art. In addition, we also performed scatter plots to visualize the relationship between accuracy and the number of parameters and identify the Pareto frontier. The plots showed that there is always a trade-off between these two factors meaning no particular method exhibited dominance or clear superiority over others. Our investigation provides valuable insights into the balance between execution time and performance, which can aid in the development of more efficient and effective machine learning models in the future. Our study makes a contribution to the area of automatic architecture search, particularly in the application of the NASH algorithm for designing CNNs.

Acronyms

AI Artificial Intelligence. 1, 29

ANOVA Analysis of Variance. 33, 47, 48, 55, 56, 63, 68

CNN Convolutional Neural Network. 1, 2, 4, 12–15, 24, 28, 30, 32, 68, 69

EA Evolutionary Algorithm. 26, 28

ILSVRC ImageNet Large Scale Visual Recognition Challenge. 24

Leaky ReLU Leaky Rectified Linear Unit. 9

ML Machine Learning. 1–3, 5, 6, 29, 69

NAS Neural Architecture Search. 5, 15, 16, 25–28, 63, 69

NASH Neural Architecture Search by Hillclimbing. 20, 21, 29–39, 64, 65, 67–69

NN Neural Network. 1–5, 9, 16, 20, 24–29, 32

ReLU Rectified Linear Unit. 9

RL Reinforcement Learning. 26, 69

RNN Recurrent Neural Network. 26

SGDR Stochastic Gradient Descent with warm Restarts. 16, 20, 22, 68, 70

SVHN Street View House Numbers. 29, 30, 35, 37, 42, 43, 45–47, 50, 51, 53–55, 58, 59, 61, 62, 66–69

tanh Hyperbolic Tangent. 8

Contents

1 Introduction

Artificial Intelligence (AI) is a concept that has been around for over 50 years, it refers to the development of computer systems or machines that can perform tasks that would typically require human intelligence [Ong17]. The very large amounts of data and the increase in computing power have led to enormous advances in recent years in this field [Liu+18]. Nowadays, AI has an important role in our everyday life and is inherently present like in social media (Facebook, Twitter, Linkedin, Reddit, etc.) [Sad+21], in smart home devices (Alexa, Google Assistant, Siri, etc.) [Mae+19], in the media industry (Youtube, Twitch, Netflix, etc.) [Veg+18] and many more. Even though not everyone knows what it really is, we find it at every step we take and it is constantly growing especially Machine Learning (ML), a subfield of AI [MD01]. ML is the process of extracting knowledge from data to help a computer learn without direct instruction [Ong17]. This enables the algorithm to continue learning and improving on its own, based on experience.

Whether it is for image recognition [SZ15] [Gir+13], speech recognition [DBB52] [Bak79] [Wil+90], machine translation [Kni97], etc. many techniques of ML exist to solve these problems. One technique that is popular for image recognition is Neural Network (NN) and more precisely Convolutional Neural Network (CNN) which are the most used type of NN [Jia17]. For example, in recognition of disease like Alzheimer's disease [KYK19], in recognition of writing by hand [SSS19], in facial recognition [Kha+19], in object detection [Che+17], etc. these are all real-world application for CNNs.

CNNs used to be designed manually by hand-crafting the architecture which is an exhausting, time-consuming process [AM17] [EMH17]. Additionally, the vast amount of possible configurations requires expert knowledge to restrict the search. Therefore, a rising demand for automating architecture engineering was the natural next step in automating ML. However, since the architecture search space is discrete, traditional optimization algorithms such as gradient descent (cfr 2.6) [Rud16] are not applicable, hence approaches based on reinforcement learning algorithms (cfr 3.3.1) [Cai+17] [Bak+16] [Wil92] and evolutionary algorithms (cfr 3.3.2) [MTH89] [SM02] are used for automated designing of neural networks. But these approaches are either very costly (requires hundreds and thousands of GPU days) or yield non-competitive results.

In this work, we seek to reduce the computational cost and increase the performances of our CNNs by using a simple hillclimbing and accelerate the learning process via knowledge transfer. The hillclimbing is an iterative heuristic method that starts with an arbitrary solution to a given problem, search for a better solution by an incremental change to the solution. The algorithm stops when there are no improvements within the next epochs [SG06]. Meanwhile, knowledge transfer is a process that accelerates the training of a *student* network by instantaneously transferring the knowledge from a *teacher* network that was already trained on the same task [CGS15]. This allows the *student* network to learn more rapidly than it would through conventional training methods. A more detailed explanation of these concepts will be covered later in section 2.5 and section 2.7 respectively.

Our first step will be a study of the hillclimbing strategy with knowledge transfer [EMH17]. By tweaking its hyperparameters (cfr 2.1.3), we will try to find the combination with the best performances. From this, we will add dropout (cfr 4.5.2) [Hin+12] and early stopping (cfr 4.5.2) [Pre12][GJP98], two regularization techniques that should help improve the generalization performance of our CNNs, and analyse their effects in practice.

But first and foremost, we will provide a theoretical background by defining important concepts in ML. Next, we will provide a detailed explanation of NNs and CNNs. We will then explain in details the components of the hillclimbing strategy with knowledge transfer. After that, we will take a look at the current state of the art. Following that, we will explain our methodology, present our results, and analyze them. Finally, we will provide a conclusion and discuss possible directions for future work.

2 Theoretical Foundations

2.1 Important concepts

In this section, we will explore several fundamental concepts of ML. To ensure that each of these concepts is clearly understood, as they will be referenced frequently, we will provide concise definitions for them.

2.1.1 Batches and epochs

The entire dataset is usually too large to process at once for a NN, so it is divided into smaller batches of equal size [Bro18]. A batch refers to a subset of the data that is used to update the model's parameters during training which helps to improve the accuracy and efficiency of the model. An epoch refers to a complete pass through the entire dataset. In other words, it is the number of times the model has seen the entire dataset. During an epoch, the model goes through all the batches in the dataset, updates its parameters after each batch, and repeats this process until it has seen the entire dataset.

2.1.2 Trainable parameters

Trainable parameters, also known as learnable parameters, are the parameters in a neural network that are updated during the training process [DKK+12]. These parameters include the weights and biases of the neural network. During training, the model attempts to learn the optimal values for these parameters by minimizing a loss function. The optimization algorithm used during training updates the trainable parameters to minimize the loss function and improve the model's performance. The number of trainable parameters in a neural network can vary depending on the architecture of the network and the number of layers and neurons used. Generally, the more parameters a model has, the more complex it is but it does not imply better results.

2.1.3 Hyperparamters

Hyperparameters are parameters that are set manually to control the behavior of a ML model during training [GBC16]. They are not learned from the data, but rather serve as configuration settings that influence the model's performance, convergence, and generalization. There are a lot of hyperparameters in a NN like the learning rate, the number of epochs, the size of the batch , the number of layers are some examples. In addition to that, for CNNs, there is also the size of the filter, the stride and the padding which makes their optimization very difficult.

2.1.4 Learning rate

The learning rate is a hyperparameter that determines the step size at which the optimizer adjusts the weights and biases of a NN during training [Sut+13]. It is one of the most important hyperparameters to tune for optimizing the performance of a NN. A learning rate that is too high can cause the model to overshoot the minimum of the loss function, while a learning rate that is too low can cause the model to take a long time to converge. The optimal learning rate depends on the specific model architecture and the dataset being used.

2.1.5 Batch normalization

Batch normalization is a technique used in deep learning to improve the training of NNs by normalizing the inputs of each layer [Bro18]. In deep NNs, the distributions of the inputs to each layer change as the parameters of the previous layers are updated during training. This causes a phenomenon known as internal covariate shift, a change in the distribution of the activations (outputs) of layers during the training process. This slows down the training of the network and makes it more difficult to find good solutions. Batch normalization addresses this problem by normalizing the inputs to each layer. Specifically, it normalizes the mean and variance of the inputs within each mini-batch of data during training, so that the mean and variance of the activations of each layer are approximately constant. This allows the subsequent layers to learn more efficiently, as the input distribution is kept more stable.

2.1.6 Backpropagation

Backpropagation is an algorithm used to train NNs by adjusting the weights of the network based on the error between the predicted outputs and the actual outputs [Rum+95]. It involves two main steps: the forward pass and the backward pass.

During the forward pass, input data is fed through the NN to compute the predicted outputs. Then, during the backward pass, the algorithm computes the gradient of the error with respect to the weights of the network, layer by layer, using the chain rule of calculus. This gradient represents the direction and magnitude of the steepest increase in the error. The weights of the network are then updated in the opposite direction of the gradient, scaled by a learning rate, to minimize the error.

2.1.7 Underfitting and overfitting

Underfitting and overfitting are common problems that can arise in ML models [Zha+17] [JK15]. Avoiding these problems is important for building models that can generalize well to new, unseen data and make accurate predictions.

Underfitting occurs when a model is too simple and has not learned enough from the training data, it is said to underfit the data. This often results in poor performance on the test sets, as illustrated in figure 2.1. Underfitting can occur when the model lacks the necessary complexity to capture the patterns and relationships in the data or when the the training data is too noisy or too small.

Overfitting occurs when a model becomes too focused on the training data and fits it too closely, it is said to overfit the data. This often results in good performance on the training set but poor generalization to new data, as illustrated in figure 2.1. Overfitting can occur when the model has learned too much from the training data or when the training data is too small.

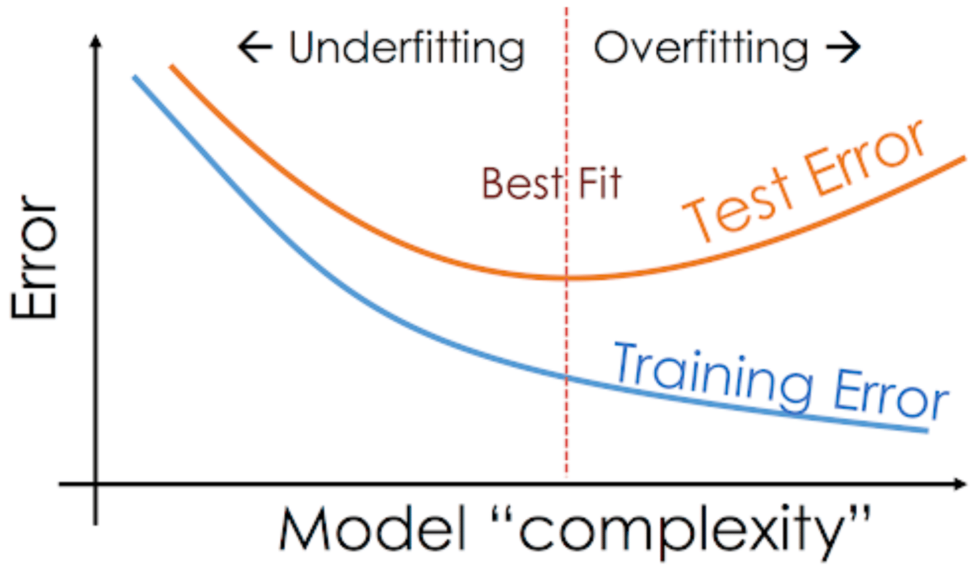


Figure 2.1: Visualization of underfitting and overfitting. (image taken from [Sax20])

2.1.8 Neural architecture search

Neural Architecture Search (NAS) is a process of automatically designing the architecture of a NN, including its topology, size, and connections, using ML techniques [EMH19]. NAS aims to automate the tedious and time-consuming process of manually designing NNs, and instead, employs algorithms to automatically search and discover optimal network architectures for a given task or dataset.

2.1.9 Binary classification problems

As illustrated in figure 2.2, binary classification problems refer to ML problems where the goal is to classify input data into one of two possible categories/classes [Bis07]. The output/prediction of the model is binary, typically represented as either 0 or 1, true or false, positive or negative. Binary classification is a fundamental problem in ML and has various applications, such as spam detection [Cra+15], fraud detection [AAO17], sentiment analysis [Mal+20], and medical diagnosis [Bha+21], among others.

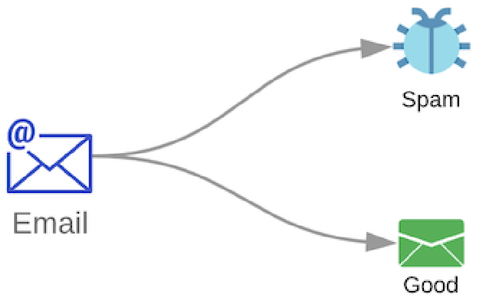


Figure 2.2: Visualization of a binary classification. (image taken from [Mur19])

2.1.10 Multi-class classification problems

As illustrated in figure 2.3, multi-class classification problems refer to ML problems where the goal is to classify input data into one of multiple possible categories/classes, typically more than two [Bis07]. The output/prediction of the model is a single class label that indicates the predicted category for the input data. Multi-class classification is commonly used in various applications, such as image recognition [SZ15] [Gir+13], speech recognition [DBB52] [Bak79] [Wil+90], document classification [Kad19], and product categorization [KA19] [Pau+20], among others.

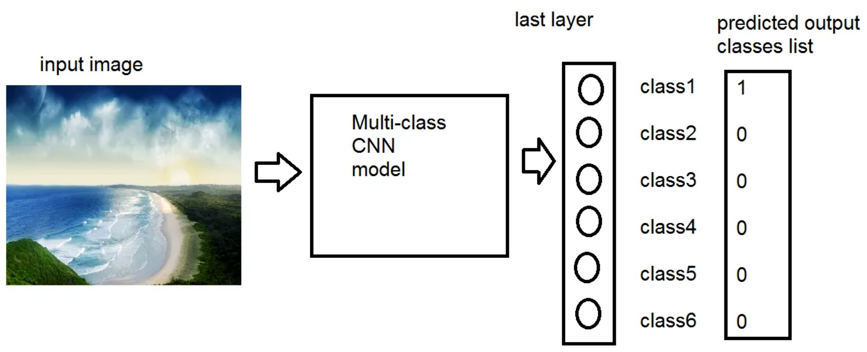


Figure 2.3: Visualization of a multi-class classification. (image taken from [Pus20])

2.2 Motivation of neural networks

The motivation behind the design of artificial neurons was to create a mathematical model that mimic the information processing capabilities of the human brain [Gup+13]. Meaning the model could learn from data, adapt to different patterns and perform tasks such as pattern recognition, classification and prediction. Artificial neurons were seen as a way to enable machines to mimic human-like cognitive abilities, such as perception, reasoning and decision-making, which were considered highly desirable for many practical applications.

2.3 Struture of a neural network

2.3.1 Neuron

Neural networks intends to simulate the behavior of the human brain as previously mentioned [MW+14]. It consists of nodes which, in the biological analogy, represent neurons.

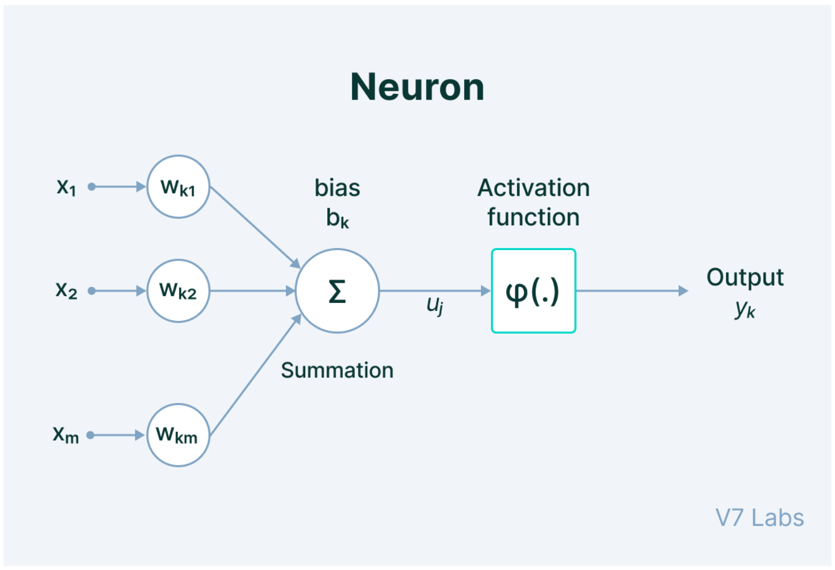


Figure 2.4: Structure of a neuron. (image taken from [Bah22])

As shown in figure 2.4, a neuron is made of various components:

- The **input** features represented by the X_i.
- The **weight** represented by W_{ki} give importance to those features that contribute more towards the learning. It does so by introducing scalar multiplication between the input value and the weight matrix.
- The **bias** represented by b_k needs to shift the value produced by the activation function. Its role is similar to the role of a constant in a linear function.

- The **transfer function** represented by Σ sum all input into one value, the net input represented by U_j, and send it to the activation function.
- The **activation function** represented by $\varphi(.)$ introduces non-linearity. Without this, the output would just be a linear combination of input values and would not be able to introduce non-linearity in the network.

Example of activation function

Sigmoid

The sigmoid activation function, also know as the logistic activation function, maps any input value to a value between 0 and 1 [SSA17]. It is often used in the output layer of a neural network for binary classification problems, where the output of the neural network needs to be transformed into a probability value between 0 and 1. It is mathematically defined in equation 2.1 and graphically defined in figure 2.5.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

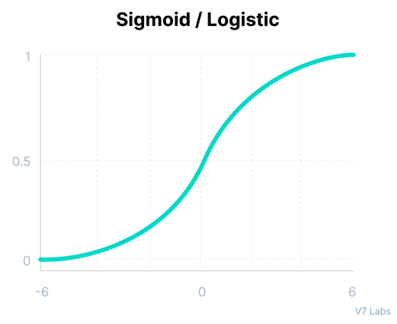


Figure 2.5: Sigmoid/logistic activation function. (image taken from [Bah23])

Tanh

The Hyperbolic Tangent (tanh) activation function is similar to the sigmoid function but maps any input value to a value between -1 and 1 [SSA17]. Like the sigmoid function, the tanh function is often used in the output layer of a neural network for binary classification problems. It is mathematically defined in equation 2.2 and graphically defined in figure 2.6.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.2}$$

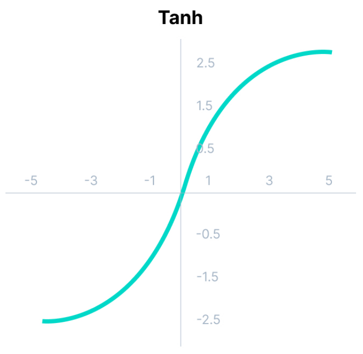


Figure 2.6: Tanh activation function. (image taken from [Bah23])

ReLU

The Rectified Linear Unit (ReLU) activation function is one of the most commonly used activation functions in deep learning [SSA17]. It has become popular due to its simplicity and effectiveness in improving the performance of deep neural networks. It is a very simple function that returns the maximum between the input value or 0. It is mathematically defined in equation 2.3 and graphically defined in figure 2.7.

$$f(x) = max(0, x) \tag{2.3}$$

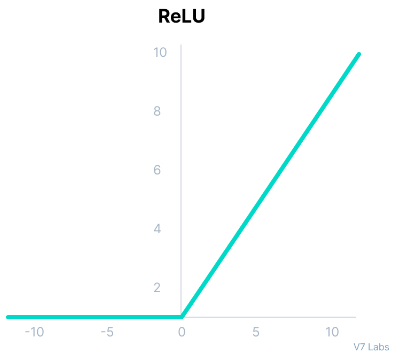


Figure 2.7: ReLU activation function. (image taken from [Bah23])

Leaky ReLU

The Leaky Rectified Linear Unit (Leaky ReLU) is an activation function that is similar to the standard ReLU function, but with a small slope for negative values [SSA17]. This means that instead of returning a 0 if the input value is negative, it will return some small negative value. The purpose of the Leaky ReLU function is to address the *dying ReLU* problem. This problem can occur when a large number of neurons in a NN end up in the negative range of the ReLU activation function, causing them to output zero. As a result, these neurons do not contribute to the learning process, and their weights are not updated during backpropagation. Over time, a significant portion of the network's neurons can become *dead* or inactive, leading to reduced model capacity and degraded performance. It is mathematically defined in equation 2.4 and graphically defined in figure 2.8.

$$f(x) = max(0.1 \cdot x, x) \tag{2.4}$$

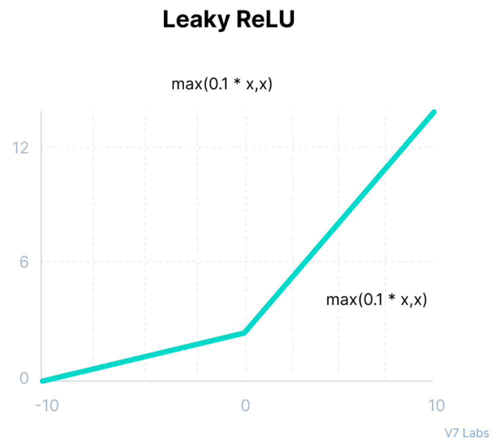


Figure 2.8: Leaky ReLU activation function. (image taken from [Bah23])

SoftMax

The softmax activation function is a combination of multiple sigmoid functions [SSA17]. It is a type of activation function that is often used in the output layer of a neural network for multi-class classification problems. It takes in a vector of real-valued inputs and produces a vector of the same size as output, where each element represents the probability of the input belonging to a particular class. It is mathematically defined in equation 2.5.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{for j} = 1, ..., K \tag{2.5}$$

where $\sigma(\mathbf{z})_j$ is the *j*-th element of the softmax output vector, z_j is the *j*-th element of the input vector, and K is the total number of classes.

2.3.2 Layers

By stacking multiple neurons together in a row, they constitute a layer and multiple layers piled next to each other create a multi-layer network, hence the name *neural network* [MW+14].

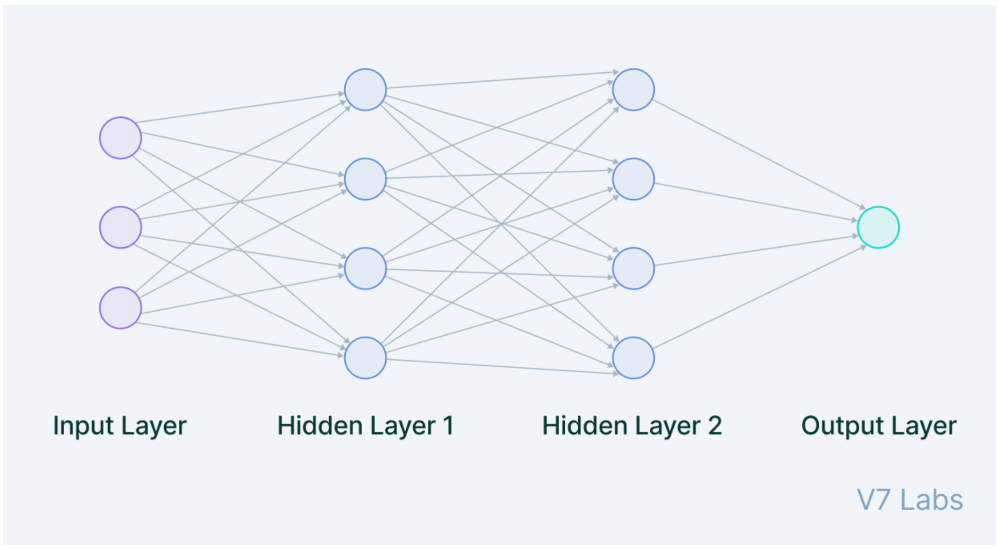


Figure 2.9: Structure of a neural network. (image taken from [Bah22])

As shown in figure 2.9, a neural network is composed of different layers:

- The **input layer** is all the data feeded to the model from an external source. It is the only visible layer in the complete neural network architecture that passes the complete information from the outside world without any computation.
- The **hidden layer** are the intermediate layers that do all the computations and extract the features from the data. There can be multiple interconnected hidden layers that account for searching different hidden features in the data.
- The **output layer** takes input from preceding hidden layers and comes to a final prediction based on the model's learning. It is the most important layer where we get the final result.

2.3.3 Example of neural network

Perceptron

The perceptron is the simplest neural network architecture [Ros58]. It is a type of neural network that takes a number of inputs, applies certain mathematical operations on these inputs, and produces an output. It takes a vector of real values inputs, performs a linear combination of each attribute with the corresponding weight assigned to each of them. The weighted input is summed into a single value and passed through an activation function.
It is an another name for the neuron 2.3.1 illustrated in the figure 2.4.

Feed-forward neural network

It is a multi-layer neural network as shown in figure 2.9, where the information is passed in the forward direction—from left to right [SKP97]. In the forward pass, the information comes inside the model through the input layer, passes through the series of hidden layers, and finally goes to the output layer. This neural network architecture is forward in nature, the information does not loop with two hidden layers.

Convolutional neural network

Convolutional Neural Network (CNN) is a type of feed-forward neural networks used in image recognition and processing that is specifically designed to automatically learn patterns or features from input images [ON15]. It is the main subject of this work and its structure is explained in greater details in the section 2.4.

2.4 Structure of a convolutional neural network

CNNs are comprised of three main types of layers: the **convolutional layer**, the **pooling layer** and the **fully-connected layer** [ON15]. The convolutional layer is the first layer of a CNN. While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object.

2.4.1 Convolutional layer

The convolutional layer is the core building block of a CNN where the original image, which is the input in a computer vision application, is *convoluted* using a *feature detector* that detects important small features of an image, such as edges. This *feature detector* is also known as **kernel** or filter and moves across the receptive fields of the image, checking if the feature is present. This process is known as a *convolution*.
The kernel is a two-dimensional array of weights, which represents part of the image. While they can vary in size, the kernel size is typically a 3x3 matrix. The kernel is then applied to an area of the image, and a scalar product is calculated between the input pixels and the kernel. This dot product is then fed into an output array (see figure 2.10). Afterwards, the kernel shifts by a distance defined manually, repeating the process until it has swept across the entire image. The final output from the series of dot products from the input and the kernel is known as a **feature map**, activation map, or

a convolved feature. Ultimately, the convolutional layer converts the image into numerical values, allowing the neural network to interpret and extract relevant patterns. In essence, a kernel can be considered the equivalent of a *neuron* in a perceptron (cfr section 2.3.3) or a fully-connected layer (cfr section 2.4.3) [LWW19].

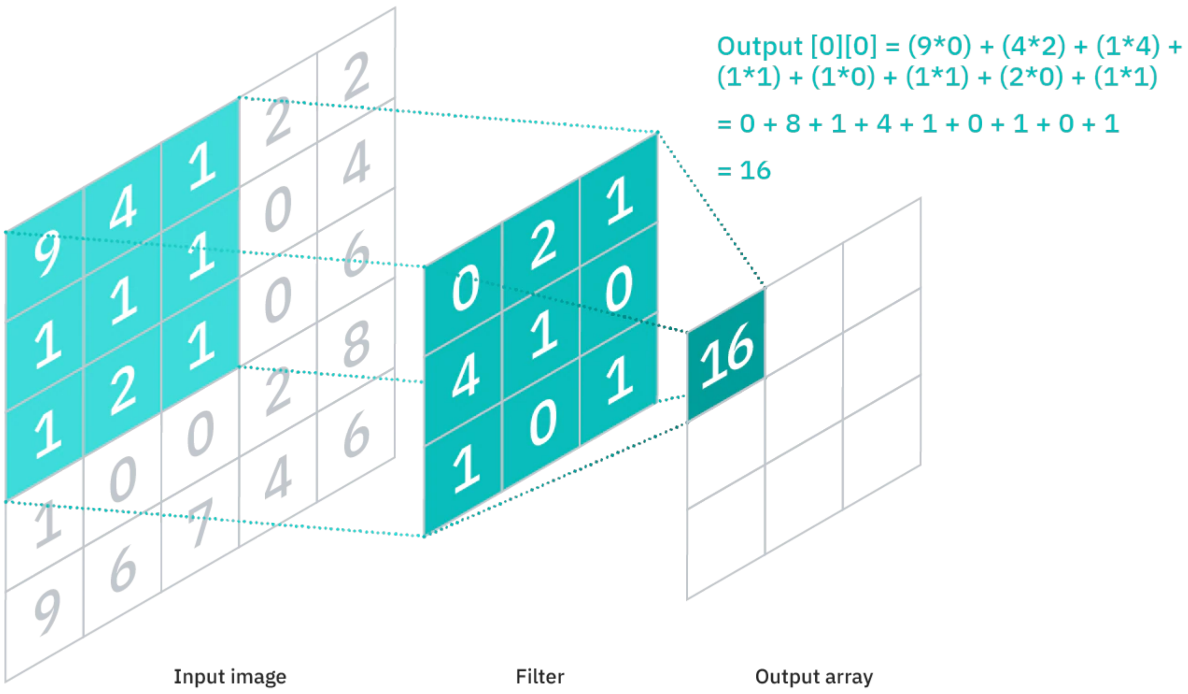

Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
Input image
Filter
Output array

Figure 2.10: A visual representation of a convolutional layer. (image taken from [Edu20])

Another convolutional layer can follow the initial convolutional layer. When this happens, the structure of the CNN can become hierarchical as the later layers can see the pixels within the receptive fields of prior layers. In other words, each convolutional layer makes up a lower-level pattern in the neural net, and the combination of its parts represents a higher-level pattern, creating a feature hierarchy within the CNN.
Convolutional layers are also able to significantly reduce the complexity of the model through the optimization of its output. These are optimised through three hyperparameters:

- The **depth** is the number of filters. Reducing this hyperparameter can significantly minimise the total number of neurons of the network, but it can also significantly reduce the pattern recognition capabilities of the model.
- The **stride** is the distance, or number of pixels, that the kernel moves over the input matrix. While stride values of two or greater is rare, a larger stride yields a smaller output.
- **Zero-padding** is usually used when the filters do not fit the input image. This sets all elements that fall outside of the input matrix to zero, producing a larger or equally sized output.

2.4.2 Pooling layer

Pooling layers conduct dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a kernel across the entire input, but the difference is that this kernel does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field, populating the output array.
As shown in figure 2.11, there are two main types of pooling:

- **Max pooling**: As the kernel moves across the input, it selects the pixel with the maximum value to send to the output array. This approach is more used.
- **Average pooling**: As the kernel moves across the input, it calculates the average value within the receptive field to send to the output array.

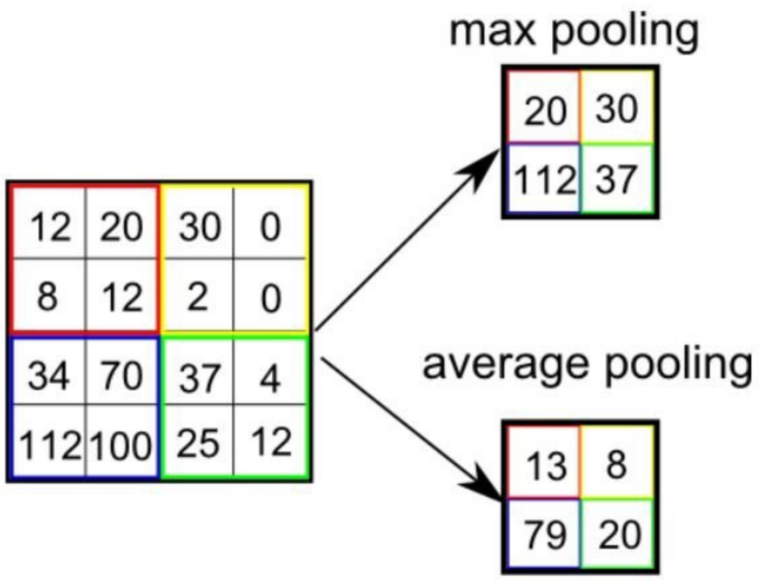


Figure 2.11: Types of pooling. (image taken from [Sah18])

While a lot of information is lost in the pooling layer, it also has a number of benefits to the CNN. They help to reduce complexity, improve efficiency, and limit risk of overfitting [ON15].

2.4.3 Fully-connected layer

The fully-connected layer contains neurons of which are connected to all the neurons in the two adjacent layers, without being connected to any layers within them. This layer performs the task of classification based on the features extracted through the previous layers and their different kernels. This output is *flattened* by turning it into a single vector before entering the layer. It leverages softmax activation function to classify inputs appropriately by producing a probability between 0 and 1 (see figure 2.12).

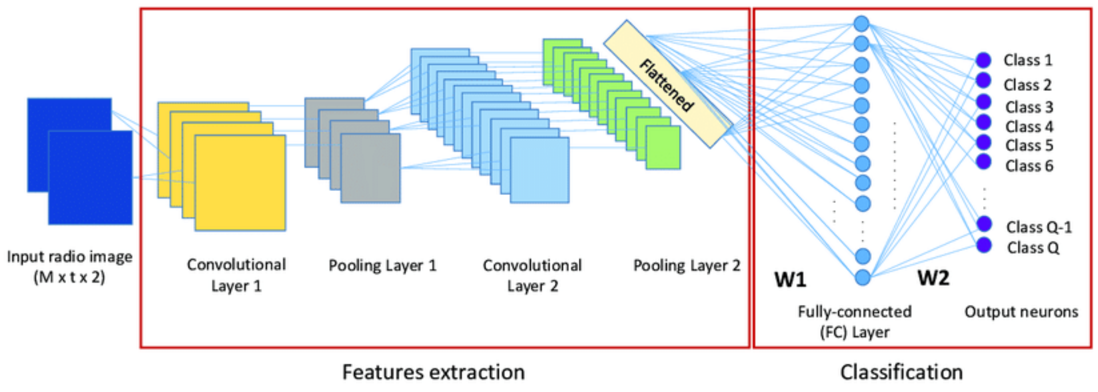


Figure 2.12: CNN with 2 fully-connected layers. (image taken from [Nji+19])

2.5 Hillclimbing

The hillclimbing is an iterative algorithm that starts with an arbitrary solution to a given problem, search for a better solution by an incremental change to the solution. The algorithm stops when there are no improvements within the next iteration. Regarding its application to NAS, hillclimbing can be applied to iteratively search for better architectures by making small modifications to the current architecture and evaluating their performance. If a better architecture is found, it is updated as the current architecture, and the search continues until a stopping criterion is met.

We decided to use hillclimbing as it is a straightforward approach. One advantage of hillclimbing for NAS is its simplicity and efficiency [Ren+21] [Zho+18b]. It requires minimal computational resources compared to other NAS methods, and can be easily parallelized to explore multiple architectures at the same time. This algorithm is really interesting considering that it can search after sufficiently good solutions in a search space too big for an exhaustive search. Indeed, it is a heuristic algorithm so the solution found may not be the optimal solution to the problem.

However, one limitation of hillclimbing for NAS is that it can easily get stuck in local optima, where small modifications do not improve the performance of the architecture. To overcome this limitation, various techniques such as simulated annealing, genetic algorithms, or reinforcement learning can be used in combination with hillclimbing to explore a wider range of architectures and avoid local optima.

2.6 Stochastic gradient descent with warm restarts

The Stochastic Gradient Descent with warm Restarts (SGDR) allows to improve the hillclimbing process by gradually annealing the learning rate over time, and periodically *restarting* the hillclimbing from a checkpoint of the model weights [LH16].

The annealing of the learning rate is done using a cosine annealing schedule, where the learning rate is decreased from its initial value to a minimum value over a set number of epochs, and then increased back to its initial value over the next set of epochs. This cycle is repeated multiple times during training, with each cycle increasing in length and decreasing in the learning rate minimum value.

The periodic *restarts* involve saving the current weights of the model at the end of each cycle, and then resetting the weights to the best performing set of weights during the previous cycle. This allows the model to escape from local minima and explore new areas of the optimization landscape, while still retaining the best performing set of weights.

SGDR has been shown to be effective in improving the optimization performance of deep NNs, especially in cases where the networks are prone to getting stuck in local minima or plateaus during training. It can also help reduce overfitting by adding regularization effects.

2.7 Network morphisms

Network morphisms allow testing many deep learning models in a short time because there is no need to train each succeeding network from scratch. The basic idea is to define a set of transformations that can be applied to a base NN architecture to produce a family of related architectures [CGS15]. During the NAS process, a set of candidate architectures is generated by applying these transformations to a base architecture without any loss of knowledge previously gained during the training. The use of network morphism operations reduced significantly the computational cost of the algorithms. It allows to transfer the knowledge to the newly created architectures and expand their capacity allowing the acquisition of better results as shown in figure 2.13. The performance of each candidate architecture is then evaluated on a validation set, and the best performing architecture is selected as the new base architecture for the next round of transformations. However, it's worth noting that these transformations are not always the best option for improving the performance of a student network. The choice of transformation depends on the specific characteristics of the task and the teacher network, and may require some experimentation to determine the best approach.

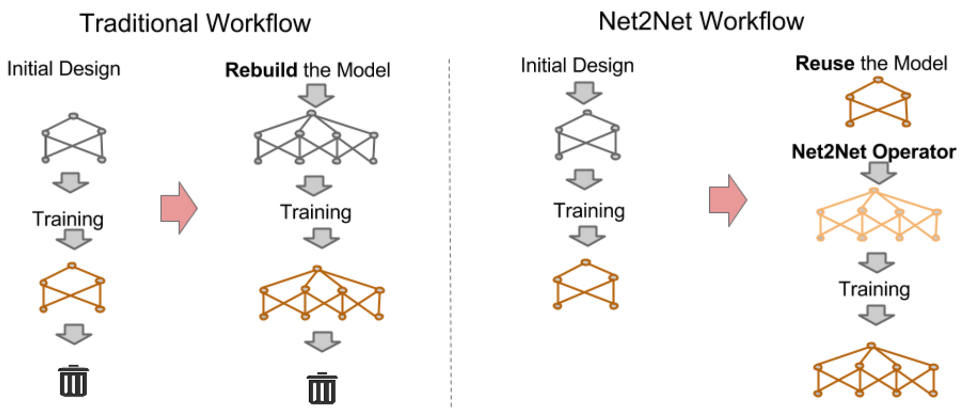


Figure 2.13: Comparison between a traditional workflow and the Net2Net Workflow. (image taken from [CGS15])

2.7.1 Deepening

As shown in figure 2.14, the goal of deepening is to increase the depth of the student network while preserving its input-output behavior, by adding more layers to the network [CGS15]. The basic idea behind deepening is to start with a shallow student network that has the same input and output dimensions as the teacher network, and then gradually add more layers to the network. The weights of the new layers are initialized based on the weights of corresponding layers in the teacher network, so that the student network starts with a similar structure to the teacher network.

Deepening can be a powerful technique for transferring knowledge from a teacher network to a student network. By adding more layers to the student network, it can capture more complex and abstract features in the input data, leading to better performance on the task. At the same time, the knowledge gained from the teacher network helps the student network avoid some of the pitfalls of training a very deep network from scratch.

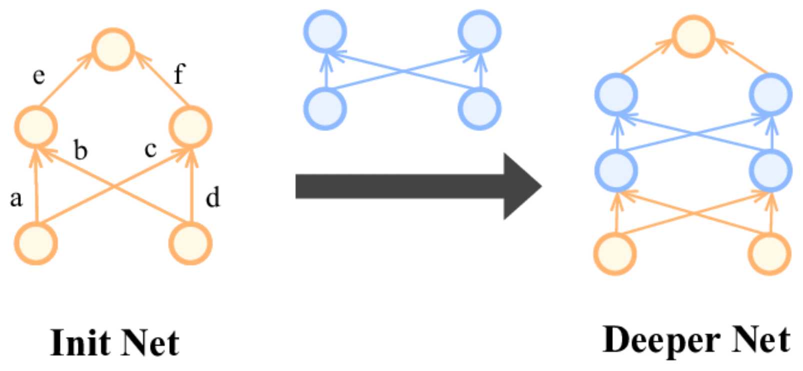


Figure 2.14: Model and its equivalent with deepening.

2.7.2 Widening

As shown in figure 2.15, the goal of widening is to increase the width of the student network while preserving its input-output behavior, by increasing the number of neurons in each layer of the network [CGS15]. The basic idea behind widening is to start with a narrow student network that has the same input and output dimensions as the teacher network, and then gradually increase the number of neurons in each layer. The weights of the existing neurons are kept fixed, while new neurons are added to the network and initialized based on the weights of corresponding neurons in the teacher network.

Widening can be a powerful technique for transferring knowledge from a teacher network to a student network, especially in cases where the teacher network is already relatively deep. By increasing the number of neurons in each layer, the student network can capture more diverse and detailed features in the input data, leading to better performance on the task. At the same time, the knowledge gained from the teacher network helps the student network avoid some of the pitfalls of training a very wide network from scratch.

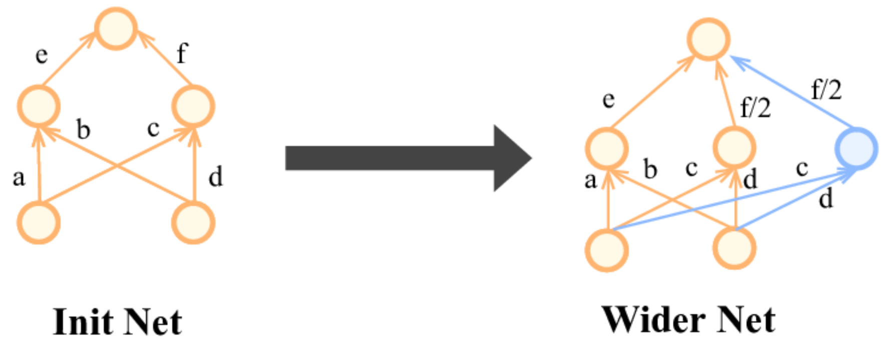


Figure 2.15: Model and its equivalent with widening.

2.7.3 Multi-branch motif

The goal of multi-branch motif is to replace a single layer in the original architecture with a more complex structure involving multiple branches [Cai+18]. Each branch in the multi-branch motif can either be just a replication of the replaced layer or can consists of multiple convolutional or other types of layers, and the outputs of these branches are combined in some way to produce the final output of the motif. This approach can be used to introduce additional capacity and complexity into the network at a specific layer, allowing for more expressive feature extraction and modeling capabilities. In our implementation, we will use this operation with two merge scheme *add* and *concatenation*.

Add scheme

As shown in figure 2.16 (middle), to create a multi-branch motif with *N* branches that is equivalent to the original convolutional layer, the branches need to be designed to replicate the output of the original convolutional layer for any input feature map *x*. The branches are merged using an *add* operation, and the allocation scheme is set to replication, where each branch is a copy of the original convolutional layer. To eliminate the effect of the number of branches *N* on the final output, each branch's output is divided by *N*. This ensures that the overall output of the multi-branch motif remains the same as the output of the original convolutional layer, irrespective of the number of branches used.

Concatenation scheme

As shown in figure 2.16 (right), when merging the branches using *concatenation*, the allocation scheme is set to replication. The filters of the original convolutional layer are then divided into *N* parts along the output channel dimension, and each part is assigned to the corresponding branch. These branches are later merged together to produce the final output of the multi-branch motif.

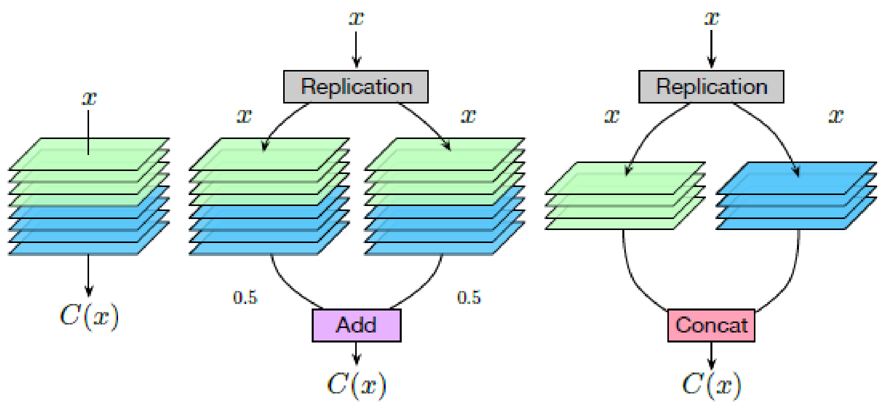


Figure 2.16: Convolution layer and its equivalent multi-branch motifs.
(image taken from [Cai+18])

2.7.4 Skip connection

As shown in figure 2.17, the goal of a skip connection is to allow the output of one layer to be directly added to the output of another layer, bypassing one or more intermediate layers [He+15]. This can be thought of as a shortcut that enables the network to more easily pass information between layers that are further apart. In our implementation, the skip connection is simulated by using a combination of *add* and *deepening*.

Skip connections are particularly useful in deep NNs, as they can help to alleviate the vanishing gradient problem that can occur during training. This problem occurs when gradients become very small as they are backpropagated through multiple layers, making it difficult for the network to learn effectively. By allowing information to be passed more easily between layers, skip connections can help to mitigate this problem.

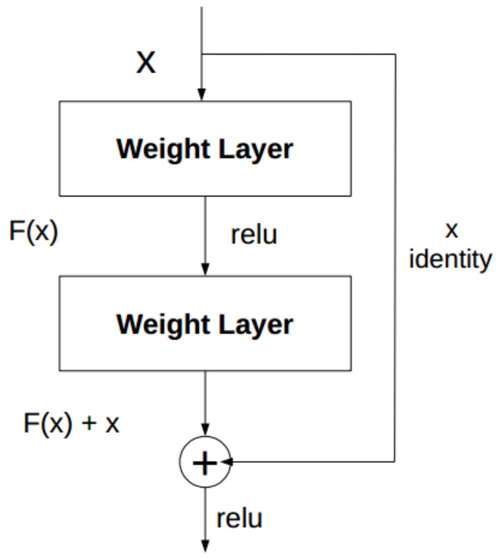


Figure 2.17: Illustration of skip connection. (image taken from [Ome+21])

2.8 Neural Architecture Search by Hillclimbing

The Neural Architecture Search by Hillclimbing (NASH) algorithm is the combination of the hillclimbing strategy, network morphisms, and training via SGDR [EMH17], as represented in figure 2.18. We start with a small, (possibly) pretrained network. Then, we apply network morphisms, sampled uniformly at random from the section 2.7, to this initial network to generate larger ones that may perform better when trained further. These new *student* networks can be seen as neighbors of the initial *teacher* network in the space of network architectures. Due to the network morphism, the *student* networks start at the same performance as their teacher. The various *student* networks can then be trained further for a brief period of time to exploit the additional capacity obtained by the network morphism, and the search can move on to the best resulting *student* network. This constitutes one step of the NASH algorithm as shown in figure 2.19. NASH can execute this step several times until performance on a validation set saturates. Note that the current best model is also considered as a student, our algorithm is not forced to select a new model but can rather also keep the old one if no other one improves upon it.

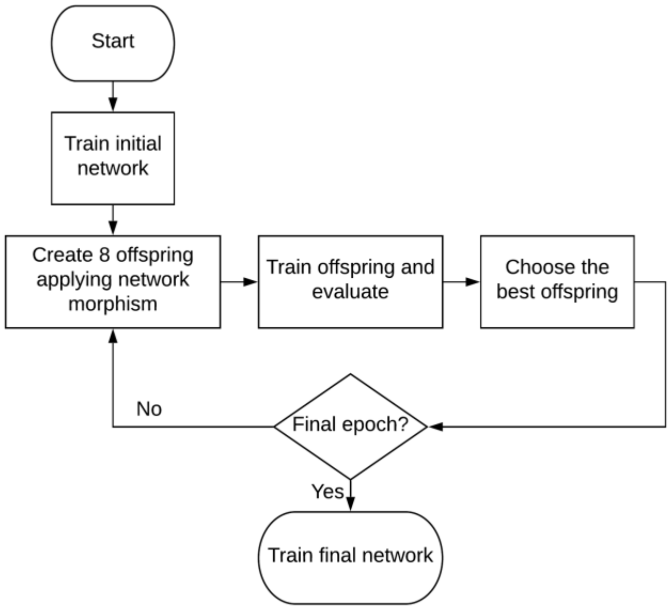

Start
Train initial network
Create 8 offspring applying network morphism
Train offspring and evaluate
Choose the best offspring
Final epoch?
No
Yes
Train final network

Figure 2.18: Schematic of the NASH algorithm. (image taken from [KGM19])

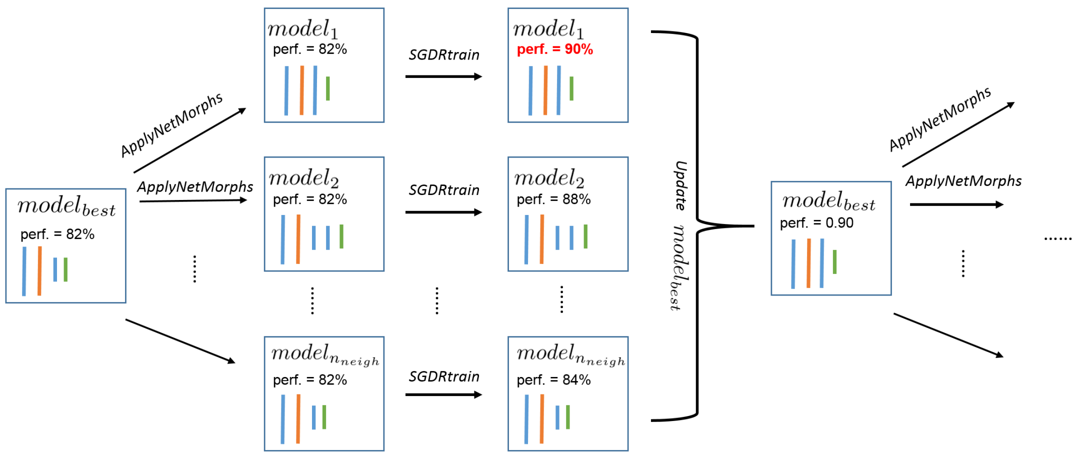


Figure 2.19: Visualization of one step of the NASH algorithm. (image taken from [EMH17])

2.8.1 Hyperparameters

This algorithm contains a certain number of hyperparameters:

- n_{steps} is the number of hillclimbing steps. It affects the quality of the final network architecture discovered by the algorithm. If the number of steps is too small, the algorithm may converge to a suboptimal architecture that is not the optimal possible solution. If the number of steps is too large, the algorithm may spend too much time exploring variations of architectures that are already close to the optimal solution, and may not have enough time to explore more distant and potentially better solutions.
- $n_{neighbours}$ is the number of neighbours created at each step in the architecture search space. A larger number of neighbors allows the algorithm to explore a more diverse set of candidate architectures, which can be beneficial for discovering better solutions in complex search spaces. However, it also increases the computational cost of evaluating the performance of each candidate architecture, as well as the difficulty of finding the optimal solution, especially if the search space is large.
- n_{NM} is the number of network morphism applied. It should be balanced between exploring a diverse set of candidate architectures and converging to promising solutions quickly, to avoid being stuck in suboptimal regions of the search space.
- $epoch_{neighbours}$is the number of epochs for training every neighbour. It can affect the accuracy and efficiency of the search process, as well as the risk of overfitting or underfitting. A small number of epochs may result in high variance in the validation accuracy and less stable search results, while a large number of epochs may lead to overfitting and poor generalization performance.
- $epoch_{final}$ is the number of epochs for final training. It can have a significant impact on the performance of the model. If the number of epochs is too low, the result may lead to underfitting. On the other hand, if the number of epochs is too high, the result may lead to overfitting.
- The initial learning rate λ_{start} is cosine annealed to λ_{end} during the SGDR training. It allows to adjust the learning rate over time to achieve better convergence and avoid oscillation or divergence during training. This approach can help the model to find a better minimum of the loss function and avoid getting stuck in a suboptimal solution. It can also prevent the model from overfitting to the training data by reducing the impact of noisy or irrelevant information in the later stages of training.

It is important to note that $epoch_{neighbours}$ should have a small value since a lot of networks need to be trained. The total number of epochs for training in our algorithm can be computed as

$$epoch_{total} = epoch_{neighbours} \cdot n_{neighbours} \cdot n_{steps} + epoch_{final} \tag{2.6}$$

3 State of the Art

3.1 Optimization of convolutional neural networks

Optimizing CNNs is an important task for achieving better performance and reducing computational resources. Over the years, variants of CNN architectures have been developed [Li+20], leading to amazing advances in the field of deep learning. We can observe this evolution by looking at the result of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), an annual computer vision competition held from 2010 to 2017 which aimed to advance the state-of-the-art in image classification and object detection. By looking at figure 3.1, we can see that the error rate fell steadily from 2010 to 2017. The top-5 error refers to the probability that all top-5 classifications proposed by the algorithm for the image are wrong. The algorithms with blue graph are CNN.

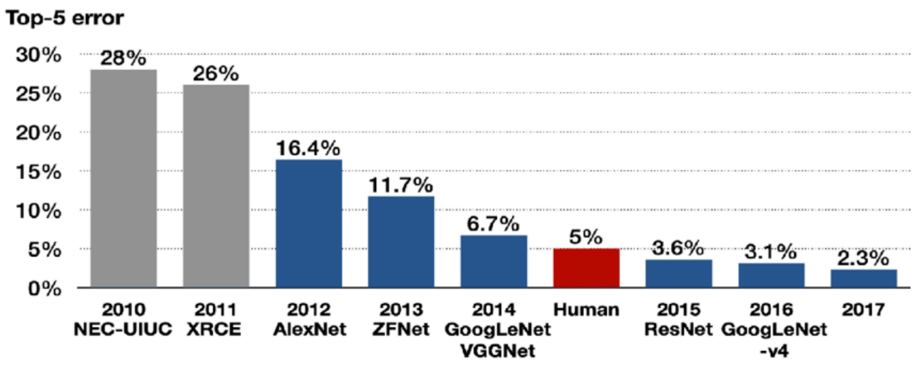


Figure 3.1: ILSVRC winning team's error rate each year
in the top-5 classification task. (image taken from [KDP20])

3.2 Hyperparameter optimization

NNs (and CNNs) are characterized by their set of hyperparameters that control the learning process. For this reason, the performance of a NN heavily depends on the value of various hyperparameters, such as learning rates and regularization constants. One set of values may result in better performance than another set. There exists a long line of research on automated methods for setting these hyperparameters.

3.2.1 Grid search

Grid search is simply an exhaustive search on every single combination of hyperparameters to find the best one. It must be guided by some performance metric, typically measured by cross-validation. [HCL+03]

3.2.2 Random search

Random search tests a certain number of combinations that are selected randomly. The benefit of this method is that it tests fewer model architectures, it requires less time and less computation to obtain results. [BB12]

3.2.3 Bayesian optimization

Bayesian optimization builds a probabilistic model of the function mapping from hyperparameter values to the objective evaluated on a validation set. It has been shown to obtain better results in fewer evaluations compared to grid search and random search, due to the ability to reason about the quality of experiments before they are run. [Ber+11] [SLA12] [Tho+13] [Feu+15]

3.2.4 Gradient descent optimization

Gradient descent optimization is to adjust the weights to minimize the loss or cost. This cost is to measure how well our model is doing. Thus, by minimizing the cost function we can find the optimal parameters that yield the best model performance. [Rud16]

3.3 Automated architecture search

Automating NAS is currently an active field of research. The reported approaches allow adaptation of a NN architecture to the wide area of tasks. Over the years, numerous NAS approaches have been proposed, leading to significant advancements in the field. While architectural choices can be treated as categorical hyperparameters and be optimized with standard hyperparameter optimization methods [Ber+11] [Men+16] [BB12] [Rud16], the current focus is on the development of *special* techniques for architectural optimization. In this section, we provide an overview of the state of the art in NAS, highlighting some of the key advancements and recent trends. The reinforcement learning and the evolutionary algorithm influenced this field the most.

3.3.1 Reinforcement learning

Reinforcement Learning (RL) involves learning the agent that samples the new NN architectures [Cai+17] [Bak+16] [Wil92]. The agent is trained to sequentially choose the type of layers (convolutional, pooling, fully-connected) and their parameters. The performance measures such as accuracy or validation loss are treated as a reward in RL applications [Zop+17]. One of the drawbacks of many RL methods is that they introduce other hyperparameters that are difficult to tune (for example the architecture of an agent NNs) and are associated with long-lasting training.

One example of NAS using RL is the work by Barret Zoph and Quoc V. Le in 2016 [ZL17]. The approach involves training a Recurrent Neural Network (RNN) controller to generate a sequence of actions that correspond to operations and connections in the NN architecture. The RNN is trained using a variant of policy gradient RL called REINFORCE, where the reward is based on the validation accuracy of the generated NN.

Another example is the work by Wei Wen et al. in 2018 [Zho+18a]. This paper presents a method for generating NN architectures in a block-wise manner, where each block corresponds to a group of layers in the network. The approach uses RL to search for the optimal configuration of each block, and then combines the blocks to form the final network architecture.

3.3.2 Evolutionary algorithms

Evolutionary Algorithm (EA) have played an important role in NAS [SM02]. This family of methods involves creating a set of architectures (called population) with random architecture. Each of the networks is then trained and evaluated on particular task e.g. classification of images. Based on the evaluation, the best network from the population can be selected to serve as the parent for offspring in the next epoch of the algorithm. Offspring are created by applying modification (mutations) in the structure of a parent network [Rea+17]. More complex algorithms allow combining structures of two parents network (crossover) [MTH89].

One example of NAS using EA is the work by Esteban Real et al. in 2019 [Rea+19]. The paper introduces a method for discovering NN architectures using a variant of EAs called regularized evolution. The approach involves creating a population of NN architectures and then using genetic operators to select the best architectures for further evaluation. The fitness of each architecture is based on its validation accuracy and its complexity, which is regularized using a combination of L1 and L2 regularization.

Additionally, we have the work by Lingxi Xie and Alan Yuille in 2017 [XY17]. This paper presents a method for evolving convolutional NN architectures that involves encoding each architecture as a binary string, which is then manipulated using genetic operators. The fitness of each architecture is based on its validation accuracy and the number of parameters, which are used to balance performance and complexity.

3.3.3 Meta-learning

Recently, there has been a growing interest in using meta-learning or few-shot learning techniques in NAS. These techniques aim to learn a meta-model that can quickly adapt to new tasks with limited data, and use this meta-model to generate architectures. They have shown promising results in discovering architectures that can achieve competitive performance with limited data, making them suitable for real-world applications where data availability is limited. Here are some real-world applications where data availability is limited.

For example, in medical image analysis, tasks such as image segmentation, lesion detection, and disease classification, often suffer from limited data availability due to factors such as privacy concerns, expensive data collection, and rare diseases. In the work of Abdelouahad Achmamad et al. in 2022 [AGR22], the authors present a few-shot learning approach for brain tumor segmentation, where a meta-learning algorithm is used to adaptively generate optimal NN architectures for segmenting brain tumors from limited labeled data.

Another example, in object detection, tasks such as pedestrian detection, object tracking, and vehicle detection, often have limited labeled data due to the need for manual annotation. In the work of Tong Yang et al. in 2018 [Yan+18], the authors introduce a meta-learning based approach for generating customized anchors for object detection with limited labeled data. Or in the work of Mingxing Tan et al. in 2020 [TPL20], the authors present an efficient object detection framework based on NAS, where a meta-learning algorithm is used to search for optimal neural network architectures for object detection tasks with limited labeled data.

3.3.4 Hardware-aware NAS

Hardware-aware NAS method aims to automatically search for architectures that can achieve the best performance on a specific hardware platform, such as a particular type of GPU, CPU, or edge device with limited computational resources. These methods take into account hardware-related factors such as the computational capacity, memory constraints, communication bandwidth, and power consumption, among others, to guide the architecture search process.

One example of hardware-aware NAS is the work by Bichen Wu et al. in 2019 [Wu+19]. The paper introduces FBNet which aims to design CNNs optimized for deployment on mobile and embedded devices . FBNet focuses on optimizing both the accuracy and efficiency of the network by leveraging a multi-objective EA to explore a large search space of possible architectures. It introduces a factorized hierarchical search space that allows for efficient and flexible architecture discovery.

Another example is the work by Andrew Howard et al. in 2019 [How+19]. The paper introduces MobileNetV3, a family of efficient CNNs designed for mobile and embedded devices. MobileNetV3 incorporates various design strategies such as network width, depth, and efficient operations like depthwise separable convolutions. It introduces a new design block called *MobileNetV3 block* that is more efficient and flexible than its predecessor, along with other architectural improvements such as squeeze-and-excitation blocks and hard-swish activation function.

3.4 Network morphism

Network morphism, as already explained in section 2.7, involves iteratively modifying a base neural network architecture in order to produce new architectures that are functionally equivalent but structurally different. This approach can be used to search for new NN architectures that are more efficient, accurate, or specialized to a particular task. Here are some recent examples that uses network morphism.

For example, we have the work of Han Cai et al. in 2020 [Cai+20]. The authors proposed the Once-for-All method, which adopts a progressive shrinking algorithm to train a single model that can be used as a *blueprint* for other models. This *blueprint* model can be used to create many different sub-models with different sizes and architectures. Moreover, the parameters learned by the *blueprint* model can be reused during the progressive shrinking for the creation of sub-models.

Another recent example is the work of Suman Sapkota and Binod Bhattarai in 2022 [SB22]. The paper presents a new method called Noisy Heuristics NAS, which uses heuristics inspired by biological neuronal dynamics to add and remove neurons and control the number of layers in the network. The method can adjust the model's capacity or non-linearity online using meta-parameters specified by the user.

4 Methodology

4.1 Objectives

In this thesis, we want to implement the NASH algorithm [EMH17] to study the different hyperparameters and ways of improving it. The code will be done using the *Python* programming language with the addition of the libraries *PyTorch* [Pas+19] and *fastai* [HG20] which are well known libraries for AI and ML.

4.2 Materials

Considering the computational load of the NN algorithms, we will use GPU to implement them to reduce the execution time. All experiments are done via the desktop virtualization service *Paperspace* [Pap]. The virtual machine has the NVIDIA Quadro M4000 8GB as GPU, the Intel Xeon E5-2623 v4 (base frequency of 2.60 GHz and a max turbo frequency of 3.20 GHz) as CPU and 30 GB of RAM. The computer runs on Windows 10 so only 1 GPU is used for the experiments.
We will use a base implementation found on GitHub [zhe21]. All revisions and extensions to the pre-existing code for the sake of our experimentations can be found on our GitHub [Zho23].

4.3 Datasets

The datasets used will be CIFAR-10, CIFAR-100 [KH+09] and SVHN [Net+11]. The training data will be augmented using random cropping, random horizontal flip and cutout (randomly masking out square regions of the input [DT17]) to improve the models' generalization.

4.3.1 CIFAR-10

The CIFAR-10 dataset is a collection of images containing 60,000 32x32 color images of 10 object classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

4.3.2 CIFAR-100

The CIFAR-100 dataset is just like the CIFAR-10 dataset, except it has 100 classes which are grouped into 20 superclasses. Each image comes with a ”fine” label (the class to which it belongs) and a ”coarse” label (the superclass to which it belongs).

4.3.3 SVHN

The Street View House Numbers (SVHN) dataset is a real-world image consisting of images of house numbers taken from Google Street View images. The images are color and have been cropped and resized to 32x32 pixels. The dataset includes over 600,000 images for training and testing.

4.4 Initial network

The choice of the initial network is a crucial part of the algorithm. A good initial network can help the training process converge more quickly and can result in better performance, while a poor initial network may lead to slower convergence and worse performance [GB10]. We choose to work with a simple CNN with the following structure Conv-BN-ReLu-MaxPool-Conv-BN-ReLu-MaxPool-Conv-BN-ReLu-FC-Softmax where Conv stands for convolutional layers, with 64 filters, 3x3 kernel, and stride 1, BN stands for batch normalization, MP stands for MaxPooling with 2x2 kernel and stride 2. A visual representation can be found in the appendix A. This network is pretrained for 20 epochs.

4.5 Experiments

In these experiments, we are interested to measure the accuracy, the loss, the execution time, the number of nodes and the number of trainable parameters. All experiments will be run 5 times on our 3 datasets (CIFAR-10,CIFAR-100 and SVHN). We will try to avoid modifying parameters that affects the total number of epochs for training (cfr equation 2.6) as these parameters should have a smaller impact on the model’s performance compared to the others.

4.5.1 First half

In this first part, we aim to improve the performances of our NASH algorithm by fine-tuning its hyper-parameters. After these experiments, we will select the combination with the best results to proceed with the second half of our experiments.

Initial network training

We train our models for 200 epochs without using the NASH algorithm. This allows us to see the potential of our base structure compared to the following experiences.

Base case

We set the following parameters for our NASH algorithm: $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 5$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{end} = 0.00$. With this, the models that are returned by our algorithm are trained for a total number of $17 \cdot 8 \cdot 5 + 100 = 780$ epochs. This will also serve as base of comparison with the others algorithms.

Modification of the n_{NM} hyperparameter

We double the number of network morphisms so $n_{NM} = 10$ and the other parameters remained unchanged. Adding more network morphisms can increase the flexibility of the search space, allowing for a wider range of architectures to be explored. This can potentially lead to better performing architectures being discovered. However, adding too many network morphisms can also increase the complexity of the search space, making it more difficult to find good architectures. So we expect the execution time to increase but in exchange of an increase in the performances of our models.

Modification of the learning rate scheduling techniques

We will use the one-cycle learning rate policy instead of the cosine annealing schedule. In this technique, the learning rate is gradually increased to a maximum value and then gradually decreased to a minimum value over the course of a single epoch [Smi17] [ST18]. This can be effective for faster convergence and better generalization. The start learning rate λ_{start} is set to 0.05 and the max learning rate is set to 0.1, the double. We expect to reach the same performance as the base case but with less execution time.

Modification of both n_{NM} and learning rate scheduling

We will also try to combine both of our methods, changing the number of network morphisms to 10 so $n_{NM} = 10$ and using the one-cycle learning rate policy instead of the cosine annealing schedule. This should allow us to be exhaustive in our experimentation and will be useful later for our analysis. We expect the execution time to increase but in exchange of an increase in the performances of our models.

4.5.2 Second half

In this second part, we aim to increase the performances of our NASH algorithm while also reducing its computational cost by incorporating regularization techniques.

Adding dropout

Dropout is a regularization technique that involves randomly setting some of the neurons to zero during training to force the network to learn more robust features, which can help prevent overfitting and improve generalization performance [Hin+12]. During evaluation, the dropout module is turned off and the full network is used to make predictions. The commonly utilized dropout rate for NNs is 0.5. However, a recent study suggests that a dropout rate of 0.2 is more suitable for CNNs [PK17]. We are not sure what kind of impact this will have on the performances and execution time.
In our experiment, we add a dropout layer after each convolutional layer. This implies the need of a new initial network because we change the initial structure of our CNN, a visual representation can be found in the appendix B. For that, we will follow the same method as the previous one (cfr section 4.4) and pretrained it for 20 epochs. We will also train it for 200 epochs without using the NASH algorithm to see the potential of this new initial structure.

Adding early stopping

Early stopping is a regularization technique that involves monitoring the validation loss during training and stopping the training process when the validation loss stops improving after a certain number of epochs by using the *patience* parameter [Pre12] [GJP98]. This can help prevent overfitting and improve generalization performance. In our experiment, the *patience* parameter is set to $10\% + 1$ of the number of training epochs. We expect to reach the same performance but with less execution time.

Adding both dropout and early stopping

We will also try to add both of our methods, dropout with a dropout rate of 0.2 and early stopping with the *patience* parameter set to $10\% + 1$ of the number of training epochs. We dub this combination *dropout early*. This should allow us to be exhaustive in our experimentation and will be useful later for our analysis. We are not sure what kind of impact this combination will have on the performances and execution time of our models.

4.5.3 Third half

In this third part, we are interested in the added value of network morphisms.

Retraining from scratch

Network morphisms in the NASH algorithm allow for simultaneous training and generation of the model. However, it remains unclear whether the final architecture discovered through this process performs better compared to training the final architecture discovered from scratch. To investigate this, we propose to reinitialize the model weights to zero and train it for 200 epochs. We will dub the entire process of using the NASH algorithm to find the model then resetting its weight to zero *NASH-reset*. We will apply this method to the best results found in the first and second halves.

4.5.4 Analysis

After all the experiments, we will analyse a bit our results. First of all, we are interested in comparing them in terms of accuracy, loss, execution time and number of trainable parameters. To do so, we will use box-plot to visualize more clearly the difference between each method and realize some two-way Analysis of Variance (ANOVA) tests. These tests allows us to confirm if there is a statistically significant difference between the means of our different experiments by looking at the p-value. If the p-value is less than 0.05, we have sufficient evidence to say that the mean values across each group are not equal and that there is a statistically significant difference between the means of the group tested. In the other case, if it is above 0.05, there is not a statistically significant difference between the means of the group tested. And secondly, we will compare our results with the current state of the art and the results of the base paper [EMH17]. In addition to this comparison, we will create a scatter plot of the accuracy versus the number of parameters. This will help us identify the Pareto frontier, which represents the optimal solutions that strike a balance between accuracy and the number of parameters.

5 Results

5.1 Initial network

In table 5.1 are showed the results of our initial network with the structure Conv-BN-ReLu-MaxPool-Conv-BN-ReLu-MaxPool-Conv-BN-ReLu-FC-Softmax after 20 epochs on our 3 datasets.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	68.97	45.73	61.64
Loss	0.937	2.121	1.140
# Nodes	12	12	12
# Trainable Parameters (x10^6)	0.54	2.01	0.54

Table 5.1: Results of our initial network after 20 epochs

As we can see, the results are average for the accuracy but as the models serve as a foundation for our algorithm it is better than having no basis at all.

5.2 First half

In this first part, we aim to improve the performances of our NASH algorithm regardless of the execution time or the complexity of our models. But measuring them could prove useful to observe their behaviour regarding the changing hyperparameter.

5.2.1 Initial network training

We can look at the results of our initial network trained for 200 epochs without using the NASH algorithm in the table 5.2.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	82.07 ± 0.39	60.96 ± 0.45	86.20 ± 0.22
Loss	0.5435 ± 0.0129	1.5781 ± 0.0182	0.4540 ± 0.0077
Execution Time (hrs)	2.22 ± 0.00	2.33 ± 0.00	3.33 ± 0.00
# Nodes	12.00 ± 0.00	12.00 ± 0.00	12.00 ± 0.00
# Trainable Parameters (x10^6)	0.54 ± 0.00	2.01 ± 0.00	0.54 ± 0.00

Table 5.2: Results of our initial network trained for 200 epochs

The findings demonstrate that the results are already much better compared to the pretrained version (cfr table 5.1) if we look at the accuracy and the loss.

5.2.2 Base case

In table 5.3, we present the results of our NASH algorithm with $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 5$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{end} = 0.00$.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	91.05 ± 0.85	68.89 ± 1.39	95.23 ± 0.71
Loss	0.3133 ± 0.0037	1.3211 ± 0.0379	0.1859 ± 0.0021
Execution Time (hrs)	26.73 ± 1.42	47.16 ± 5.52	33.96 ± 2.96
# Nodes	99.80 ± 12.73	89.60 ± 7.65	88.20 ± 7.95
# Trainable Parameters (x10^6)	10.26 ± 6.74	14.64 ± 8.19	9.62 ± 6.86

Table 5.3: Results of our base case

The results indicate that they are pretty good. We manage to reach correct accuracies but not excellent accuracies in a fair amount of time. However, this is due to the fact that our models are quite complex by looking at the number of nodes and the number of trainable parameters. We can already see this outperforms the initial network after 200 epochs (cfr table 5.2).

5.2.3 Modification of the n_{NM} hyperparameter

Table 5.4 shows the results of our NASH algorithm with $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 10$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{end} = 0.00$.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	89.78 ± 1.45	69.06 ± 1.12	95.52 ± 0.58
Loss	0.3317 ± 0.0048	1.4563 ± 0.0070	0.1703 ± 0.0074
Execution Time (hrs)	89.84 ± 25.35	120.21 ± 25.38	99.79 ± 3.56
# Nodes	160.40 ± 13.48	217.00 ± 18.83	133.40 ± 23.01
# Trainable Parameters (x10^6)	57.12 ± 20.44	49.37 ± 30.41	19.74 ± 15.71

Table 5.4: Results with $n_{NM} = 10$

While increasing the number of network morphisms did not improve the performance for CIFAR-10, there is a slight improvement for CIFAR-100 and SVHN. However, the execution time increased considerably and the models became much more complex than the base case (cfr table 5.3). In addition to that, we can observe that the standard deviation is quite high. This can be explained by the fact that increasing the number of network morphism increases the chance of our models to become more complex depending on which network morphism has been chosen.

5.2.4 Modification of the learning rate scheduling techniques

We can look at the results of our NASH algorithm with $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 5$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{max} = 0.1$ and with one-cycle in the table 5.5.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	91.08 ± 0.97	70.32 ± 1.21	95.48 ± 0.38
Loss	0.3137 ± 0.0046	1.2060 ± 0.0092	0.1835 ± 0.0041
Execution Time (hrs)	25.54 ± 2.44	36.51 ± 2.33	29.88 ± 6.01
# Nodes	82.00 ± 12.79	109.60 ± 14.48	88.80 ± 8.56
# Trainable Parameters (x10^6)	18.04 ± 8.71	9.41 ± 5.96	12.20 ± 10.85

Table 5.5: Results with one-cycle

We can see that the results are slightly better than the base case (cfr table 5.3) but with less execution time. However, the models have become more complex by looking at either the number of nodes or the number of trainable parameters or both.

5.2.5 Modification of both n_{NM} and learning rate scheduling

Table 5.6 shows the results of our NASH algorithm with $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 10$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{max} = 0.1$ and with one-cycle.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	91.91 ± 0.43	68.77 ± 1.07	94.35 ± 1.01
Loss	0.3066 ± 0.0051	1.4278 ± 0.0962	0.2218 ± 0.0104
Execution Time (hrs)	74.80 ± 27.82	113.10 ± 24.13	70.87 ± 3.57
# Nodes	158.00 ± 42.65	182.60 ± 37.14	167.60 ± 36.36
# Trainable Parameters (x10^6)	42.83 ± 25.33	36.25 ± 32.31	24.31 ± 23.59

Table 5.6: Results with $n_{NM} = 10$ and one-cycle

A slight increase of performance only for CIFAR-10 can be observed compared to the base case (cfr table 5.3) but the models still remain very complex. Now if we compared to the results of $n_{NM} = 10$ (cfr table 5.4), we can see that the addition of one-cycle reduce the execution time for all datasets and the complexity but only for CIFAR-10 and CIFAR-100.

5.2.6 Best results

These experiments show that our NASH algorithm is quite robust, the performance is stable and reliable. Overall, the models with the best performances are the ones with one-cycle (cfr table 5.5), they beat the base case by a small margin for an increase in their complexity. We will use this combination of hyperparameters for the following experiments.

5.3 Initial network with dropout

Table 5.7 shows the results of our initial network with the structure Conv-BN-ReLu-Drop-MaxPool-Conv-BN-ReLu-Drop-MaxPool-Conv-BN-ReLu-Drop-FC-Softmax after 20 epochs on our 3 datasets.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	65.38	39.40	75.31
Loss	0.994	2.367	0.751
# Nodes	15	15	15
# Trainable Parameters (x10^6)	0.54	2.01	0.54

Table 5.7: Results of our initial network with dropout after 20 epochs

As we can see, the results are average by looking at the accuracy and only SVHN has better result than the initial network without dropout (cfr table 5.1).

5.4 Second half

In this second part, we aim to increase the performances of our NASH algorithm while also reducing its computational cost.

5.4.1 Initial network with dropout training

We can look at the results of our initial network with dropout trained for 200 epochs without using the NASH algorithm in the table 5.8.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	79.62 ± 0.24	57.33 ± 0.10	89.27 ± 0.21
Loss	0.5931 ± 0.0073	1.6061 ± 0.0053	0.3688 ± 0.0064
Execution Time (hrs)	2.22 ± 0.00	2.33 ± 0.00	3.33 ± 0.00
# Nodes	15.00 ± 0.00	15.00 ± 0.00	15.00 ± 0.00
# Trainable Parameters (x10^6)	0.54 ± 0.00	2.01 ± 0.00	0.54 ± 0.00

Table 5.8: Results of our initial network with dropout trained for 200 epochs

The observed results are already much better compared to the pretrained version (cfr table 5.7) if we look at the accuracy and the loss.

5.4.2 Adding dropout

In table 5.9, we can see the results of our NASH algorithm with $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 5$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{max} = 0.1$ and with one-cycle and with a dropout layer after each convolutional layer with a dropout rate of 0.2.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	91.74 ± 0.68	70.17 ± 1.48	95.33 ± 0.49
Loss	0.2975 ± 0.0281	1.0884 ± 0.0802	0.1729 ± 0.0172
Execution Time (hrs)	32.54 ± 15.70	51.11 ± 10.63	52.91 ± 9.36
# Nodes	89.20 ± 7.41	86.60 ± 8.01	88.20 ± 9.02
# Trainable Parameters (x10^6)	11.41 ± 6.95	11.42 ± 5.67	8.82 ± 2.94

Table 5.9: Results with dropout

The addition of dropout allows to decrease the loss and even make our models less complex whether by decreasing the number of nodes or the number of trainable parameters or both compared to the case without dropout (cfr table 5.5). Nevertheless, this improvements comes with an increase in the execution time.

5.4.3 Adding early stopping

We can look at the results of our NASH algorithm with $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 5$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{max} = 0.1$ and with one-cycle and with early stopping with the *patience* parameter set to $10\% + 1$ of the number of training epochs in table 5.10.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	87.14 ± 0.88	65.91 ± 0.71	93.26 ± 0.91
Loss	0.4284 ± 0.0224	1.3517 ± 0.0332	0.2461 ± 0.0311
Execution Time (hrs)	9.27 ± 0.79	14.39 ± 3.08	16.91 ± 1.85
# Nodes	107.80 ± 6.96	102.60 ± 4.58	98.00 ± 12.85
# Trainable Parameters (x10^6)	7.78 ± 3.12	8.73 ± 2.05	8.67 ± 6.99

Table 5.10: Results with early stopping

We can see that adding early stopping decreases the performances of our models by a small amount but considerably reduces the execution time compared to the case without early stopping (cfr table 5.5). The models also become less complex if we look at the number of trainable parameters.

5.4.4 Adding both dropout and early stopping

Table 5.11 shows the results of our NASH algorithm with $n_{steps} = 5$, $n_{neighbours} = 8$, $n_{NM} = 5$, $epoch_{neighbours} = 17$, $epoch_{final} = 100$, $\lambda_{start} = 0.05$, $\lambda_{max} = 0.1$ and with one-cycle and with dropout early.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	89.46 ± 0.79	67.61 ± 1.25	95.29 ± 0.35
Loss	0.3283 ± 0.0217	1.2121 ± 0.0274	0.1785 ± 0.0114
Execution Time (hrs)	13.69 ± 2.87	13.21 ± 1.63	19.36 ± 3.05
# Nodes	94.60 ± 3.44	88.60 ± 7.22	91.60 ± 10.66
# Trainable Parameters (x10^6)	5.15 ± 3.022	15.55 ± 6.08	10.39 ± 6.99

Table 5.11: Results with dropout early

The addition of dropout early allows us to get close to the results of the case without dropout early (cfr table 5.5) but for less execution time. The same observation can be made compared to the case with only dropout (cfr table 5.9).

5.4.5 Best results

We consider the case with dropout early to be the best results of this second half. Even though there is a small drop in performances, we think the decrease in the execution time is far more important in the context of our research.

5.5 Third half

In this third part, we are interested in the added value of network morphisms.

5.5.1 Retraining from scratch

We can look at the results of our weight reset and training for 200 epochs on one-cycle in table 5.12 and on dropout early in table 5.13, note that early stopping was deactivated for this experiment. Moreover, the execution time shown in both tables refers specifically to the weight reset and the training. However, it is important consider the entire NASH algorithm process for finding the final architecture. Therefore, we have introduced a new row called *total execution time*, which represents the time of the entire NASH-reset process.

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	88.52 ± 1.10	66.91 ± 1.29	93.47 ± 0.91
Loss	0.4475 ± 0.0328	1.5022 ± 0.0991	0.3110 ± 0.0687
Execution Time (hrs)	10.07 ± 1.88	11.77 ± 2.46	16.49 ± 5.18
Total Execution Time (hrs)	35.61 ± 3.08	48.28 ± 3.39	46.37 ± 7.93
# Nodes	82.00 ± 12.79	109.60 ± 14.48	88.80 ± 8.56
# Trainable Parameters (x10^6)	18.04 ± 8.71	9.41 ± 5.96	12.20 ± 10.85

Table 5.12: Results of the weight reset and training for 200 epochs on one-cycle

Dataset	CIFAR-10	CIFAR-100	SVHN
Accuracy (%)	91.06 ± 0.16	68.17 ± 0.64	94.19 ± 1.28
Loss	0.2940 ± 0.0213	1.2701 ± 0.0781	0.1986 ± 0.0324
Execution Time (hrs)	11.06 ± 2.24	16.23 ± 6.63	17.82 ± 6.14
Total Execution Time (hrs)	24.75 ± 3.64	29.44 ± 6.83	37.18 ± 6.86
# Nodes	94.60 ± 3.44	88.60 ± 7.22	91.60 ± 10.66
# Trainable Parameters (x10^6)	5.15 ± 3.022	15.55 ± 6.08	10.39 ± 6.99

Table 5.13: Results of the weight reset and training for 200 epochs on dropout early

Comparing the results, we observe that using NASH-reset on one-cycle leads to a slight decrease in model performances compared to the case without (cfr table 5.5). However, the performances of dropout early improve compared to the case without (cfr table 5.11) and become comparable to the one-cycle without dropout early (cfr table 5.5). Moreover, we can notice that the total execution time is still lower than the execution time of the one-cycle.

6 Analysis

We will now use boxplots to provide a visual summary of the distribution of our results for our different methods. This will show the median, the middle value of a dataset when it is arranged in order. It is considered a *better* measure of the middle value than the mean because it is less sensitive to outliers. An outlier is an observation that is significantly different from other observations and can have a significant impact on the results of statistical analyses e.g. the mean. After that, we will conduct a two-way ANOVA analysis to determine whether there is a relationship between the number of network morphisms and the learning rate scheduler, as well as between dropout and early stopping. The aim of this analysis is to identify whether these hyperparameters are independent or dependent on each other in their effects on the accuracy and the execution time, which are the primary metrics of interest in this research.

6.1 First half

6.1.1 Box-plot

Accuracy

Figure 6.1 illustrates that the boxplots for both the base case and the one-cycle case exhibit similar shapes and median positions compared to the other cases. Similarly, in figure 6.2, the boxplot for the base case and the $n_{MN} = 10$ case demonstrate similarities in terms of shape and median position compared to the other cases. In figure 6.3, the boxplots for the base case, the $n_{MN} = 10$ case and the one-cycle case share a similar median position. However, it is worth noting that the base case and the one-cycle case display an outlier.

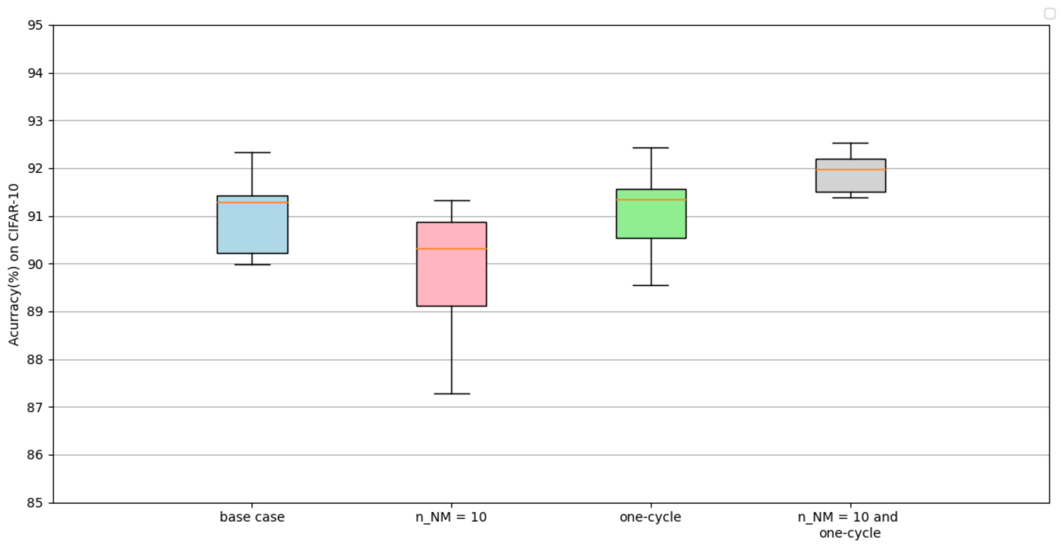


Figure 6.1: Box-plot of the accuracy on CIFAR-10

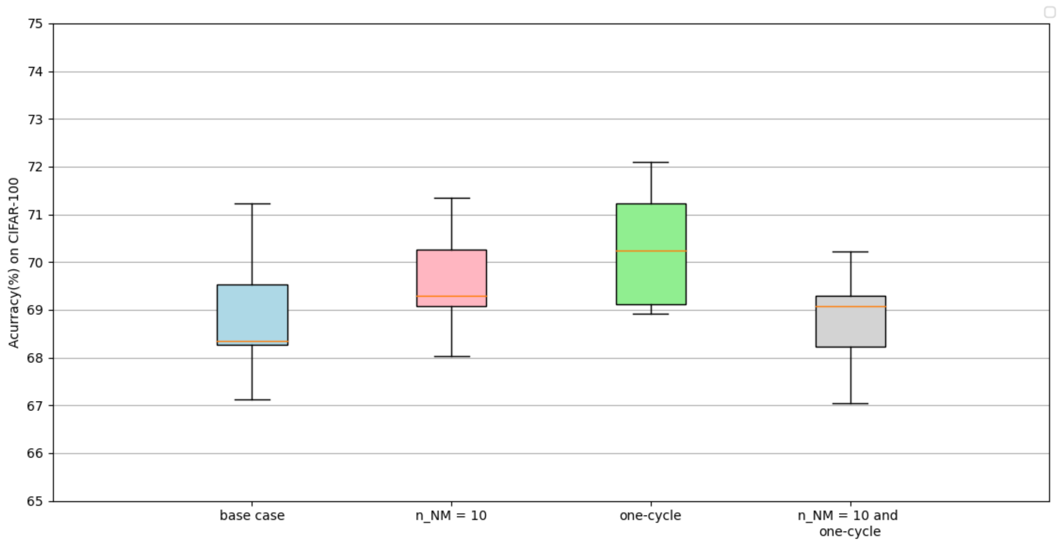


Figure 6.2: Box-plot of the accuracy on CIFAR-100

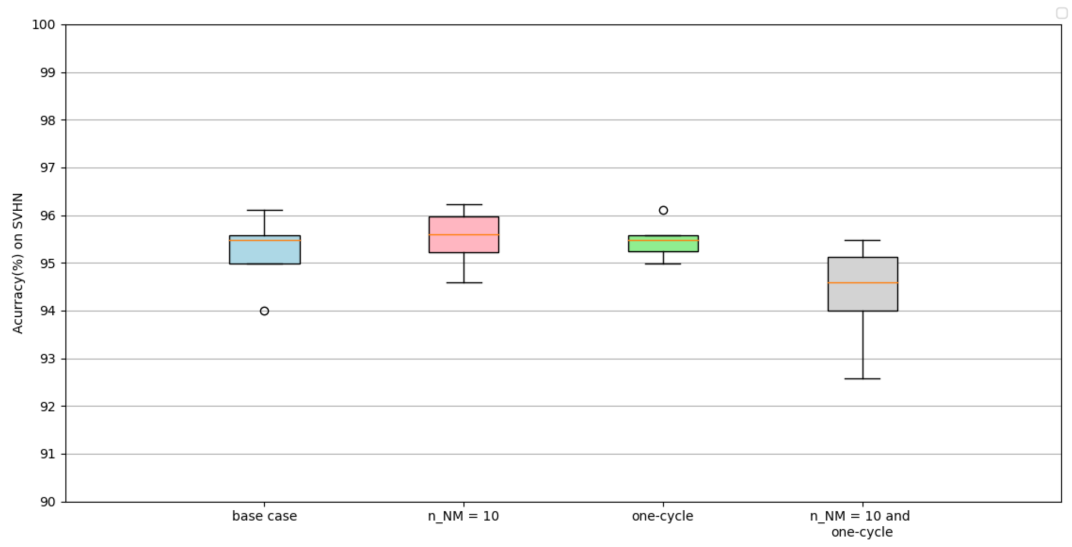


Figure 6.3: Box-plot of the accuracy on SVHN

Loss

As we can see in figure 6.4, the boxplot for the base case and the one-cycle case demonstrate similar shapes and median positions when compared to the other cases. In figure 6.5, the boxplots for all cases exhibit significant differences in position, but only the boxplot for the $n_{MN} = 10$ and one-cycle cases display a distinct shape. Notably, the base case includes an outlier. Figure 6.6 illustrates that the boxplots for the base case and the one-cycle case resemble each other in terms of shape and median position relative to the other cases. However, the base case contains one outlier, while the one-cycle case has two outliers.

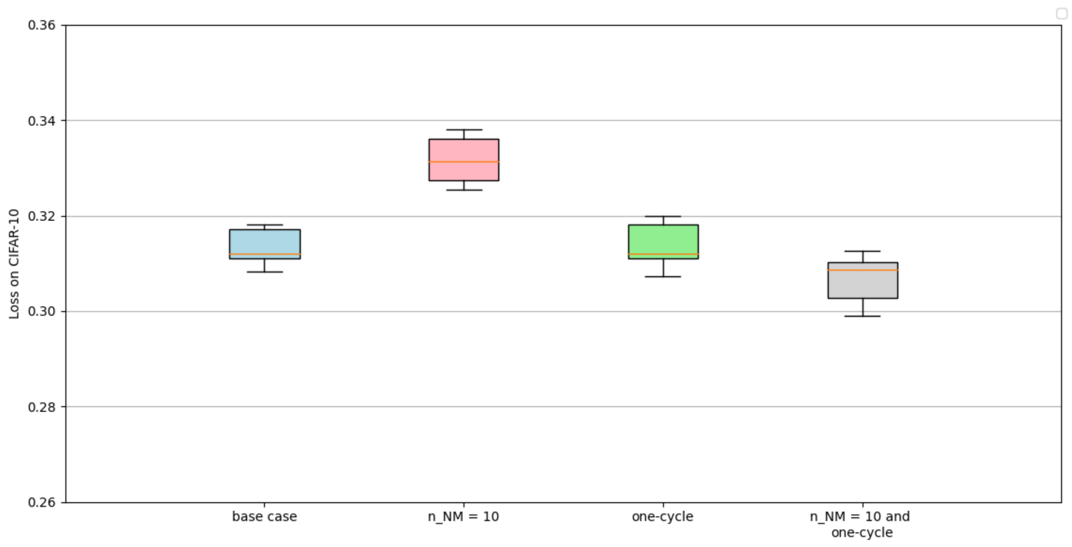


Figure 6.4: Box-plot of the loss on CIFAR-10

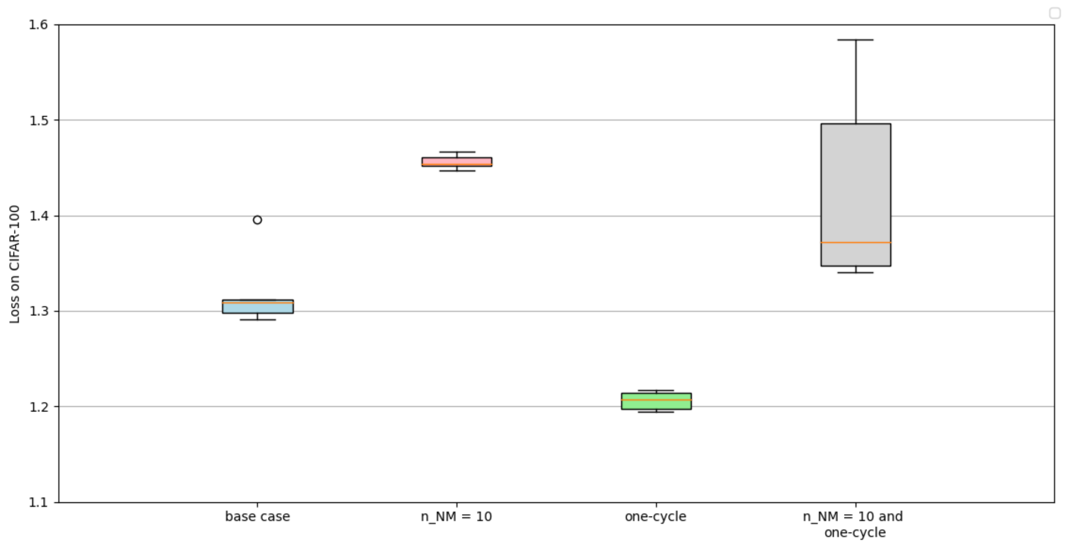


Figure 6.5: Box-plot of the loss on CIFAR-100

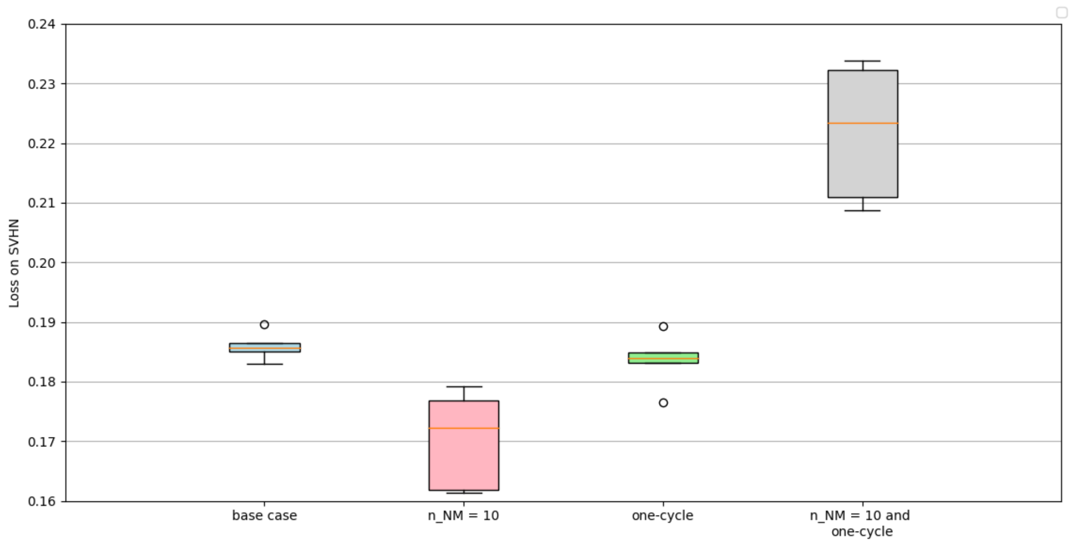


Figure 6.6: Box-plot of the loss on SVHN

Execution time

As we can see in figure 6.7, the boxplot for the base case and the one-cycle case share a similar position and both exhibit an outlier. While in figure 6.8, the boxplots for all cases show significant differences in both position and shape. Notably, the $n_{MN} = 10$ case and the $n_{MN} = 10$ with one-cycle case include an outlier. Figure 6.9 reveals substantial differences in position and shape among the boxplots for all cases. It is noteworthy that the base case and the one-cycle case contain two outliers.

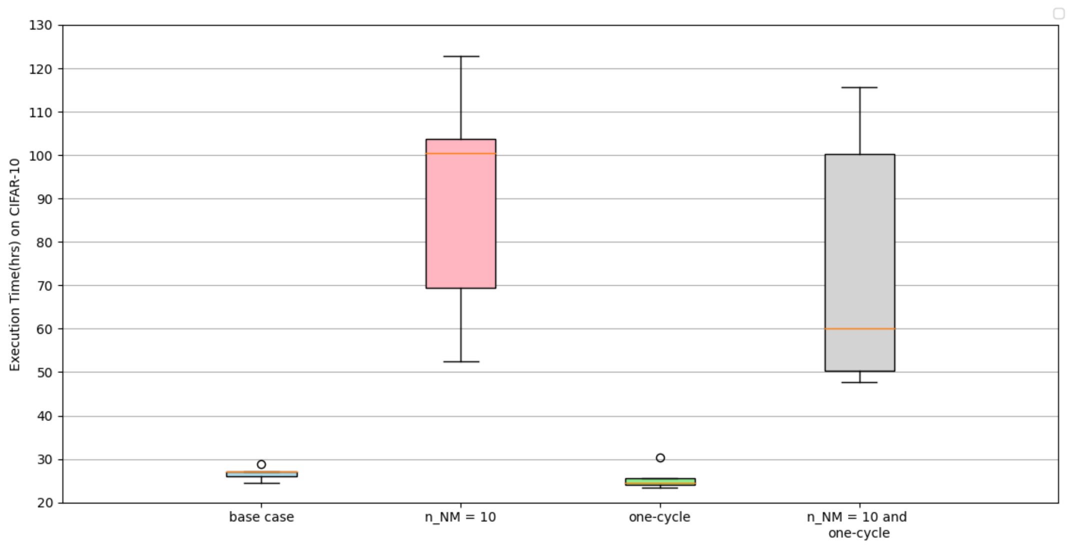


Figure 6.7: Box-plot of the execution Time on CIFAR-10

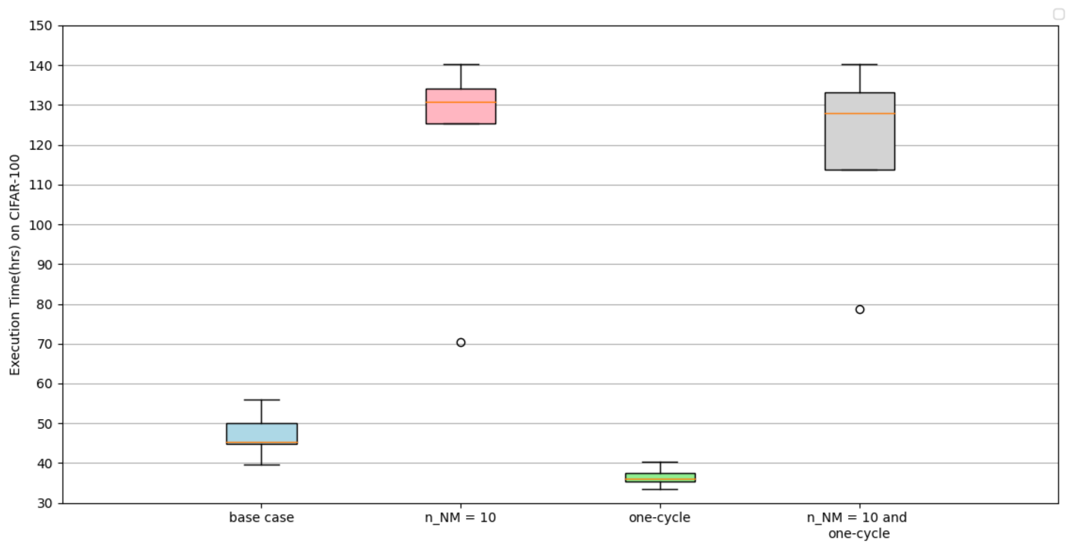


Figure 6.8: Box-plot of the execution Time on CIFAR-100

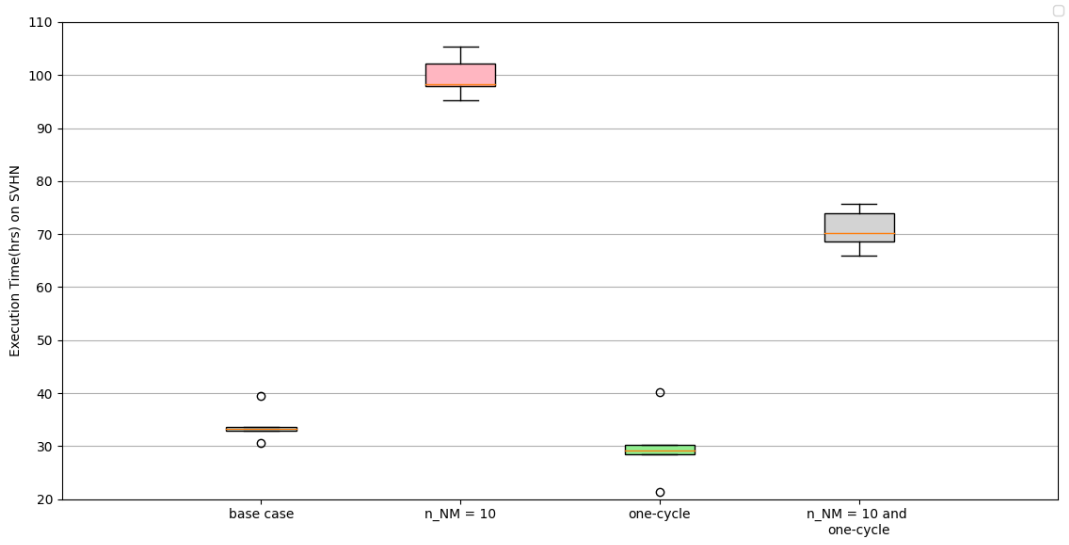


Figure 6.9: Box-plot of the execution Time on SVHN

Number of trainable parameters

Figure 6.10 displays boxplots for all cases, revealing variations in both position and shape among them. In figure 6.11, the boxplots for all cases demonstrate differences in both position and shape. Notably, the base case and the one-cycle case exhibit an outlier. Figure 6.12 illustrates that the boxplot for the base case and the one-cycle case share similar shapes and median positions when compared to the other cases. It is worth mentioning that the one-cycle case includes an outlier.

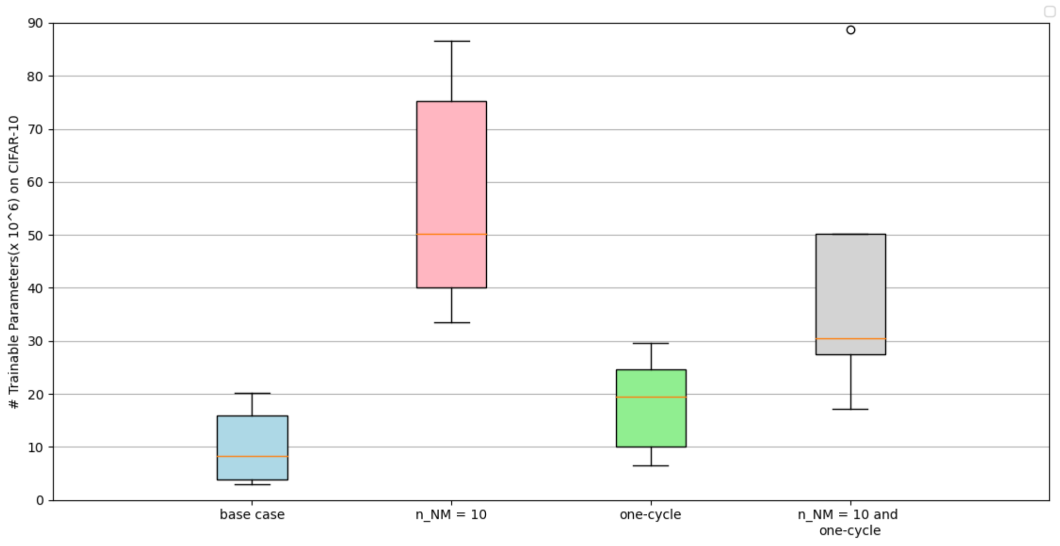


Figure 6.10: Box-plot of the number of trainable parameters on CIFAR-10

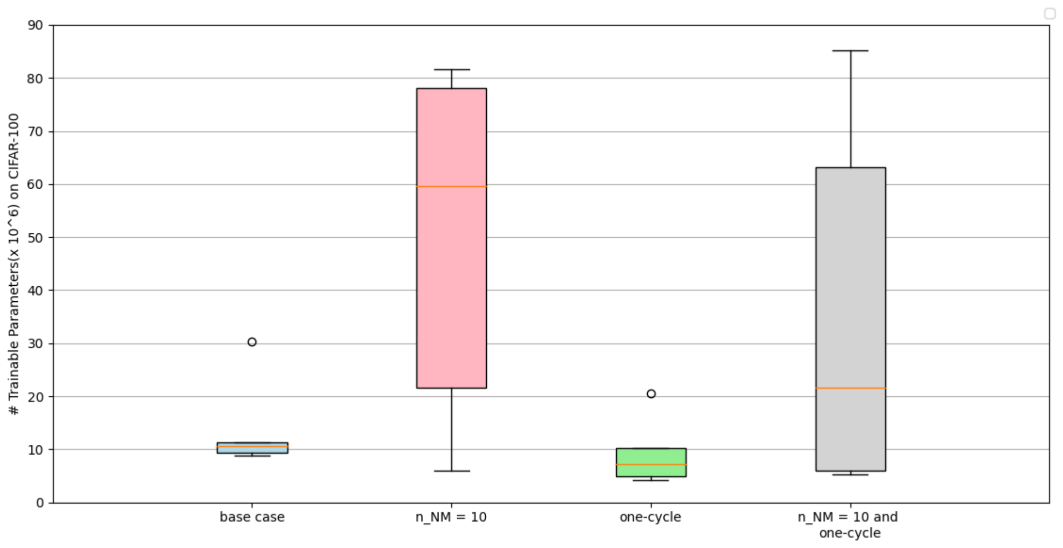


Figure 6.11: Box-plot of the number of trainable parameters on CIFAR-100

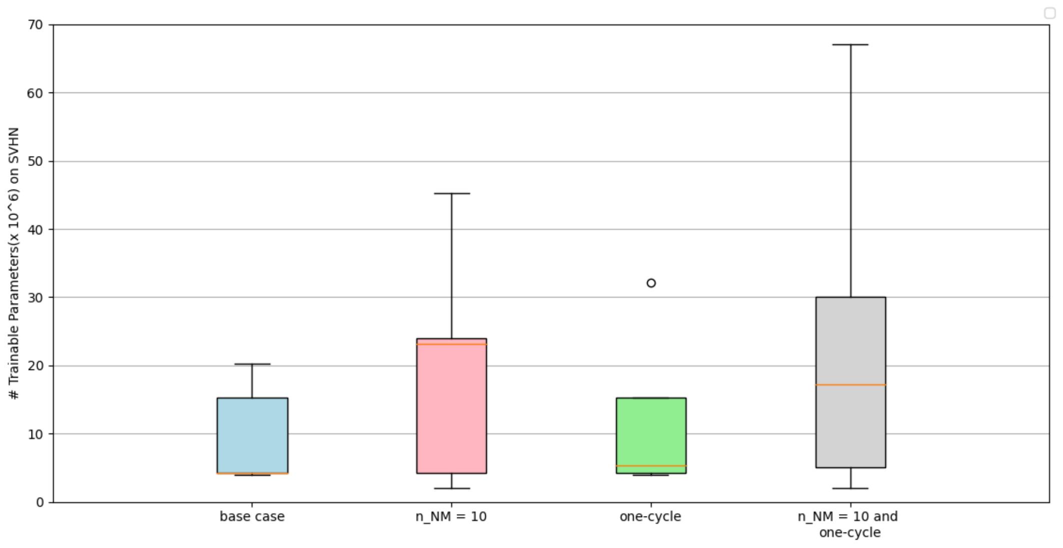


Figure 6.12: Box-plot of the number of trainable parameters on SVHN

6.1.2 Two-way ANOVA

Accuracy

If we take a look at the last column (which represents the p-value) in table 6.1, we can see that, except for the one-cycle in CIFAR-10, every p-values are higher than 0.05. From this, we can conclude that only the one-cycle has an effect on the accuracy but only for CIFAR-10 and that the n_{MN} parameter and one-cycle are independent from each other for the accuracy.

Accuracy on CIFAR-10					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
n_{MN}	1.000000	0.235445	0.235445	0.189581	0.669082
one-cycle	1.000000	5.864445	5.864445	4.722061	0.045142
n_{MN} + one-cycle	1.000000	5.523005	5.523005	4.447132	0.051072
Residual	16.000000	19.870800	1.241925	NaN	NaN
Accuracy on CIFAR-100					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
n_{MN}	1.000000	0.903125	0.903125	0.496252	0.491279
one-cycle	1.000000	0.447005	0.447005	0.245622	0.626913
n_{MN} + one-cycle	1.000000	6.350645	6.350645	3.489571	0.080182
Residual	16.000000	29.118280	1.819892	NaN	NaN
Accuracy on SVHN					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
n_{MN}	1.000000	0.882000	0.882000	1.392183	0.255283
one-cycle	1.000000	1.058000	1.058000	1.669988	0.214613
n_{MN} + one-cycle	1.000000	2.534720	2.534720	4.000900	0.062745
Residual	16.000000	10.136600	0.633538	NaN	NaN

Table 6.1: Two-way ANOVA test on the accuracy for the n_{MN} parameter and the learning rate scheduler

Execution time

If we take a look at the last column (which represents the p-value) in table 6.2, we can see that, without surprise, the p-value of the n_{MN} parameter on all datasets is lower than 0.05 meaning it has an impact on the execution time. In addition, the one-cycle also has an impact on the execution time but only for SVHN. Furthermore, the p-value of the combination of the n_{MN} parameter and the one-cycle being lower than 0.05, we can conclude that these two parameters are dependent on each other for the execution time but only for SVHN.

Execution time on CIFAR-10					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
n_{MN}	1.000000	15782.897460	15782.897460	35.456353	0.000020
one-cycle	1.000000	329.441026	329.441026	0.740091	0.402339
n_{MN} + one-cycle	1.000000	239.847034	239.847034	0.538817	0.473544
Residual	16.000000	7122.175287	445.135955	NaN	NaN
Execution time on CIFAR-100					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
n_{MN}	1.000000	27988.856107	27988.856107	70.927671	0.000000
one-cycle	1.000000	394.241329	394.241329	0.999063	0.332415
n_{MN} + one-cycle	1.000000	15.687069	15.687069	0.039753	0.844476
Residual	16.000000	6313.779791	394.611237	NaN	NaN
Execution time on SVHN					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
n_{MN}	1.000000	14264.136721	14264.136721	647.843697	0.000000
one-cycle	1.000000	1360.940676	1360.940676	61.810739	0.000001
n_{MN} + one-cycle	1.000000	771.614246	771.614246	35.044913	0.000022
Residual	16.000000	352.285881	22.017868	NaN	NaN

Table 6.2: Two-way ANOVA test on the execution time for
the n_{MN} parameter and the learning rate scheduler

6.2 Second half

6.2.1 Box-plot

Accuracy

In figure 6.13, the boxplots for all cases exhibit variations in position and shape. Notably, the dropout early case includes two outliers. Figure 6.14 illustrates that the boxplot for the one-cycle case and the dropout case share a similar median position but differ in terms of shape. It is worth mentioning that the dropout case contains an outlier. As depicted in figure 6.15, the boxplots for the one-cycle case, dropout case and dropout early case display a similar median position but differ in shape. It is worth noting that both the one-cycle case and the early stopping case have an outlier.

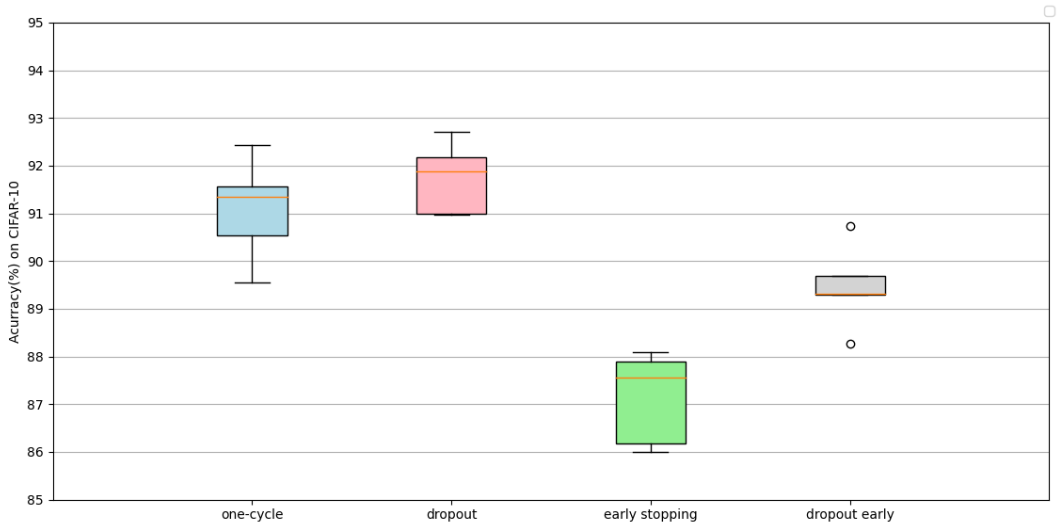


Figure 6.13: Box-plot of the accuracy on CIFAR-10

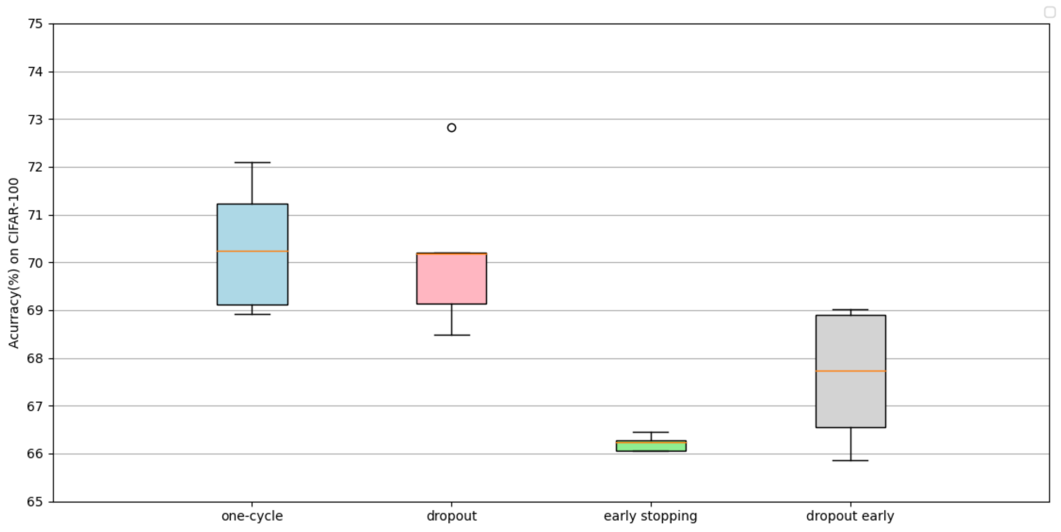


Figure 6.14: Box-plot of the accuracy on CIFAR-100

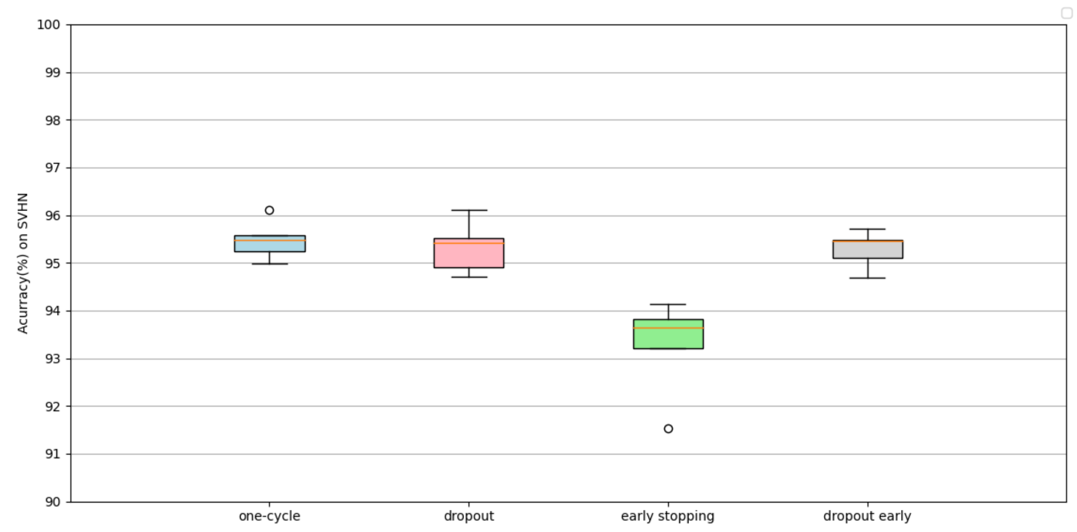


Figure 6.15: Box-plot of the accuracy on SVHN

Loss

As we can see in figure 6.16, the boxplot for all cases differ in terms shape. Notably, the early stopping case contains an outlier. In figure 6.17, the boxplots for all cases exhibit differences in both position and shape. As depicted in figure 6.18, all cases, except for the one-cycle case, share a similar boxplot shape. It is noteworthy that the one-cycle case includes two outliers.

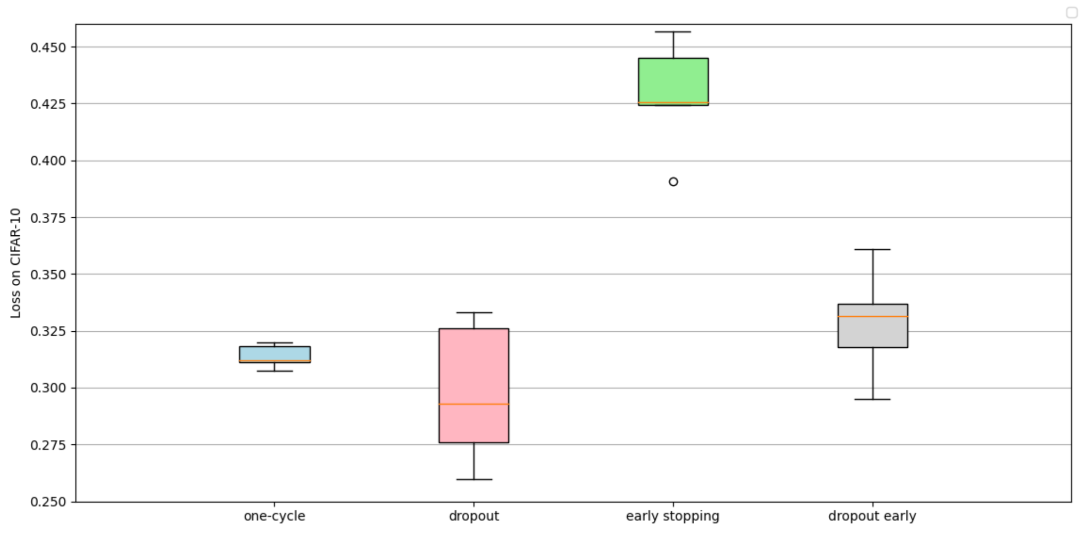


Figure 6.16: Box-plot of the loss on CIFAR-10

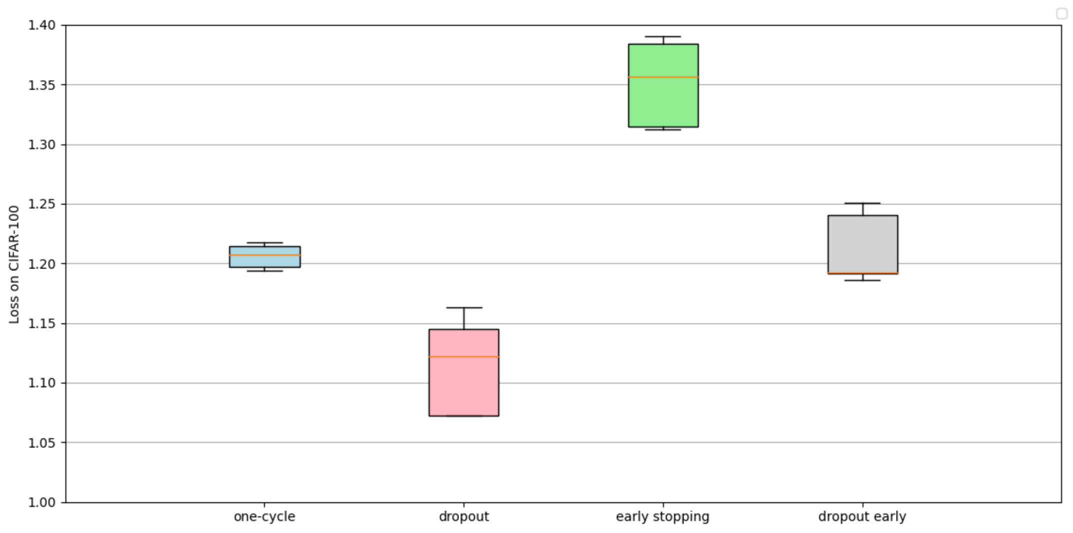


Figure 6.17: Box-plot of the loss on CIFAR-100

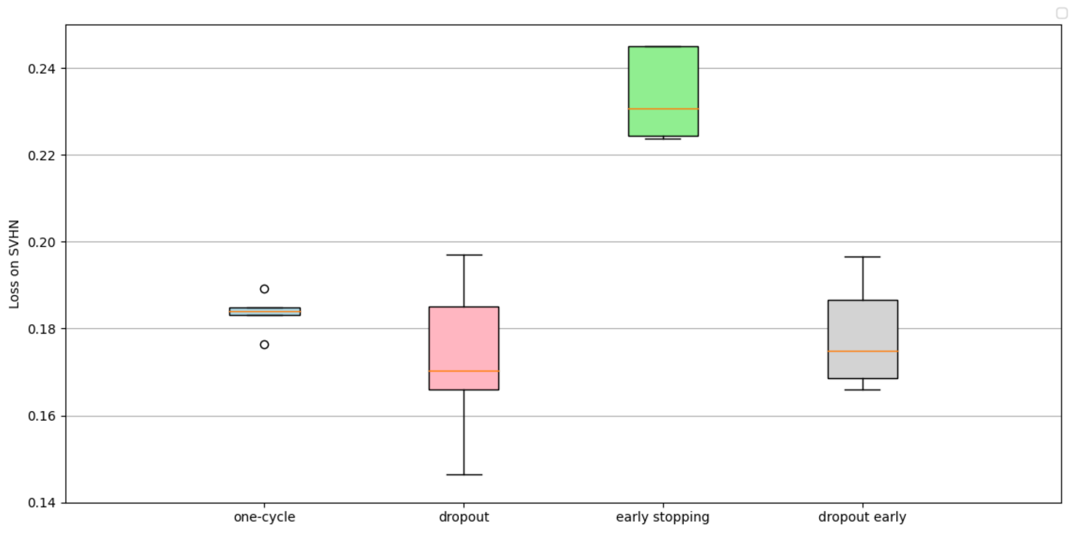


Figure 6.18: Box-plot of the loss on SVHN

Execution time

In figure 6.19, the boxplots for all cases exhibit variations in both position and shape. Notably, the one-cycle case includes an outlier. As we can see in figure 6.20, the boxplot for all cases differ in terms of position and shape. It is worth mentioning that the dropout case and the dropout early case contain an outlier. As shown in figure 6.21, the boxplots for all cases demonstrate variations in position. It is noteworthy that the dropout case has an outlier, while both the one-cycle case and the dropout early case have two outliers.

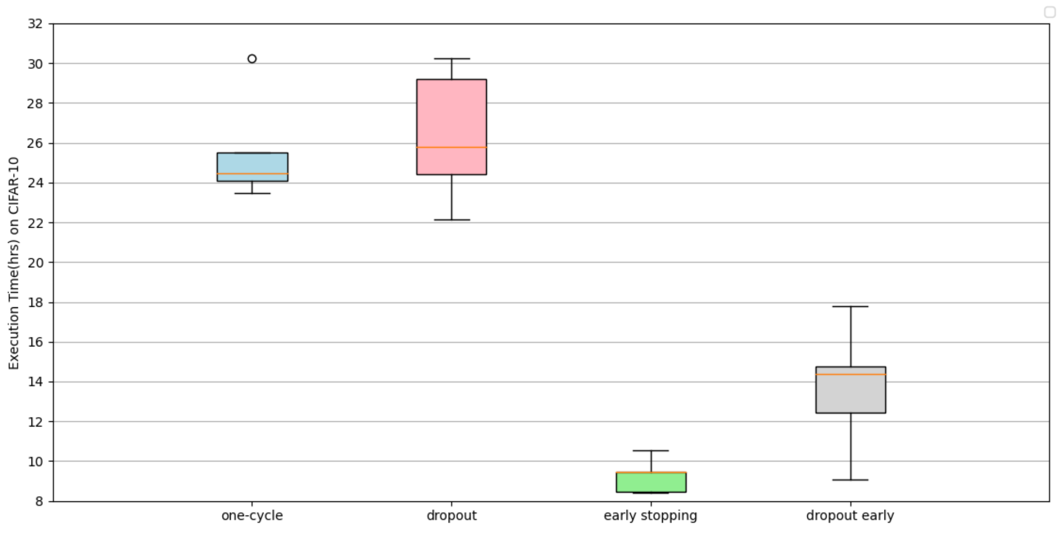


Figure 6.19: Box-plot of the execution Time on CIFAR-10

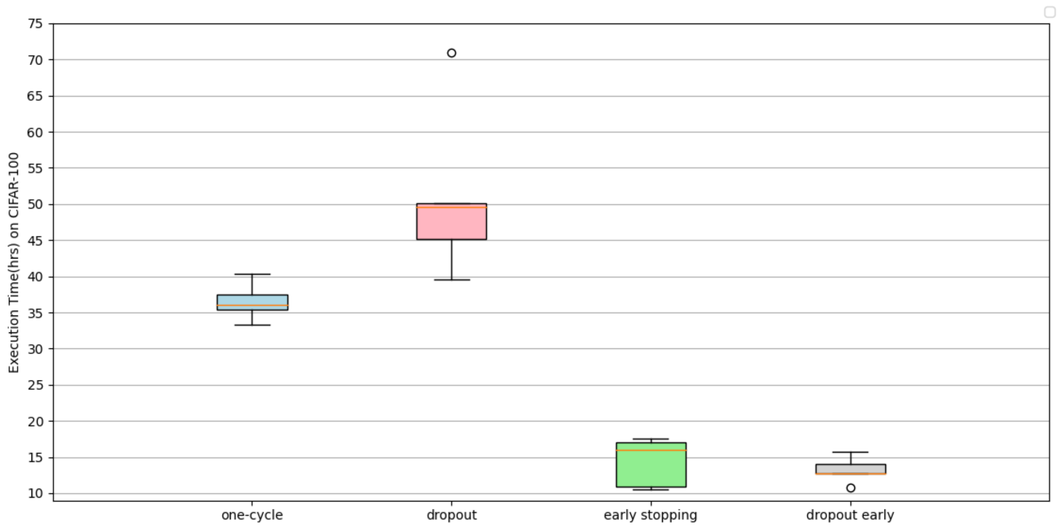


Figure 6.20: Box-plot of the execution Time on CIFAR-100

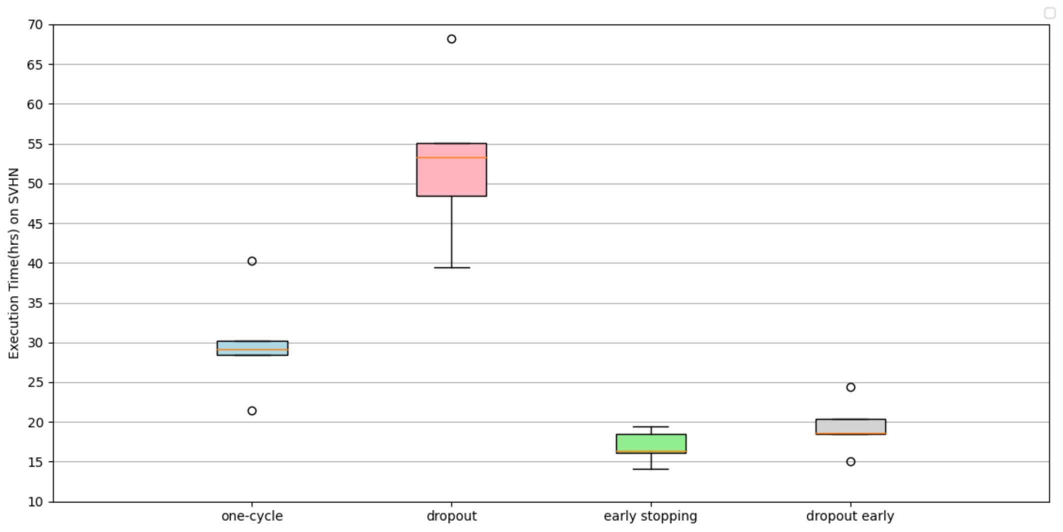


Figure 6.21: Box-plot of the execution Time on SVHN

Number of trainable parameters

In figure 6.22, the boxplots for all cases exhibit variations in both position and shape. Notably, the dropout early case contains an outlier. Figure 6.23 illustrates that the boxplot for the dropout case and the early stopping case share a similar median position. It is worth mentioning that every case, except the early stopping case, includes an outlier. As depicted in figure 6.24, the boxplots for the one-cycle case and the dropout early case display a similar shape and median position.

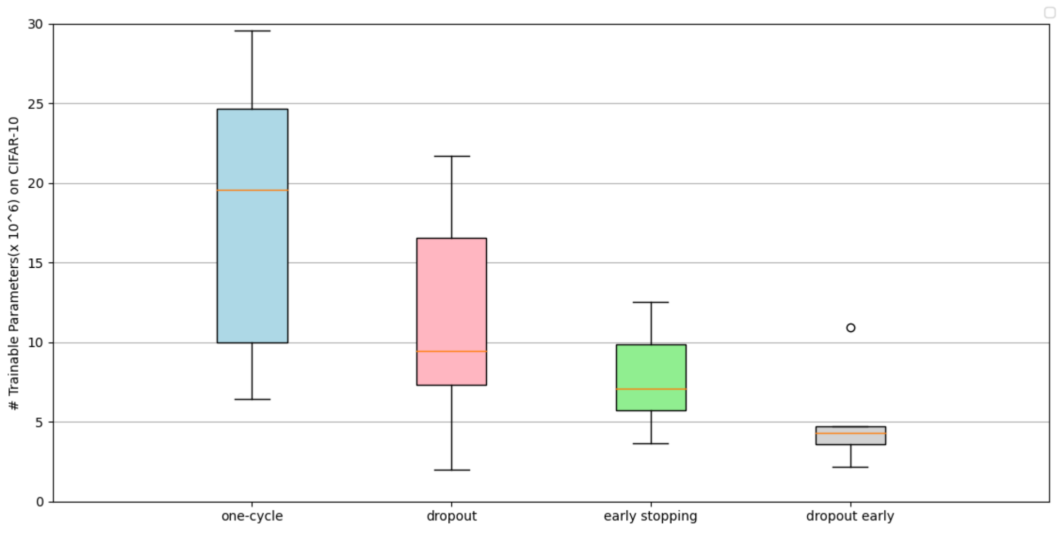


Figure 6.22: Box-plot of the number of trainable parameters on CIFAR-10

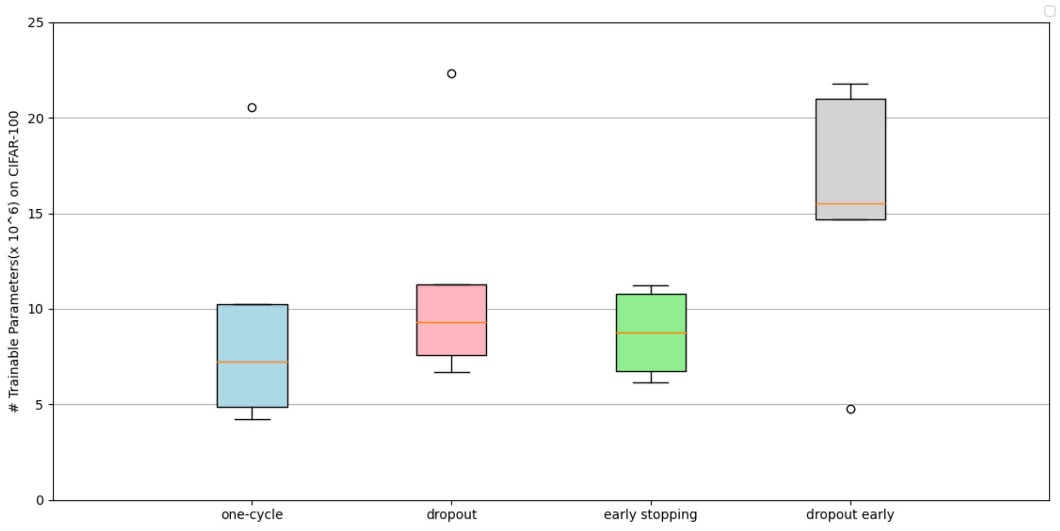


Figure 6.23: Box-plot of the number of trainable parameters on CIFAR-100

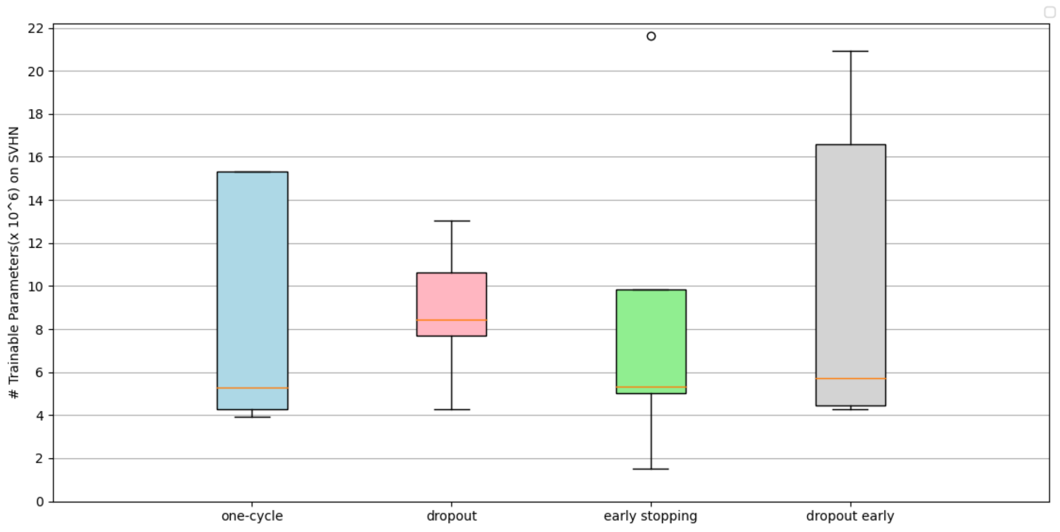


Figure 6.24: Box-plot of the number of trainable parameters on SVHN

6.2.2 Two-way ANOVA

Accuracy

If we take a look at the last column (which represents the p-value) in table 6.3, we can see that the early stopping has an effect on the accuracy for all datasets as its p-value is lower than 0.05. The same could be said for the dropout but only on SVHN. Moreover, the combination of both also has a p-value lower than 0.05 which means they are dependent on each other for the accuracy but only for SVHN.

Accuracy on CIFAR-10					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
dropout	1.000000	11.115405	11.115405	12.631824	0.002643
early stopping	1.000000	48.391605	48.391605	54.993429	0.000001
dropout early	1.000000	3.452805	3.452805	3.923854	0.065064
Residual	16.000000	14.079240	0.879952	NaN	NaN
Accuracy on CIFAR-100					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
dropout	1.000000	3.049805	3.049805	1.703492	0.210291
early stopping	1.000000	60.865605	60.865605	33.996959	0.000026
dropout early	1.000000	4.352445	4.352445	2.431092	0.138509
Residual	16.000000	28.645200	1.790325	NaN	NaN
Accuracy on SVHN					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
dropout	1.000000	4.380480	4.380480	10.361685	0.005360
early stopping	1.000000	6.384500	6.384500	15.102038	0.001312
dropout early	1.000000	5.896980	5.896980	13.948848	0.001805
Residual	16.000000	6.764120	0.422757	NaN	NaN

Table 6.3: Two-way ANOVA test on the accuracy for the dropout and the early stopping

Execution time

If we take a look at the last column (which represents the p-value) in table 6.4, we can see that, without surprise, the p-value of the early stopping on all datasets is lower than 0.05 meaning it has an impact on the execution time. However, the dropout also has an impact on the execution time for CIFAR-100 and SVHN. Moreover, the combination of both also has a p-value lower than 0.05 which means they are dependent on each other for the execution time but only for SVHN.

Execution time on CIFAR-10					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
dropout	1.000000	202.004485	202.004485	2.149596	0.161985
early stopping	1.000000	1657.177484	1657.177484	17.634566	0.000679
dropout early	1.000000	18.655972	18.655972	0.198524	0.661884
Residual	16.000000	1503.572000	93.973250	NaN	NaN
Execution time on CIFAR-100					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
dropout	1.000000	224.839747	224.839747	5.507096	0.032144
early stopping	1.000000	4503.656362	4503.656362	110.309975	0.000000
dropout early	1.000000	311.503133	311.503133	7.629779	0.013881
Residual	16.000000	653.236498	40.827281	NaN	NaN
Execution time on SVHN					
	DF	Mean Sq	Sum Sq	F	Pr(>F)
dropout	1.000000	811.277513	811.277513	19.001607	0.000487
early stopping	1.000000	2706.465099	2706.465099	63.390376	0.000001
dropout early	1.000000	529.537319	529.537319	12.402735	0.002830
Residual	16.000000	683.123284	42.695205	NaN	NaN

Table 6.4: Two-way ANOVA test on the execution time for the dropout and the early stopping

6.3 Third half

6.3.1 Box-plot

Accuracy

In figure 6.25, the boxplots for all cases exhibit variations in both position and shape. Notably, the dropout early case contains two outliers. Figure 6.26 illustrates that the boxplots for all cases display differences in both position and shape. It is worth mentioning that the one-cycle reset case includes an outlier. As depicted in figure 6.27, the boxplot for the one-cycle case and the dropout early case share a similar median position but differ in terms of shape. It is worth noting that both the one-cycle case and the one-cycle reset case contain an outlier.

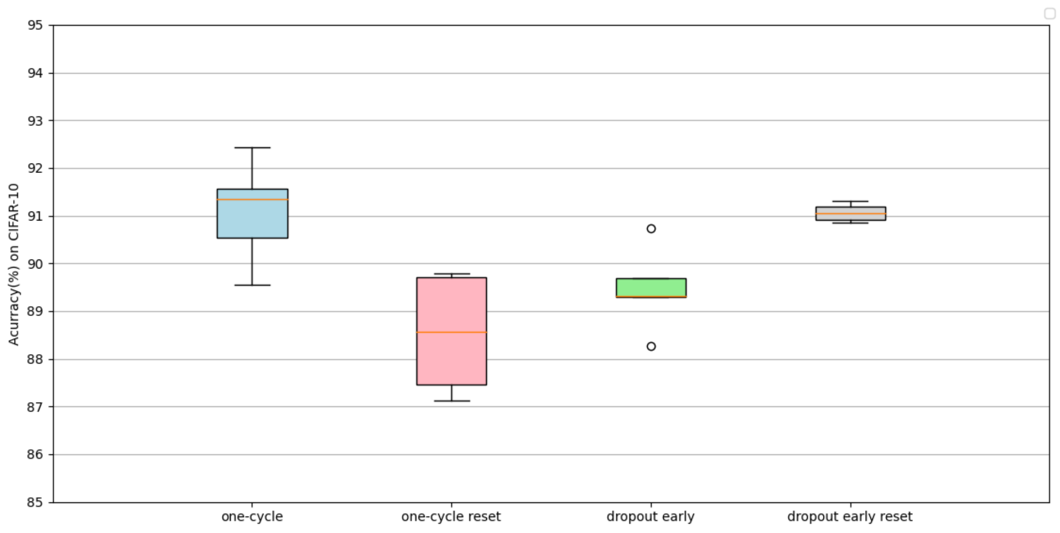


Figure 6.25: Box-plot of the accuracy on CIFAR-10

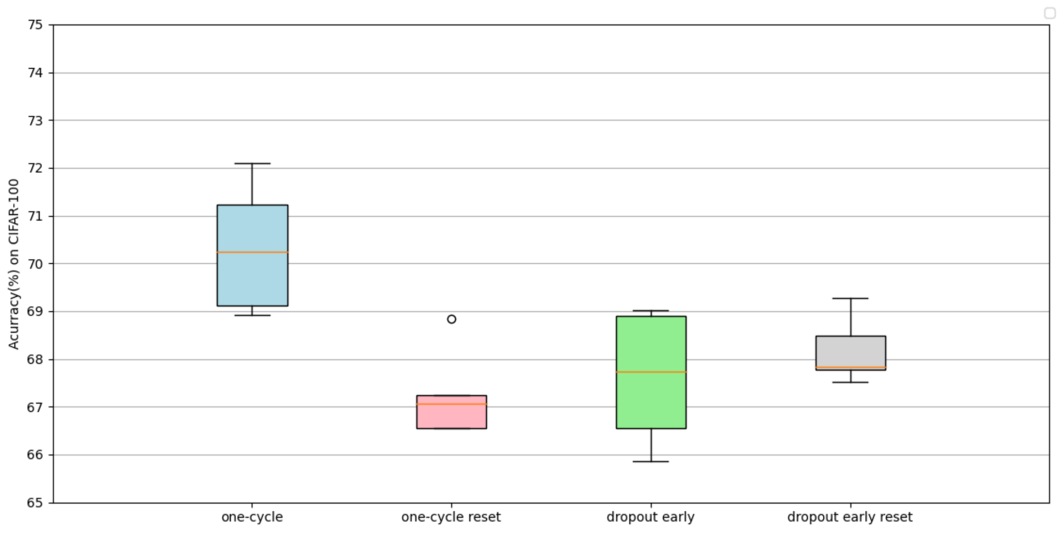


Figure 6.26: Box-plot of the accuracy on CIFAR-100

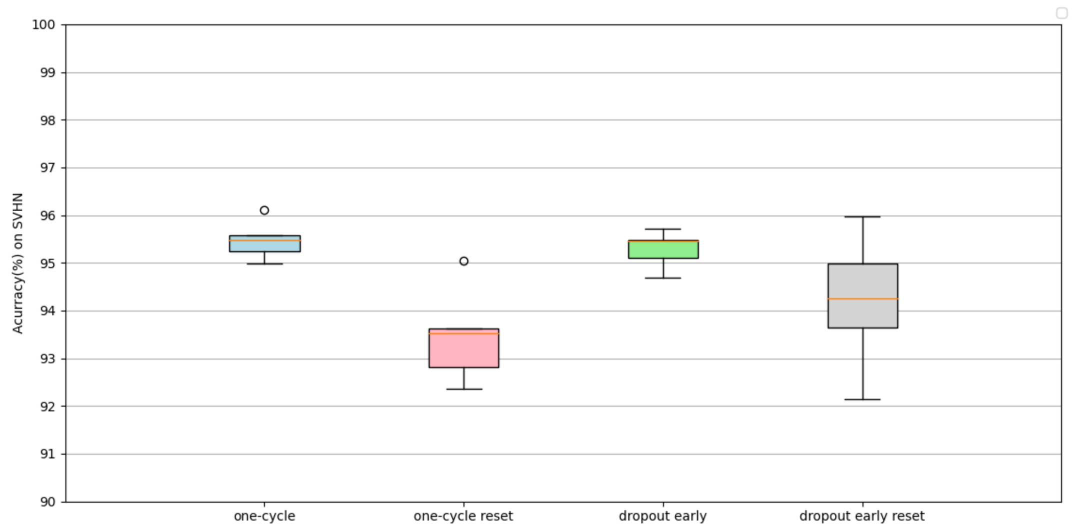


Figure 6.27: Box-plot of the accuracy on SVHN

Loss

As we can see in figure 6.28, the boxplot for all cases differ in terms shape. Figure 6.29 illustrates that the boxplots for all cases display differences in both position and shape. As depicted in figure 6.30, all cases, except for the one-cycle reset case, share a similar boxplot position. It is worth mentioning that both the one-cycle case and the one-cycle reset case include two outliers.

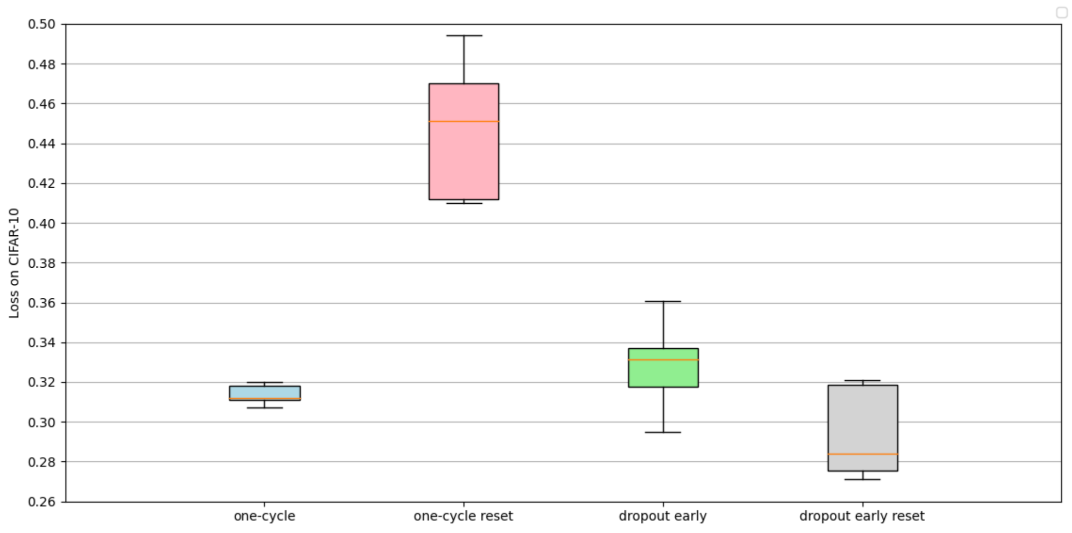


Figure 6.28: Box-plot of the loss on CIFAR-10

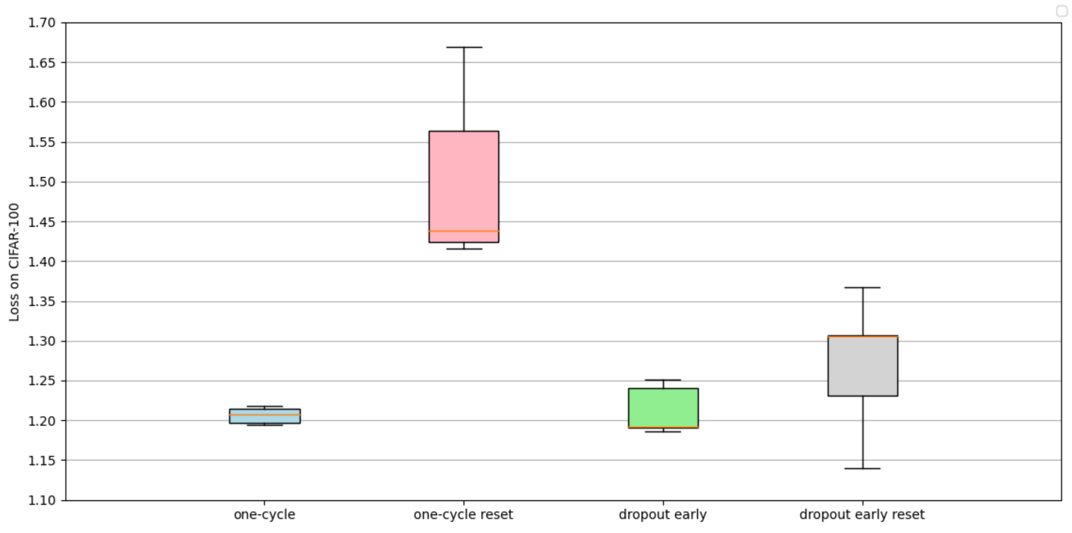


Figure 6.29: Box-plot of the loss on CIFAR-100

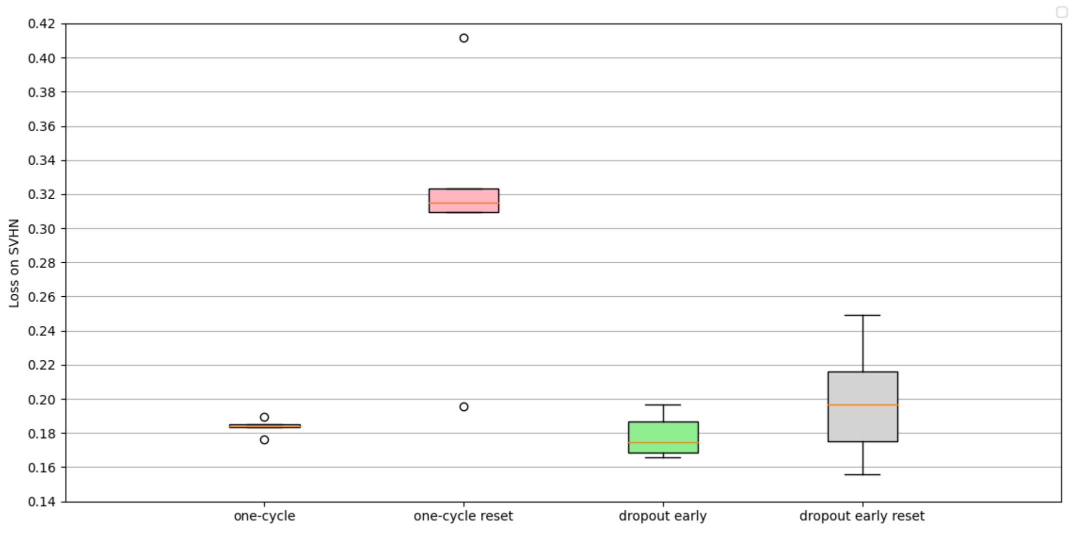


Figure 6.30: Box-plot of the loss on SVHN

Execution time

In figure 6.31, the boxplots for all cases exhibit variations in both position and shape. Notably, the one-cycle case and the one-cycle reset case include an outlier. As we can see in figure 6.32, the boxplot for all cases differ in terms of position and shape. It is worth noting that the dropout early case has an outlier. As depicted in figure 6.33, the boxplots for all cases demonstrate variations in position. It is noteworthy that the one-cycle reset case has an outlier, while both the one-cycle case and the dropout early case include two outliers.

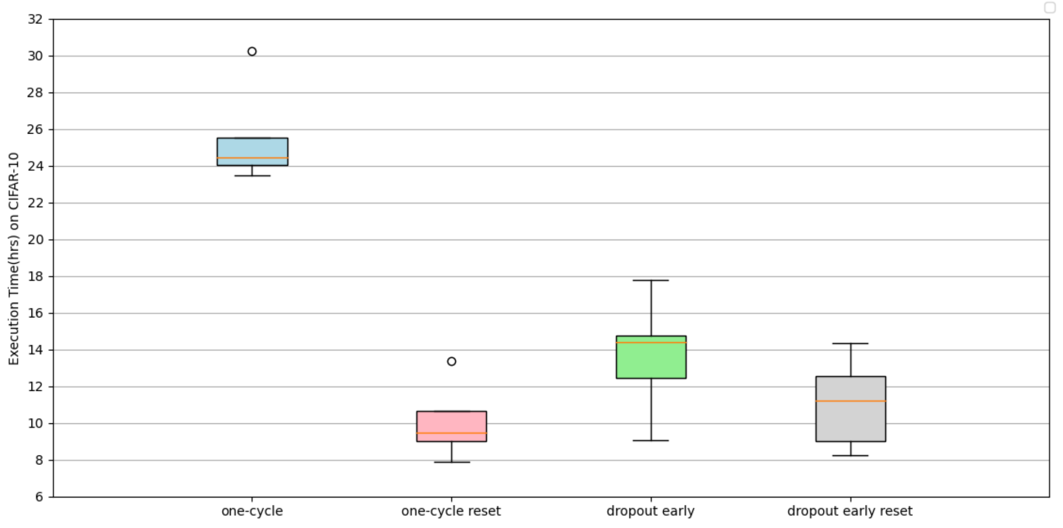


Figure 6.31: Box-plot of the execution Time on CIFAR-10

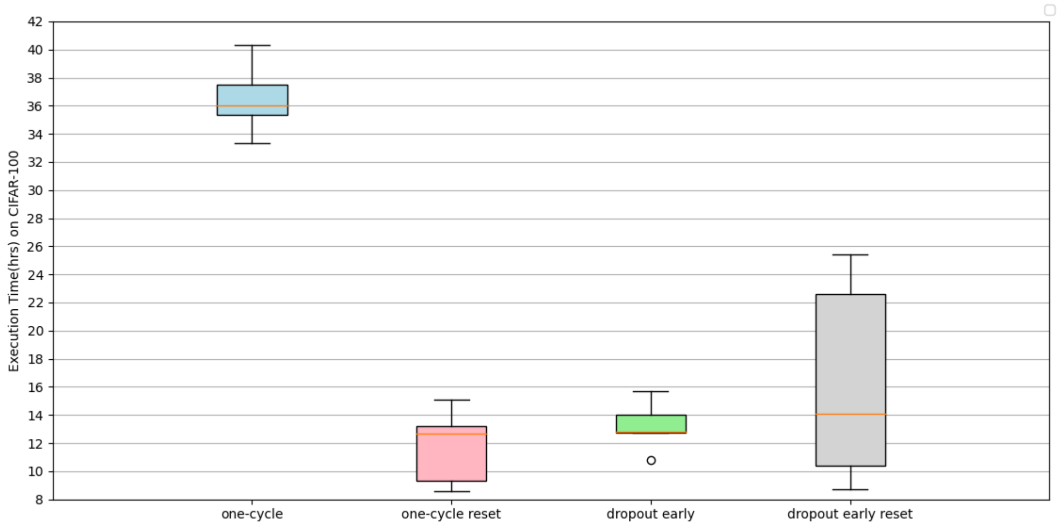


Figure 6.32: Box-plot of the execution Time on CIFAR-100

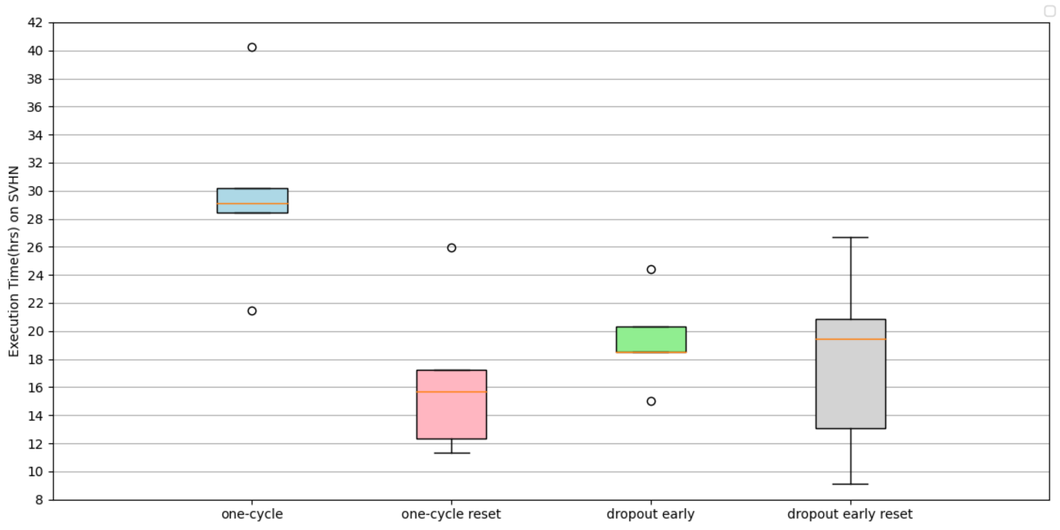


Figure 6.33: Box-plot of the execution Time on SVHN

Number of trainable parameters

As the number of trainable is not affected by the weight reset, the boxplot is the identical for the one-cycle case and the one-cycle reset case, as well as for the dropout early case and the dropout early reset case, as seen in figure 6.34, in figure 6.35 and in figure 6.36.

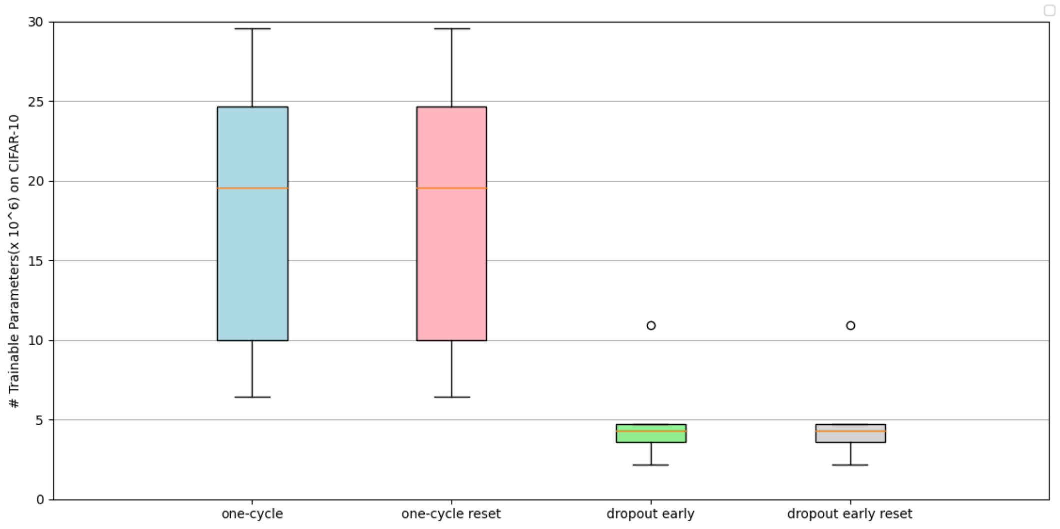


Figure 6.34: Box-plot of the number of trainable parameters on CIFAR-10

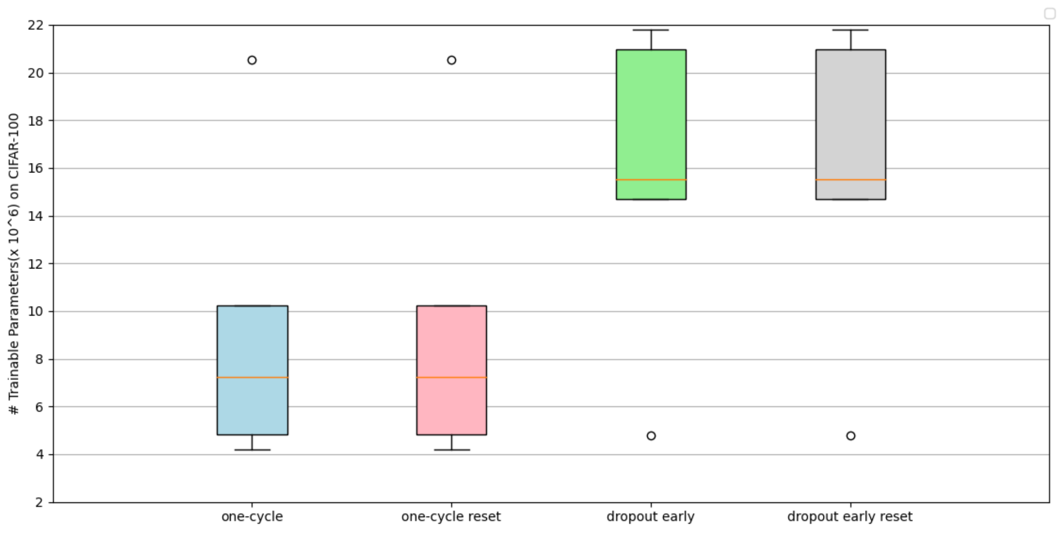


Figure 6.35: Box-plot of the number of trainable parameters on CIFAR-100

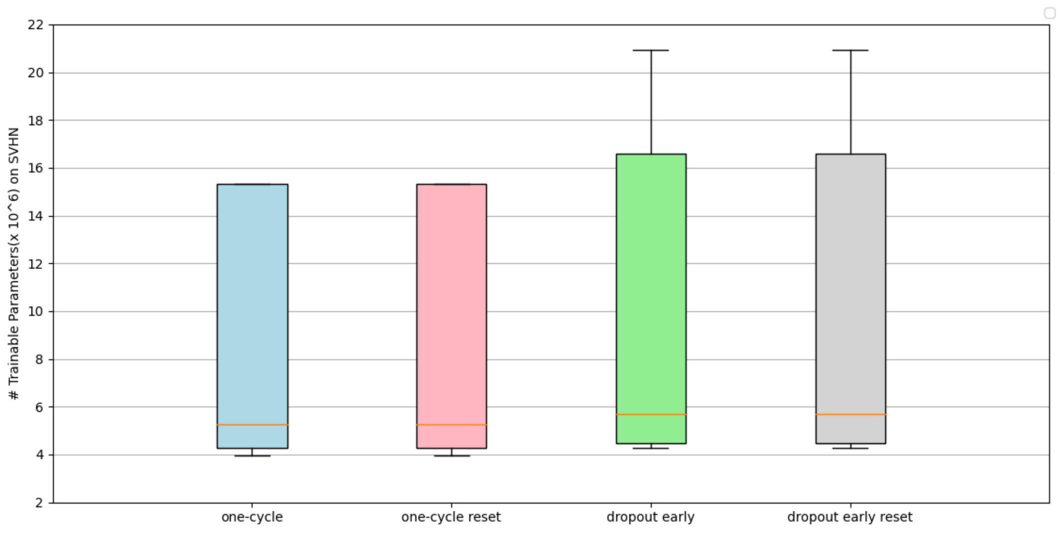


Figure 6.36: Box-plot of the number of trainable parameters on SVHN

6.3.2 Two-way ANOVA

We have already done this test for one-cycle in section 6.1.2 and for dropout early in section 6.2.2.

6.4 Comparison

We can now compare our methods with the current state of the art and other works which uses gradient-based NAS. We will assess the differences in terms of accuracy, GPU days and the number of trainable parameters. GPU days represent the total execution time divided by the number of GPUs used during the experiment, providing an evaluation of the time taken for the experiment to complete.

6.4.1 CIFAR-10

We can look at the results for CIFAR-10 in table 6.5. Our algorithm demonstrates performance that is comparable to our base paper with the same amount of time, but falls short on achieving the state of the art results. But is worth noting that our best performing models are significantly less complex than the state of the art ones. While in figure 6.37, we can see the scatter plot of these results for the accuracy versus the number of parameters. We can notice that the Pareto frontier consists of a single red line. This means that there is always a trade-off between the number of parameters and accuracy, and there is no dominance or clear superiority of any specific point over others.

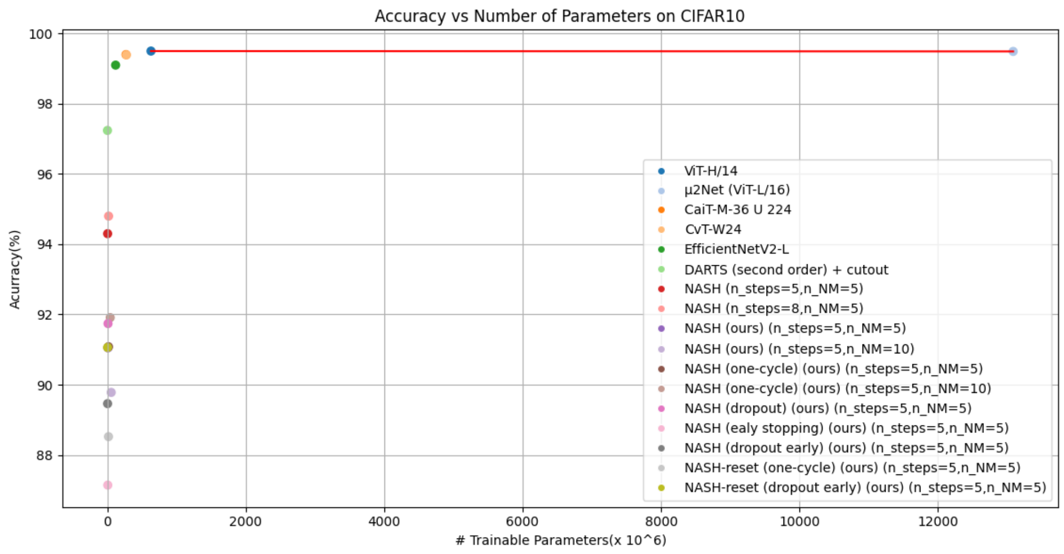


Figure 6.37: Scatter plot of the accuracy versus the number of parameters on CIFAR-10

Model	Accuracy (%)	GPU Days	# Parameters (x10^6)
ViT-H/14 [Dos+21]	99.50	2500 (on TPU)	632
µ2Net (ViT-L/16) [GD22]	99.49	_	13087
CaiT-M-36 U 224 [Tou+21]	99.40	_	271
CvT-W24 [Wu+21]	99.39	_	277
EfficientNetV2-L [TL21]	99.10	1	121
DARTS (second order) [LSY19] + cutout	97.24	4	3.3
NASH [EMH17] ($n_{steps} = 5, n_{NM} = 5$)	94.30	0.5	5.7
NASH [EMH17] ($n_{steps} = 8, n_{NM} = 5$)	94.80	1	19.7
NASH [ours] ($n_{steps} = 5, n_{NM} = 5$)	91.05	1.1137	10.26
NASH [ours] ($n_{steps} = 5, n_{NM} = 10$)	89.78	3.7433	57.12
NASH (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 5$)	91.08	1.0641	18.04
NASH (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 10$)	91.91	3.1166	42.83
NASH (dropout) [ours] ($n_{steps} = 5, n_{NM} = 5$)	91.74	1.3558	11.41
NASH (early stopping) [ours] ($n_{steps} = 5, n_{NM} = 5$)	87.14	0.3862	7.78
NASH (dropout early) [ours] ($n_{steps} = 5, n_{NM} = 5$)	89.46	0.5704	5.15
NASH-reset (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 5$)	88.52	$1,4837$	18.04
NASH-reset (dropout early) [ours] ($n_{steps} = 5, n_{NM} = 5$)	91.06	$1,0312$	5.15

Table 6.5: Comparison with the state of the art on CIFAR-10

6.4.2 CIFAR-100

We can look at the results for CIFAR-100 in table 6.6. We can observe that our accuracy is quite low on this very complex dataset compared to the state of the art but it is worth mentioning that the accuracy is in proximity to the results reported in our base paper. Moreover, our algorithm takes more execution time. But again, it is worth noting that our best performing models are significantly less complex than the state of the art ones. While in figure 6.38, the scatter plot illustrates the relationship between the accuracy and the number of parameters. The Pareto frontier, represented by the red line, indicates that a trade-off always exists between these two variables. This suggests that improving accuracy often requires an increase in the number of parameters and vice versa, just like for the CIFAR-10 dataset.

Model	Accuracy (%)	GPU Days	# Parameters (x10^6)
EffNet-L2 (SAM) [For+21]	96.08	_	64
Swin-L + ML-Decoder [Rid+21]	95.10	_	_
µ2Net (ViT-L/16) [GD22]	94.95	_	13087
CvT-W24 [Wu+21]	94.09	_	277
EfficientNetV2-L [TL21]	92.30	1	121
NASH [EMH17] ($n_{steps} = 8, n_{NM} = 5$)	76.60	1	22.3
NASH [ours] ($n_{steps} = 5, n_{NM} = 5$)	68.89	1.9650	14.64
NASH [ours] ($n_{steps} = 5, n_{NM} = 10$)	69.06	5.0087	49.37
NASH (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 5$)	70.32	1.5212	9.41
NASH (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 10$)	68.77	4.9495	36.25
NASH (dropout) [ours] ($n_{steps} = 5, n_{NM} = 5$)	70.17	2.1295	11.42
NASH (early stopping) [ours] ($n_{steps} = 5, n_{NM} = 5$)	65.91	0.5995	8.73
NASH (dropout early) [ours] ($n_{steps} = 5, n_{NM} = 5$)	67.61	0.5504	15.55
NASH-reset (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 5$)	66.91	$2,0116$	9.41
NASH-reset (dropout early) [ours] ($n_{steps} = 5, n_{NM} = 5$)	68.17	$1,2266$	15.55

Table 6.6: Comparison with the state of the art on CIFAR-100

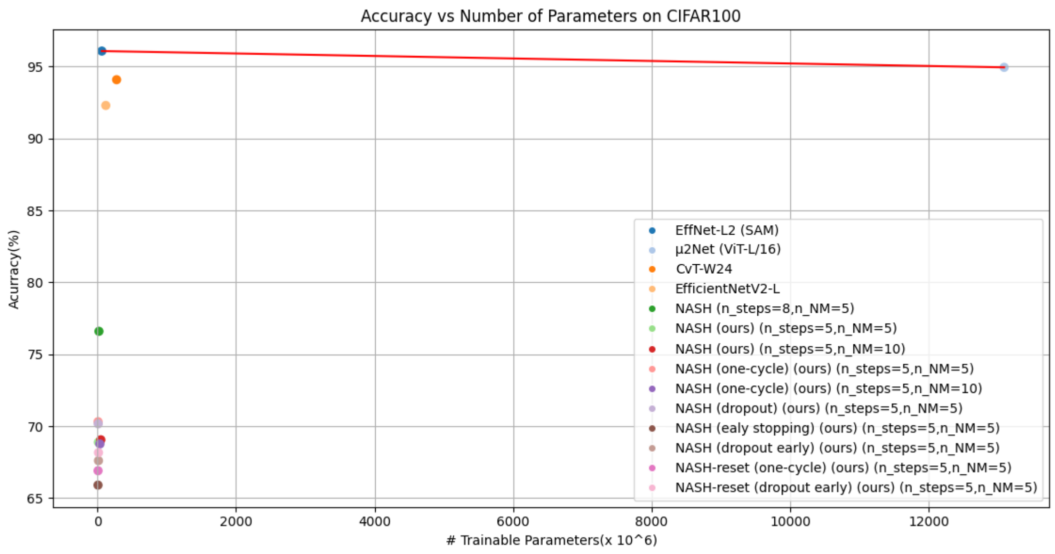


Figure 6.38: Scatter plot of the accuracy versus the number of parameters on CIFAR-100

6.4.3 SVHN

We can look at the results for SVHN in table 6.7. However, there are limited information available on the execution time (GPU days). We can see that with this simple dataset, the accuracies are not really far. Again, it is worth noting that our best performing models are less complex than the state of the art ones. In figure 6.39, the scatter plot displays the relationship between accuracy and the number of parameters. The Pareto frontier, represented by a single red line, indicates that there is always a trade-off between these two factors. Just like the previous datatsets, there is no dominance or clear superiority of any particular point over others, emphasizing the absence of a solution that excels in both accuracy and a smaller number of parameters simultaneously.

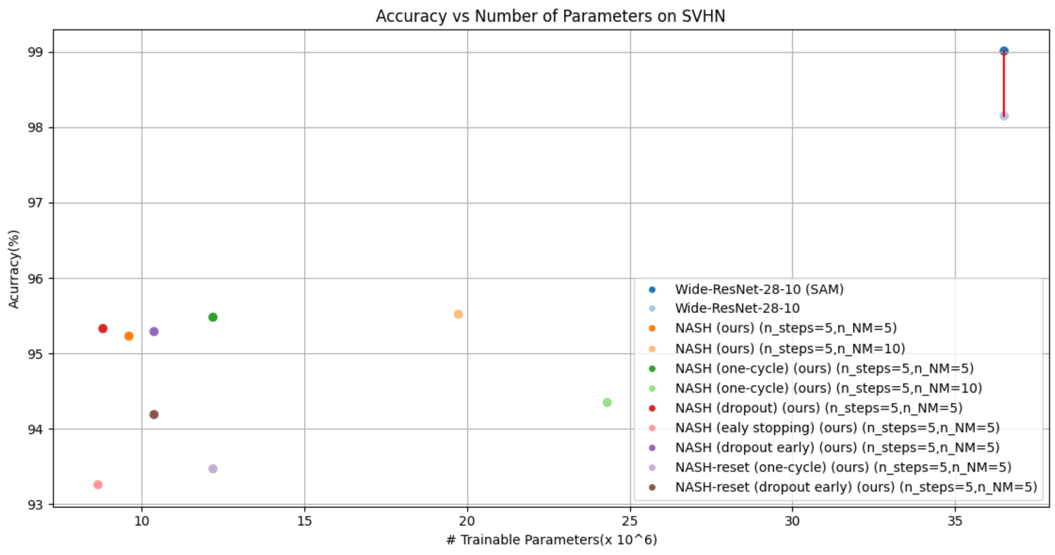


Figure 6.39: Scatter plot of the accuracy versus the number of parameters on SVHN

Model	Accuracy (%)	GPU Days	# Parameters (x10^6)
Wide-ResNet-28-10 (SAM) [For+21]	99.01	_	36.5
Wide-ResNet-28-10 [HCR22]	98.15	_	36.5
NASH [ours] ($n_{steps} = 5, n_{NM} = 5$)	95.23	1.4150	9.62
NASH [ours] ($n_{steps} = 5, n_{NM} = 10$)	95.52	4.1579	19.74
NASH (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 5$)	95.48	1.2450	12.20
NASH (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 10$)	94.35	2.9529	24.31
NASH (dropout) [ours] ($n_{steps} = 5, n_{NM} = 5$)	95.33	2.2045	8.82
NASH (early stopping) [ours] ($n_{steps} = 5, n_{NM} = 5$)	93.26	0.7045	8.67
NASH (dropout early) [ours] ($n_{steps} = 5, n_{NM} = 5$)	95.29	0.8066	10.39
NASH-reset (one-cycle) [ours] ($n_{steps} = 5, n_{NM} = 5$)	93.47	$1,9321$	12.20
NASH-reset (dropout early) [ours] ($n_{steps} = 5, n_{NM} = 5$)	94.19	$1,5492$	10.39

Table 6.7: Comparison with the state of the art on SVHN

7 Conclusion

In this work, we were interested by the Neural Architecture Search by Hillclimbing (NASH) algorithm, a method to automatically design architecture for CNNs. This method is the combination of the hill-climbing strategy, network morphisms, and training via SGDR. First, our aim was to find the best combination of hyperparameters without touching those related to the number of epochs, as it was expected to have a comparatively lesser effect on the performance of the model. Our focus was on improving the performance of the models, without considering their complexity or execution time as it was not the main focus of this research. However, measuring them could allow us to observe their behaviour with regard to the changing hyperparameters. Subsequently, we wanted to study the effect of incorporating dropout and early stopping techniques. This time, our focus was on increasing the performances while also reducing the computational costs. Following that, we selected the best models from our previous two experiments and reset their weights. The objective of this third experiment was to determine the additional value of network morphisms in our models. All these experiments were conducted on three different datasets, namely CIFAR-10, CIFAR-100, and SVHN.

Our first results showed that the NASH algorithm is a robust algorithm, with consistent and reliable performance. However, we noticed that switching the learning rate scheduler from cosine annealing to one-cycle resulted in a slight improvement in the overall performance. From this finding, our subsequent results indicated that incorporating both dropout and early stopping techniques led to a minor decrease in the overall performance of the models. Nevertheless, this reduction in performance was accompanied with a considerable decrease in the execution time. In our third set of results, we observed that weight reset had a detrimental effect on the one-cycle, leading to a decline in performances. However, for dropout early, it resulted in improved performances, nearly identical to the one-cycle approach, while still maintaining lower execution times compared to the one-cycle.

Then, we performed an analysis of our results. We first used boxplots to visually summarize the distribution of our results for our different methods. This enabled us to identify the median and detect any outliers that may have impacted the mean of our results. Following that, we performed a two-way ANOVA to investigate the possible dependency between the hyperparameters we modified. The results indicated that the n_{NM} parameter and the one-cycle are dependent for the execution time only on the SVHN dataset, while the dropout and the early stopping are dependent on each other for both the accuracy and the execution on the same dataset. Furthermore, we compared our models to the state of the art and other gradient-based NAS methods. We showed that our models fall short in terms of performance compared to the state of the art on CIFAR-10 and CIFAR-100, but are less complex than them. Nevertheless, for both datasets, we were able to achieve comparable performances to our base paper using the same amount of time. On the other hand, for the SVHN dataset, we managed to reach performances that are comparable to the state of the art. In addition, we employed scatter plots to visualize the relationship between accuracy and the number of parameters. This allowed us to identify the Pareto frontier, representing optimal solutions that strike a balance between accuracy and the number of parameters. However, our plots showed that there is always a trade-off between these two factors. Across all three datasets, no particular method exhibited dominance or clear superiority over others, highlighting the absence of a solution that excelled in both accuracy and a smaller number of parameters simultaneously.

Our work highlighted the challenges involved in striking a balance between execution time and performance. While techniques like early stopping and dropout can lead to faster execution times, they can also have a slight impact on the overall accuracy of the models. Our research provides valuable insights into this trade-off, which can be useful in developing more efficient and effective ML models in the future. Overall, our study contributes to the field of automatic architecture search and provides valuable insights into the use of the NASH algorithm for designing CNNs.

7.1 Future work

Following this study, there are various potential ways for improving the NASH algorithm, as well as exploring alternative research directions. Here are some potential ideas to improve the NASH algorithm:

- **Exploration strategies**: Incorporate different exploration strategies into the hillclimbing process to encourage the search for diverse architectures. For instance, one could incorporate random search, mutation-based search, or RL to explore the architecture space.
- **Network morphism search**: Use a more extensive set of network morphisms to generate a broader range of architectures.
- **Regularization techniques**: Investigate the use of more regularization techniques such as weight decay [HP88] or injecting noises [Wan+13] to reduce overfitting and improve the generalization of the models.
- **Architectural hyperparameters**: Optimize the values of architectural hyperparameters such as the number of layers, number of filters, and kernel sizes to further improve model performance.

- **Exploration with more diverse datasets (especially larger datasets)**: Explore the use of the proposed methods with larger datasets such as ImageNet [Den+09], MNIST [Den12], or Oxford-IIIT-Pet [Par+12] to evaluate its performance on more complex problems.
- **Use different optimization techniques**: Instead of using SGDR with cosine annealing schedule or one-cycle, other optimization techniques like Adam [KB17] or Adagrad [DHS11] could be used with different learning rate schedules.
- **Consider network compression techniques**: After the search, the architecture could be further optimized by using techniques such as pruning [Bla+20] [Xu+20], quantization [Gho+21] [Kri18], or knowledge distillation [Gou+21] [CH19] to make the model smaller and faster while maintaining its accuracy.

Bibliography

[AGR22] Achmamad, Abdelouahad, Fethi Ghazouani, and Su Ruan (2022). “Few-shot learning for brain tumor segmentation from MRI images”. 1, pp. 489–494.

[AM17] Albelwi, Saleh and Ausif Mahmood (2017). “A Framework for Designing the Architectures of Deep Convolutional Neural Networks”. *Entropy* 19.6. ISSN: 1099-4300. DOI: 10.3390/e19060242. URL: https://www.mdpi.com/1099-4300/19/6/242.

[AAO17] Awoyemi, John O, Adebayo O Adetunmbi, and Samuel A Oluwadare (2017). “Credit card fraud detection using machine learning techniques: A comparative analysis”, pp. 1–9.

[Bah22] Baheti, Pragati (2022). “The Essential Guide to Neural Network Architectures”. URL: https://www.v7labs.com/blog/neural-network-architectures-guide#what-are-neural-networks.

[Bah23] — (2023). “Activation Functions in Neural Networks [12 Types & Use Cases]”. URL: https://www.v7labs.com/blog/neural-networks-activation-functions.

[Bak+16] Baker, Bowen et al. (2016). “Designing Neural Network Architectures using Reinforcement Learning”. DOI: 10.48550/ARXIV.1611.02167. URL: https://arxiv.org/abs/1611.02167.

[Bak79] Baker, James K (1979). “Trainable grammars for speech recognition”. *The Journal of the Acoustical Society of America* 65.S1, S132–S132.

[BB12] Bergstra, James and Yoshua Bengio (2012). “Random search for hyper-parameter optimization.” *Journal of machine learning research* 13.2.

[Ber+11] Bergstra, James et al. (2011). “Algorithms for Hyper-Parameter Optimization”. 24. Ed. by J. Shawe-Taylor et al. URL: https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[Bha+21] Bhavsar, Kaustubh Arun et al. (2021). “Medical diagnosis using machine learning: a statistical review”. *Computers, Materials and Continua* 67.1, pp. 107–125.

[Bis07] Bishop, Christopher M. (2007). “Pattern Recognition and Machine Learning (Information Science and Statistics)”.

[Bla+20] Blalock, Davis et al. (2020). “What is the state of neural network pruning?” *Proceedings of machine learning and systems* 2, pp. 129–146.

[Bro18] Brownlee, Jason (2018). “What is the Difference Between a Batch and an Epoch in a Neural Network”. *Machine Learning Mastery* 20.

[Cai+17] Cai, Han et al. (2017). “Efficient Architecture Search by Network Transformation”. DOI: `10.48550/ARXIV.1707.04873`. URL: `https://arxiv.org/abs/1707.04873`.

[Cai+18] Cai, Han et al. (2018). “Path-Level Network Transformation for Efficient Architecture Search”. *arXiv preprint arXiv:1806.02639*.

[Cai+20] Cai, Han et al. (2020). “Once-for-All: Train One Network and Specialize it for Efficient Deployment”. arXiv: `1908.09791 [cs.LG]`.

[Che+17] Chen, Chenyi et al. (2017). “R-CNN for small object detection”, pp. 214–230.

[CGS15] Chen, Tianqi, Ian Goodfellow, and Jonathon Shlens (2015). “Net2Net: Accelerating Learning via Knowledge Transfer”. DOI: `10.48550/ARXIV.1511.05641`. URL: `https://arxiv.org/abs/1511.05641`.

[CH19] Cho, Jang Hyun and Bharath Hariharan (2019). “On the efficacy of knowledge distillation”, pp. 4794–4802.

[Cra+15] Crawford, Michael et al. (2015). “Survey of review spam detection using machine learning techniques”. *Journal of Big Data* 2.1, pp. 1–24.

[DBB52] Davis, Ken H, R Biddulph, and Stephen Balashek (1952). “Automatic recognition of spoken digits”. *The Journal of the Acoustical Society of America* 24.6, pp. 637–642.

[Den+09] Deng, Jia et al. (2009). “Imagenet: A large-scale hierarchical image database”, pp. 248–255.

[Den12] Deng, Li (2012). “The mnist database of handwritten digit images for machine learning research”. *IEEE Signal Processing Magazine* 29.6, pp. 141–142.

[DT17] DeVries, Terrance and Graham W. Taylor (2017). “Improved Regularization of Convolutional Neural Networks with Cutout”. arXiv: `1708.04552 [cs.CV]`.

[DKK+12] Dongare, AD, RR Kharde, Amit D Kachare, et al. (2012). “Introduction to artificial neural network”. *International Journal of Engineering and Innovative Technology (IJEIT)* 2.1, pp. 189–194.

[Dos+21] Dosovitskiy, Alexey et al. (2021). “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. *ICLR*.

[DHS11] Duchi, John, Elad Hazan, and Yoram Singer (2011). “Adaptive subgradient methods for online learning and stochastic optimization.” *Journal of machine learning research* 12.7.

[Edu20] Education, IBM Cloud (2020). “Convolutional Neural Networks”. URL: `https://www.ibm.com/cloud/learn/convolutional-neural-networks`.

[EMH19] Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter (2019). “Neural Architecture Search: A Survey”. arXiv: 1808.05377 [stat.ML].

[EMH17] Elsken, Thomas, Jan-Hendrik Metzen, and Frank Hutter (2017). “Simple And Efficient Architecture Search for Convolutional Neural Networks”. DOI: 10.48550/ARXIV.1711.04528. URL: https://arxiv.org/abs/1711.04528.

[Feu+15] Feurer, Matthias et al. (2015). “Efficient and Robust Automated Machine Learning”. 28. Ed. by C. Cortes et al. URL: https://proceedings.neurips.cc/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf.

[For+21] Foret, Pierre et al. (2021). “Sharpness-Aware Minimization for Efficiently Improving Generalization”. arXiv: 2010.01412 [cs.LG].

[GD22] Gesmundo, Andrea and Jeff Dean (2022). “An Evolutionary Approach to Dynamic Introduction of Tasks in Large-scale Multitask Learning Systems”. arXiv: 2205.12755 [cs.LG].

[Gho+21] Gholami, Amir et al. (2021). “A survey of quantization methods for efficient neural network inference”. *arXiv preprint arXiv:2103.13630*.

[GJP98] Girosi, Federico, Michael Jones, and Tomaso Poggio (Oct. 1998). “Regularization Theory and Neural Networks Architectures”. *Neural Comput* 7. DOI: 10.1162/neco.1995.7.2.219.

[Gir+13] Girshick, Ross B. et al. (2013). “Rich feature hierarchies for accurate object detection and semantic segmentation”. *CoRR* abs/1311.2524. arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[GB10] Glorot, Xavier and Yoshua Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”.

[GBC16] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). “Deep Learning”. http://www.deeplearningbook.org.

[Gou+21] Gou, Jianping et al. (2021). “Knowledge distillation: A survey”. *International Journal of Computer Vision* 129, pp. 1789–1819.

[Gup+13] Gupta, Neha et al. (2013). “Artificial neural network”. *Network and Complex Systems* 3.1, pp. 24–28.

[HP88] Hanson, Stephen and Lorien Pratt (1988). “Comparing biases for minimal network construction with back-propagation”. *Advances in neural information processing systems* 1.

[He+15] He, Kaiming et al. (2015). “Deep Residual Learning for Image Recognition”. *CoRR* abs/1512.03385. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[Hin+12] Hinton, Geoffrey E et al. (2012). “Improving neural networks by preventing co-adaptation of feature detectors”. *arXiv preprint arXiv:1207.0580*.

[HCR22] Hounie, Ignacio, Luiz F. O. Chamon, and Alejandro Ribeiro (2022). “Automatic Data Augmentation via Invariance-Constrained Learning”. arXiv: 2209.15031 [cs.LG].

[How+19] Howard, Andrew et al. (Oct. 2019). “Searching for MobileNetV3”.

[HG20] Howard, Jeremy and Sylvain Gugger (2020). “Fastai: A Layered API for Deep Learning”. *Information* 11.2. ISSN: 2078-2489. DOI: 10.3390/info11020108. URL: https://www.mdpi.com/2078-2489/11/2/108.

[HCL+03] Hsu, Chih-Wei, Chih-Chung Chang, Chih-Jen Lin, et al. (2003). “A practical guide to support vector classification”.

[JK15] Jabbar, H and Rafiqul Zaman Khan (2015). “Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study)”. *Computer Science, Communication and Instrumentation Devices* 70, pp. 163–172.

[Jia17] Jia, Xin (2017). “Image recognition method based on deep learning”, pp. 4730–4735. DOI: 10.1109/CCDC.2017.7979332.

[Kad19] Kadhim, Ammar Ismael (2019). “Survey on supervised machine learning techniques for automatic text classification”. *Artificial Intelligence Review* 52.1, pp. 273–292.

[KDP20] Kang, Dae-Young, Pham Duong, and Jung-Chul Park (Sept. 2020). “Application of Deep Learning in Dentistry and Implantology”. *The Korean Academy of Oral and Maxillofacial Implantology* 24, pp. 148–181. DOI: 10.32542/implantology.202015.

[KYK19] Kavitha, Muthusubash, Novanto Yudistira, and Takio Kurita (2019). “Multi instance learning via deep CNN for multi-class recognition of Alzheimer’s disease”, pp. 89–94. DOI: 10.1109/IWCIA47330.2019.8955006.

[Kha+19] Khan, Suleman et al. (2019). “Facial recognition using convolutional neural networks and implementation on smart glasses”, pp. 1–6.

[KB17] Kingma, Diederik P. and Jimmy Ba (2017). “Adam: A Method for Stochastic Optimization”. arXiv: 1412.6980 [cs.LG].

[Kni97] Knight, Kevin (Dec. 1997). “Automating Knowledge Acquisition for Machine Translation”. *AI Magazine* 18.4, p. 81. DOI: 10.1609/aimag.v18i4.1323. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/1323.

[Kri18] Krishnamoorthi, Raghuraman (2018). “Quantizing deep convolutional networks for efficient inference: A whitepaper”. *arXiv preprint arXiv:1806.08342*.

[KA19] Krishnan, Abhinandan and Abilash Amarthaluri (2019). “Large scale product categorization using structured and unstructured attributes”. *arXiv preprint arXiv:1903.04254*.

[KH+09] Krizhevsky, Alex, Geoffrey Hinton, et al. (2009). “Learning multiple layers of features from tiny images”. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[KGM19] Kwasigroch, Arkadiusz, Michal Grochowski, and Mateusz Mikolajczyk (2019). “Deep neural network architecture search using network morphism”, pp. 30–35. DOI: 10.1109/MMAR.2019.8864624.

[Li+20] Li, Zewen et al. (2020). “A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects”. arXiv: 2004.02806 [cs.CV].

[LSY19] Liu, Hanxiao, Karen Simonyan, and Yiming Yang (2019). “DARTS: Differentiable Architecture Search”. arXiv: 1806.09055 [cs.LG].

[Liu+18] Liu, Jiaying et al. (2018). “Artificial Intelligence in the 21st Century”. *IEEE Access* 6, pp. 34403–34421. DOI: 10.1109/ACCESS.2018.2819688.

[LWW19] Liu, Qiang, Lemeng Wu, and Dilin Wang (2019). “Splitting Steepest Descent for Growing Neural Architectures”. DOI: 10.48550/ARXIV.1910.02366. URL: https://arxiv.org/abs/1910.02366.

[LH16] Loshchilov, Ilya and Frank Hutter (2016). “Sgdr: Stochastic gradient descent with warm restarts”. *arXiv preprint arXiv:1608.03983*.

[Mae+19] Maedche, Alexander et al. (2019). “AI-based digital assistants: Opportunities, threats, and research perspectives”. *Business & Information Systems Engineering* 61, pp. 535–544.

[MW+14] Maind, Sonali B, Priyanka Wankar, et al. (2014). “Research paper on basic of artificial neural network”. *International Journal on Recent and Innovation Trends in Computing and Communication* 2.1, pp. 96–100.

[Mal+20] Malviya, Sunil et al. (2020). “Machine learning techniques for sentiment analysis: A review”. *SAMRIDDHI: A Journal of Physical Sciences, Engineering and Technology* 12.02, pp. 72–78.

[Men+16] Mendoza, Hector et al. (24 Jun 2016). “Towards Automatically-Tuned Neural Networks”. Proceedings of Machine Learning Research 64. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, pp. 58–65. URL: https://proceedings.mlr.press/v64/mendoza_towards_2016.html.

[MTH89] Miller, Geoffrey, Peter Todd, and Shailesh Hegde (Jan. 1989). “Designing Neural Networks using Genetic Algorithms.”, pp. 379–384.

[MD01] Mjolsness, Eric and Dennis DeCoste (2001). “Machine Learning for Science: State of the Art and Future Prospects”. *Science* 293.5537, pp. 2051–2055. DOI: 10.1126/science.293.5537.2051. eprint: https://www.science.org/doi/pdf/10.1126/science.293.5537.2051. URL: https://www.science.org/doi/abs/10.1126/science.293.5537.2051.

[Mur19] Murugesan, Mallesh (2019). “Text Classification: Binary to Multi-label Multi-class classification”. URL: https://abeyon.com/textclassification/.

[Net+11] Netzer, Yuval et al. (2011). “Reading digits in natural images with unsupervised feature learning”. URL: http://ufldl.stanford.edu/housenumbers.

[Nji+19] Njima, Wafa et al. (July 2019). “Deep CNN for Indoor Localization in IoT-Sensor Systems”. *Sensors* 19, p. 3127. DOI: 10.3390/s19143127.

[ON15] O’Shea, Keiron and Ryan Nash (2015). “An Introduction to Convolutional Neural Networks”. *CoRR* abs/1511.08458. arXiv: 1511.08458. URL: http://arxiv.org/abs/1511.08458.

[Ome+21] Omee, Sadman Sadeed et al. (Sept. 2021). “Scalable deeper graph neural networks for high-performance materials property prediction”.

[Ong17] Ongsulee, Pariwat (2017). “Artificial intelligence, machine learning and deep learning”, pp. 1–6. DOI: 10.1109/ICTKE.2017.8259629.

[Pap] Paperspace (n.d.). “Paperspace: Cloud computing, evolved” (). URL: https://www.paperspace.com/.

[PK17] Park, Sungheon and Nojun Kwak (2017). “Analysis on the dropout effect in convolutional neural networks”, pp. 189–204.

[Par+12] Parkhi, Omkar M. et al. (2012). “Cats and Dogs”.

[Pas+19] Paszke, Adam et al. (2019). “Pytorch: An imperative style, high-performance deep learning library”. *Advances in neural information processing systems* 32.

[Pau+20] Paulucio, Leonardo S et al. (2020). “Product Categorization by Title Using Deep Neural Networks as Feature Extractor”, pp. 1–7.

[Pre12] Prechelt, Lutz (2012). “Early stopping—but when?” *Neural networks: tricks of the trade: second edition*, pp. 53–67.

[Pus20] Pushprajmaraje (2020). “Simple Multi-Class Classification using CNN for custom Dataset.” URL: https://medium.com/analytics-vidhya/multi-class-classification-using-cnn-for-custom-dataset-7759865bd19.

[Rea+17] Real, Esteban et al. (2017). “Large-Scale Evolution of Image Classifiers”. DOI: 10.48550/ARXIV.1703.01041. URL: https://arxiv.org/abs/1703.01041.

[Rea+19] Real, Esteban et al. (2019). “Regularized Evolution for Image Classifier Architecture Search”. arXiv: 1802.01548 [cs.NE].

[Ren+21] Ren, Pengzhen et al. (2021). “A comprehensive survey of neural architecture search: Challenges and solutions”. *ACM Computing Surveys (CSUR)* 54.4, pp. 1–34.

[Rid+21] Ridnik, Tal et al. (2021). “ML-Decoder: Scalable and Versatile Classification Head”. arXiv: 2111.12933 [cs.CV].

[Ros58] Rosenblatt, Frank (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review* 65.6, p. 386.

[Rud16] Ruder, Sebastian (2016). “An overview of gradient descent optimization algorithms”. *CoRR* abs/1609.04747. arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[Rum+95] Rumelhart, David E et al. (1995). “Backpropagation: The basic theory”. *Backpropagation: Theory, architectures and applications*, pp. 1–34.

[Sad+21] Sadiku, M et al. (2021). “Artificial Intelligence in Social Media”. *International Journal of Scientific Advances* 2.1, pp. 15–20.

[Sah18] Saha, Sumit (2018). “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way”. URL: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`.

[SB22] Sapkota, Suman and Binod Bhattarai (2022). “Noisy Heuristics NAS: A Network Morphism based Neural Architecture Search using Heuristics”. arXiv: `2207.04467 [cs.LG]`.

[Sax20] Saxena, Sharoon (2020). “Underfitting vs. Overfitting (vs. Best Fitting) in Machine Learning”. URL: `https://www.analyticsvidhya.com/blog/2020/02/underfitting-overfitting-best-fitting-machine-learning/`.

[SG06] Selman, Bart and Carla P Gomes (2006). “Hill-climbing search”. *Encyclopedia of cognitive science* 81, p. 82.

[SSA17] Sharma, Sagar, Simone Sharma, and Anidhya Athaiya (2017). “Activation functions in neural networks”. *Towards Data Sci* 6.12, pp. 310–316.

[SSS19] Siddique, Fathma, Shadman Sakib, and Md. Abu Bakr Siddique (2019). “Recognition of Handwritten Digit using Convolutional Neural Network in Python with Tensorflow and Comparison of Performance for Various Hidden Layers”, pp. 541–546. DOI: `10.1109/ICAEE48663.2019.8975496`.

[SZ15] Simonyan, Karen and Andrew Zisserman (2015). “Very Deep Convolutional Networks for Large-Scale Image Recognition”. arXiv: `1409.1556 [cs.CV]`.

[Smi17] Smith, Leslie N (2017). “Cyclical learning rates for training neural networks”, pp. 464–472.

[ST18] Smith, Leslie N. and Nicholay Topin (2018). “Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates”. arXiv: `1708.07120 [cs.LG]`.

[SLA12] Snoek, Jasper, Hugo Larochelle, and Ryan P Adams (2012). “Practical Bayesian Optimization of Machine Learning Algorithms”. 25. Ed. by F. Pereira et al. URL: `https://proceedings.neurips.cc/paper/2012/file/05311655a15b75fab86956663e1819cd-Paper.pdf`.

[SM02] Stanley, Kenneth O. and Risto Miikkulainen (2002). “Evolving Neural Networks through Augmenting Topologies”. *Evolutionary Computation* 10.2, pp. 99–127. DOI: `10.1162/106365602320169811`.

[Sut+13] Sutskever, Ilya et al. (2013). “On the importance of initialization and momentum in deep learning”, pp. 1139–1147.

[SKP97] Svozil, Daniel, Vladimír Kvasnicka, and Jiří Pospichal (1997). “Introduction to multi-layer feed-forward neural networks”. *Chemometrics and Intelligent Laboratory Systems* 39.1, pp. 43–62. ISSN: 0169-7439. DOI: `https://doi.org/10.1016/S0169-7439(97)00061-0`. URL: `https://www.sciencedirect.com/science/article/pii/S0169743997000610`.

[TL21] Tan, Mingxing and Quoc V. Le (2021). “EfficientNetV2: Smaller Models and Faster Training”. arXiv: `2104.00298 [cs.CV]`.

[TPL20] Tan, Mingxing, Ruoming Pang, and Quoc V Le (2020). “Efficientdet: Scalable and efficient object detection”, pp. 10781–10790.

[Tho+13] Thornton, Chris et al. (2013). “Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms”. KDD ’13, pp. 847–855. DOI: `10.1145/2487575.2487629`. URL: `https://doi.org/10.1145/2487575.2487629`.

[Tou+21] Touvron, Hugo et al. (2021). “Going deeper with Image Transformers”. arXiv: `2103.17239 [cs.CV]`.

[Veg+18] Vega, Maria Torres et al. (2018). “A review of predictive quality of experience management in video streaming services”. *IEEE Transactions on Broadcasting* 64.2, pp. 432–445.

[Wan+13] Wan, Li et al. (2013). “Regularization of neural networks using dropconnect”, pp. 1058–1066.

[Wil92] Williams, Ronald J. (May 1992). “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. *Mach. Learn.* 8.3–4, pp. 229–256. ISSN: 0885-6125. DOI: `10.1007/BF00992696`. URL: `https://doi.org/10.1007/BF00992696`.

[Wil+90] Wilpon, Jay G et al. (1990). “Automatic recognition of keywords in unconstrained speech using hidden Markov models”. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 38.11, pp. 1870–1878.

[Wu+19] Wu, Bichen et al. (2019). “FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search”. arXiv: `1812.03443 [cs.CV]`.

[Wu+21] Wu, Haiping et al. (2021). “CvT: Introducing Convolutions to Vision Transformers”. arXiv: `2103.15808 [cs.CV]`.

[XY17] Xie, Lingxi and Alan Yuille (2017). “Genetic CNN”. arXiv: `1703.01513 [cs.CV]`.

[Xu+20] Xu, Sheng et al. (2020). “Convolutional neural network pruning: A survey”, pp. 7458–7463.

[Yan+18] Yang, Tong et al. (2018). “Metaanchor: Learning to detect objects with customized anchors”. *Advances in neural information processing systems* 31.

[Zha+17] Zhang, Chiyuan et al. (2017). “Understanding deep learning requires rethinking generalization”. arXiv: `1611.03530 [cs.LG]`.

[zhe21] zhengjian2322 (2021). “net2net”. URL: `https://github.com/zhengjian2322/net2net`.

[Zho+18a] Zhong, Zhao et al. (2018a). “BlockQNN: Efficient Block-wise Neural Network Architecture Generation”. arXiv: 1808.05584 [cs.CV].

[Zho+18b] Zhong, Zhao et al. (2018b). “Practical Block-wise Neural Network Architecture Generation”. arXiv: 1708.05552 [cs.CV].

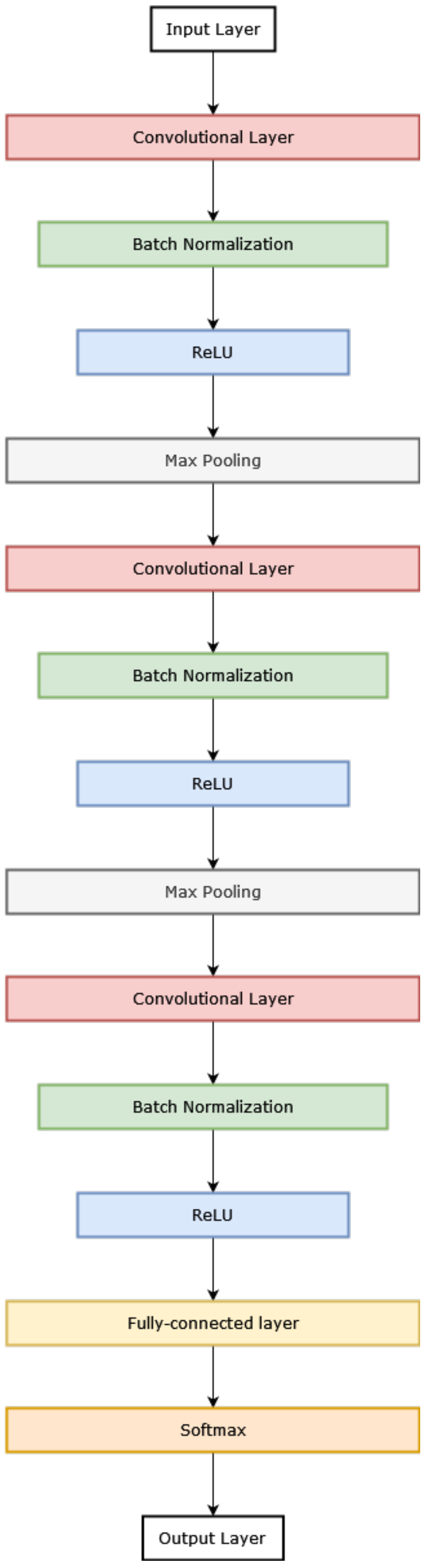
[Zho23] Zhou, Anthony (2023). “cnn-nash”. URL: https://github.com/anthzhou/cnn-nash.

[ZL17] Zoph, Barret and Quoc V. Le (2017). “Neural Architecture Search with Reinforcement Learning”. arXiv: 1611.01578 [cs.LG].

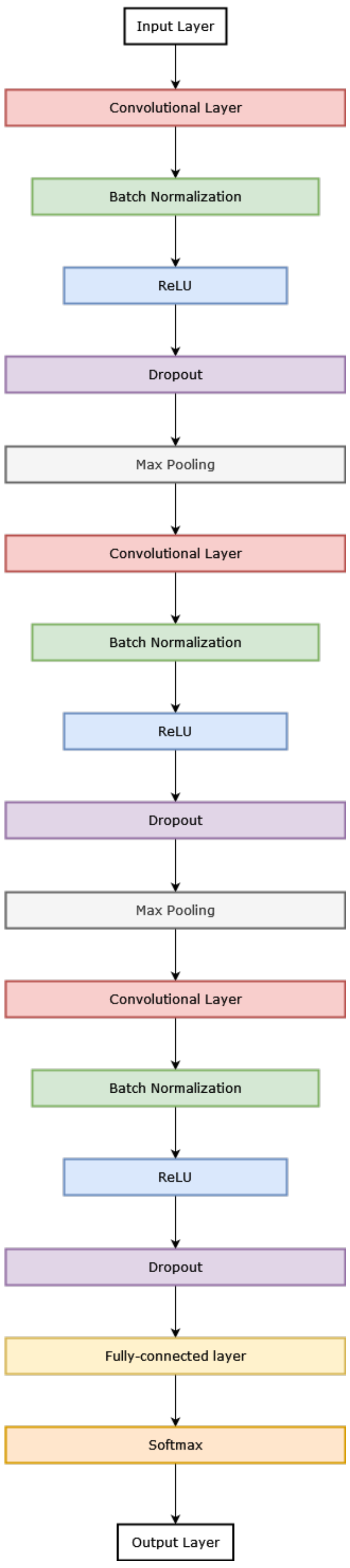
[Zop+17] Zoph, Barret et al. (2017). “Learning Transferable Architectures for Scalable Image Recognition”. DOI: 10.48550/ARXIV.1707.07012. URL: https://arxiv.org/abs/1707.07012.

Appendices

Initial network structure



B Initial network structure with dropout


Input Layer
Convolutional Layer
Batch Normalization
ReLU
Dropout
Max Pooling
Convolutional Layer
Batch Normalization
ReLU
Dropout
Max Pooling
Convolutional Layer
Batch Normalization
ReLU
Dropout
Fully-connected layer
Softmax
Output Layer