Internship thesis

performed by

Timothée Neitthoffer

Computer Science Department

Academic year 2019 - 2020

Neural architecture growing, pruning and search

Place of the internship
Inria

Internship supervisor
Yannis Avrithis

INSA Rennes supervisor
Mireille Ducassé

PFE defended on 01/09/2020

Contents

1 Introduction

1.1 Neural Network and deep learning

Machine learning Computers are more and more present in our society and the tasks we want them to perform are increasingly complex. Tasks such as recognizing handwriting, classifying data or playing a game are hard to address directly and it has been found that learning from the data performs well on such tasks. This category of algorithms that learns from the data is called Machine Learning and is part of the Artificial Intelligence family. Among the multiple machine learning algorithms, this internship's work is centered on neural network.

Neural Network A task can be viewed as a function mapping the inputs to the desired output. For example, in an image classification task, the inputs would be the pixels of the image and the output would be what's in the image. Neural networks model a generic function that can approximate other functions by modifying its parameters. This generic function is a succession of multiple simpler functions called *layers*. The aim of a neural network is then to modify its parameters to fit the function of the task. The process of fitting the neural network to the task function is called *training*.

Convolution The computer vision domain is widely used as support for multiple works on neural network because of its complexity and the margin of progression of its algorithms. In this internship, an image classification task (a subcategory of computer vision) is used as support for the experiments. Neural networks that are designed for such task need to take advantage of the properties of an image to perform better.

In an image, the pixels that are close to each other are usually strongly linked, a succession of black pixels can form an edge for example. This implies that to understand a part of the image, there is no need to look at the whole image at the same time. Considering this property, another type of functions for neural network was designed that is called *convolutions*. This type of function is similar to a succession of filters looking for particular elements in the image. Using such operation in neural networks has achieved high performances in a lot of computer vision problems for a long time (convolutions were first designed in the 1980 [1]) and are nowadays widely used for tasks involving images.

The neural networks used in this internship are based on convolutions.

Deep learning As the tasks we want to address are harder and harder, the networks used need to model more complex functions. This leads to networks that are deeper. But as they stack multiple layers, those deep networks are harder to optimise and face many problems. Recent researchs help to understand them better and deep neural networks are now used in most of the tasks.

Neural networks usually have a linear flow of information, having connections from one layer to the next. However, it is also possible to use much more complex architectures with connections from one layer to multiple others or with multiple layer branches. Defining the architecture is then a complex task that needs lots of trials and is not always certain to define the optimal architecture.

1.2 Network Architecture Search

Neural Networks' performance highly depends on the architecture of the chained layers it is composed of. Considering this, a lot of time is spend to design architectures that are competitive on the different tasks while keeping the number of computations low enough to be trained and used. As such thing becomes harder and harder as the networks increase in size and complexity, a new field of research has emerged that aims to solve this problem by making it automatic. This field called *network architecture search* aims to create algorithms that find the optimal sequence of layers and flow of information between them on a given task. However, in order to determine the correct architecture of layers it is necessary to also learn the parameters of the network, making the number of computations needed heavy. Moreover, the number of possible combinations of layers is immense, making this objective even more difficult. To achieve such a feat, different techniques and algorithms such as evolutionary (genetic) algorithms, reinforcement learning and more recently differentiable programming, are developed. Those techniques however still require lots of computations (4 GPU days for one of the fastest while other can take up to 2000 GPU days; c.f. [17]) and sometimes don't scale well to more complex tasks requiring bigger architectures.

1.3 Context of the work

After the course on artificial intelligence given by Pascal Garcia at the INSA Rennes, I was deeply interested in this domain and more precisely neural networks and the way they work. To learn more about it, I followed online lessons alongside my studies at the INSA. This internship allowed me to work on the subjects I found most interesting as I was able to discuss this internship with my supervisor. Moreover, I was considering working in the research field but, having no experience in it, I was still hesitant. This internship was the opportunity to learn more about it and experience it in spite of the particular conditions faced this year.

This *Projet de Fin d'Etude* was realised in the Linkmedia team of the INRIA Rennes. It address the subject of architecture search for convolutional neural networks using architecture growing and pruning.

1.3.1 INRIA Rennes and Linkmedia team

INRIA INRIA Rennes is a research center in computer science created in 1980. It is composed of 28 teams in Rennes for a total of 600 people, researchers and supports. It works in collaboration with public research institutes, innovation centers, regional authority, companies and universities. Its scope of research span from cyber-security, human-computer interactions to digital health and digital ecology.

Linkmedia Inside this organisation, the Linkmedia team works on content-based media linking in order to create links between multimedia collections. Its work concerns pattern recognition, description and structuring over a collection and linking of content. It is composed of 15 team members and 11 PhD students.

1.3.2 Goals of the internship

Nowadays, neural networks achieve incredible performances on lots of tasks such as computer vision thanks to the discovery of better architectures. Those architectures are the results of years of research and of the work of multiple researchers. The performances of the architectures vary depending on the number of layers stacked, the type of layer, the parameters of the layers or how the information flows in between the layers. Moreover, the architectures used are different for every new task. Considering the time and the efforts put to come up with only one architecture for a certain task, algorithms are designed to search for the architecture of neural networks.

This internship aims to investigate this particular domain using a combination of several ideas. The architecture search will be done using pruning techniques over the complete search-space of the network in an iterative way so as to deal with the huge size of the search space (connections between the layers). In order to clarify this aim, the different points will be discussed in the Background section (2).

1.3.3 Planning

This internship is divided into three main parts spanning from the beginning of March to the end of August (see Gantt diagram in appendix figure 24).

- The first part focuses on the way to make this problem iterative as it will be the main basis of our algorithm. It is important because the behavior of the algorithm may changes if the pruning and the addition of dense connections were done at first in an non-iterative way.
- The second part includes the pruning technique so as to be able to put it in place before working on the full search space, allowing faster experiments.
- The third part allows the search to be performed on the expanded space by connecting every layers to the previous ones.

In this manner, we progressively build our method based on the results of the previous experiments.

2 Background

Artificial Intelligence and Neural Networks are nowadays the main trend in computer vision and other human-like tasks. In spite of its rising usage, the concepts they are based on are sometimes overlooked. In order to better understand the subject and the elements discussed in this Projet de Fin d'Etude thesis, I will describe in this section the different concepts used.

2.1 Neural Networks and Deep Learning

2.1.1 Origins of neural networks

Since the beginning of computers, engineers and researchers imagine algorithms that could perform human-like tasks such as identifying images [4], reading text from images [12], making links between medias [18] or playing a game [20]. One of the assumption lots of techniques are based on is that there exists a mathematical function such that given the right inputs, can solve the given task. Considering this, a way to perform the task is to try to approximate this function by looking at the training set.

For example, our task is to determinate whether a dot is *blue* or *orange* with respect to its coordinates (x, y). Given the first training dataset in figure 1, it is possible, by taking measurements to find the function that separates the two classes (in green). Problems where this function is linear are called *linearly separable problems* and are simple to solve. However, most problems are not linearly separable and the distribution of the inputs may be very complex. Given a complex dataset (such as the second dataset in figure 1), it is no longer trivial and fast to take measurements on the training set to find the separation. Other algorithms to approximate the discriminating function are then needed.

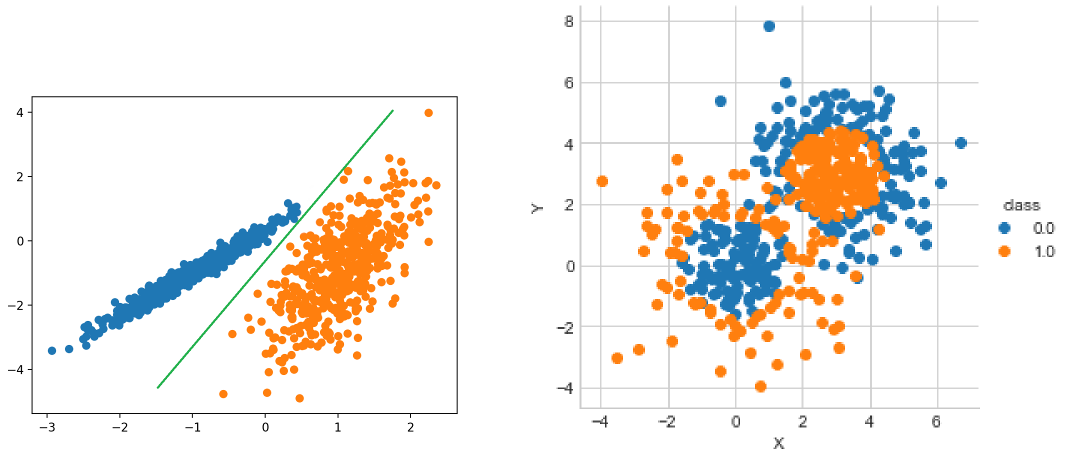


Figure 1: Linearly separable dataset and non linearly separable dataset.

Fitting a function We previously saw that there exists a function mapping the inputs to the decision for every task. This particular function is unknown. So instead of trying to figure this function out by looking at the dataset, it is possible to combine more general functions together so as to come closer to the real function. Those general functions then adjust their parameters by learning the approximation on the examples of the dataset (example in figure 2).

For example, let's say that the function we try to approximate is $y = 3 * sin(x) + x$ (the target function) and to do so, we use the combination of two functions: αx and β which gives the function $y = \alpha x + \beta$ (the approximation function). Then, using an optimisation algorithm to adjust the parameters (α, β) by taking some points in $\mathbb{R}$ as inputs (x) and the value of those points as a target output, the approximation function fits the target function and is able to give outputs close to the ones from the target function.

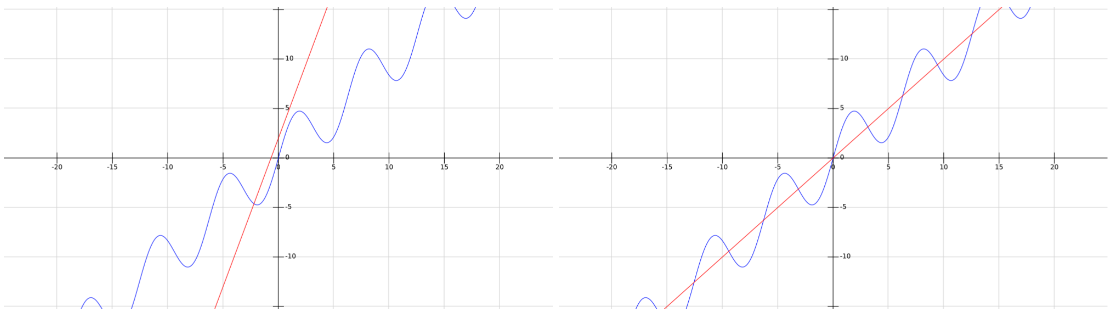


Figure 2: Target function and approximation function: before optimisation (left), after optimisation (right).

The approximation function can also be considered as a vector multiplication between the input vector and the parameter vector that maps the $\mathbb{R}^2$ space of the input to a single dimension in $\mathbb{R}$:

$$\begin{pmatrix} y \end{pmatrix} = \begin{pmatrix} 1 \\ x \end{pmatrix}^t \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{1}$$

Under this form, it is possible to generalise for the case where we want to map inputs from a $\mathbb{R}^m$ to a $\mathbb{R}^n$ output space by increasing the number of rows and columns in the parameter matrix (previously vector):

$$\begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}^t = \begin{pmatrix} x_0 \\ \vdots \\ x_m \end{pmatrix}^t \begin{pmatrix} \alpha & \cdots & \gamma \\ \vdots & \ddots & \\ \beta & & \delta \end{pmatrix} \tag{2}$$

Such vector/matrix multiplication is also represented under another form in the machine learning domain to ease the notation (see figure 3).

The algorithm pictured in figure 3 is a linear classifier. Under this form, every matrix multiplication results in a layer where a neuron is the value of one of the dimensions (for example, in the right figure of figure 3, the output layer was a layer composed of n dimensions where the neurons are of value y_i).

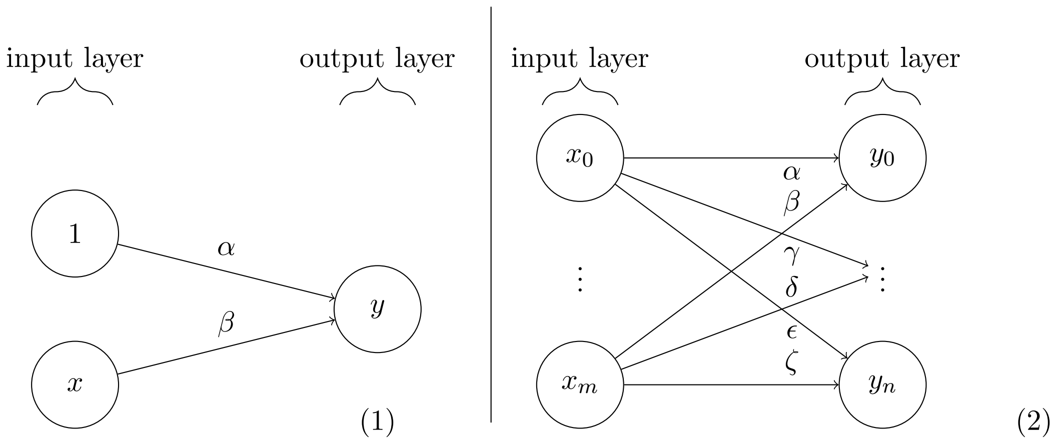


Figure 3: Left: representation of equation (1); Right: representation of equation (2). Usually, the parameters are omitted so as to keep the figure light.

Linear classifiers Linear classifiers are able to learn from high dimensional data where measurement based techniques may either fail or take a lot of time to infer the output. But as its name indicates, it is only able to address linear problems even if we add more matrix multiplications between the inputs and outputs as the combination of linear transformations only results in a linear transformation.

In order to be able to tackle non-linear problems, it is needed to bring some non-linear functions in our approximation function. One solution would be to bring our inputs, using some non-linear transformation into a space where they can be linearly separable. For example, we could modify the equation 1 by applying to the input vector this transformation:

$$\begin{pmatrix} 1 \\ x \\ sin(x) \end{pmatrix}$$

The non-linearity is brought by the $sin(x)$ function that uses the previous input x. Such transformation would be very efficient as it is possible to learn the exact form of the target function. However, this technique requires a prior knowledge over the problem treated as one set of transformations can perform well on one task but not on the others. So, it isn't simple to put efficiently in place.

Another solution is to bring this non-linearity through all our matrix multiplications so that each of our transformations is non-linear. This method allows to bring the information into higher non-linear spaces at every matrix multiplication (represented in figure 4).

By stacking multiple matrix multiplications and non-linear functions, we finally obtain a neural network.

2.1.2 Neural networks

As seen previously, neural networks are algorithms that model a global function. This function is a succession of intermediary mathematical functions which outputs are the inputs

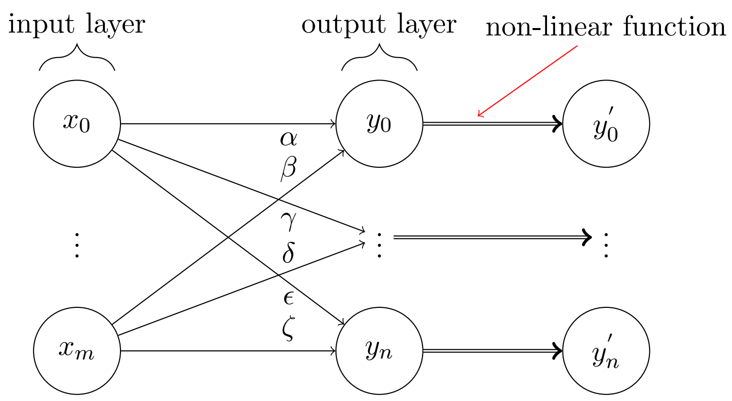


Figure 4: Addition of a non-linear function to figure 3 in order to address non-linear tasks.

of the next function. It aims to fit the target function defined by the task it learns on and represented by the data. The outputs of an intermediary function is called a *layer*. A layer can have multiple dimensions and a dimension is called a *neuron*.

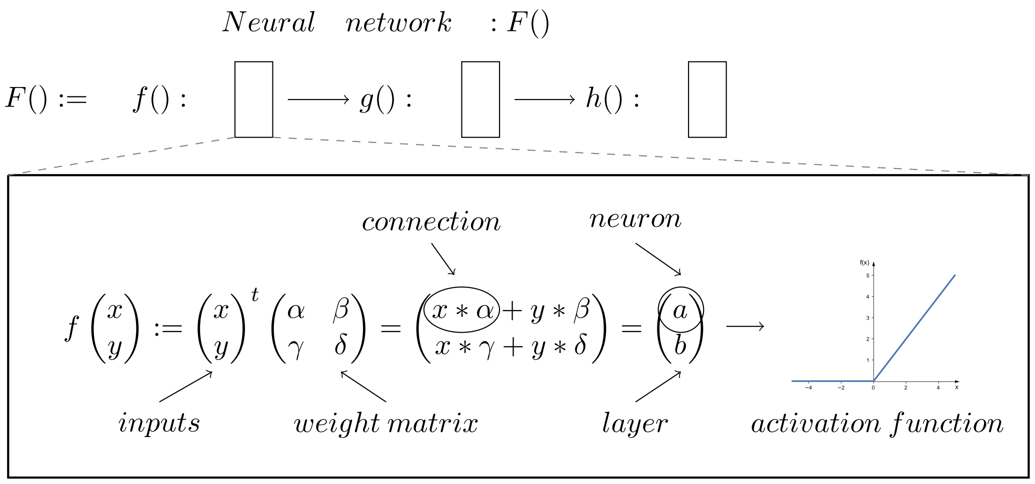
As the global function should be general enough to be able to fit any kind of target function, the intermediary functions are kept as simple as possible and usually are a vector-matrix multiplication between the inputs of the intermediary function (the values of the previous layer) and a *weight matrix*. It is those weights that algorithms optimise by finding the best value of the weights so that the network can properly perform the given task. A multiplication between an input element and a weight is called a *connection*. The matrix multiplication operation is called *fully connected* operation.

To be able to address non-linear problems, the matrix multiplication is followed by a non-linear function called *activation function* (Sigmoïd or ReLU for example). The description of the succession of intermediary functions with their parameters (number of neurons for example) and how they are connected is what is called the *architecture* of the neural network. If there are more than one layer in the neural network (without counting the input layer nor the output layer) then it can be referred to as a *Deep Neural Network* that is part of *Deep Learning*.

In the representation of the figures 3, 4, 5 and 6, a layer is the output of an intermediary function. However the name layer is usually used to refer to the operation leading to the output, meaning the intermediary function. In this thesis, a layer will be referring to the intermediary function.

2.1.3 Convolutions

Even if analyzing an image seems trivial for our brain, we sometimes fail to recognize something or think it is something else. In those image related domains such as object recognition or image classification, it is necessary to exploit hypothesis so as to allow the algorithms to compete with humans.



Neural network : F()
F() := f() : → g() : → h() :
connection
neuron
$f\begin{pmatrix}x\\y\end{pmatrix} := \begin{pmatrix}x\\y\end{pmatrix}^t \begin{pmatrix}\alpha & \beta\\ \gamma & \delta\end{pmatrix} = \begin{pmatrix}x*\alpha + y*\beta\\ x*\gamma + y*\delta\end{pmatrix} = \begin{pmatrix}a\\b\end{pmatrix} \longrightarrow$
inputs
weight matrix
layer
activation function

Figure 5: Mathematical representation of a neural network.

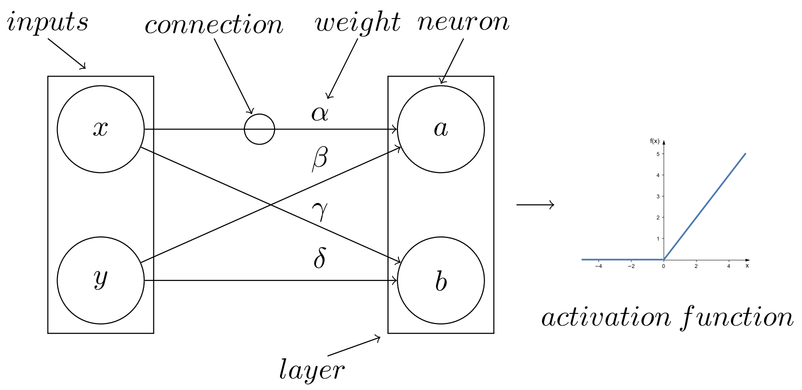


Figure 6: Machine Learning representation of a layer.

Convolution assumption and operation The main assumption made is that pixels close one to another in an image are strongly linked while distant pixels are independent. Based on this hypothesis, another type of layer was designed in 1981 by Fukushima K. [1]: the *convolutional layer* or *convolution*. A convolution performs multiplications over a range of the inputs with a weight matrix (the *kernel*) before summing them. Then, it slides step by step through the remaining inputs, repeating the operation at every step (see figure 7). As seen for the matrix multiplication layer, in order to be able to address non-linear problems, this operation is combined with non-linear functions. The square or rectangle defined by the weight matrix over the image is called a *window*. The number of pixels the window shifts at each step is called the *stride*. In figure 7, the stride value is 1 making the window overlap on some pixels. It is possible to have strides of higher number such that not all pixels overlap between the different steps. Generally, the stride is not higher than the kernel size.

A convolution usually possesses multiple weight matrices where each one computes a feature, making the operation similar to filtering particular components of the image.

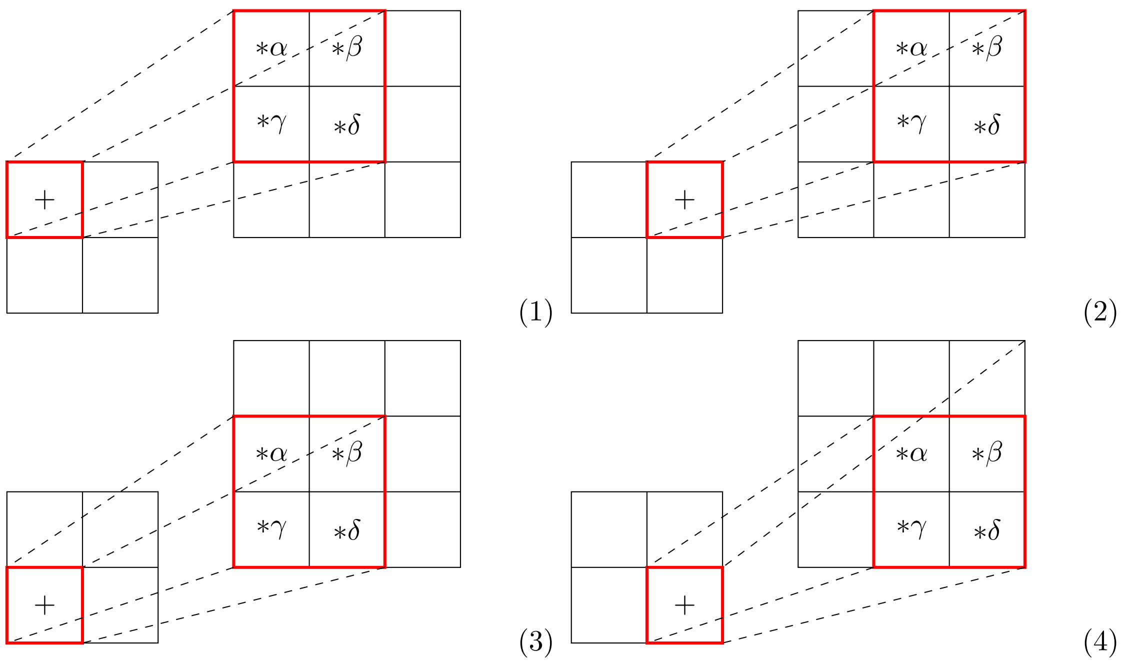


Figure 7: A convolutional layer operation between a 3*3 image and a 2*2 weight matrix.

In figure 7, the 3*3 image has been reduced to a 2*2 feature map (can be seen as an image with only the desired feature in). It is possible to modulate the size of this output feature map by adding *"null pixels"* around the input image, meaning adding rows and/or columns of empty pixels, making the input image look larger. This technique is called *padding*. Another way that can be used in combination with the previous one is modifying the size of the kernel. In figure 7, if the kernel is of size 3*3, then the convolution results in a 1*1 feature map. In a similar way, if the kernel is of size 1*1 then the output is a feature map of the same size of input. Usually, the padding is adjusted so that the size of the output is the same as the size of the input.

Neural networks that are using convolutions are called *Convolutional Neural Network* (CNN). Standard convolutions uses a kernel of size $3 * 3$.

Pooling In CNN, the convolutional layers are paired with another type of operation that is called *pooling*. Similar to the convolution, it performs an operation over a window of the inputs. The main differences are that the operation doesn't require a weight matrix or anything but set operations such as picking the maximum value of the window or calculating the average of the values, and that the steps made are most of the time of the size of the window. This means that there is no overlaps made. This operation is used to reduce the size of the feature map and perform a down-sampling, reducing the number of computations to perform in the next layers. The size of the window determine the down-sampling ratio. For example, a pooling of 2*2 will reduce the size of the image by 2 (see figure 8).

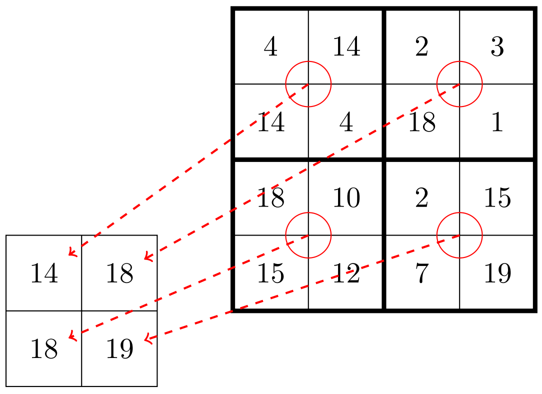


Figure 8: Maximum pooling with a 2*2 window.

2.1.4 Batch Normalization

Neural networks learn from the data in the batches and update their weights based on them. However, the gradient received from the batch is noisy as it considers only a part of the training set. One of the effect caused by this noise is that the network has to learn a new distribution with different mean and variance for the inputs at every batch. Moreover, every weight is updated under the assumption that all the other weights do not change. However all weights are changing at every iteration. Weight changes in layer $k-1$ affect the distribution of the inputs of the layer k in addition to the noise from the batch. Thus, making the previous assumption on the gradient step of the layer k is weak particularly in the first phase of the training where changes can be dramatic. It can make convergence slower and more difficult. This problem called *covariate shift* [6] can impede the learning process especially for deeper network.

To mitigate this effect, a normalization layer has been introduced named Batch Normalization [6]. This layer brings the inputs of the next layer to the same mean and variance and is performed as follow. A standard layer operation is of the form:

$$x_{(k)} = h_k(W_k^t x_{(k-1)}) \tag{3}$$

Where h is a non-linear function, W_k is the weight matrix of the layer and $x_(k)$ is the output of the layer k. Let's consider that

$$z^k = W_k^t x_{(k-1)} \tag{4}$$

Then, the normalization would be:

$$\tilde{z}^k = \frac{z^k - \mathbb{E}[z^k]}{\sqrt{\mathbb{V}[z^k]}} \tag{5}$$

The variance and mean are calculated based on the batches and updated little by little all along the training. By applying such normalization we force the inputs to every layers to be close to a normal distribution. This eases the dependencies between the layers and makes it so changes in a layer are not invalidated by changes in another layer. In most of the CNN, the pattern Convolution-BatchNormalization-ReLU is describe as a single layer and stacked to create the network.

In addition to reducing the covariate shift, it has been observed that batch normalization introduces other benefits. This additional layer also allows the use of higher learning rate to converge faster and also have a stabilizing effect that makes the network less sensitive to initialization randomness.

2.1.5 Optimisation of neural networks

In order to fit the target function, it is necessary to optimize the different weight matrices of the neural network. This optimization can be done with several methods depending on the nature of the task. For example, if we don't know what the best output is for an input (like for playing chess), techniques such as reinforcement learning or genetic algorithm are used. However, if the best output is known, the techniques used are gradient descent based [10] (even if it is possible to use the previous ones, they are several time slower than gradient descent).

In this internship, we apply our algorithms on an image classification task. As we know the content of every image of the dataset, we can use gradient descent based techniques to optimize our networks.

Weight adjustment The neural network we want to optimise is usually filled with values from a random distribution. The weights are then non optimal for the task we want to perform. In order to know how we modify them so as to give the correct answer, we have to compare the output of the network and the correct output. This comparison is done by a function called a *loss function.* The loss function gives a single value as output representing how big is the error between the two answers. That way, we want to modify our weights such that the value of the loss function is minimal. However, due to the high number of parameters in the neural network, it isn't possible to solve analytically the minimum of the loss function to find the optimal weights. Another way to do it is following the slope of the function by updating little by little the weight in this direction, descending the gradient.

For every weight, the gradient of the loss function with respect to this particular weight is calculated (usually using the gradient chain rule) and the weight is updated using the following equation:

$$w^t = w^{t-1} - \alpha * \frac{\partial Loss(network\ output, correct\ output)}{\partial w^{t-1}} \tag{6}$$

Where w^t is the weight at time t, *Loss* is the loss function used and α is the learning rate such that $\alpha \in]0, 1]$.

As the loss function is a convex function, the algorithm is certain to converge to a minimum. Yet, this minimum can be a local minimum and not the minimum of the whole function.

The goal of the learning rate is to smooth the descent so as no to make "big jumps" that could make it miss the minimum or even increase the loss. Lots of variants of this update exist such as adding momentum or computing an adaptive gradient [7].

This algorithm of gradient descent can be illustrated as trying to go down a mountain in the fog. You look for where the slope is the steepest and you go down by some meters depending on the strength of the slope and repeat the operation. The shape of the mountain is the shape of the loss function and the slope is the gradient of this loss function.

Batch updates There are different strategies to apply this update. For example, it is possible to apply this update at every example of our dataset. However, by doing this, the network will update itself to follow only one point in the space while ignoring other possibilities. This results in an irregular descent as the gradient won't give a general path but a short-sighted one. On the contrary, we take in account every example of the dataset to have the whole picture of the gradient. But such a thing is prohibitively expensive in practice. Since datasets are composed of tens of thousands of example, performing one update would require a lot of computations, too many to be practical. An intermediary solution is then preferred: computing gradient with only a portion of the dataset. This portion of the dataset is called a *batch*. The size of the batch may vary depending on the size of the dataset and of the network.

Learning schedule In most cases, passing through the dataset only once isn't enough for the network to converge to a minimum. Then, the process of learning pass several times through the dataset. A pass on the dataset is called an *epoch*. The number of epochs is determined by the size of your model as well as the difficulty of the task. However, there is no techniques to easily determine the learning rate, the batch size or the number of epochs. These parameters are defined by experience and trial and error.

It can be interesting also to modify the learning rate as the number of epoch increases. Indeed, a larger learning rate will perform large updates while smaller one will perform small updates. Both effects are useful as the first one allows to converge faster but can't be precise whereas the second one takes time to converge but takes precise steps. Then, it is beneficial to have a larger learning rate during the first epochs to converge faster to the minimum and then to have a smaller learning rate to be able to precisely reach the minimum.

2.1.6 General problems

However, as efficient as they can be, neural networks also face several problems due to their construction and the way they are trained.

Vanishing Gradient When calculating the gradients, it can be seen that the weights from the first layers receive lower gradients than the ones from the last layers. This is due to how those weights impact the value of the loss. The farther a weight is from the output, the more transformations there are before the output, so its impact is reduced. This imply that it receives lower gradient. This problem is called *vanishing gradient* [3].

Overfitting During the training, the network learns multiple times the same examples through the different epochs. However, learning on the same examples can lead the network to only focus on those examples and not be able to generalise to inputs outside of the dataset.

This problem is called *overfitting* and occurs when there is little data to train on, too many epochs or when the network is too big for the task needed (it tries to model a much more difficult function).

2.2 Progressive inference

Usually, neural networks are constituted as one block that is then optimized so that it can infer the output based on the inputs. However, this kind of inference face problems as learning it is a process that needs a lot of examples and that by going deeper and deeper, the network try to look at something complicated, forgetting simpler results.

To answer such problems, different progressive inference techniques were developped.

2.2.1 Growing an architecture

Some architectures are facing problems of overfitting due to how deep they are and that they learn on small datasets of labeled examples (examples for which we know the desired output). It is possible to grow the network so as to reach the ideal depth that does not overfit while having a high accuracy [14].

Growing a network means to add, given a criterion, a new set of layers to the existing one (see figure 25). These new layers usually have the same structure (for example, two convolutions of given width with a pooling at the end) called building block. And each time the criterion is valid, a new building bloc is added to the architecture. This allow to facilitate the training for the first layers while preventing overfitting in the later one. However, such technique may require more epochs to be performed as the last layers are added later in the training schedule.

The criterion for the growing can be of multiple origins. For example, it is possible to grow the network at some predetermined epochs (at the 10^{th} and 30^{th} for instance), it can be loss-based as when the loss stop decreasing, the network has converged and can't learn further as it is.

2.2.2 Shallow Deep Network

As the inference goes deeper in the network, it will try to find more complex scheme. However, some exemples can be very simple to determine and correctly answered before the end of the training. Using this observation, [22] presents a way to keep the quality of the simpler examples from being distorded in the later layers that is called Shallow Deep Netowrk. What they call *shallow deep* is the uses of output layers (the one that takes the decision whether it is a cat or a plane) at intermediary stages. This method has been designed to address what is called the *overthinking problem*. This problem is an observation that for some inputs, the correct result can be decided before reaching the final layer of the network. Moreover, some of the correct decisions made at the early layers are no longer correct as they continue toward the final layer. This can be caused by the network trying to decide based on small details that work well on complex inputs but that are confusing and unnecessary on easier inputs.

To know if an intermediary output (Internal Classifier in figure 9) has found a solution and that it is possible stop the process here, the criterion is based on the confidence of the output. The confidence corresponds to the probability of the input to belong to the intermediary output decision. It is this confidence that the outputs try to learn. This means that the outputs, given the input, compute values for the different classes (cat or plane) and that the higher this value is, the more probable it is for the input to belong to this class. Then, it is possible to set a parameter $q \in [0, 1]$ that represent the threshold of the decision. If the value of an output is greater than q, then a decision can be taken.

The uses of intermediary output layer enable the network to make a decision as early as possible, saving computations and gaining accuracy as the inputs that would have been mistaken at the latest layers are decided by the first ones. It also optimises the network to learn discriminant features at each stages, specializing the different outputs and allowing for a better global accuracy.

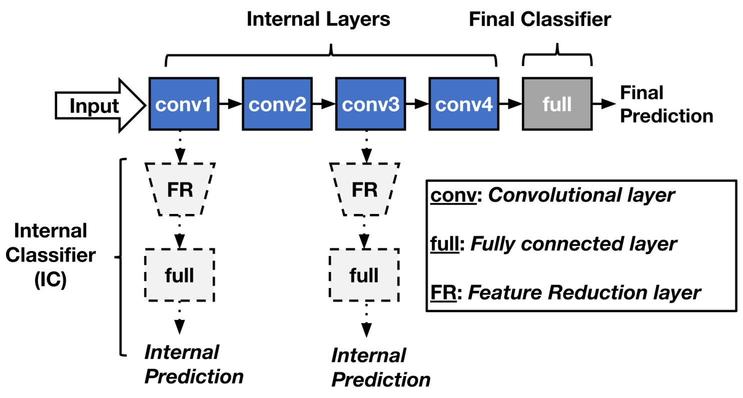


Figure 9: Example of Shallow Deep network. The Internal Classifiers represent the set of layers used to perform a prediction.

Source: Shallow Deep Network paper [22]

2.3 Pruning a network

Pruning has been a research subject since a long time and is more and more active with the efforts made to bring deep networks to mobile devices that are resource limited [2] [25].

As neural networks are becoming deeper and deeper, the number of connections increase exponentially. This increase of connections leads to computational overheads that slows the inference speed of the networks and induces a high memory cost.

It has been found that there exist redundant or unnecessary connections over the network [2]. In addition, some connections are more important to the output than others. The removal of non-necessary connections alleviates the memory and computational cost of the network while keeping a matching accuracy [16].

The pruning is generally viewed as setting and keeping to zero the pruned weights. This is usually done by element-wise multiplying the weights with a binary mask before the operation performed by the layer. The mask weight is 0 if the connection is pruned and 1 if the connection is kept.

Pruning elements The pruning algorithms can differ on several points [24](see figure 26):

- they can prune weight by weight by assuming no structured relationship between pruned weights [16](*Unstructured* figure 27) or prune a complete neuron or filter by assuming structured relationships between the weights [13] (*Structured* figure 28). It can be done in a local (single or small set of layer, see figure 29) or global (the whole network, see figure 30) fashion.
- They can select the elements to keep based on the magnitude of the weights [15], on gradient measurements [16] or via a mask learned by some algorithm.
- They can perform the pruning once (*One Shot*) or do it little by little (*Iterative*) as in figure 31 and
- they can decide to prune at the end of the training, during the training or even before the training.

Traditional pruning Traditionally, pruning is performed on a large neural network that is fully trained and the weights are kept as they are. This technique allows to reduce the number of parameters in a network while keeping the accuracy of the subnetwork close to the original. Training a pruned subnetwork from scratch (without previous training or little) was observed to achieve lower accuracy and thus is not considered.

This observation seems strange as the subnetwork after pruning and reinitialization should be able to perform at least as well as the subnetwork pruned from the completely trained big network as they both has the same structure. However, it has been found that the design and learning dynamics are different between the big pruned network and the subnetwork trained from scratch.

2.3.1 The Lottery Ticket Hypothesis

The *lottery ticket hypothesis* is a metaphor used in the paper of the same name [15] to describe the existence of subnetworks in an over-parameterized big network that can at least match the performance of this fully trained network. Such subnetworks are called *winning tickets* because they have weights that are initialized in a fashion that allows them to approximate well the global function compared to others. In that way, they have won the initialisation lottery.

Traditionally, after pruning a trained network and if we want to train it from scratch (trained as if it was a new network), it is considered good practice to reinitialize randomly the weights. Whereas, given the lottery ticket hypothesis, the performance of the found subnetworks is dependant to the initialisation. As such, the reinitalization throw off their good properties. The idea is then to find the winning subnetworks and train them while keeping their initialization intact.

With such technique, they managed to keep only up to 10% of the original network without degrading the accuracy. However, this technique was found and tested on small computer vision tasks and sometimes, it is needed to tweak the learning rate and its' schedule to achieve state of the art results.

Stabilizing the lottery hypothesis In a following paper (Stabilizing the Lottery Ticket Hypothesis [21]), Frankle and Carbin investigate the possibility to reset the weights not at the initialization, but instead to their value at epoch k. Indeed, it has been observed that, due to the randomness of the initialisation, the subnetworks are particularly sensitive to noise in the beginning of the training. If it is performed with caution (begin with a smaller learning rate to reduce individual misstep before rising it to its original level), it becomes possible to train subnetworks that are more stable (the variations of the loss are smoother).

From this observation, they modified their former hypothesis and tested the stability of the new found winning tickets. It was found that resetting the weights at epoch k instead of epoch 0 gives better performance and stability to the subnetworks, allowing to perform stronger pruning for the same accuracy.

2.3.2 Single shot Network Pruning: SNIP

Lots of pruning methods need the network to be trained to convergence before pruning it and sometimes, reinitializing the subnetwork to train it from scratch. Such techniques need to perform at least two trainings (one for the big network and one for the subnetwork) or even more if an iterative pruning is used as it would need $n + 1$ trainings if n prunings are performed. This is an expensive task both in computation and time.

In order to overcome the burden of having to train every time a pruning have to be performed, research have been done to perform the pruning before the end of the training and as early as possible. Single shot Network Pruning [16] (aka SNIP) achieves such a feat by allowing the network to be pruned without the need for training.

Method In order to become independent from the training, SNIP tries to find the important connections of the network using the data of the training. Moreover, as there is no training, it is important to also make the decision independent from the weight values as they are not trained. To do so, auxiliary variables $c \in \{0, 1\}^m$ are used representing the connectivity of the parameters w. A variable is assigned to a weight and acts similar to a mask. In that way, the weight is separated from whether there is a connection or not.

However, as it is, the variables are discrete and not continuous, making it impossible to determine via gradient based techniques. To allow the use of those techniques to evaluate the importance of the connection, the constraint on the value of c is relaxed to a continuous space. Then, the importance of the connection and its effect on the loss function are determined by the gradients they receive. The gradients of the connectivity parameters are computed by forwarding a batch of examples and computing the loss of the network with respect to both the weights and the connectivity parameters. Each variable is then ranked by the gradient it receives, a higher gradient meaning that the connection is more important. The weights are reinitialized so as to make the pass of the batch independent from the initial weights and then set back to their value after the pass. Then, from the ranking of the connectivity by their gradient, the mask with the required level of sparsity is build by only keeping the top $k\%$ connections.

With the lottery hypothesis However, results of this technique degrade on bigger networks. A partial reason why this technique scales badly is given by Frankle and Carbin in Stabilizing the Lottery Ticket Hypothesis [21]. They remark that such techniques scale poorly as they want to perform the pruning right at the beginning before any training. In their paper, they show that performing pruning on a network after some epochs of training allows more stability in the subnetwork and a better scaling to bigger architectures.

The combination of the results of those two papers shows that it is possible to perform pruning at high level of sparsity on bigger architectures even at the beginning of the training, after some epochs.

2.4 Dense connections

Skip connections Traditional networks have the information flow from one layer to the next. Recent architectures introduce skip connections [5] [8] to allow both a better propagation of the gradient to the first layers and a better transmission of information (see figure 10). After every layer, the input space is transformed into a more and more non-linear space. Such transformations may loose important information that are present in the first layers. The skip connections allow the network to keep such information even into the later layers.

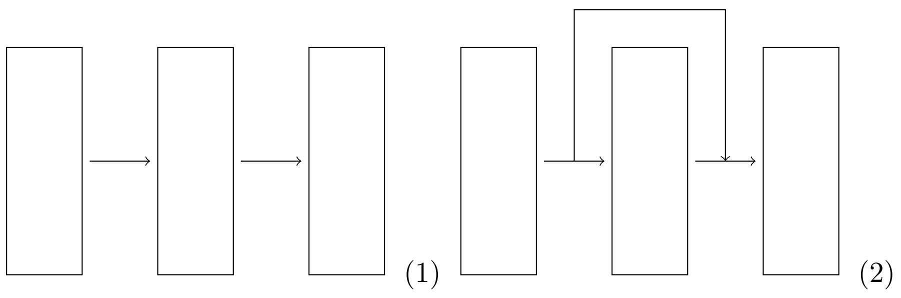


Figure 10: Traditional network setup (1); network with a skip connection (2).

The skip connection bring information of the previous layers to the next one. There are two ways used to mix this information:

- It is possible to concatenate it. For example, you have a network that has 10 outputs for the layers 1 and 2; by concatenation, the layer 3 will have 20 inputs, increasing the number of weights in the weight matrix to handle the number of inputs.
- The other possibility is to add the inputs together before feeding the input to the next layer. That way, the number of weights doesn't increase. However, this method may need to format the inputs from the layer 1 to adapt it to the shape of the output of layer 2 to perform the addition. Moreover, as the former inputs are transformed, it may impede the information flow of the network. For example, the n-1 layer output is 2 and

the n-2 layer output is 3, then the input received is 5; it is the same 5 that is received if the n-1 value is 3 and the n-2 value is 2 while it may mean something totally different.

DenseNet Some computer vision tasks are hard tasks that need very deep networks to perform well (some are over a 100 layers [23]). Due to the complexity of the tasks, lots of work are done to improve the architectures of the networks on those tasks.

DenseNet [9] is a network that connects every layer to the previous layers that have the same output shape (see figure 11). To merge the different outputs into a single input for the next layer, it uses the concatenation strategy. Each layer of this architecture output k filters (*growth rate* in the DenseNet paper), which means that the n^{th} layer receives $k * (n - 1)$ filters. To keep the number of operations from exploding, a bottleneck convolutional layer is used (see figure 12). This bottleneck layer performs a convolution using a $1 * 1$ kernel that output $4 * k$ filters for a $3 * 3$ kernel convolution (a standard convolution).

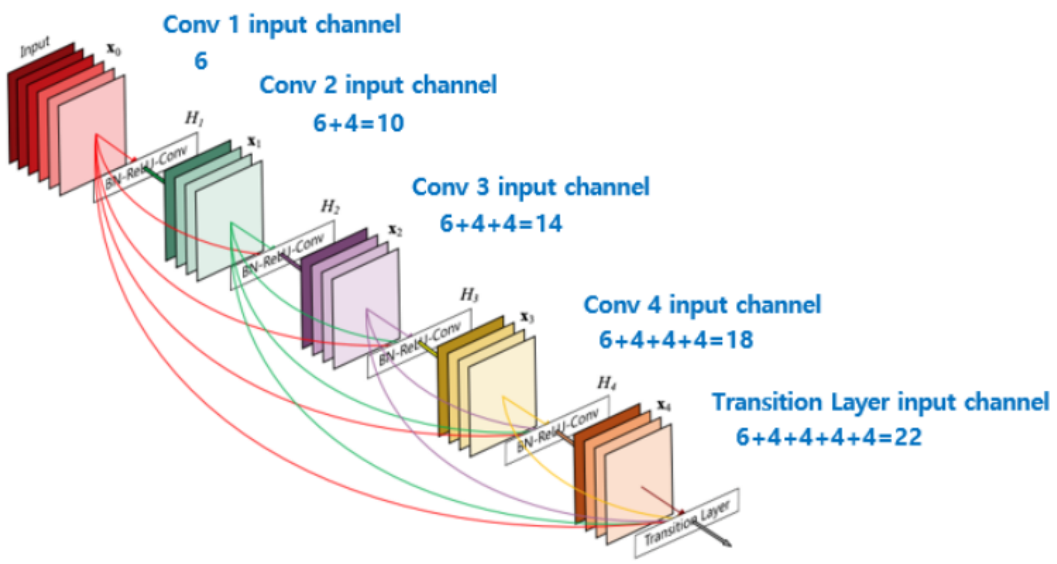


Figure 11: DenseNet architecture: every layer has access to every previous layer output. Every sheet represents a filter and every sheet color represents a convolution.

DenseNet article

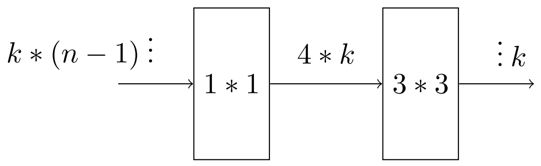


Figure 12: Bottleneck layer block. The k parameter represent the *growth rate* used in [9] and is the number of output filters of every layer.

While it seems that such a design will lead to architectures with exponential increase of parameters on big networks, it has been observed that in fact, it needs less parameters to reach the same level of accuracy than standard networks. Part of this result comes from the fact that there is no need for the network to learn redundant filters at each layer to keep interesting former information as they are available at every layer via the skip connections. Moreover, as there is a better flow of the gradient via those long skip connections, it is faster to train [9].

However, as it is, the skip connections are stopped at *transition layers* that performs down-sampling (reducing the size of the information) and do not connect to further layers. To overcome the transition layers, it would require the next layers to down-sample in the same way the skip connections coming from before the transition layers.

2.5 Network Architecture Search

Defining a network architecture is crucial as it greatly impacts its performance. On the other hand, finding an architecture that fits a task better than the state of the art is not a trivial matter. It takes lots of work for researchers and engineers to develop such architectures as the possible combinations between the different layers are immense. To answer this problem and remove the human variable from the process, a new field of research has emerged: network architecture search (NAS). It aims to automatically discover what is the best sequence of layer and their connections for a given task.

To solve this hard problem, different algorithms have been developed using different paradigm such as reinforcement learning [11] or genetic algorithm [19]. Solving such tasks takes hundreds or thousands of GPU days which makes the use of those algorithm nearly impossible. For those algorithms, the search is done in a discrete space as the presence of a layer and the presence of a connection between two layers are binary parameters.

Network architecture search example The representation of an architecture search algorithm can be drawn as such:

- A certain number of nodes are defined. Those nodes are the intermediary results of the network.
- They are connected to form an acyclic graph that represents the flow of information in the network.
- The connections between the nodes are the layer operations, taken from a set of predefined operations, that are performed over the network.

The operations used can be convolutions of different kernel size or width (number of filters), pooling, identity or a special operation called *zero* operation that symbolizes the absence of connection between the nodes. An example of this representation can be seen at figure 13.

The aim is then to correctly select the operations on each node connection. To do so, multiple techniques exists.

One of them would be to select an operation randomly for each connection of the network and to create multiple networks using the same process. Then, after training the networks, the best are selected and other networks are created by combining the operations of the best.

Some of the operations of the new networks are then randomly modified to be sure that some possibilities are not overlooked. This process is then repeated several time until no better networks are found (for instance figure 13). This example can give an idea on how to perform architecture search using genetic algorithm.

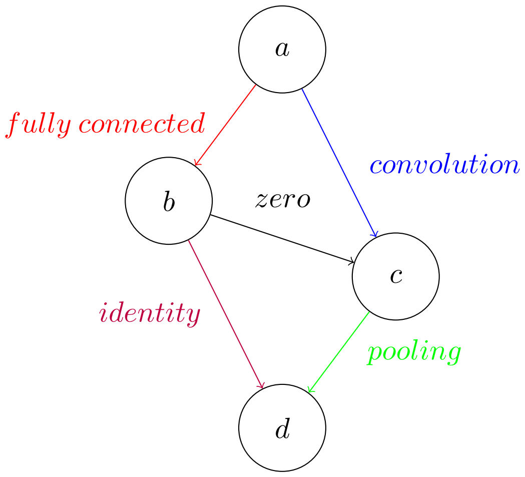


Figure 13: Example of outcome architecture for an architecture search

Differentiable architecture search More recently, another kind of algorithm has been proposed using the differentiable programming paradigm. This new algorithm allows the search to be made in a differentiable and continuous space.

The previous algorithms consider the presence of the operation as a discrete parameter, thus it is not possible to use the efficient gradient based methods that need a continuous space. In DARTS [17] (Differentiable ARchitecTure Search), they solve such problem by relaxing the discrete presence of the operation. The previous setup can be seen as if for each connection, the probability of the selected operation is 1 and the probability of the other operations is 0.

The relaxation allows those binary parameters to take value in $\mathbb{R}$. So as to be able to continue working with probabilities to select the best operation, the *Softmax* function is used to bring the $\mathbb{R}^n$ vector of relaxed parameters to a $[0, 1]^n$ vector of probabilities. This relaxation makes the search space of those parameters continuous and thus allowing the use of the gradient based methods.

The optimization then modifies the parameters such that the operation that gives the best accuracy has a higher value. At the end of the training, the operations with the higher values are selected for each connection.

This step makes possible the use of well known and high-performance optimisation algorithms that speed the process, keeping the search time to only several days. This algorithm represented in figure 14 is better described in the research paper DARTS [17]. So far, methods based on it has given similar or better results with less time than other methods.

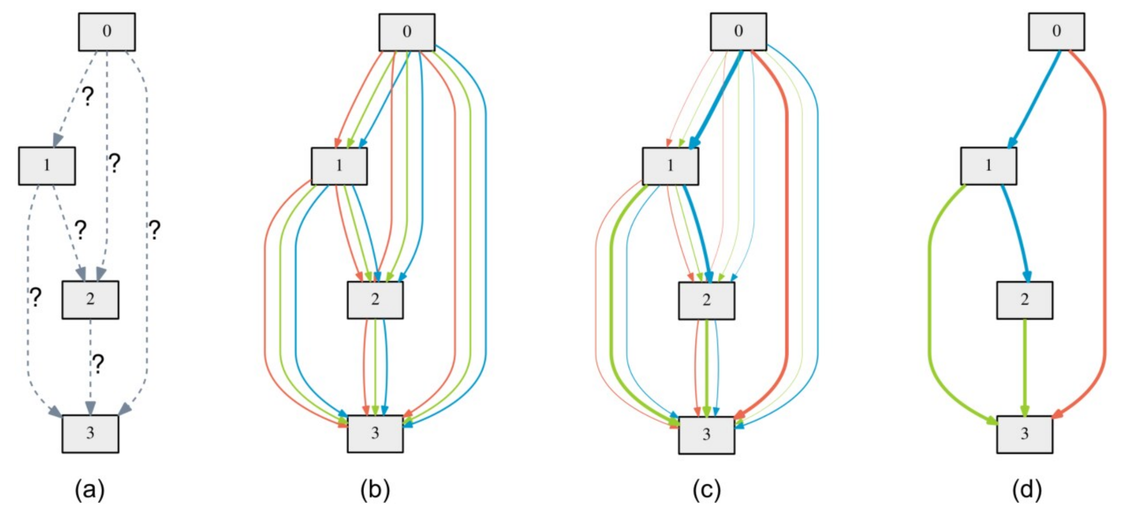


Figure 14: DARTS setup example
(a) Operations on the edges are initially unknown. (b) Continuous relaxation of the choice of the operation by placing the candidate operations on each edge. (c) Learning the parameters of each operation (d) Choosing the best operation based on the parameters.

Source: DARTS paper [17]

Limits to DARTS DARTS has allowed network architecture search to be performed using efficient techniques. However, contrary to the reinforcement or genetic version that perform the search on the whole network, DARTS performs the search for *cells*. Different kind of cells are usually learned for different roles in the network. For example, in image classification, two kind of cells are used: a convolutional cell for a combination of convolutional layers and a reduction cell for the pooling phase. Those cells are stacked to constitute the full network. This setup needs less parameters than searching on a full network and performs well while taking much less time, however it looses in potential as the search is only performed on stacked cells and not on a complete architecture. Moreover, to speed the search phase, the number of stacked cells is lower than the number of cells in the network trained. This implies that the function of the network during the search and the function of the network during the training are different. Even if it makes DARTS run faster than other algorithms, it also limits the search capacity of this algorithm. Moreover, it has been proved that every operation in DARTS doesn't have the same chances of being selected and that bias exists (see [26]). Researchers are now trying to bring the idea of a differentiable search space to the search level of the reinforcement and genetic methods while keeping its lower search time.

2.6 Contributions

In this internship, we address network architecture search using a different approach than the ones presented before. Instead of trying to find the optimal sequence of layers, we aim to find a high performance network (that can be sub-optimal) in less time than the other methods. Moreover, unlike DARTS, we aim to learn the architecture of the full network.

First, we define the different layers that will be used (for example, 2 convolutions of 16 filters, a pooling and a fully connected layer) as well as some structure in those layers (for example, 2 convolutions then the pooling with at the end the fully connected layer). With those layers, we create a fully dense network meaning that every layer has access to the previous layers' output. Then, using a pruning technique, we find the best connections between layers, searching for the best flow of information between them.

As the network is a dense network, the pruning will operate as the path and operation finder in this dense network representing the search space. This means that, if an operation is not necessary then, every connection to it will be pruned and on the contrary, if an operation is highly relevant, there will be a lot of connections preserved. That way, we search for the best association of layers and connections in the search space. However, the layer sequence found is highly dependant of the initial architecture of the dense super-network. This implies that we may not find the optimal architecture for the task, but only the best architecture for the task with respect to the dense super-network.

However, as we are dealing with a dense network with possibly millions of connections, training such a network requires a lot of time and computation power. Using pruning as early as possible can ease the process but as seen in section 2.3, pruning after some epochs remains superior to pruning without training and allows to use bigger models.

To deal with this amount of connections, it is possible to iteratively grow the network until it reaches the desired size. Using a Shallow Deep Network (see section 2.2.2) with a growing policy, it is possible to iteratively prune the network, making the number of connections to deal with a lot smaller than with the full super-network.

In this Projet de Fin d'Etude thesis, we propose another architecture search method based on a dense search space using pruning as the search agent along with a growing schedule to alleviate the computational cost. The aim is not to find the optimal architecture for the task but to find the best architecture defined in the search space in a smaller amount of time compared to other NAS methods.

We base our work on the results given by recent works in the field of pruning, dense network and growing network and combine them to perform something different from their original field: network architecture search.

3 Growing a network

We start with growing the network. This step is done first as it has a big impact on the training schedule. Indeed, as we begin with a smaller network and then add layers, the later layers will be less trained than the earlier ones. Considering the impact of growing on the training process, it is important to perform it first.

Combination of growing and Shallow Deep Network The idea of using internal classifiers actually fits well with a growing network. Indeed, it is possible to grow several layers with an internal classifier attached to it and to repeat the operation (see figure 15). Usually, in the growing strategies [14], the final layer was the same and so is trained with an increasing number of layers between the input and itself. This can lead to a learning problem as the output is learning first on one function (the previous layers) and then on another function (previous layers plus the grown layers). By using internal classifiers, it is possible to train the growing architecture and the internal classifiers as a whole without misleading the output decision.

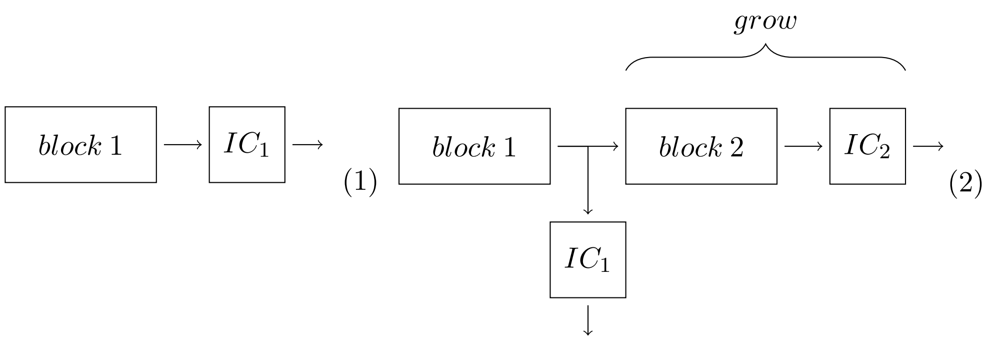


Figure 15: Growing with internal classifier strategy. A block is a set of layer.

This association and the training process is performed as follow: we start with a first block of layers that is randomly initialised and that is connected to an internal classifier that acts as the output. Then, we train this network for some epochs. When the criterion chose decides to grow the second block, the first internal classifier then becomes an output branch and the second block is linked to the last layer of the first bloc. The whole is then trained again and the operation is repeated until the final size of the network is reached. That way, we build blocks by blocks a SDN using a growing schedule IC by IC.

Training schedule The training schedule of a network is the set of hyper-parameters that influence the learning process such as the learning rate or the number of epochs. However, defining such schedule can be hard. There is no easy way to find the best hyper-parameters for the network (learning rate, number of epochs, ...) with respect to the task. Those parameters are usually defined after multiple trials.

In order to reduce the number of hyper-parameters, we set the epochs where the growing happens. That way, we reduce the number of trials needed.

Objective of the experiments To summarize, we aim to find a training schedule with its hyper-parameters (learning rate, number of epochs, learning rate adjustment). In order to define an objective to the performance of the schedule and not try everything to find the most optimal solution, we want the growing network to reach at least the same level of accuracy than the same network architecture without growing.

3.1 Experimental setup

As a starter code, we use the code given in the Shallow Deep Network paper [22].

Dataset In this repository, they define different networks that are used to test their method. These networks are defined for different tasks. As we want to be able to perform multiple experiments without spending too much time, we chose a task that is already present in the repository and that doesn't require to train for too many epochs: CIFAR 10. CIFAR 10 is an image classification task using 32*32 pixel images that belongs to ten classes (cat, plane, ...). It is a benchmark commonly used in SOTA researchs. This dataset is composed of two sets: a *learning set* and a *test set*. The learning set contains 60000 examples and is used during the training phase. The learning set is divided into a *training set* that is used to update the parameters of the network and the *validation set* that is used to evaluate the accuracy of the network during the training. The training set contains 50 000 examples and 10 000 for the validation set. The test set contains 10 000 examples and is used for evaluating the accuracy of the network after the training.

Baselines The ResNet [5] architecture is simple to use and has great results on image classification tasks. It also fits well with our experiments as it is composed of similar blocs stacked together, making them easy to grow. We decided to use this architecture already implemented in the code for our experiments.

However, the ResNet architecture in the code isn't implemented to allow a growing architecture. It was then necessary to create the growing procedure and modify the training process, making the network grows IC by IC.

In order to find the best schedule, it is necessary to launch a lot of experiments. However the size of the predefined networks (more than 50 layers) makes the training too long to be performed multiple time. To reduce the training time, we decided to use only 21-layer deep networks with ICs placed at 30%, 60% and 80% of the network. In the original SDN setup, the ICs are placed every 15% but, as we are dealing with smaller networks, we only kept some of them. It is possible to explore the impact of having more growing epochs and thus more IC on the schedule and accuracy. However this is not the main subject of this work so the number and the place of the ICs are kept constant.

The baselines used for comparison are a ResNet network and a ResNet network with internal classifiers (see figure 16). We then compare the accuracy of the ResNet grown network to the one of the baselines. All three networks have the same architecture (number and type of layer, connections). As the initialization of the networks and the order of the training examples are random, we perform several time the same experiment (5 times in our tests) and use the mean accuracy. To be sure that the mean accuracy is representative, we also compute the standard deviation of the experiments.

During the training, we select the best network based on its accuracy on the validation set and use this network to compute its accuracy on the test set. The results we display are from the test set.

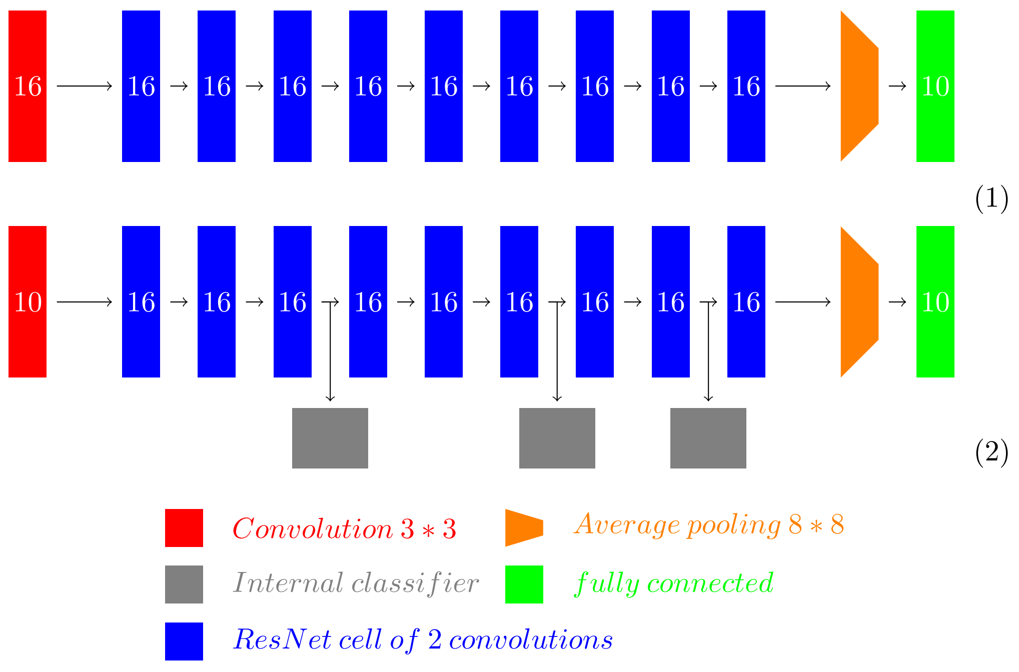


Figure 16: Baseline architectures: (1) ResNet without ICs, (2) ResNet with ICs; the number in the convolutions and ResNet cells is the number of output filters of the layer; the number in the fully connected layer is the size of the output. The layers between the ICs are grown together and are called a *block*.

In this thesis, the ResNet cells that are stacked to compose the network are called *cells*. The set of cells grown between each internal classifier is called a *block*.

3.2 Experiments

In order to have a starting point to search, we can reuse the schedule of the networks in the Shallow Deep Network paper. However, this schedule is defined for a network that is bigger compared to the networks that will be trained. The first step is then to adapt the schedule to the baselines that will serve as reference.

To perform a first test, we tried the schedule used for the bigger network on our baselines (see table 1). As this schedule was optimised for a similar architecture and the same task, our schedule should be alike.

In this thesis, *milestones* are the epochs where the learning rate is modified and *gammas* are the multiplicative coefficient that modify the learning rate. For example, if the learning rate is 1, the milestones are 20 and 40 and the gammas are 0.1 and 0.1, at epoch 20 the learning rate changes to 0.1 and at epoch 40 the learning rate becomes 0.01. The *accuracy* is expressed in percentage of examples that are correctly predicted by the network over the total number of examples in the validation set.

epochs	learning rate	milestones	gammas
100	0.1	35, 60, 85	0.1, 0.1, 0.1

Table 1: Schedule of the baseline.

After monitoring the evolution of the training, we observed that the accuracy increases smoothly and then converge at the end of the training. This implies that the schedule of the bigger network also works for the baselines. The accuracies of the baselines are shown in table 2.

baseline (base)	81.77
baseline with IC (base+ic)	67.02, 77.11, 81.47, 82.11

Table 2: Accuracy of the baselines on the test set; the 4 numbers of the baseline with IC are the accuracies of the internal classifiers.

Now that we know the accuracy we must reach, we can begin the experiments on the growing network. To reduce the number of hyper-parameters in the schedule to explore, we have set the growing epochs to the 25th, 50th and 75th epoch. As the growing network is of the same architecture as the base+ic, the schedule should be similar. However, as the last layer is grown at the 75th epoch, for this layer to have as much training as the base+ic, the total number of epochs must be at least 175.

As we are growing the network, to make sure that the network is trained enough, the number of epochs is increased to 200 epochs. Moreover, the milestones are delayed so that every layers get to learn in a similar way as the network without growing.

However, as the learning rate was used for a bigger network and that the growing network begin with only a very small network, the learning is unstable. This instability is due to the loss reaching a high value and thus making the weights unable to be updated correctly. To correct this problem, the initial learning rate is reduced to 0.01.

The experiments will test different number of epochs, milestones and gammas to find a fitting schedule.

epochs	milestones	gammas	accuracies (mean)				accuracies (std)
			IC1	IC2	IC3	final	
200	100, 133, 166	0.1, 0.1, 0.1	76.37	81.14	81.69	81.91	0.59, 0.34, 0.49, 0.57
200	100, 133, 166	0.1, 0.1, 0.01	75.37	80.86	81.71	81.79	1.18, 0.14, 0.79, 0.90
200	100, 133, 166	0.1, 0.01, 0.1	76.78	81.39	81.64	81.86	0.47, 0.40, 0.27, 0.21
200	100, 133, 166	0.1, 0.01, 0.01	76.46	81.42	81.60	81.97	0.47, 0.52, 0.44, 0.52
200	100, 133, 166	0.1, 0.01, 0.001	76.73	81.53	82.08	81.98	1.16, 0.71, 0.85, 0.88

Table 3: Accuracies on the test set of the growing network for different gammas.

Table 3 shows that there isn't much difference between the various gammas. The best schedule here is with the gamma at (0.1, 0.01, 0.01). Even if (0.1, 0.01, 0.001) holds a better result and even has its third IC reaching 82%, its standard deviation is much higher whereas it is not much better. However, it also shows that it is possible to reach higher accuracy with the network. To explore this further, we perform some more experiments with a higher number of epochs. This higher number of epoch also allows us to have more epochs between the milestones. The next experiments are then done with 250 epochs and the milestones at 120, 160, 180.

epochs	milestones	gammas	accuracies (mean)				accuracies (std)
			IC1	IC2	IC3	final	
250	120, 160, 180	0.1, 0.1, 0.1	76.66	81.18	81.58	81.63	0.24, 0.59, 0.26, 0.62
250	120, 160, 180	0.1, 0.1, 0.01	76.06	81.42	81.63	81.83	0.78, 0.45, 0.44, 0.47
250	120, 160, 180	0.1, 0.01, 0.1	76.54	81.60	82.00	82.15	1.37, 0.61, 0.54, 0.70
250	120, 160, 180	0.1, 0.01, 0.01	76.42	81.81	82.19	82.30	0.52, 0.60, 0.56, 0.58

Table 4: Accuracies of the networks on the test set for different gammas.

Table 4 shows the results of the experiments. It holds bigger differences than the previous table but also some better results. The best schedule here is with (0.1, 0.01, 0.01) gamma which is similar to the previous experiments. This schedule presents the best results with low standard deviations and even better results than the standard network with IC. The schedule that is selected is shown in table 5.

epochs	learning rate	milestones	gammas	growing epochs
250	0.01	120, 160, 180	0.1, 0.01, 0.01	25, 50, 75

Table 5: Selected schedule.

4 Pruning connections

The second element of our method is to perform the pruning over the network. The pruning is used as the search agent that selects the best connections between the layers in our method. Before working on the whole search space, that is on the densely connected super-network, the pruning is performed on the growing network. As it is a smaller search space, it is faster to perform the experiments on it and to solve the integration problems when adding the pruning algorithm.

4.1 Method

Aim of the pruning With this part, we aim to experiment on pruning strategies as well as study the evolution of the accuracy with respect to how much we prune the network (the sparsity level). The pruning method that will then be used will be able to reach the lowest level of kept connections with the highest mean accuracy while also being stable with a low standard deviation. As the network we use is a lot smaller than networks presented in the pruning papers, there are less redundant connections between the layers. This means that the accuracy drops faster as the sparsity increases.

Pruning strategies In our final method, due to the dense connections, the final network without pruning would have millions or more connections. While pruning such number of connections isn't hard, training such a network even for some epochs requires a lot of computation. To overcome this, we use a growing network and train it as it grows deeper. The objective is then to prune the network so as to keep the number of connections from reaching such a high number. To do so, we perform the pruning while still growing the network. It is then necessary to adapt the pruning methods to the growing architecture. Considering this, the pruning is performed once the layers are grown and trained on few epochs as explained in the section 2.3.1.

The pruning can be done in two ways:

- The *one shot* strategy prunes the block only once by a certain ratio (see figure 17 (1)). After some training epochs, the first block is pruned to the desired sparsity. The network is then trained again and a second block is added. The two blocks are trained for some epochs. The second block is pruned while the first block isn't. The cycle is then repeated until all the network is pruned.
- The *iterative* strategy prunes the block little by little until it reaches the desired sparsity (see figure 17 (2)). After some training epochs, the first bloc is pruned less than the final sparsity. The network is then trained again and a second block is added. The two blocks are trained for some epochs. The first and the second blocks are then pruned. The connections that were previously pruned remain pruned and the new pruned connections are taken from the non pruned connections. If a block reachs the desired level of sparsity, no more pruning is performed on the block and the sparsity can't be lower than the desired level. The cycle is then repeated until all the network is pruned.

This makes the pruning schedule linked to the growing schedule as the pruning of a layer happens before the next layers are grown.

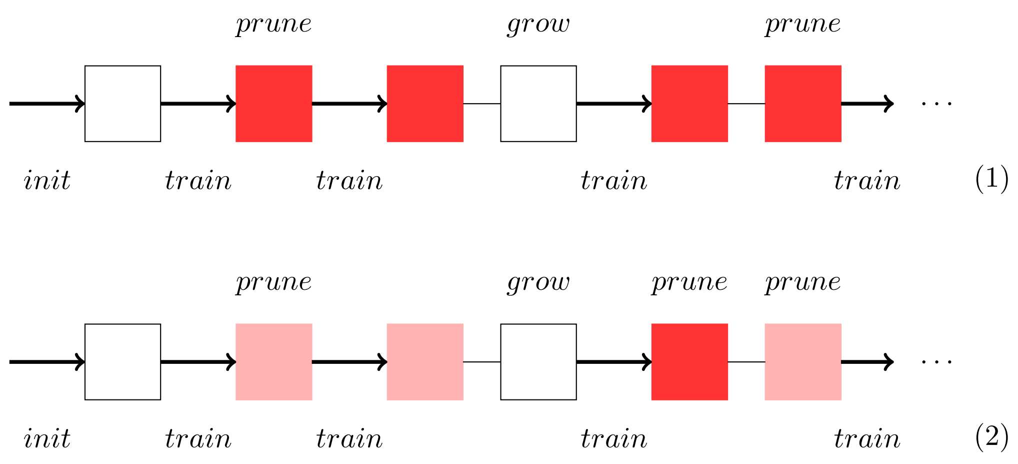


Figure 17: One shot (1) and iterative pruning (2); the redder the sparser.

Criterion The pruning must be performed during the training. However, most of the criteria used in pruning do it at the end of the training or need to train multiple times. Those criteria can't be used as the search space is very large, making them computationally expensive. The SNIP criterion (see section 2.3.2) however, possesses the qualities we are looking for. It is possible to use it since the beginning of the training, it doesn't need to perform multiple training sessions of the same network and it is simple to use as it only needs to compute the first order derivative (gradient) of the connections.

4.2 Experiments

A growing schedule has been found via the previous experiment (section 3). To this schedule, we integrate the pruning schedule using the properties of SNIP and performing the pruning after some training as it is advised in [21].

Pruning at different levels First, pruning with different level of sparsity is to be performed on the growing network to create a reference to compare the performance of the different methods used. For this experiment, a one shot pruning strategy is used as it is the one presented in the SNIP paper. The schedule used can be seen in table 6. Each layer is trained during 10 epochs before being pruned and the ratios of kept connections are 10, 20, 30, 40, 50, 60, 70, 80, 90.

epochs	learning rate	milestones	gammas	growing	pruning
250	0.01	120, 160, 180	0.1, 0.01, 0.01	25, 50, 75	10, 35, 60, 85

Table 6: Schedule of the growing pruned network.

From the figure 18 it can be seen that the accuracy rapidly decreases once it reaches 40% of kept connections and that the later classifier then drops behind the first classifiers in terms of accuracy. As we are using a much smaller network than the one used in pruning papers,

keep ratio	accuracy (mean)				accuracy (std)
	IC1	IC2	IC3	final	
100%	76.42	81.81	82.19	82.30	0.52, 0.60, 0.56, 0.58
90%	76.28	81.11	81.22	81.47	1.46, 0.54, 0.40, 0.44
80%	73.75	80.57	81.13	81.35	0.97, 0.51, 0.90, 0.54
70%	73.82	80.33	80.91	81.00	3.40, 0.59, 0.73, 0.97
60%	74.43	80.02	80.83	81.07	1.04, 0.48, 0.43, 0.30
50%	73.36	78.93	79.08	79.11	0.35, 0.50, 1.10, 1.06
40%	72.83	77.58	76.93	76.34	0.85, 0.70, 1.10, 0.83
30%	71.34	72.87	70.94	72.49	1.05, 0.79, 0.81, 0.92
20%	68.39	63.57	65.60	63.49	1.22, 5.19, 2.96, 3.12
10%	56.27	54.19	49.51	41.99	1.85, 6.44, 9.37, 8.85

Table 7: Accuracy and standard deviation of the growing network with respect to the ratio of kept connections.

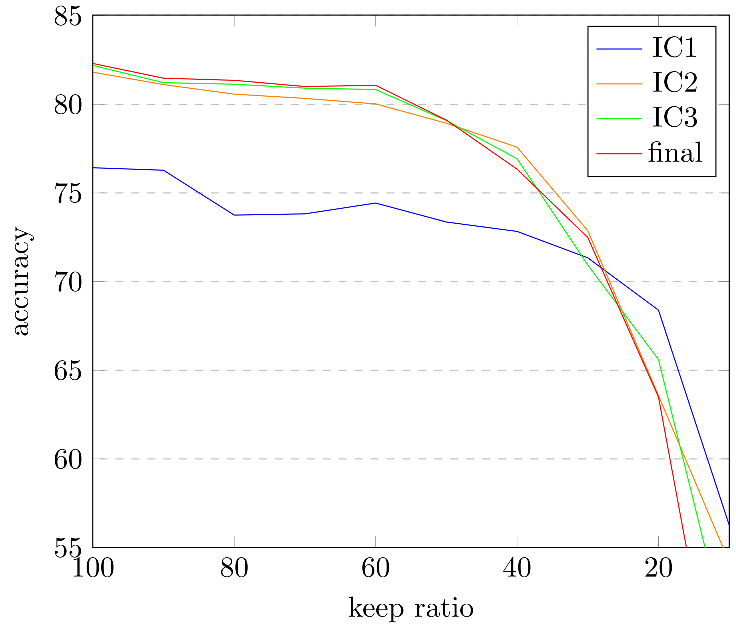


Figure 18: Evolution of the accuracy with respect to the keep ratio.

there are less redundant connections and thus, it is harder to keep the same accuracy when reaching smaller keep ratio. Table 7 shows that the more the network is pruned, the more the standard deviation increases, especially for very low level of kept connections. This indicates that the network looses its stability when reaching those levels.

Pruning with various batch sizes The SNIP criterion uses a batch of data to find the important connections. It is interesting to see the influence of the size of the batch on the accuracy as using more data should be able to give better gradients. To be able to see if this influence is beneficial even when the network start to have accuracy drop, it is performed with a keep ratio of 50%. The original batch size is 128 examples.

batch size	accuracy (mean)				accuracy (std)
	IC1	IC2	IC3	final	
128	73.36	78.93	79.08	79.11	0.35, 0.50, 1.11, 1.06
256	69.87	79.04	79.08	79.81	2.16, 0.56, 0.73, 0.54
512	72.25	78.17	78.58	78.33	1.96, 0.79, 0.76, 0.60
640	73.94	79.09	79.23	78.90	1.46, 0.78, 0.73, 0.49
768	74.85	79.56	79.85	79.50	0.39, 0.35, 0.56, 1.04
896	74.65	79.32	79.36	79.52	1.01, 0.37, 0.40, 0.53

Table 8: Accuracy and standard deviation on the test set with respect to the batch size used for the criterion.

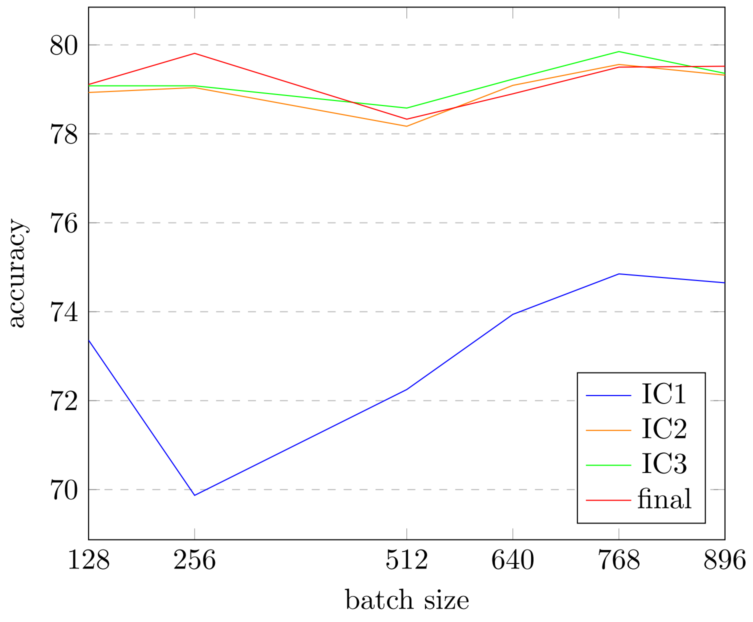


Figure 19: Evolution of the accuracy at 50% of the network pruned with different batch sizes.

From looking at figure 19 it seems that the influence of the batch size is important for the first internal classifier. But this one holds a high standard deviation that can explain the fluctuation in accuracy (most of the time ≥ 1 see table 8). Moreover, the other IC stay within a 2% range with a limited standard deviation (≤ 1.00). As the accuracy doesn't increase with the batch size, it can be said that its influence is limited on the accuracy. For the other experiments, the original batch size of 128 is then used.

Pruning without reinitialization Before performing the search of important connections, SNIP method reinitializes the weights of the network. However, referring to the lottery ticket hypothesis[15], this is detrimental to the learning afterward. In order to check if that hypothesis also applies for the SNIP criterion, we rerun the first experiment with different pruning levels without reinitialization.

keep ratio	mean accuracy				standard deviation
	IC1	IC2	IC3	final	
90%	76.24,	81.00,	81.47,	81.82	0.90, 0.48, 0.33, 0.23
	-0.04	-0.11	+0.25	+0.35	
80%	75.57,	80.67,	81.25,	81.53	0.54, 0.62, 0.81, 0.67
	+1.82	+0.1	+0.12	+0.18	
70%	74.68,	80.55,	81.00,	81.36	1.04, 0.69, 0.64, 0.65
	+0.86	+0.22	+0.09	+0.36	
60%	73.86,	79.86,	80.75,	80.85	0.45, 0.31, 0.62, 0.74
	-0.57	-0.16	-0.08	-0.22	
50%	72.36,	79.23,	80.34,	80.54	1.59, 0.88, 0.51, 0.44
	-1.00	+0.3	+1.26	+1.43	
40%	71.76,	78.29,	79.61,	79.57	1.53, 0.25, 0.49, 0.47
	-1.07	+0.71	+2.68	+3.23	
30%	69.50,	77.37,	78.40,	78.48	0.93, 0.71, 0.48, 0.43
	-1.84	+4.50	+7.46	+5.99	
20%	66.11,	74.58,	76.22,	76.39	2.36, 1.06, 0.39, 0.42
	-2.28	+11.01	+10.62	+12.90	
10%	56.04,	67.21,	69.70,	68.55	2.38, 1.41, 0.91, 0.97
	-0.23	+13.02	+20.19	+26.56	

Table 9: Accuracy at different keep ratio for pruning without reinitialization. Comparison to pruning with reinitialization.

The results of this experiment in table 9 show that without reinitialisation, the pruned network can reach a higher accuracy and is more stable. The standard deviation is lower and kept ≤ 1 even at highly pruned levels. This difference in accuracy and deviation is more and more important as the number of kept connections drop. As this experiment demonstrate the importance of non reinitializing the parameters when using highly pruned network, we now only perform pruning this way.

Iterative pruning Up until now, the pruning has been performed in a One-Shot way (see figure 17). In *the Lottery Ticket Hypothesis* [15], Frankle and Carbin have found the iterative pruning to work better than in one-shot. However, they are using another pruning criterion that removes the top k% of the weights with the smallest magnitude and are performing the pruning at the end of each training session before resetting the weights to their original value. As we want to keep the computations needed as low as possible, it is not possible to perform multiple training sessions. However, we can still perform the pruning iteratively during one training session. To do so, we are using the method described in figure 17 and define more pruning epochs. For the experiments, the pruning epochs are $10, 35, 60, 85, 95, 105, 115$.

Then, to be able to compare the results with the previous experiment, we perform the pruning with different keep ratio and make the blocks reach their final sparsity in three pruning steps.

keep ratio	mean accuracy				standard deviation
	IC1	IC2	IC3	final	
80%	74.09	80.33	81.34	81.78	1.44, 1.16, 0.58, 0.40
	-1.48	-0.34	+0.09	+0.25	
70%	73.94	80.71	81.40	81.70	0.78, 0.42, 0.46, 0.52
	-0.74	+0.16	+0.4	+0.34	
60%	71.27	79.56	81.21	81.61	1.05, 0.80, 0.45, 0.57
	-2.59	-0.30	+0.46	+0.76	
50%	67.90	78.37	80.51	81.30	1.13, 0.84, 0.36, 0.28
	-4.46	-0.86	+0.17	+0.76	
40%	63.29	76.00	79.26	80.54	0.98, 2.29, 0.33, 0.33
	-8.47	-2.29	-0.35	+0.97	
30%	58.36	74.78	78.17	79.17	4.69, 1.01, 1.12, 2.09
	-11.14	-2.59	+0.23	+0.69	
20%	50.13	68.74	75.64	78.58	3.07, 2.14, 0.49, 0.25
	-15.98	-5.84	+0.58	+2.19	

Table 10: Accuracy at different keep ratio for the iterative strategy. Comparison with the one shot strategy.

Table 10 shows that the iterative pruning increases the accuracy of the later classifiers to the detriment of the earlier classifiers. However, this increase is not significant most of the time and the accuracy drop of the first classifiers is huge ($\approx 10\%$) and brings instability. Even if the benefits of the iterative pruning are not enough to be considered under this setup, it is possible that it can still be interesting when using dense connections.

pruning epochs	impact of batch size	reinitialization	strategy
10, 35, 60, 85	none	without	one-shot

Table 11: Final pruning method designed from the experiments.

The final setup for the pruning method is detailed in table 11.

5 Adding dense connections

The third element of our method is to add the dense connections over the network. Those dense connections build the super-network that is the final search space of the method we aim to experiment on.

The aim of the experiments is to validate the integration of the dense connections. Moreover, in the previous experiment, the choice of the strategy of the pruning was hard. Indeed, the iterative strategy has a little higher accuracy on the last classifiers but harm the previous ones. It is interesting to perform again this experiment with the addition of dense connections to see how those results can change.

Integration of dense connections Up until now, we have been using the ResNet architecture for its good results and properties on image classification tasks. However, the idea of a DenseNet is to connect each layer to every previous one. This would be difficult to do starting from the way ResNet is implemented as it works with residual cells (see figure 20) that are agnostic of the other cells and just send their results to the higher network architecture. Using a real DenseNet would require to implement it from scratch and as the architecture would be different, we would lose the ability to compare to a similar baseline for every experiment. It is then necessary to adjust the ResNet architecture to the DenseNet connections.

The ResNet architecture is based on residual cells that are composed of two CBR (Convolution - Batch Norm - ReLU) layers with a skip connection between the inputs and the output of the second layer. If it is not possible to correctly implement dense connections between every layers without having to start from scratch, it is possible to make those dense connections between the different cells of the ResNet architecture as shown in figure 20.

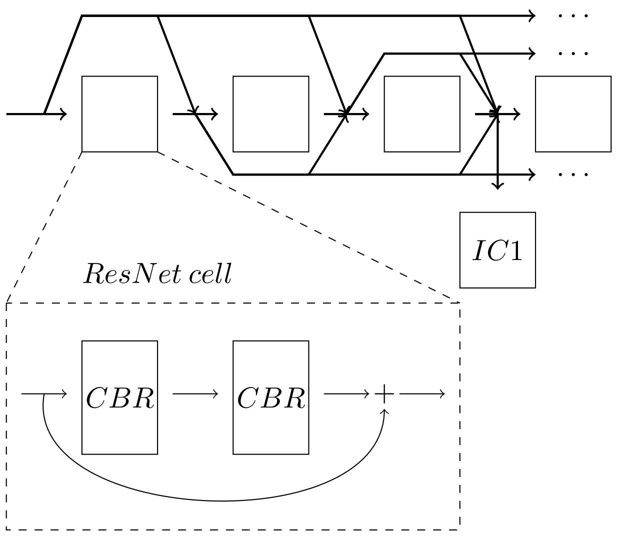


Figure 20: Dense ResNet architecture.

5.1 Experiment

In the previous experiments the different elements of the pruning method have been defined. However, to be able to compare the networks of this experiments to the previous one, the kept ratio can't be used. Indeed, as we have added more connections, the architecture of the networks are not comparable. To do so, a similar base of all network must be used: the number of parameters of the network. The number of parameters represents the number of connections in the network and so, is linked to its complexity.

The network is divided into 4 blocks that end with an IC. The number of parameters of each block is:

- Block 0: 25 008,
- Block 1: 47 872,
- Block 2: 45 568,
- Block 3: 50 944,

for a total of 169 392 parameters.

Comparison of one shot and iterative pruning It has been found in the previous experiments that using an iterative strategy can improve the accuracy of the last classifier at the detriment of the other classifiers. It is interesting to determine if the addition of the dense connections has an impact on this observation.

parameters	keep ratio	mean accuracy				standard deviation
		IC1	IC2	IC3	final	
135 512	80%	76.75	83.40	84.14	85.12	0.78, 0.74, 0.43, 0.29
118 572	70%	74.87	82.54	83.56	84.50	0.80, 0.41, 0.71, 0.30
101 633	60%	73.90	81.85	83.28	84.39	0.51, 0.29, 0.20, 0.20
84 696	50%	72.22	81.58	83.06	84.40	1.09, 0.77, 0.41, 0.26
67 755	40%	67.84	81.06	82.64	83.77	1.29, 0.40, 0.28, 0.26
50 816	30%	66.23	79.29	81.29	82.56	2.72, 0.82, 0.53, 0.59
33 876	20%	46.00	76.46	80.14	81.54	9.63, 0.87, 0.38, 0.33
16 937	10%	41.47	69.48	75.04	77.46	2.87, 2.03, 0.33, 0.60

Table 12: Accuracy and standard deviation of the growing dense network with one shot pruning.

Table 13 shows that, as in the previous pruning experiment (see table 10) the iterative pruning allows a higher accuracy for the later classifiers while being detrimental to the earlier ones when compared to the one shot strategy (see table 12). However, it can be seen that in this case, the difference in accuracy consistently increases and nearly reach 3% while keeping a low standard deviation. As we are more interested in the accuracy of the final classifier, using the iterative pruning when reaching high level of sparsity is efficient.

# params	ratio	mean accuracy				standard deviation
		IC1	IC2	IC3	final	
118 574	70%	74.49	82.67	83.62	84.72	0.53, 0.60, 0.78, 0.84
		-0.38	+0.13	+0.06	+0.22	
101 635	60%	72.30	82.70	83.98	85.10	1.65, 0.48, 0.46, 0.38
		-1.60	+0.85	+0.7	+0.71	
84 696	50%	70.29	81.62	83.55	84.82	1.40, 0.53, 0.71, 0.60
		-1.93	+0.04	+0.49	+0.42	
67 756	40%	65.68	80.89	82.51	83.91	2.14, 0.46, 0.29, 0.32
		-2.16	+0.17	-0.13	+0.14	
50 817	30%	63.00	78.77	82.51	84.19	1.60, 0.34, 0.25, 0.19
		-3.23	-0.52	+1.22	+1.63	
33 878	20%	52.88	74.83	80.70	83.13	1.88, 1.27, 0.80, 0.45
		+6.88	-1.63	+0.56	+1.59	
16 939	10%	34.75	63.33	75.38	80.34	5.83, 5.17, 1.06, 0.36
		-6.72	-6.15	+0.34	+2.88	

Table 13: Accuracy at different keep ratio of the dense network using iterative pruning. Comparison with the one shot strategy.

Decreasing keep ratio It can be seen that the first blocks accuracy drops fast when reaching a high level of sparsity. This can be because every block has a different number of parameters, the earlier classifiers reach a low number of parameters during the pruning, making them more unstable and with a lower accuracy. In addition, as the quality of the earlier classifiers drops, it is possible that it impede the learning of the later classifiers. To verify this hypothesis, we proceed by defining different pruning limits for every blocks: the deeper, the lower the pruning limit is.

To be able to compare the previous strategy that uses a constant keep ratio with this one, we trained networks that have the same global sparsity. The results of those networks are shown in table 15. The global sparsities are shown in table 14.

decreasing keep ratio	global sparsity
$0.8, 0.7, 0.6, 0.5$	0.65
$0.7, 0.6, 0.5, 0.4$	0.53
$0.6, 0.5, 0.4, 0.3$	0.43
$0.5, 0.4, 0.3, 0.2$	0.33
$0.4, 0.3, 0.2, 0.1$	0.23

Table 14: Table of the global sparsity of the network for the corresponding decreasing ratio.

In table 16, we compare the networks with constant pruning ratio and the networks with decreasing pruning ratio. It can be seen that using a decreasing ratio strategy increase the accuracy of every classifier. It is really efficient for the earlier classifier and is slightly beneficial

parameters	keep ratio	mean accuracy				standard deviation
		IC1	IC2	IC3	final	
106 328	65%	74.57	82.68	83.66	84.67	0.77, 0.44, 0.36, 0.33
89 777	53%	72.83	82.07	83.30	84.23	0.70, 0.39, 0.49, 0.41
72 836	43%	69.44	81.30	82.84	83.99	1.44, 0.29, 0.68, 0.45
55 897	33%	65.95	80.23	82.09	83.09	2.77, 0.78, 0.46, 0.53
38 958	23%	59.00	77.32	79.95	81.38	10.50, 1.10, 0.57, 0.18

Table 15: Accuracy and standard deviation of the growing dense network with one shot pruning.

# params	keep ratio	accuracy (mean)				accuracy (std)
		IC1	IC2	IC3	final	
106 328	80, 70, 60, 50	76.25	83.24	83.86	84.71	0.50, 0.44, 0.48, 0.47
		+1.68	+0.56	-0.3	+0.04	
89 389	70, 60, 50, 40	75.49	82.66	83.65	84.51	0.51, 0.41, 0.25, 0.34
		+2.66	+0.59	+0.35	+0.28	
72 450	60, 50, 40, 30	73.43	81.97	83.16	84.06	0.83, 0.18, 0.20, 0.37
		+3.99	+0.67	+0.32	+0.07	
55 510	50, 40, 30, 20	70.72	81.63	82.74	83.80	2.62, 0.68, 0.72, 0.56
		+4.77	+1.4	+0.65	+0.71	
38 571	40, 30, 20, 10	68.98	80.54	81.61	82.47	1.69, 0.57, 0.54, 0.44
		+9.98	+3.22	+1.66	+1.09	

Table 16: Accuracy at different keep ratio for the blocks of the dense growing network with one-shot pruning. Comparison with one shot pruning from table 15.

for the later one if we compare the number of parameters of the whole network, but highly beneficial if we look at the number of parameters of the concerned classifiers. This means that it is possible to prune more the later connections rather than the earlier ones.

We also can compare the one-shot pruning with the iterative pruning for the decreasing ratio to see its influence.

From table 17, the iterative pruning increases the later classifier's accuracy to the detriment to the earlier ones. However, the decrease in accuracy is less important than for the other experiments (-2.54% compared to $-3 \sim -6\%$). The iterative strategy is then better for this setup.

5.2 Comparison to baselines

In the previous experiments, we have defined the setup of our method. It is then interesting to compare the results of the architecture learned by our method to the results of other architectures.

To compare to them, it is possible to use the number of parameters of the network or the number of computations needed to perform the decision at the last classifier as common

# params	keep ratio	accuracy (mean)				accuracy (std)
		IC1	IC2	IC3	final	
106 328	80, 70, 60, 50	76.08	82.68	83.76	84.53	0.26, 0.28, 0.50, 0.44
		-0.17	-0.56	+0.40	-0.18	
89 389	70, 60, 50, 40	75.12	82.29	83.27	84.61	0.37, 0.61, 0.64, 0.31
		-0.37	-0.37	-0.38	+0.1	
72 450	60, 50, 40, 30	73.18	82.16	83.49	84.52	0.50, 0.74, 0.70, 0.55
		-0.25	+0.19	+0.33	+0.46	
55 510	50, 40, 30, 20	70.28	80.82	82.69	84.12	2.25, 0.72, 0.96, 0.70
		-0.44	-0.81	-0.05	+0.32	
38 571	40, 30, 20, 10	66.44	79.20	82.07	83.54	1.69, 1.17, 0.39, 0.16
		-2.54	-1.34	+0.46	+1.07	

Table 17: Accuracy at different keep ratio for the blocks of the dense growing network with iterative pruning. Comparison with decreasing ratio for one shot pruning.

denominator.

ResNet architecture First, we compare it with ResNet architectures similar to the baseline of the first experiments in figure 21. Those ResNets have different architectures (more or less ResNet cells) so as to compare with various number of parameters. What interests us the most is the performance of the final classifer, so we will focus on it in the following graphs. Besides, this makes the graph simpler and improves readability.

By comparing the results of our method with the ResNet architecture, our method has a better accuracy in the final classifier by approximatly 2%. This difference increases when reaching a lower number of parameters. Moreover, our method is less affected by the drop in accuracy when reaching a low number of parameters.

Dense architecture To be able to determine if our method really discovers architectures that are better than the one we can predefine, we also must compare to a dense architecture. Indeed, our method is based on such dense architecture and comparing to it is important. To have a more general view of the differences, we compare the two architectures using the number of parameters and the number of computations needed to reach the final classifier.

From figures 22 and 23, it can be seen that our method, with similar number of parameters and similar number of computations perform better (by approximatly 2%).

It can be said that as we outperform both ResNet and a DenseNet version by 2% that both baselines share the same level of accuracy. However, the different number of parameters of our method are from the same network but with various keep ratio. As the ResNet architecture holds less parameters than the dense one, we had to prune more to reach this number of parameters, potentially harming badly the accuracy. Perhaps, if the original network was smaller and that less pruning was done, the difference would have been bigger.

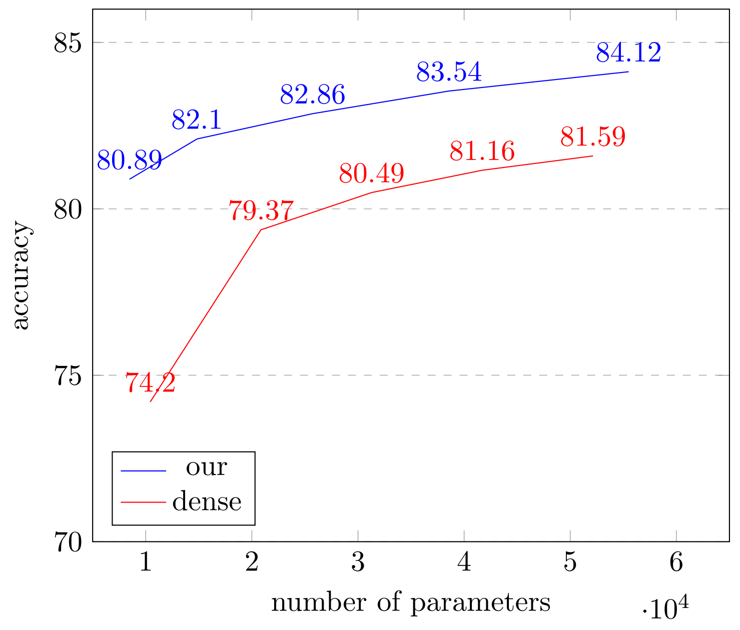


Figure 21: Evolution of the accuracy of the final output for a ResNet architecture and our method w.r.t. the number of parameters.

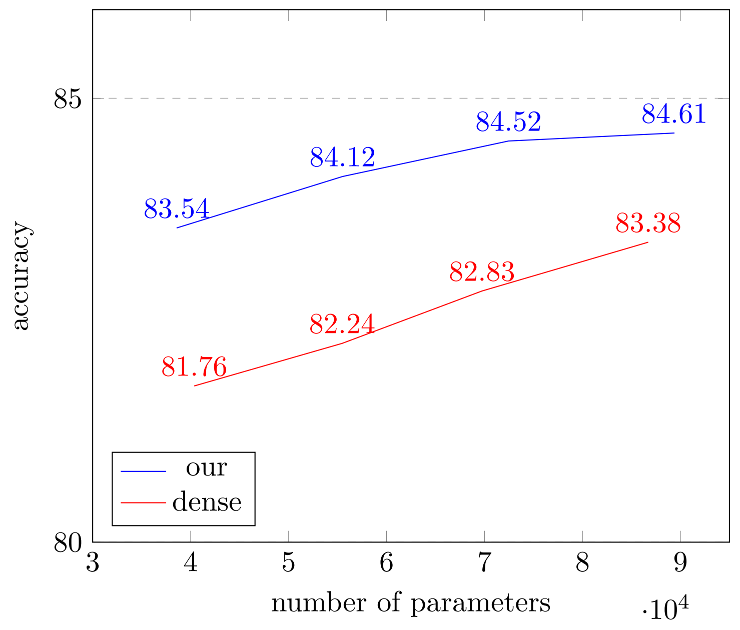


Figure 22: Evolution of the accuracy of the final output for a Dense architecture and our method w.r.t. the number of parameters.

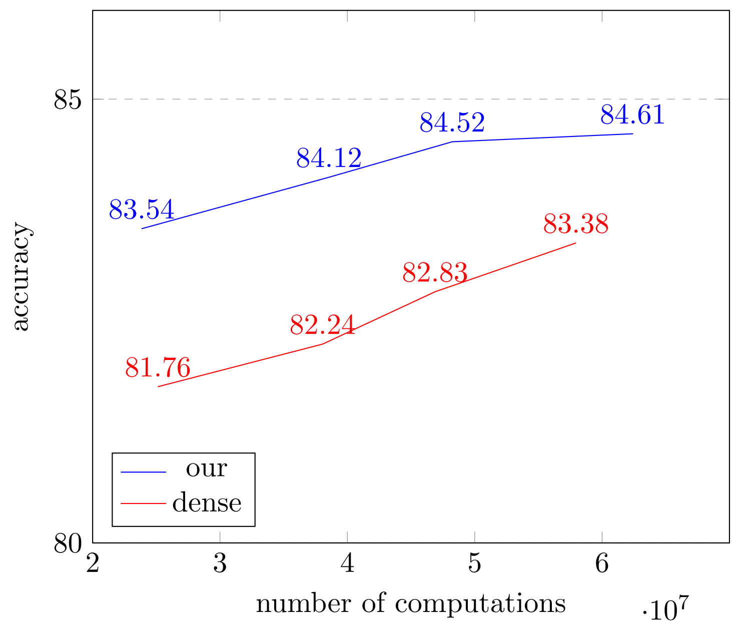


Figure 23: Evolution of the accuracy of the final output for a Dense architecture and our method w.r.t. the number of computations.

6 Conclusion

In this internship, I worked in the Linkmedia research team from March to August. This internship was the opportunity to learn more about the research field and work on a subject corresponding to my interests.

6.1 Summary of contributions

During this thesis, we introduce a new method to automatically discover a network architecture for a given task. It uses an overparameterized dense super-network as the search space. It then defines the optimal succession of layers and the optimal connections by using a pruning criterion. We also integrate a growing strategy to reduce the overhead caused by the huge number of parameters from the dense network. This method is designed to be efficient and take less time than other NAS methods. However, we do not search for the optimal architecture for the task but for the best architecture defined by the search space.

We show that the architectures discovered by this method holds better accuracy than the ResNet and DenseNet baselines for the same number of parameters. As the concepts this method is based on are generic and can be applied to various architectures (such as simple fully-connected or recurrent neural network) and not only convolutional network, our method can also be extended to these architectures.

6.2 Future work

Our method presents promising results but there are ways to potentially improve it.

- In the experiments, the networks used were rather small. Investigating our technique on bigger architectures would validate the results obtained so far.
- It would also be interesting to compare the results of our method with other network architecture search methods both in term of performance (accuracy/number of parameters or computations) and of time needed to find the architecture.
- The current method doesn't search for the optimal number of layers. A direction would be to grow the network similarly to what is done in [14] by using a criterion for growing.
- Currently, our method partially searches for the width of the different layers as it would prune all connections to a filter or a neuron if this one is unnecessary. It would be interesting to investigate the combination of structured and unstructured pruning to search first for the number of filters and then the connections for example.
- In the method presented, we do not investigate the search of down sampling in the network. This can be investigated with the growing strategy with growing a layer performing down sampling and seeing by pruning if the operation is interesting.

7 Appendix

7.1 Gantt diagram of the internship

The *familiarization* is the time needed to learn the different tools used at INRIA as well as the structure of the code of the Shallow Deep Network repository.

The *experiments* are the periods where the tests on the different parts of the method were performed

The *writing* is the period where I wrote this thesis.

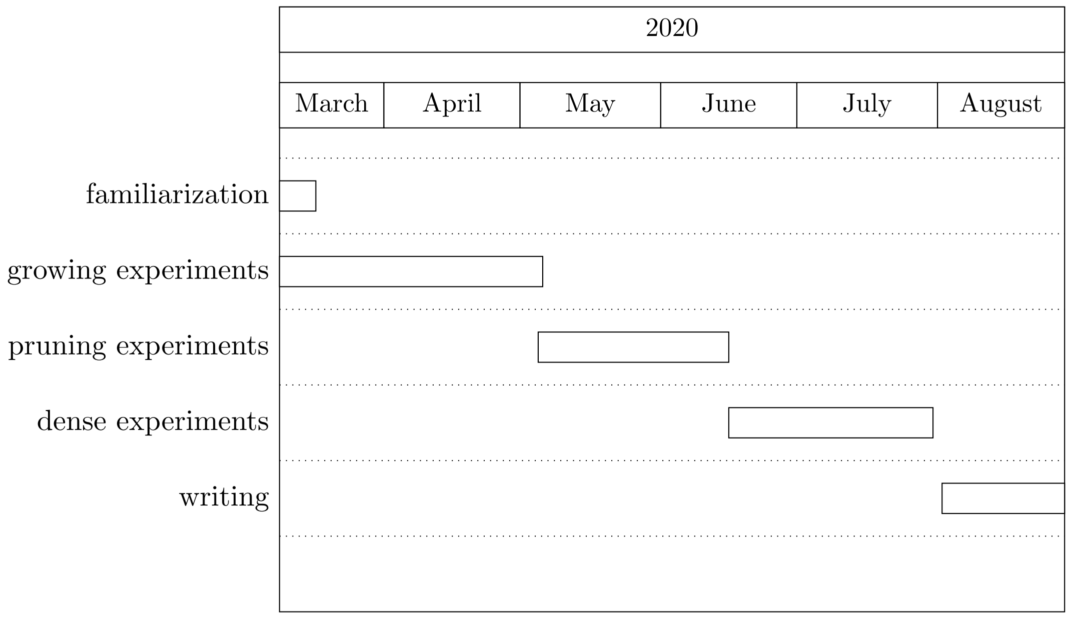


Figure 24: Gantt diagram of this internship

7.2 Growing an architecture

This figure 25 shows the way layers are grown in most of existing methods such as [14]. The layers are grown between the previous existing layers and the output layer.

7.3 Pruning

The figure 26 shows the different kind of elements that defines a pruning method. The following figures 27, 28, 29, 30 and 31 represent those different elements so as to better explain them.

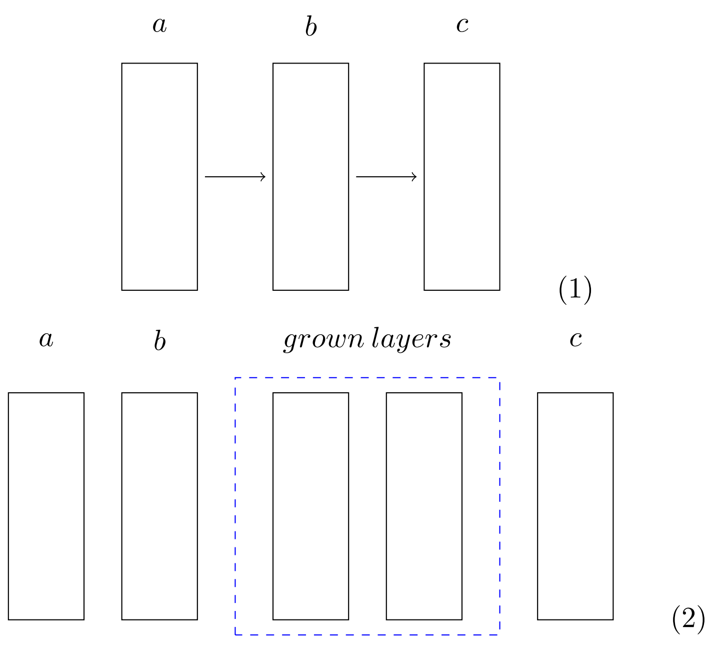


Figure 25: Growing new layers in a network.

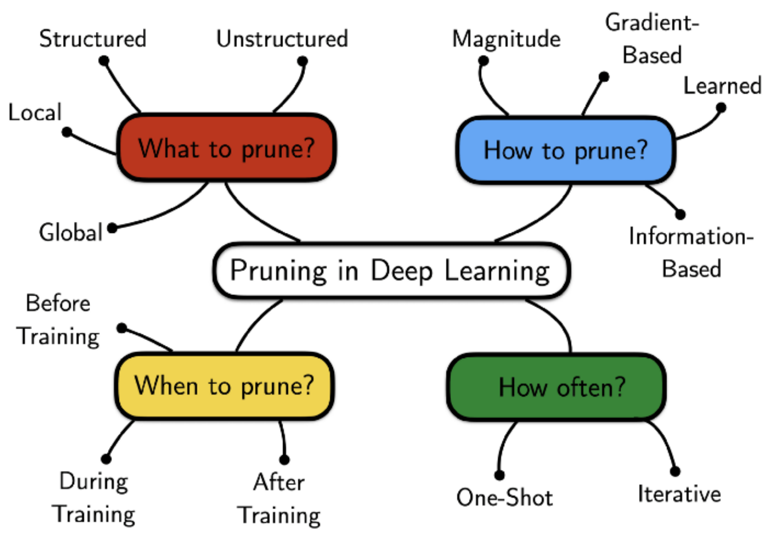


Figure 26: The different elements of pruning (from Robert Lange in a toward-data-science post [24]).

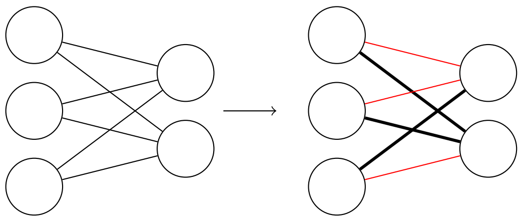


Figure 27: Unstructured pruning; red connections are pruned.

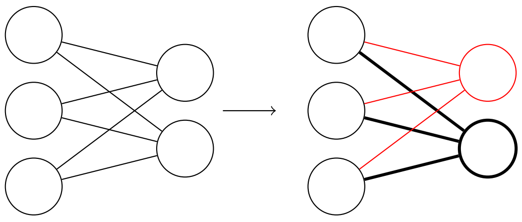


Figure 28: Structured pruning; red structures are pruned.

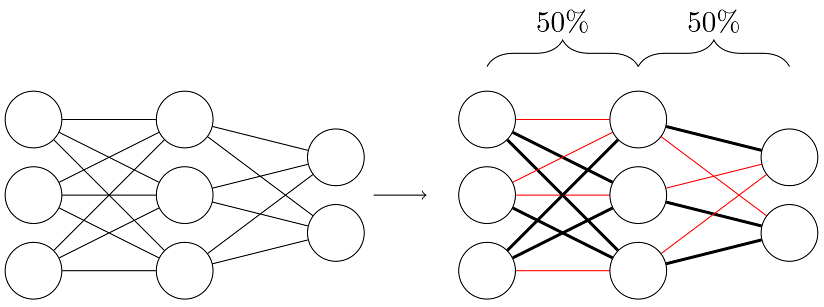


Figure 29: Local pruning; red connections are pruned

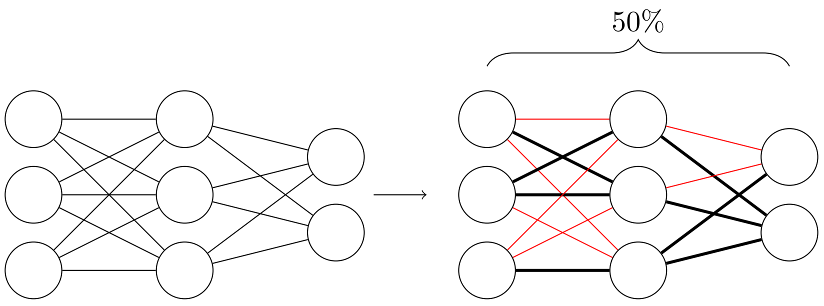


Figure 30: Global pruning; red connections are pruned

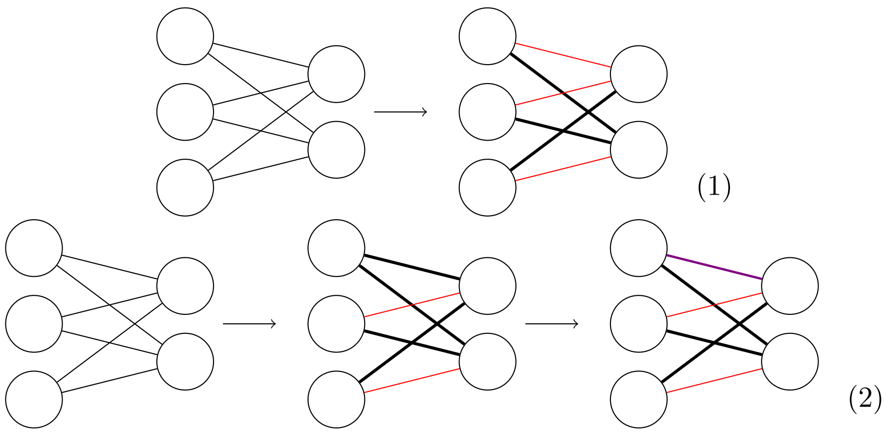


Figure 31: One-Shot (1) and Iterative pruning(2); red connections are pruned the first time; violet connections are pruned at the second time.

References

[1] Kunihiki Fukushima & Sei Miyake. “Neocognitron: a new algorithm for pattern recognition tolerant of deformations and shifts in position”. In: (1981). URL: http://www.cs.cmu.edu/afs/cs/user/bhiksha/WWW/courses/deeplearning/Fall.2016/pdfs/Fukushima_Miyake.pdf.

[2] John S Denker Yann LeCun and Sara A Solla. “Optimal brain damage”. In: *Advances in neural information processing systems*. 1990, pp. 598–605.

[3] S. Hochreiter. “Untersuchungen zu dynamischen neuronalen Netzen”. In: 1991.

[4] Q. Li et al. “Medical image classification with convolutional neural network”. In: *2014 13th International Conference on Control Automation Robotics Vision (ICARCV)*. 2014, pp. 844–848.

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

[6] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[7] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2015).

[8] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. “Highway Networks”. In: *CoRR* abs/1505.00387 (2015). arXiv: 1505.00387. URL: http://arxiv.org/abs/1505.00387.

[9] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. “Densely Connected Convolutional Networks”. In: *CoRR* abs/1608.06993 (2016). arXiv: 1608.06993. URL: http://arxiv.org/abs/1608.06993.

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

[11] Barret Zoph and Quoc V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *CoRR* abs/1611.01578 (2016). arXiv: 1611.01578. URL: http://arxiv.org/abs/1611.01578.

[12] Christian Bartz, Haojin Yang, and Christoph Meinel. “STN-OCR: A single Neural Network for Text Detection and Text Recognition”. In: *CoRR* abs/1707.08831 (2017). arXiv: 1707.08831. URL: http://arxiv.org/abs/1707.08831.

[13] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel Pruning for Accelerating Very Deep Neural Networks”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Sept. 2017.

[14] Guangcong Wang et al. “Deep Growing Learning”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[15] Jonathan Frankle and Michael Carbin. “The Lottery Ticket Hypothesis: Training Pruned Neural Networks”. In: *CoRR* abs/1803.03635 (2018). arXiv: 1803.03635. URL: http://arxiv.org/abs/1803.03635.

[16] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. “SNIP: Single-shot Network Pruning based on Connection Sensitivity”. In: *CoRR* abs/1810.02340 (2018). arXiv: 1810.02340. URL: http://arxiv.org/abs/1810.02340.

[17] Hanxiao Liu, Karen Simonyan, and Yiming Yang. “DARTS: Differentiable Architecture Search”. In: *CoRR* abs/1806.09055 (2018). arXiv: 1806.09055. URL: http://arxiv.org/abs/1806.09055.

[18] Niluthpol Chowdhury Mithun et al. “Learning Joint Embedding with Multimodal Cues for Cross-Modal Video-Text Retrieval”. In: *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval*. ICMR ’18. Yokohama, Japan: Association for Computing Machinery, 2018, pp. 19–27. ISBN: 9781450350464. DOI: 10.1145/3206025.3206064. URL: https://doi.org/10.1145/3206025.3206064.

[19] Yiheng Zhu et al. “GP-CNAS: Convolutional Neural Network Architecture Search with Genetic Programming”. In: *CoRR* abs/1812.07611 (2018). arXiv: 1812.07611. URL: http://arxiv.org/abs/1812.07611.

[20] Christopher Berner et al. “Dota 2 with Large Scale Deep Reinforcement Learning”. In: *ArXiv* abs/1912.06680 (2019).

[21] Jonathan Frankle et al. “The Lottery Ticket Hypothesis at Scale”. In: *CoRR* abs/1903.01611 (2019). arXiv: 1903.01611. URL: http://arxiv.org/abs/1903.01611.

[22] Yiğitcan Kaya, Sanghyun Hong, and Tudor Dumitras. “Shallow-Deep Networks: Understanding and Mitigating Network Overthinking”. In: *Proceedings of the 2019 International Conference on Machine Learning (ICML)*. Long Beach, CA, June 2019.

[23] Wu Zifeng, Shen Chunhua, and van den Hengel Anton. “Wider or Deeper: Revisiting the ResNet Model for Visual Recognition”. In: *Pattern Recognition* 90 (2019), pp. 119–133. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2019.01.006. URL: http://www.sciencedirect.com/science/article/pii/S0031320319300135.

[24] Robert Tjarko Lange. “The Lottery Ticket Hypothesis: A Survey”. In: (2020). URL: https://roberttlange.github.io/posts/2020/06/lottery-ticket-hypothesis/.

[25] Wei Niu et al. “PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 907–922. ISBN: 9781450371025. DOI: 10.1145/3373376.3378534. URL: https://doi.org/10.1145/3373376.3378534.

[26] Pan Zhou et al. “Theory-Inspired Path-Regularized Differential Network Architecture Search”. In: *ArXiv* abs/2006.16537 (2020).

Résumé

Durant ce stage, j'ai travaillé au sein de l'équipe Linkmedia sur une nouvelle approche pour la recherche d'architecture de réseaux de neurones.
La recherche d'architecture de réseaux de neurones permet de trouver de manière automatique l'architecture optimale d'un réseaux de neurones soit, la succession de couche optimale ainsi que la manière dont elles sont connectées pour résoudre un problème donné (reconnaître un chat, d'un chien, d'un vélo). Il s'agit d'une tâche difficile car le nombre de combinaison de couches et de connections possibles est immense.
Nous traitons cet objectif en combinant les idées de différents domaines. Tout d'abord, nous définissons un réseaux de neurones dense (toutes les couches ont accès aux sorties des précédentes) qui contient toutes les connections possibles entre les couches. Ce super-réseaux décrit toutes les séquences de couches possibles en permettant de ne pas en utiliser certaines grâce aux connections denses. Ensuite, nous gardons les meilleures connections trouvées par un critère d'élagage qui retire alors les connections non nécessaires. De cette façon, nous trouvons l'architecture optimale définie au sein du super-réseaux.
Cependant, le nombre de connections qui composent ce réseaux est immense ce qui entraîne un lourd coût lors de l'entrainement de celui-ci. Afin de répondre à ce problème, nous procédons à la recherche d'architecture de manière itérative, ajoutant petit à petit de plus en plus de couches jusqu'à atteindre le super-réseau final et en effectuant l'élagage à chaque couche ajoutée. Ainsi, nous empêchons le nombre de connections d'augmenter de manière exponentielle.
Pour résumer, nous proposons une nouvelle méthode de recherche d'architecture basée sur les récents résultats en élagage de réseaux de neurones, en réseaux denses et en croissance de réseaux de neurones et leur combinaison.

Abstract

In this internship, I worked with the Linkmedia team on a new approach to address network architecture search.
Network architecture search tries to automatically find the optimal network architecture, meaning the optimal succession of layers and how they are connected to each other to answer a particular objective (identify a cat from a dog or a bicycle). This is a difficult task as the number of possible combinations of layers and connections is intractable.
We answer this task by combining ideas from different fields. We first define a dense (every layer has access to the previous layer outputs) network that holds every possible connection between the layers. This super-network composes different possible sequences of layers by having the possibility to skip the unnecessary layers. Then, we select the most important connections using a pruning criterion that removes the unnecessary connections. That way, we find the optimal architecture defined in the super-network.
However, the number of connections in the super-network is immense, leading to computation overhead when training it. To alleviate this problem, we procede to the search in an iterative manner, adding little by little the layers of the final super-network and pruning them in the process. That way, we keep the number of connections from increasing exponentialy.
To summarize, we propose a new network architecture search method based on recent results in the fields of pruning, dense and growing networks and the combination them.