MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN DATA SCIENCE

Automated and Disciplined ConvNet Architecture Exploration

Gratia, Antoine

Award date:
2021

Awarding institution:
University of Namur

Link to publication

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Automated and Disciplined ConvNet Architecture Exploration

Antoine GRATIA

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Université de Namur
Faculté d'informatique
Année académique 2020–2021

Automated and Disciplined ConvNet Architecture Exploration

Antoine GRATIA

Maître de stage : Mathieu ACHER

Promoteur : ____________________ (Signature pour approbation du dépôt - REE art. 40)
Gilles PERROUIN

Co-promoteur : Paul TEMPLE, Benoit FRENAY

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Acknowledgments

I would like to thank all the people who contributed to the success of my internship and who helped me to write this master thesis. I would like to express my gratitude to them. I would like to thank the team for their good humour despite the sanitary conditions which are not optimal. I also thank you for your advice and your remarks concerning my work.

First of all, I would like to express my deepest gratitude to my internship supervisor Prof. Mathieu Acher for taking time to have meetings every day with sometimes some technical difficulties. I thank you for helping me, guiding me through this internship.

I would also like to pay my special regards to my master's thesis supervisor Dr. Gilles Perrouin for his patience, his availability and above all his judicious advice, which contributed to my reflection.

I wish to show my deep gratitude to Dr. Paul Temple and Prof. Benoît Frénay. First of all, Paul, thank you for giving me the meetings, time and answering my questions about what a PhD is and how it works, as well as your experience. Next, Benoit, thank you for your availability and your tips and for answering my questions regarding Machine Learning. They have been of great support in the development of this master thesis.

I would like to thank Irisa/Inria Rennes, France, for welcoming me (virtually) for this internship. In particular, many thanks to the DiverSE team in which I was a part during these few months.

Abstract

Convolutional neural networks (CNNs) are widely used for diverse tasks, such as image recognition and analysis. Recently, research aimed at finding CNN architectures that can be used in various contexts/applications and provide the latest performance has yielded fruitful results, resulting in numerous recommendation models tailored for more or less specific purposes. Finding the right CNN is a challenging issue: there are many possible architectures, hyperparameters, and frameworks that can be considered. From a software engineering perspective, having such diversity can be difficult to deal with when trying to maintain a system or trying to reason effectively (for example, consider choosing the best solution for deployment on a system with high potential impact on daily life). In this master thesis, we investigate how variability can be expressed to derive different CNN variants. We develop a generator on top of Keras for deriving variants of LeNet, ResNet, and DenseNet architectures. Our results show that we can reach accurate results on MNIST and CIFAR-10. The next step of our work is to improve the generator for other architectures (e.g. Xception, SqueezeNet,...) and find optimal ways to explore the configuration space.

Contents

1 Introduction

Today, advances in hardware and algorithms have enabled the development of extremely complex but that are extremely efficient regarding the performance (accuracy) and that are also precise in object recognizing of deep learning techniques. This allows for a widespread usage of these techniques in numerous fields: medical imaging, recognition of entities in images, natural language processing, etc. A very common application of Deep Learning (DL) is the support of automated classification tasks. There is no single algorithm (realized via a DL architecture) that copes optimally with all possible tasks and contexts, in other terms “one size fits all” does not apply to Machine Learning (ML). Consequently, many algorithms and architectures were subsequently proposed in the scientific literature. A model of deep neural networks with multiple layers has been proposed [7]. This master thesis offers to explore the variability of several popular state-of-art convolutional neural network architectures, also called ConvNets: LeNet, ResNet, DenseNet. We do it in a disciplined way, by having a common workflow to desing increasingly complex and knowledgeable (via constraint) generators able to generate cross-architecture variants. In short, this master thesis makes the following contributions:

- The modelling of the variability via state machine for the part of the CNNs architecture;
- The modelling of the variability via a feature model for the hyperparameters part of CNNs architecture;
- The construction of a cross-architecture generator.

In Section 2, we introduce the background concepts on Convolutional Neural Network and the software variability. In Section 3 we discuss the context, motivation and overview of the state of the art of this work. Section 4 formulate the different research questions and the experimental protocol. In Section 5, we present variation points in CNN architectures. Then, in Section 6 we model CNN variants with feature models and state machines. Section 7 present three architecture generators that exploit our models. In Section 8, we present our results and in Section 9 we conclude.

2 Background

This section introduces variability and deep learning concepts, core to this master thesis.

2.1 Software Product Lines

Software Product Lines (abbreviated SPLs) are “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way” [4]. Central to the design of SPLs is *commonality & variability analysis*, *i.e.* the design of elements that are shared by all the members of a SPL (commonalities) and those that are specific to one or several members of the SPL (variation points). commonality and variability analysis therefore enable to specify all member in a SPL. This is specification is often provided in the form of a variability model (example in Section 2.1.1 Figure 1).

2.1.1 Feature Models

Feature model [24] is currently the most popular model for expressing variability. It is a specific type of variability model whose focus is on expressing variability through variation points. The feature model provides information about how variation points are related and what choices can be made in order to derive a given member of the SPL or *variant*.

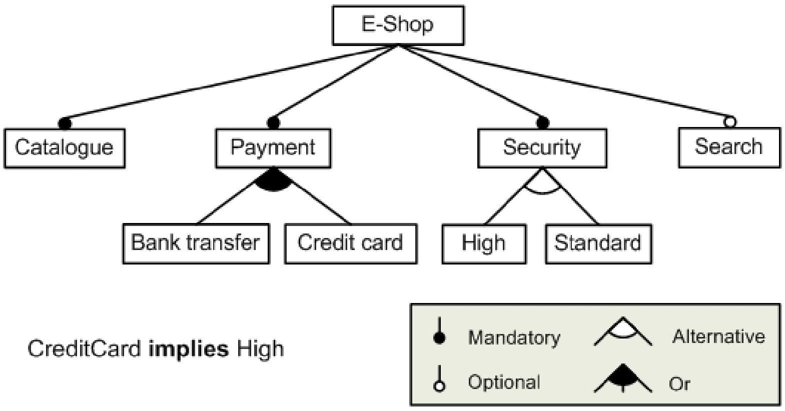


Figure 1: A feature model representing a configurable e-shop system

Figure 1 shows that for an e-shop we necessarily need a catalog, a payment method and security. But research is optional. For the payment we can choose bank transfer or credit card or both. For security we can choose either High or Standard. We also have a constraint on the credit card, which implies a High security.

2.2 Deep Learning

Deep Learning can be viewed as a subset of Machine Learning studying computer algorithms that learns based on data samples (or examples) [29]. Deep Learning uses a neural network architecture (Figure 2), the term "Deep" refers to the number of layers of neurons in the network. Indeed, a traditional neural network generally contains two or three layers unlike deep learning nets that can contain hundreds. Deep learning additionally requires several thousands of inputs (or data samples) to be accurate.

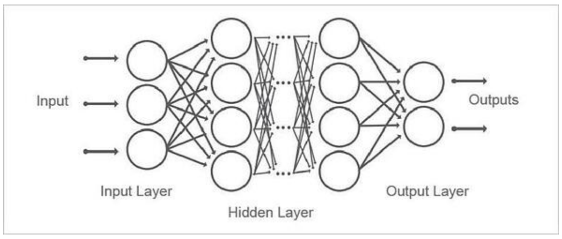


Figure 2: Deep Learning Neural Network

In this work, we focus on Convolutional Neural Networks (CNNs) (see next section). Other architectures exist like Recurrent Neural Networks (RNNs), AutoEncorder (AE), ... [1]

2.3 Convolutional Neural Network

Convolutional neural networks (CNNs, or ConvNet for short) is a class of deep neural networks, most commonly applied to images processing. The name "convolutional neural network" comes from the mathematical operation called convolution that they use. Figure 3 show an example of a CNN architecture but there exists several architectures because the sequence/number of layers, etc. is configurable. This master thesis attempts to model ConvNets architectures as variants of a SPL. In the following, we present the main state-of-art ConvNet architectures.

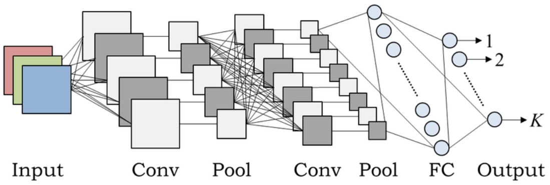


Figure 3: Example of CNN architecture

2.3.1 Architectures

2.3.1.1 LeNet

LeNet-5[1] is a kind of CNN architecture made with seven layers. The composition layer consists of 2 convolutional layers (i.e. it is a operation apply on the input data more information Section 5.1.1), 2 subsampling layers (i.e. it is a way to reduce the quality of the input more information in Section 5.1.2) and 3 fully connected layers (i.e. they are used to make the classification more information in Section 5.1.5). The number present in the name of the architecture corresponds to the number of layers. For example, LeNet-5: that means this 5-layer architecture. Note that a convolution layer plus a subsampling layer is counted as a single layer. So we have 2 convolution layers and 2 subsampling layers which are equal to 2 plus the 3 fully connected layers, we thus get 5 layers.

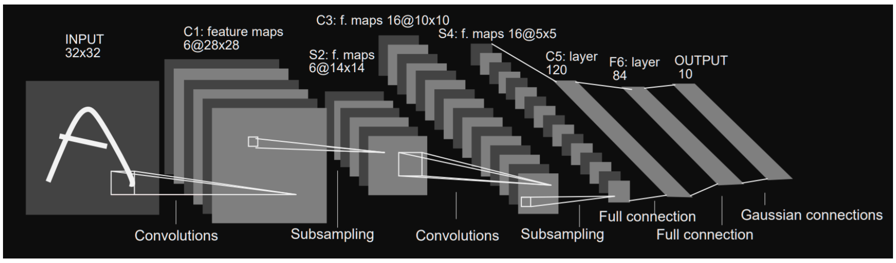


Figure 4: LeNet-5 architecture

Figure 4 shows a description of the LeNet-5 architecture, as illustrated in the original paper [22].

[1] LeNet is a set of architecture and LeNet-5 is an architecture among the existing one

2.3.1.2 Residual Network

Residual Network (ResNet) [9] is a kind of network that introduces the residual connection. A residual connection (also known as skip connection) is an additional connection added between the different layers of a neural network which allows one or more layers of processing to be avoided.

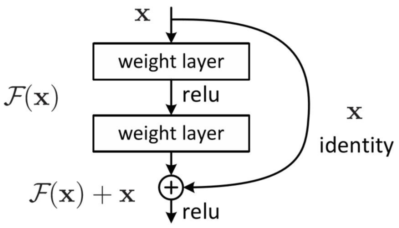


Figure 5: Residual block

In Figure 5, f (x) a function applied on input x (it can be a convolution, a matrix multiplication or a batch normalisation) and "x" (identity) allows the gradient to pass directly. By stacking these layers, the gradient could theoretically "jump" over all the intermediate layers and reach the bottom without being modified.

2.3.1.3 DenseNet

For many intensive prediction problems, low-level information is shared between input and output, so it is hoped that this information will be passed directly through the network. Another way to achieve skip connections is to concatenate previous feature maps. The most famous deep learning architecture is DenseNet [11]. Figure 6 shows an example of feature reusability by concatenating with 5 Convolutional layers.

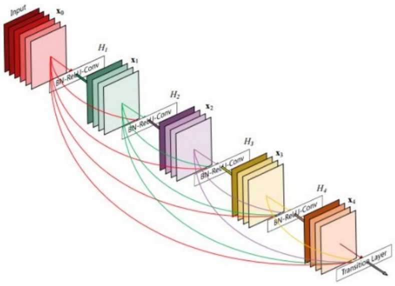


Figure 6: Example of DenseNet

This architecture makes extensive use of function concatenation to ensure the maximum information flow between the various layers in the network. This is achieved by directly connecting all layers to each other through series, as opposed to ResNets. In fact, we want to connect the characteristic channel dimensions. This will result in

- A large number of functional channels on the last layer of the network;
- A more compact model;
- extreme feature reusability.

3 Context

This section presents two articles that are directly related to and inspired our work. They were the focus of two reading group lectures in the Diverse team.

3.1 Automated Search for Configurations of Deep Neural Network Architectures

Ghamizi *et al.* designed a tool to configure and specialise DNNs [7]. For this, they create a variability model that captures all the variation points of DNN models. They represent the whole variability of DNNs with a Feature Model (FM) focuses on the architecture. They eliminated all the optimizations from the training process, such as increased data, decreased learning rate of modern regularization technology, and improved stochastic gradient descent methods to perform experiments within a reasonable time. This means that the model can be improved and have a larger space of possibilities but its risk of being more complex to find efficient architectures.
They address the following research questions :

- Can we develop a variability model that represents all possible DNN architectures?
- Can we effectively search the configuration space and identify well-performing DNN architectures?
- Does the technique find DNN architectures that outperform the state of the art?

They use PLEDGE [10], that is a tool which samples very dissimilar configurations from a feature model using an evolutionary algorithm. Since the feature model represents architectural choices, automatically sampled configurations are architectural variants. They create two manually configured architecture LeNet5 and SqueezeNet. They chose those architectures because of their popularity in the past twenty years. They compare LeNet5 generated by the tool with the version they have manually built by comparing the average accuracy over multiple runs.

They experiment their tool but they impose 2 constraints to reduce the training time. They do not optimise the training (e.g. optimisers, batch size, loss function). They use 2 datasets MNIST and CIFAR10. They also use two metrics (namely accuracy and efficiency). Efficiency represent a trade off between correctness of predictions and computation resources.

For the result of the paper, they test their FM, by comparing the LeNet5 architecture of their model with the one implemented directly in Tensorflow/Keras (with the same parameters). Results show that their FM can represent the space of DNN architectures and can successfully automate the generation of their variants. As they search the configuration space, they found that

- architectures have a real impact on accuracy;
- architectures with high accuracy are not necessarily the ones with the largest size;
- framework has the ability to form and specialize architectures for a particular domain.

They seek for architectures that outperform the state-of-the-art. They generate architectures three times under different constraints The first sample is architectures obtained by applying the general setup which thus cannot be substantially larger than LeNet5. The second one, LeNet5 architectures with different parameterisations (S2) the third one, enforce smaller architectures (S3) testing on MNIST & CIFAR-10 They conclude that their approach indeed manages to outperform an established architecture that was designed manually and that effective architectures are not necessarily the largest.

The contribution of the paper is :

- demonstrated how to model architectures with a FM;
- application of variability management techniques;
- studying most popular DNN architectures;
- the hierarchical structure of FMs also allows the addition of new dimensions of variability. While the current FM focuses on the inner constituents of DNN architectures;
- creating fully automated process that searches and deploys DNN architectures.

This paper inspired the work reported in this dissertation. They use feature models to model variability but we think that the feature model is not enough to model the variability, as we will demonstrate hereafter. Their study is limited to the architecture itself. We incorporate knowledge during hyperparameter exploration in our work.

3.2 Auto-Keras: An Efficient Neural Architecture Search System

Auto-Keras [15] stands for AutoML (automated Machine Learning) and Keras which is Deep learning library. The purpose of AutoML is to enable people with limited machine learning background knowledge to use machine learning models easily.
They proposed a new network architecture search (NAS) method, which can select the best architecture for a specific task. They graphically represent the collection of possible architectures. The node represents the architecture, and the edge represents the morph (transformation) of the architecture. They use Bayesian Optimization to find the best architecture by navigating the graph.
They propose a kernel function to compare different networks, higher is the result of the function, higher is the difference between two architectures. The process is composed of 3 steps that can be repeated :

- Update : train a Gaussian Process (GP) on the previous tested architecture and their accuracy to estimate the accuracy of the connex networks.
- Generate : predict the best morph to apply and generate the new architecture.
- Observe : test the new architecture

The research questions on the paper are :

- How effective is the search algorithm with limited running time?
- How much efficiency is gained from bayesian optimization and network morphism?
- Does the proposed kernel function correctly measure the similarity among neural networks in terms of their actual performance?

The contributions of the paper are :

- To propose an algorithm for efficient neural architecture search based on network morphism guided by Bayesian optimization;
- Conduct intensive experiments on benchmark datasets to demonstrate the superior performance of the proposed method over the baseline methods;
- Develop an open-source system, namely Auto-Keras, which is one of the most widely used AutoML systems.

The paper has a new approach with the network morphism. Then they create a graph based on a architecture manualy-built and they apply their gaussian search. But there are some limitations:

- the morphism operations (edge of the graph) are limited;
- the architecture is manualy build so if the base is not performing well, the morphism operation will not improve it radically.

As for Gamhizi et al., the exploration space is (deliberately) limited. However the optimisation strategy developed in this paper is complementary to ours (we restrict our work to exploration) and technically compatible since we are also using the keras framework.

3.3 Challenge

Machine learning frameworks have made the application of advanced Deep Learning architectures relatively easy (generally few lines of python code). However, only experts can explore the neural nets search space wisely, without exploring invalid combination of parameters, or combinations of layers that do not compile. Our goal is to extend the exploration space to multiple architectural styles (as opposed to previous works) while limiting the chances to derive invalid/useless architectures through the addition of knowledge during generation. We detail in the following sections how we reached that goal by developing variability-aware and model-based architecture exploration generators.

4 Questions & Experimental Protocol

This section discusses the research questions and the strategy used.

4.1 Research Questions

Our purpose is to find a way to explore the architecture space to a large extent to find the best way to solve a specific task. For this we rely on a variability model, the model captures all the variation points and validity constraints of the CNN model. As usually performed in a configurable system, the configuration selected from the variability model is linked to specific implementations (libraries like Tensorflow and Keras in our example) to derive deployable CNNs. We focus on the CNN architecture and hyperparameters, while ignoring the training process (training rate, cost function, gradient descent method, etc.) and data set preparation (such as data expansion). In view of this, our research questions are:

RQ1: How can we model the variability in a neural network ?
We started to model variability via feature models. And as sub-question: will feature models be sufficient to model the variability? If not, is there another way to model them? And we asked ourselves the question of what is the best language to build this generator? Beyond the programming language, we thought rather of a language which would be able to structure the architecture.

RQ2: How can we engineer a generator of CNN architectures?
We aim to build a generator capable of configuring CNN architectures. We also thought about a method to use to allow improving the generator and compare the architecture generated to study them and retain various improvements.

RQ3: How does generated architecture variants compare to the state of the art?
We aim to evaluate our generated architectures and to be able to compare them to the state of the art. We asked ourselves the question: is accuracy alone a representative measure of the performance? Is accuracy the only one that we can take into account? What is the performance of the generated architectures without specialization (i.e. mixed architectures)? And additionally we wonder: are these generated architectures able to outperform baseline architectures?

4.2 Experimental Protocol

4.2.1 Strategy

The main strategy was to begin with a simple generator and let it explore the space as much as possible, then improve the generator incrementally and correct its errors by adding some constraints (i.e. domain knowledge extracted and retrieved through experiments). This strategy aims at increasing chances of creating functional neural architectures while keeping as much as possible the potential to explore the configuration space. Iterations are based on the workflow which is described in the next section. At each iteration we fix various errors like python errors. And if the generated architecture is able to run then we look at the precision, the training time of the architectures and if there is overfitting. If an architecture has a lower accuracy than random. We build it to then correct it and retain a new rule / constraint in order to improve the generator. Our generators therefore increasingly explore the architecture space in a disciplined manner.

4.2.2 Empirical Workflow

Figure 7 shows our empirical strategy to improve the generator. First of all, regarding DSL and variability models, we wondered if our DSL was expressive enough, in particular to integrate new architectures without modifying the language. For the addition of an architecture, we looked at the research papers [22, 9, 11] as well as their implementations. We observed and analysed the various points that went wrong for the expressiveness of DSL. So after extracting the missing expressiveness points we could add them and start over (RQ1). Secondly, we use our DSL to obtain results, then we observe these results and analyze them. From these analyzes, we can extract improvements / knowledge that can be integrated into the generator. Further improvements can be added in an iterative manner following the same process (RQ2).

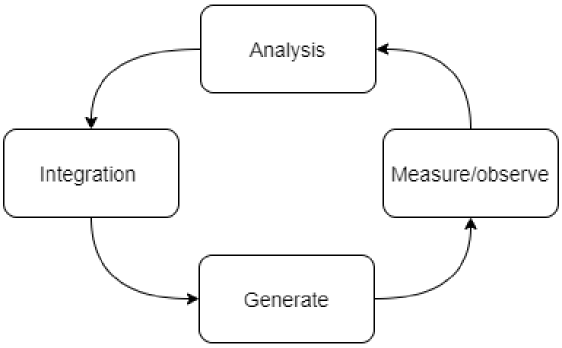


Figure 7: Empirical workflow

5 Variation Points

This section presents the retained variation points to describe ConvNets' architectural variability.

5.1 Layers

A layer in the deep learning model is the structure or network topology in the model architecture. It obtains information from the previous layer and then passes the information to the next layer. Deep learning has several famous layers which we will explain in the following section.

5.1.1 Convolution

Convolution is a mathematical operation that uses a kernel (also called filter or matrix). A kernel is a small matrix of weights. This kernel "slides" over the data, performing an elementwise multiplication with the part of the input it is currently over, and then summing up the results into a single output pixel.

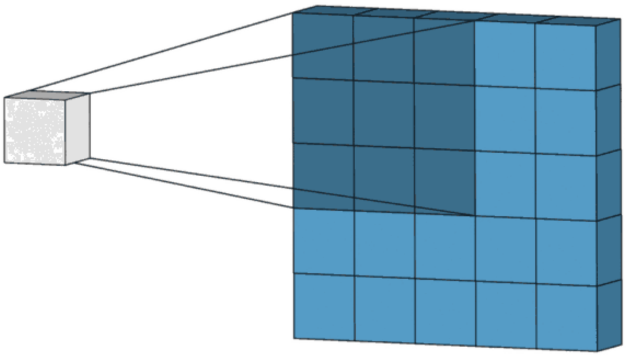


Figure 8: Visualisation of a convolution

In Figure 8, the input is the 5x5 matrix, the kernel is a 3x3 dark blue matrix at the top left corner and the result of the convolution is the grey-single element in front of the matrix. The kernel repeats this process for every location it slides over, converting a matrix of features into another matrix of features. The output features are essentially the weighted sums (with the weights being the values of the kernel itself) of the input features located roughly in the same location of the output pixel on the input layer.

5.1.2 Subsampling

Subsampling is a technique that has been designed to reduce the quality (i.e. the size) of the image by skipping some of the value or by combining several values into one (average, weighted average, mean, etc.). Thus, subsampling reduces the number of feature maps as we move through the network.

5.1.3 Pooling

Pooling is a sample-based discretization process. The objective is to down-sample an input representation (image, hidden layer output matrix, etc.), by reducing its dimensionality and making assumptions about the characteristics contained in the grouped sub-regions.

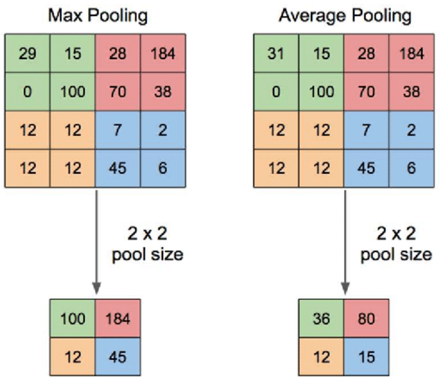
5.1.3.1 Average Pooling

This operation computes the average of values inside a window which slides over the input (see Figure 9).

5.1.3.2 Max-Pooling

This operation computes the maximum of values inside a window which slides over the input (see Figure 9).

5.1.3.3 Examples



Max Pooling

29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

100	184
12	45

Average Pooling

31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

36	80
12	15

Figure 9: Max and Average Pooling

5.1.4 Global Pooling

The global pooling works like the pooling but instead of calculating the max or the average on a window, it calculates on the image in a global way.

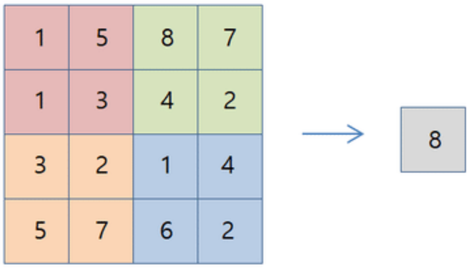


Figure 10: Example of global max pooling

5.1.5 Fully Connected

Stricto sensu, a fully connected layer is not a layer but rather a way to connect two layers. Precisely, all neurons of the first layer are all connected (hence its name) to the neurons of the second layer. This layer is used in convolutional neural network for the classification part.

5.1.6 Flatten

A flatten layer allows to convert matrix of size $N \times M$ to vector of size $1 \times (N \times M)$. (See Figure 11)

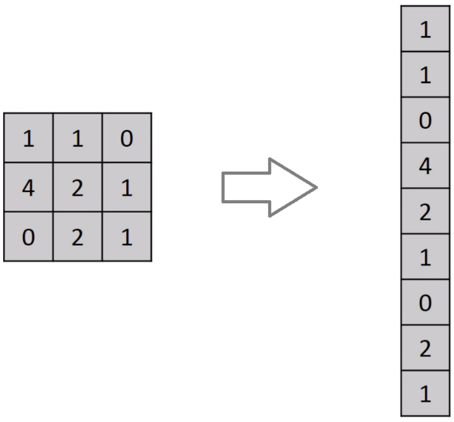


Figure 11: Example of Flatten on matrix 3×3

5.1.7 Dropout

The term “dropout” refers to dropping out units (hidden and visible) in a neural network. Dropout introduces the probability (i.e. a rate between 0 and 1) at which outputs of the layer are dropped out (i.e. deactivate), or inversely, the probability at which outputs of the layer are retained (e.g. a rate of 0.5 deactivates 50% of the neurons of the layer). Dropout is a way to prevent neural networks from overfitting [27].

5.1.8 Batch Normalisation

Batch normalisation is proposed as a technique that helps to coordinate multiple layers of updates in the model [14].

5.2 Hyperparameters

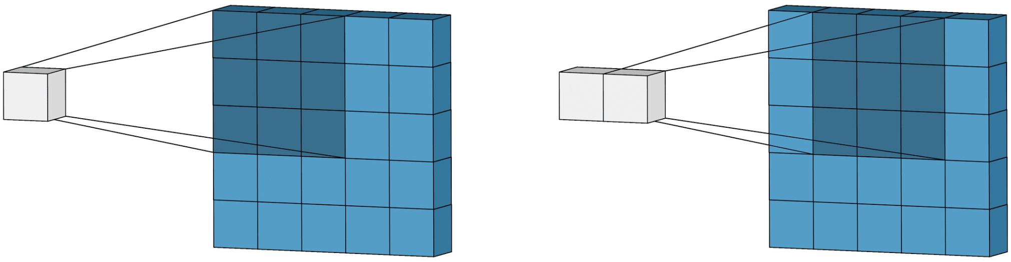
Hyperparameters are parameters that need to be fixed at some point as they may influence model performances. Their value can be influenced by the datasets characteristics that are used (separated classes, number of instances, number of classes, data distributions, etc.) the task-at-hand and probably other factors. Hereafter we explain hyperparameters that are of interest for the remaining of this work.

5.2.1 Padding

Padding allows the use of additional "fake" pixels (usually 0, hence the term "zero padding" is often used) to fill the edges. In this way, the kernel can allow the original edge pixels to be located at its centre while sliding, and at the same time extend to pseudo-pixels outside the edges, thereby generating an output of the same size as the input.

5.2.2 Stride

Stride is used to downsample an image or any other block processing of images (or matrices), the idea is to avoid overlapping of the kernel to force it to "slide". The concept of the stride is to skip some of the slide locations of the kernel. Figure 12 shows an example of how stride works. We have the convolution kernel (3x3 matrix) which slides on the input (5x5 matrix). Figure 12b shows this sliding step after Figure 12a. We notice that we have shifted by 1 pixels to the right: it means that we have a stride with a value of 1.



(a) convolution at time 0

(b) convolution at time 1

Figure 12: Stride example

5.2.3 Activation Function

The activation function is a mathematical function applied to a signal at the output of a neuron.

5.2.3.1 ReLu

The Rectified Linear Unit activation function or ReLu for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero.

$$f(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x \geq 0 \end{cases}$$

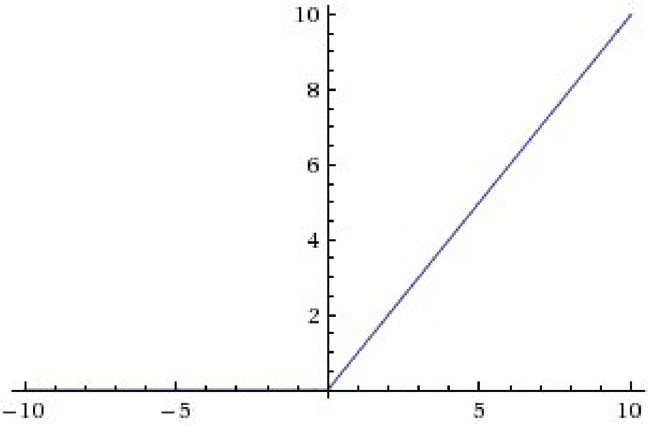


Figure 13: Activation function ReLU

5.2.3.2 SeLu

Scaled Exponential Linear Unit [19], or SeLu for short, is similar to ReLu but the main change is that it performs self-normalization. There are two reasons why you should use SELUs instead of ReLUs:

- Similar to ReLUs, SELUs enable deep neural networks since there is no problem with vanishing gradients.
- SELUs and its internal normalization is faster than other activation functions, even if they are combined with external normalisation (like Batch Normalisation).

$$f(a,x) = \lambda \begin{cases} a(e^x - 1), & \text{for } x < 0 \\ x, & \text{for } x \geq 0 \end{cases}$$

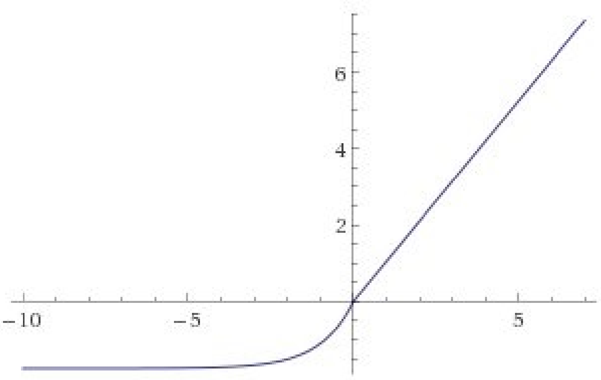


Figure 14: Activation function SeLu

5.2.3.3 Softmax

Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. The softmax function $\sigma : \mathbb{R}^K \to [0,1]^K$ is defined by the formula:

$$\sigma(\vec{a})_i = \frac{e^{a_i}}{\sum_k e^{a_k}}, \text{ for } i = 1, ..., K \text{ and } \vec{a} = (a_1, ..., a_k) \in \mathbb{R}^K$$

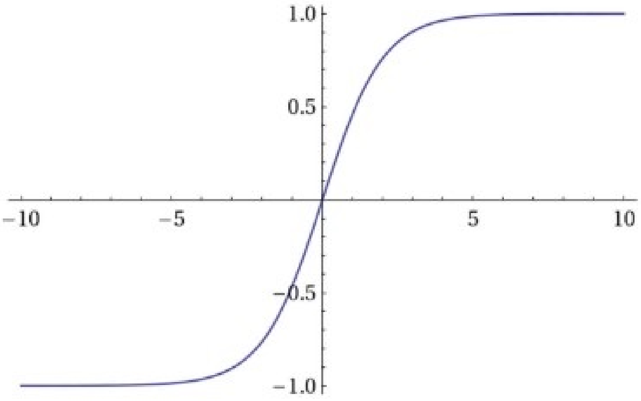


Figure 15: Activation function Softmax

5.2.3.4 TanH

The hyperbolic tangent, or TanH is very similar to the sigmoid[16] activation function and even has the same S-shape. The function takes any real value as input and outputs values in the range -1 to 1.

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}, \text{ for } x \in \mathbb{R}$$

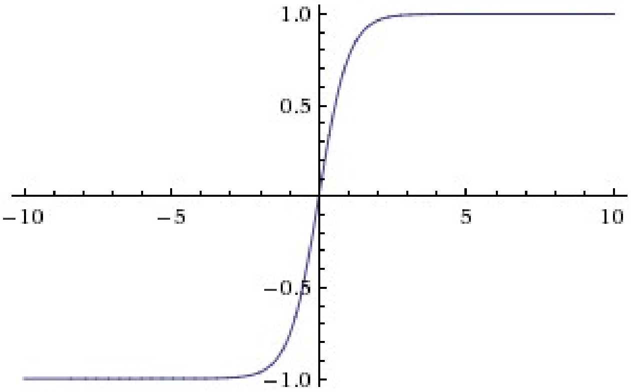


Figure 16: Activation function TanH

5.2.4 Loss Function

The loss function (or objective function) is associated with the final layer to calculate the classification error.

5.2.4.1 Categorical Cross-Entropy

Categorical cross-entropy, also called softmax loss, is a softmax activation function (see before) and follow by a cross entropy loss. This loss function is used for multi-class classification.

$$CE(y, y') = -\sum_i y_i \log(y'_i)$$

with y expected value and y' predicted value. Thus y'_i is the predicted value (y') for the class i
There exist others loss function like Binary cross-entropy which is the same as categorical cross-entropy but for binary classification, Mean Square Error (MSE) ... The most commonly used can be found on the Keras webpage dedicated to loss functions[2]

5.2.5 Optimization Function

The optimization function (also known as optimizer) is an algorithm used to minimize the loss function. There are a lot of optimizers [26] like stochastic gradient descent (SGD), Adamax but in this work, we focus on Adam.

5.2.5.1 Learning Rate

The learning rate controls how fast the model adapts to the problem. Given a smaller change in the weight of each update, a smaller learning rate requires more training time, while a larger learning rate leads to rapid changes and requires less training time.

Too large learning rate will cause the model to converge to sub-optimal solutions too quickly, while too small learning rate will cause the process to get stuck.

[2] https://keras.io/api/losses/

5.2.5.2 Adam

Adaptive Moment Estimation (adam) [18] is a method to calculate the adaptive learning rate(η) for each parameter. In addition to storing the exponential decay average of the past squared gradient v_t like Adadelta [32] and RMSprop, Adam also retains the exponential decay average of past gradients m_t, similar to momentum [25]:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{v-1} + (1 - \beta_2) g_t^2$$

β_1 and β_2 are fixed parameters, m_t and v_t are respectively estimations of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

then, they $\hat{m}_t$ and $\hat{v}_t$ use these to update the parameter:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

The authors of Adam propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

5.3 Summary

The Sections 5.1 and 5.2 presented two categories of variation points: layers (e.g. convolution layer, pooling layer, ...) and hyperparameters (e.g. padding, stride, activation function, ...) which will be used for the models of variability. The next section details more of the variability models.

6 Variability Models

This section details the models used for modelling the variability, the problems encountered and provides an answer to our first research question.

6.1 Feature Models

6.1.1 Architecture

6.1.1.1 Description

Figure 17 is based on the study of the ResNet [9], LeNet5 [22] and Densenet [11] architectures as presented in Section 2.3.1. Initially, the neural network architecture is divided into 3 parts. Those parts are called Input, HiddenLayer and Output. Based on the 3 architectures we have identified five "categories" of hidden layer, each having a specific role: the Classification part, which use the characteristics extracted from the previous part to help the output to classifies the image, has one concrete variation point called Dense. Then the "FeatureExtraction" extracts the main characteristics of the input that will be used by the Classification part. This variation point is decomposed into three new variation points: Batch normalisation, Convolution and Pooling. We also have a dropout variation point. The Flatten and GlobalPooling are used to link the FeatureExtraction and Classification parts of the network.

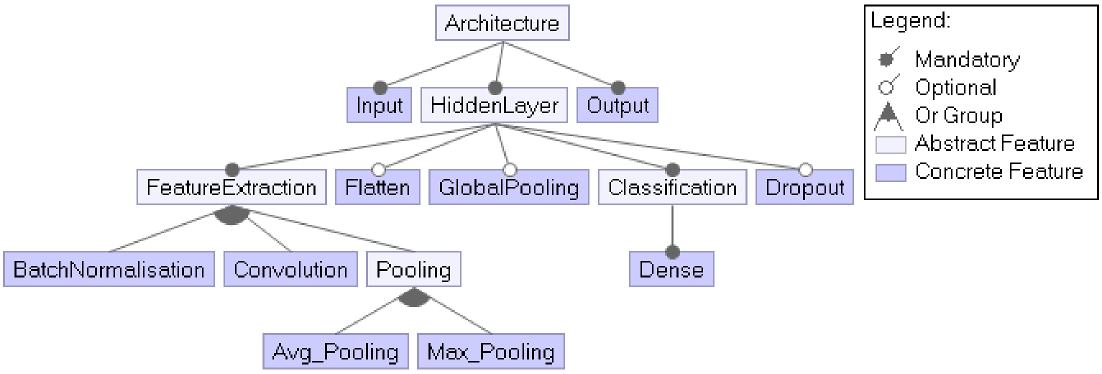


Figure 17: CNNs Feature Model

6.1.2 Hyperparameters

6.1.2.1 Description

Figure 18 is also based on the study of ResNet and LeNet5 architectures as well as the documentation of Keras [17] / TensorFlow [28].

For the Input, it has been noticed that its main point of variation is the size of the input image.

For the Output, it has two points of variability: The number of neurons depends on the number of desired classes (for example: if we classify images of cats and dogs there will therefore be 2 two neurons).

TThe activation funciton depends on the kind of task to realize i.e. if we have a binary classification it is necessary to use a logistic function (Sigmoid for example) but if we have a multi-class classification, it is advised to use a function like softmax (normalized exponential function) for example.

For Convolution, it is made up of five variability points:

The Kernel corresponds to the size of the filter. It is always a square matrix with a minimum size of one element (1x1). The Padding can take as value: same, valid, or an integer. The Stride takes as value an integer. The Number of filters corresponds to the number of feature maps and it is an integer value. The possible value for activation function are describe in Section 5.2.3

For pooling, it has three points of variability: Kernel, Padding and Strides (same value as Convolution).

For the dense part, it has two points of variability: The number of neurons takes an integer value The possible value for activation function are describe in Section 5.2.3

Dropout has the dropout rate parameter which takes value between 0 and 1.

Batch normalisation has the epsilon parameter which takes float value close to 0 like 10^{-3}, 10^{-5},...

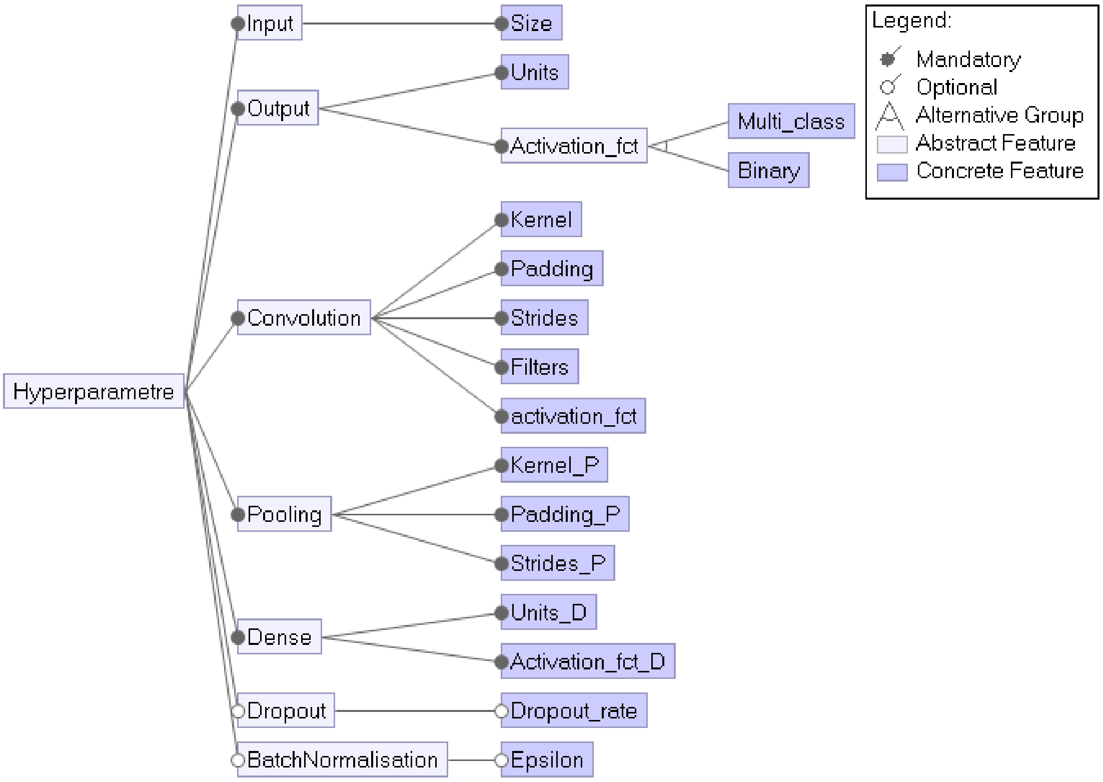


Figure 18: Feature Model of Hyper-parameters

6.1.3 Problem

Figure 17 gathers all the different variation points/layers/functionalities that we observed in the analysis of ResNet, LeNet5, and Densenet architectures. However, trying to generate an architecture out of this representation can create a model which would start with an Output or HiddenLayer directly instead of an Input which is not desirable.

Figure 19 shows an example of an architecture valid for the feature model but invalid from an implementation perspective: it will return an error if we try to build it. Indeed, we can notice three sequencing problems: the position of the output which should be at the end of the architecture, the first Dense which should be after the flatten (i.e. green bullet) and Pooling + Convolution which should be rearranged in Convolution + Pooling and be placed before the flatten.

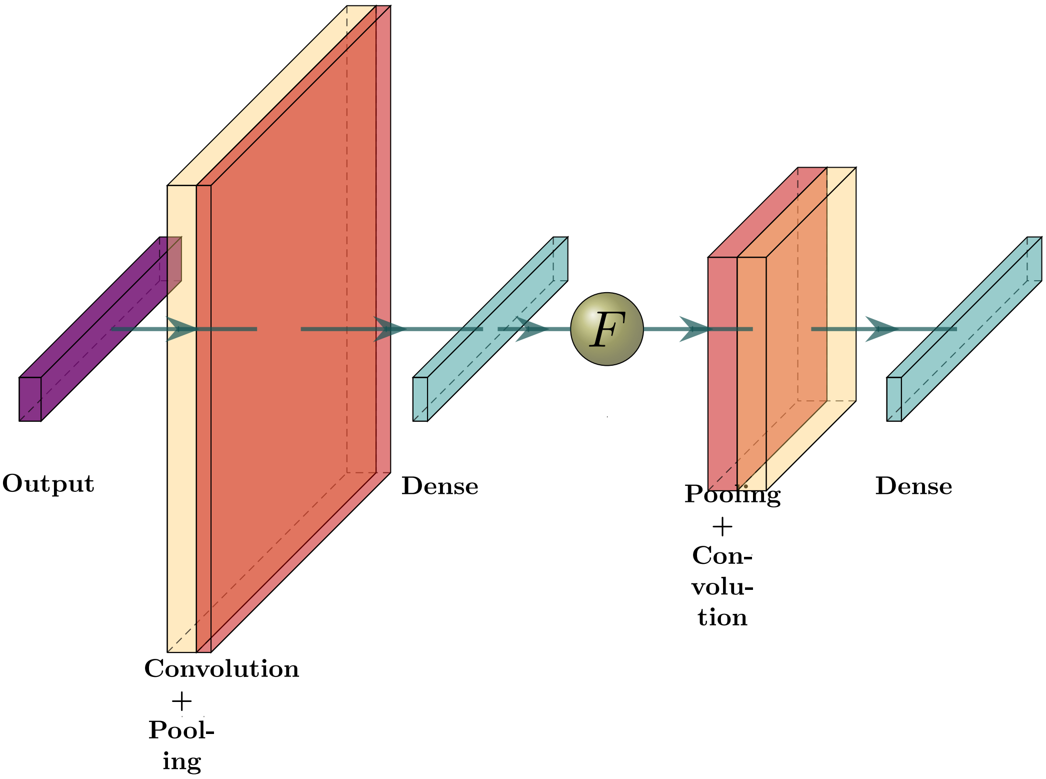


Figure 19: Example of an architect validated by the feature model

Another representation is therefore necessary to take into account the possible sequences of these variation points(see Section 6.2).

6.2 State Machines

Feature models are not expressive enough to model ConvNets, in particular due to the sequencing of these variation points. We therefore looked for a new way to model such sequences. State machines naturally integrates a notion of time and sequencing (*e.g.*, [8]). We therefore extend our formalism in order to generate valid ConvNets variants.

6.2.1 LeNet State Machines

First we built a state machine based on the LeNet architecture (see Figure 20).

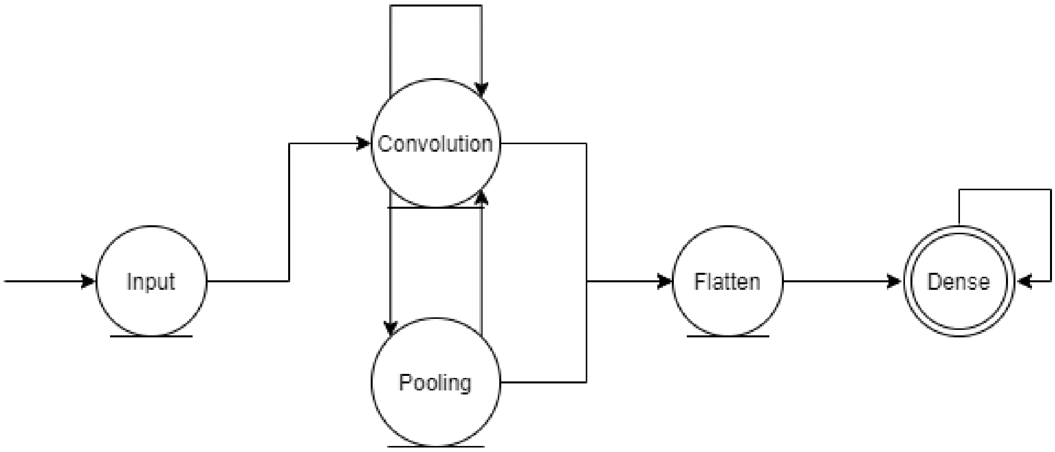


Figure 20: LeNet State Machine

The architecture LeNet is constructed as follows: it begins with the single entry point is the input layer followed by an alternation between convolution and pooling, continues with flatten and ends with a succession of dense. This state machine allows us to find the state of the art (LeNet5) and also to vary the architecture, that is to say we can build without pooling with only convolution, flatten and dense layers. We will cover different experiments based on this state machine in Section 7.1.

6.2.2 ResNet State Machines

The previous version of the state machine was limited to model LeNet-like architecture. However, other base architectures exist. It lead us to evolve this state machine to be able to model a more diverse set of architectures, and we decided to add the ResNet architecture. So we modified the LeNet based state machine to have a new ResNet based state machine.

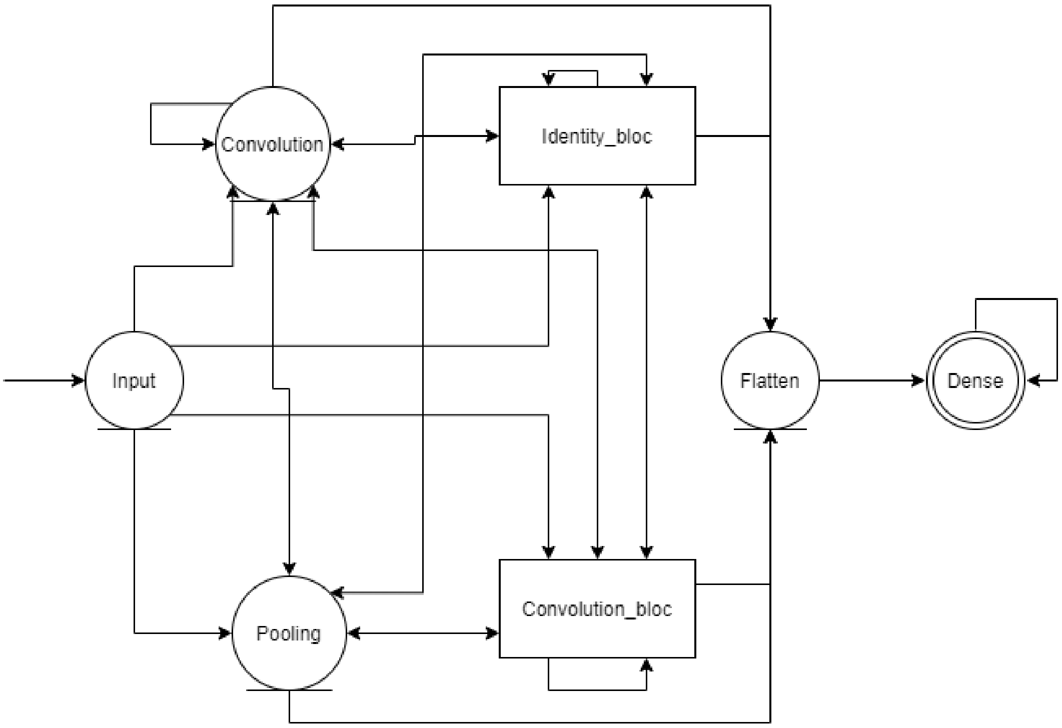
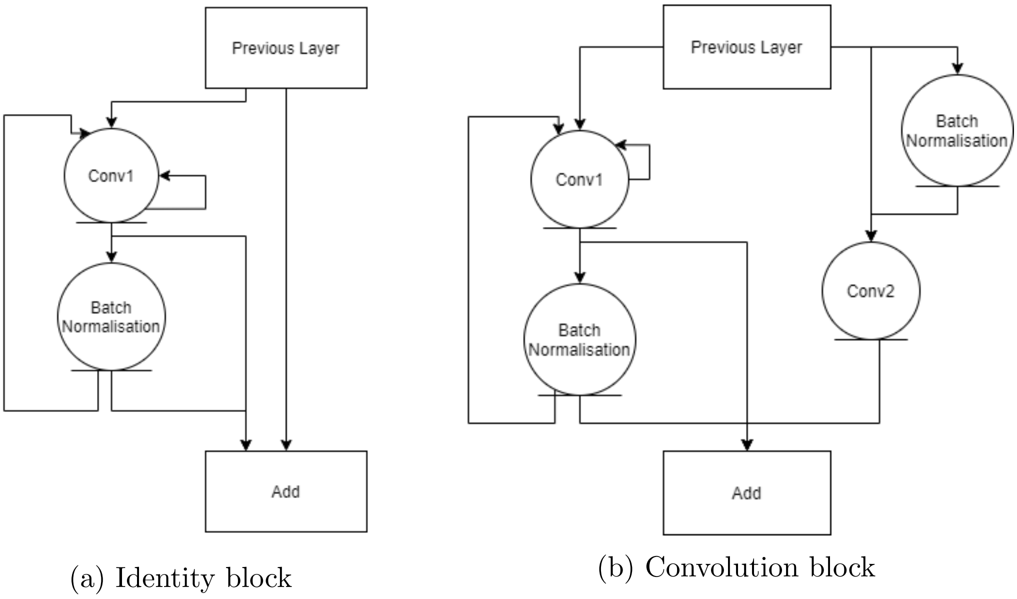


Figure 21: ResNet State Machine

We can notice the addition of two new blocks: identity block and convolution block. The identity block allows to keep the size of the image and to propagate it and with regard to the convolution block it allows to reduce the size of the image.



(a) Identity block

(b) Convolution block

Figure 22: Resnet block

This state machine allows us to build architecture based on ResNet. It is also capable of finding the state of the art such as ResNet18 or ResNet34, while adding variability, in particular in the arrangement of the layers. We can also notice that we can find the previous state machine and therefore allow building a LeNet type architecture. See Section 7.2 for details.

6.2.3 DenseNet State Machines

Our motivation for this state machine was the same as the previous. We wanted to improve the state machine with a new architecture and we decided to add the DenseNet architecture.

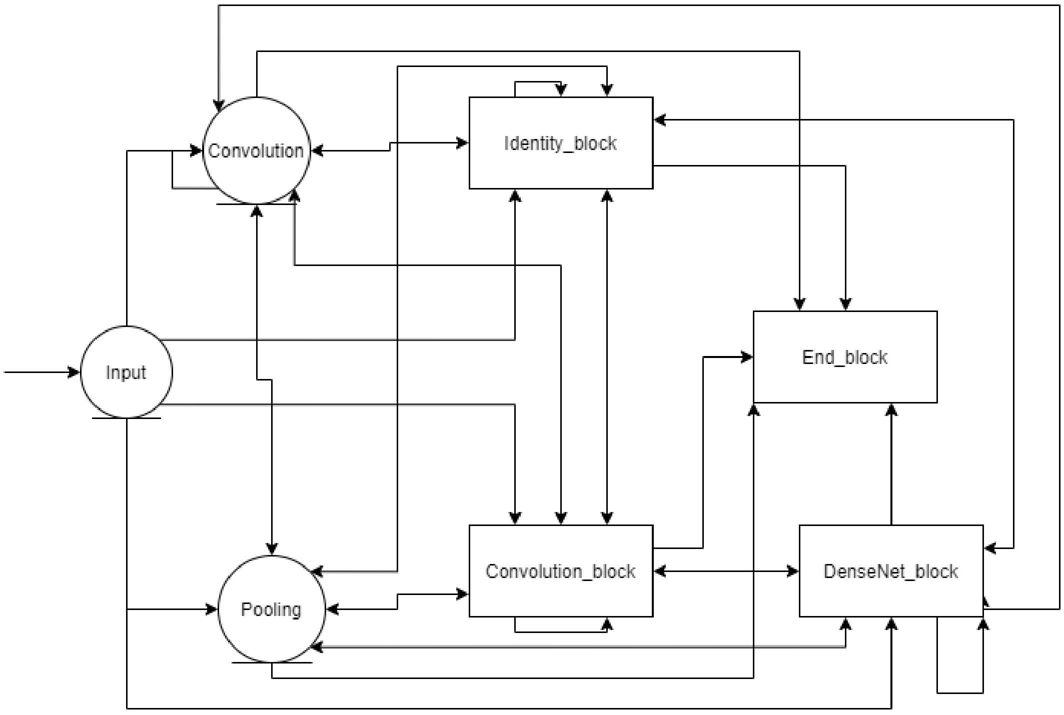
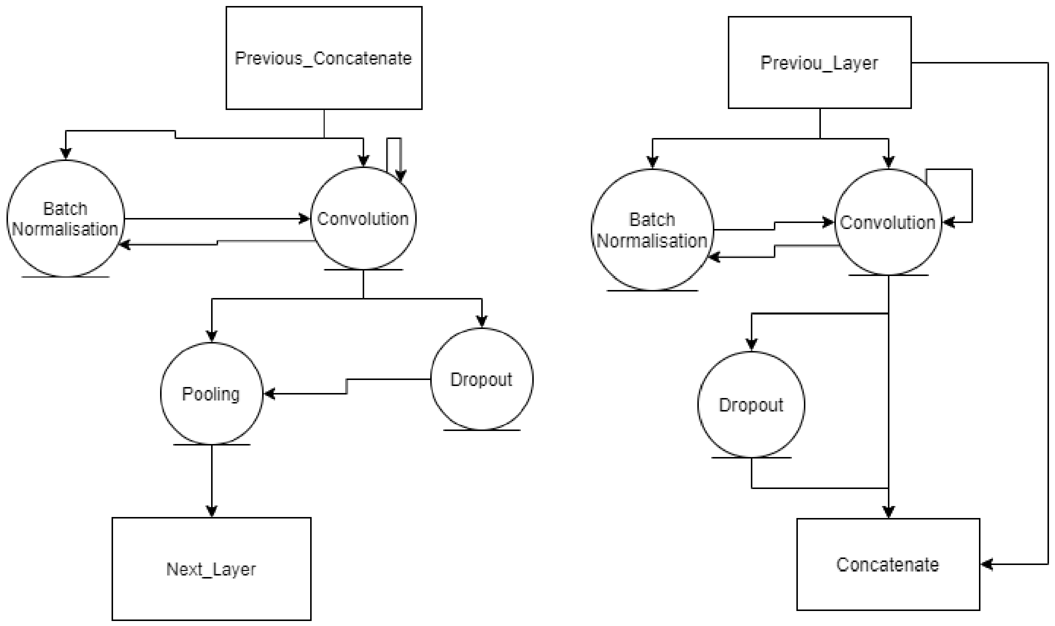


Figure 23: DenseNet Generator State Machine

We can notice the addition of a new block: DenseNet block. This block is made up of 2 blocks: Densenet convolution block and transition block. Densenet convolution block allows adding channels / filters without reducing the size of the image and then we have a transition block, it allows to reduce the number of channels as well as the size of the input.



(a) Transition block

(b) Dense convolution block

Figure 24: DenseNet block

We also changed the end of the architecture to allow global pooling. So the architecture can end with either a global pooling layer followed by a Dense layer or a flatten layer followed by a succession of Dense layers.

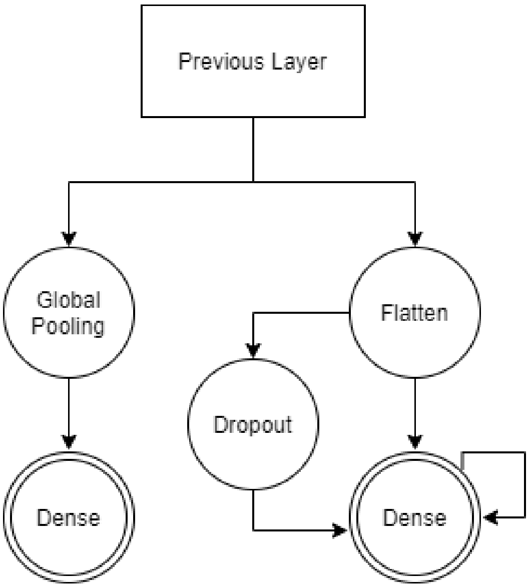


Figure 25: End Architecture block

This state machine allows us to build architecture based on DenseNet. It is also capable of finding the state of the art such as DenseNet121[11], while adding variability, in particular in the arrangement of the layers. We can also notice that we can find the previous state machine and therefore allow building a LeNet type architecture. We can also build architectures that are not specific to LeNet, ResNet and DenseNet but rather merge them to create new architectures. See Section 7.3 for more details.

6.3 RQ1: Modelling Variability

We have modelled the variability of the architecture in two different ways: the first is the feature model and as we have seen previously (Section 6), it is not sufficient in particular due to the sequencing. The second is the state machine which allows the sequencing of the different layers. Regarding the hyperparameters, a feature model was expressive enough.

We looked for a way to structure the different layers as well as their hyperparameters. We decided to integrate our modelling efforts in a Domain Specific Language (DSL) [23]. A domain specific language is a (generally) high-level language capturing the essence of a domain. For example, LaTeX, the language we used to write this master thesis, can be thought of DSL for typesetting (scientific) documents. In our case the domain is the one of ConVNets architecture. Our DSL is expressed in the Json format.

7 Generators

This section reports on the design of the three generators we have built, answering RQ3.

7.1 LeNet Generator

We start with random configurations drawn from the feature model. The generated configurations are valid by definition (see Section 5.1 Feature Models). However Python threw errors because of the value of parameters or the scheduling of layers. For instance, missing input/output or combination of parameters resulting in negative image size (e.g. too large kernel size or stride values). So we notice that the knowledge contained in the feature model are not enough and we had to add some constraints. By including more and more constraints (that ML or domain experts may know directly), performance of the generated models improved until reaching state-of-the-art performances.
From this first attempt on the model generation, we have learnt that:

- Limited the size of the architecture;
- The order of the layers is important;
- The kernel, padding and strides trio is important to manage the reduction of the image;
- The kernel must be superior or equal to strides;
- The units at the Dense level must be decreasing and smaller than the input.

As we manage to converge on the results of the state of the art, we have added a new architecture to our generator. We will discuss it in the next section.

7.2 ResNet Generator

We started by experimenting this generator with the ResNet state machine as well as the constraints of the previous generator (LeNet). Overall the results were close to those from the state-of-the-art. Nonetheless, some architectures did not do better than random accuracy. The problem comes from the units in the dense layers. In fact, the higher the input of dense layer (i.e. the feature extraction output), the greater the reduction of the input should be. This reduction is done via a certain percentage. For example a percentage of 80, that means that we keep 80% of the value of the input. After having solved the problem, the results of the generator were around those of the state of the art.
And from these different experiments we were applying these various constraints:

- The bigger the input sent to the dense layer, the higher its reduction must be;
- Padding was fixed because it is difficult to compute for the merge part (add). When using residual connections, the two paths must have the same output size. Kernel, stride and padding have to be configured. In addition, changing the value of one of them impacts the others. That is why we have fixed the value of padding.

But what about DenseNet? DenseNet is a ResNet to which we have pushed the skip connection to its peak but with the current generator it is not possible to make a network like DenseNet. The concept of DenseNet is to ensure the maximum flow of information between the different layers of the network (as described in the section 2.3.1.3). Figure 24b shows a skip connection (on the right) that ensures the flow of information and the function that groups the two paths is Concatenate contrary to ResNet which uses the function Add (Figure 22b). So we have to improve the generator so that it can produce a network like DenseNet (see next section).

7.3 DenseNet Generator

This generator is slightly different from the two previous ones because it allows building four types of architectures: LeNet, ResNet, DenseNet or mixed of three and specified powers which are desired. We started by experimenting this generator with the DenseNet state machine as well as the constraints of the previous generator (ResNet). We very quickly noticed that the number of the epochs was not sufficient, in particular because the number of layers increased as well as the amount of training parameters, the architectures did not have time to converge. So we increased the maximum number of epoch to 50 with the possibility to perform early stops (see Figure 26). The results obtained are rather heterogeneous, i.e. we have as many good results capable of approaching the state of the art as bad results but always better than the random. We also noticed cases of overfitting.

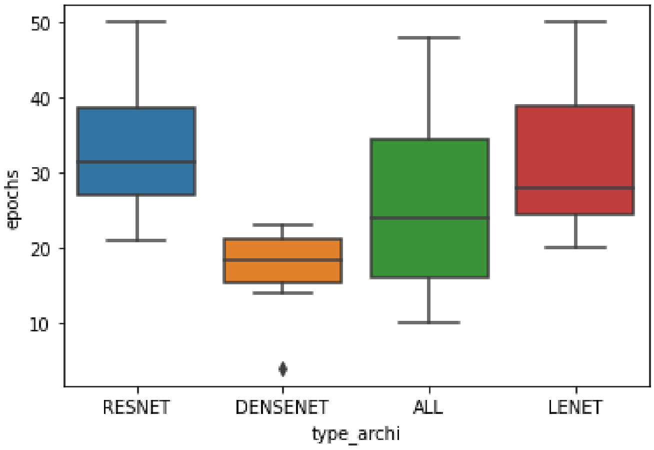


Figure 26: Box-plot for the number of epochs per architecture generated

Figure 26 shows the number of epochs required by type of architecture (the “ALL” label represents the mixed architecture). We can notice that 5 epochs was clearly not enough. We also notice that there can be outliers, they are architecture which does not manage to converge or which lose the gradient.

7.4 RQ2: Engineeing a Generator

We have improved our generator on the basis of different constraints such as the so-called strong constraints. These constraints make it possible to reduce the search space and therefore to avoid all these interpretation errors thrown by Python. For example it is useless to explore part of the space in which the architecture begins with the output or to reduce an image of size 1/1. We also have the so-called soft constraints, they allow to specify the architecture in such a way as to avoid the architecture which loses the growth or the one which does less well than the random one. Such errors in the architecture definition which are not detected by Python. As an example, we saw in Section 7.1 that may be the kernel must be greater than or equal to the strides.

8 Results

This section exposes the experimental parameters, results with respect to the constraints and a response to RQ3.

8.1 Experimental Settings

8.1.1 Generator Workflow

Our generators are built according to the workflow presented in Figure 27.

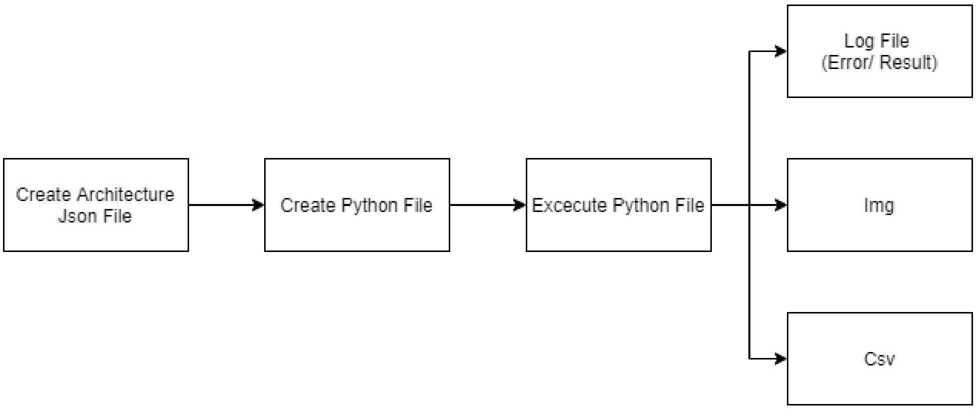
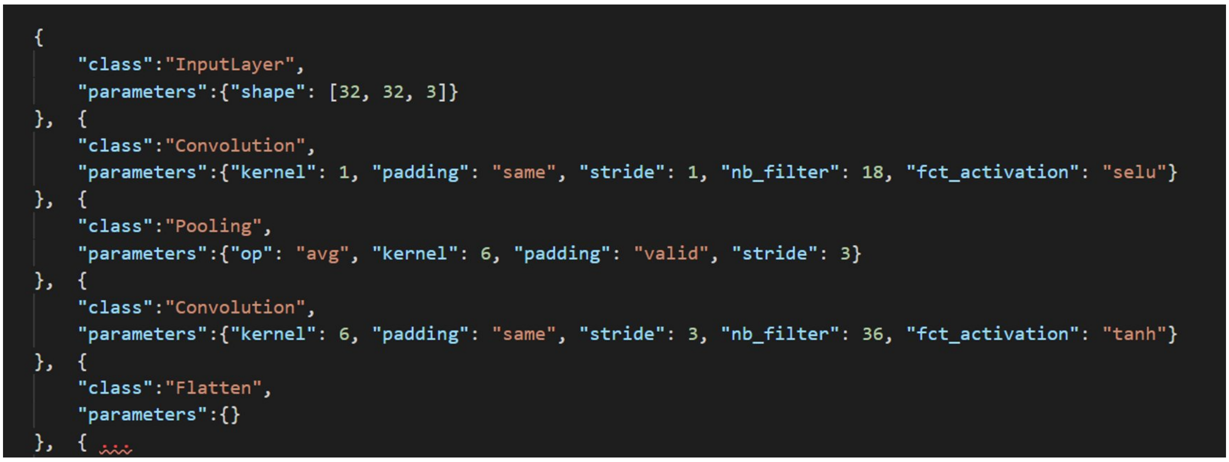


Figure 27: Generator workflow

First, the architecture is generated via the generator and saved in our DSL. Figure 28 shows an example of a part of this DSL, we can see its structure. The name of the layer is represented by the term “class” in the DSL and the hyperparameters of the layer by the term “parameters”. We can therefore see that our generated architecture is represented in our DSL in the form of a list of layers and their respective hyperparameters. Then this DSL is used for creating a python file that contains the Python code and calls to Keras or TF in order to run a CNN model (with training, evaluation, etc.). After that the python file executed and it generated:

- The log file for the error/result;
- An image that represents the trained model using the graph description language (DOT) and save in png file;
- It updates a comma separated value (CSV) file for further analysis.



```
{
    "class":"InputLayer",
    "parameters":{"shape": [32, 32, 3]}
}, {
    "class":"Convolution",
    "parameters":{"kernel": 1, "padding": "same", "stride": 1, "nb_filter": 18, "fct_activation": "selu"}
}, {
    "class":"Pooling",
    "parameters":{"op": "avg", "kernel": 6, "padding": "valid", "stride": 3}
}, {
    "class":"Convolution",
    "parameters":{"kernel": 6, "padding": "same", "stride": 3, "nb_filter": 36, "fct_activation": "tanh"}
}, {
    "class":"Flatten",
    "parameters":{}
}, { ...
```

Figure 28: Example of part of DSL

8.1.2 Datasets

For training, we used 2 datasets called MNIST and CIFAR10:

- MNIST was used as a sanity test (i.e. very brief run-through of the functionality of our generator to assure that it works roughly as expected), widely studied and is now known to have characteristics (e.g., data distribution) that are fairly easy to process. It has 10 classes with 60 000 images in the training set and 10 000 in the test set;

- CIFAR10 was used for the other experiments. CIFAR10 is a much more challenging dataset used in evaluation of state-of-art ML models. It has also 10 classes with 50 000 images in the train set and 10 000 in the test set.

8.1.3 Training

We did not explore variability during training and we decided to set the following as constants in our experiments:

- We used Adam as an optimization function;

- We used categorical cross entropy as a loss function.

8.1.4 Hardware

We started our experiments on an inspiron 15 5000 laptop, i5-10210u 1.6Ghz (4cores), 8GB ram, nvidia mx250. Nevertheless, the training time started to increase (about 1 hour per training) in particular due to the complexity of architectures (ResNet, DenseNet) and increase in the number of epochs. Therefore we made a request in order to be able to use a more powerful machine and we were able to have access to a remote machine in the Diverse team in Rennes: Intel (R) Xeon (R) Gold 6238 CPU @ 2.10GHz (88 cores), 187Gb ram, NVIDIA Tesla T4.

8.2 Constraints Results

Table 1a presents the results obtained during the generation via the feature model which we discussed in Section 7.1 and Table 1b presents the results obtained during the generation with all the constraints seen in Section 7.1. We can see improvements from an accuracy point of view, notably by avoiding python errors. The training time is already more acceptable compared to the accuracy (e.g. 255 sec / 0.089 accuracy). We have improved the results of Table 1b by applying the constraints presented in Sections 7.2 and 7.3 besides the following section details the last results obtained with the DenseNet generator.

Training Time (s)	**Test Accuracy**
88	0.0003
36	0.0
0	Error
0	Error
0	Error
0	Error
255	0.089
0	Error
0	Error
0	Error

(a) Results with the LeNet Generator without constrains

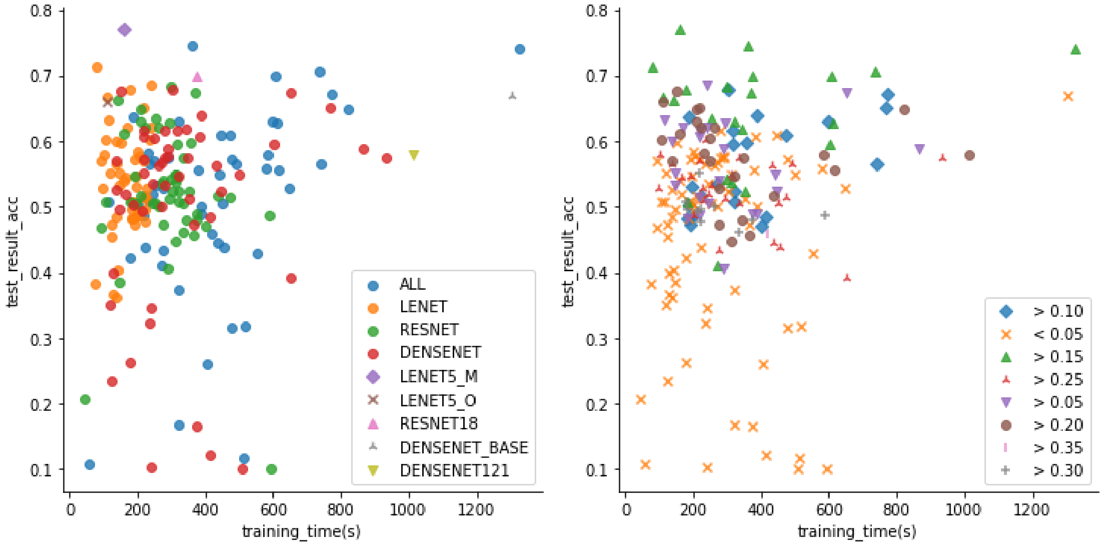
Training Time (s)	**Test Accuracy**
66	0.49
54	0.55
85	0.63
150	0.1
105	0.1
107	0.51
59	0.63
40	0.60
264	0.1
112	0.47

(b) Results with the LeNet Generator with constrains

Table 1: comparative tables between two LeNet generator results

8.3 RQ3: Performance of Generated Architectures Compared to the State of the Art

Based on the DenseNet generator (see Section 6.3) we carried out an experiment on a larger scale. We started with 200 generated architectures (50 instances per type of architecture). Figure 29 shows the results obtained during this experiment. Figure 29a which represents the accuracy and the training time of the generated architectures (circle) compared to the state-of-the-art (others). The state of the art consists of two LeNet5, Resnet18 and two DenseNet. Figure 29b represents the difference between the accuracy during the training and that of the test. This difference is labelled by groups of values of 0.05 (i.e. < 0.05, > 0.05, > 0.10, ... , > 0.35). This figure allows to visualise the overfitting of the various architectures.



(a) Training time and accuracy per type of architecture

(b) Training time and accuracy per overfitting label

Figure 29: Result from last experiment with Generator DenseNet

These figures show us that the current generator can build architecture capable of competing the state of the art but it can also generate very bad architecture (10% accuracy). In the following we will see the results in more detail. We will compare the trends in terms of accuracy (Figure 30), training time (Figure 31) and number of epochs (Figure 32) of these different types of architecture.

Figure 30 therefore shows us the accuracy of the test dataset. We can see that there are outliers in particular the architectures which do not manage to converge. Yet, we can notice that the trend between the different types of architectures remains similar.

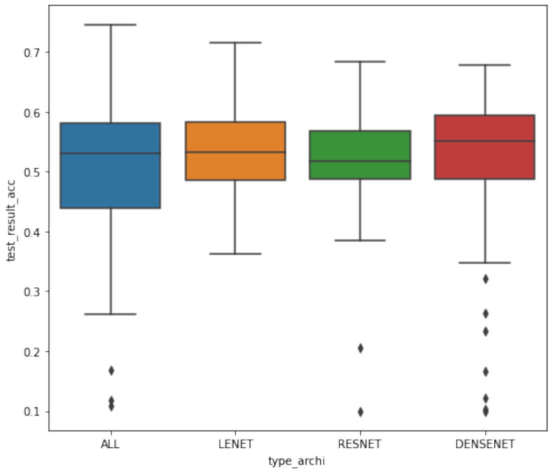


Figure 30: Accuracy on testset per type of architecture

Regarding Figure 31, this shows us the training time for the different types of architecture and we can see that LENET architecture is the one that takes the least time, because it is the least complex architecture among the four unlike DENSENET architecture which must be the most complex since it tries to combine the other three.

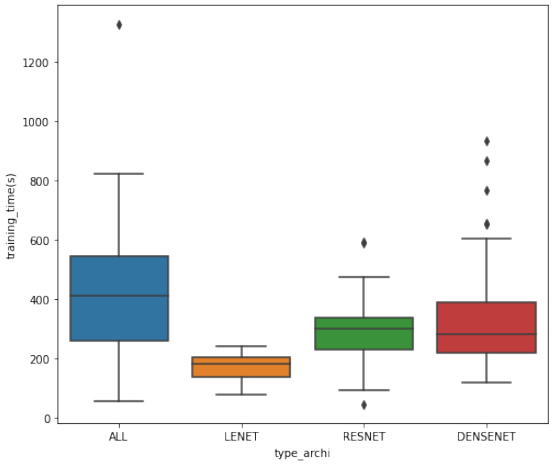


Figure 31: Training Time per type of architecture

Figure 32 shows the number of epochs by type of architecture. These results confirm the conclusions we drew from Section 7.3, the number of epochs we had chosen previously (i.e. 5) was clearly not sufficient as shown in this figure. Most architectures need at least 20 epochs to converge. Obviously, the architectures which are difficult to converge stop before the fact that some architecture has few epochs.

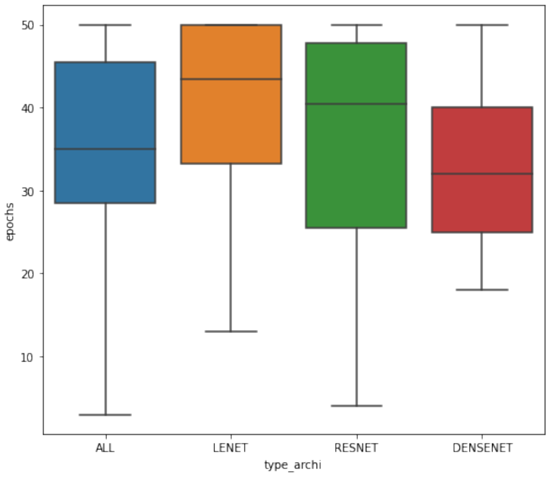


Figure 32: Epochs per type of architecture

To continue this analysis, we compute a confidence interval to see the performance of the current generator. A confidence interval is a range of values that measure the degree of uncertainty or certainty in a sampling method. This interval is bounded above and below the statistic's mean, that likely would contain an unknown population parameter. Confidence level refers to the percentage of probability, or certainty, that the confidence interval would contain the true population parameter when we draw a random sample many times. We therefore computed our confidence interval with a confidence level of 95%. We are therefore 95% certain that most of these samples represent the generator population.

The top graph in Figure 33 represents the confidence interval on the accuracy of the test, this interval is bounded between 0.27-0.75. This means that we are sure that 95% of the generated architecture will give an accuracy on the test set included in this interval. The bottom graph represents the confidence interval on the accuracy of the training and this interval is bounded between 0.27–1.

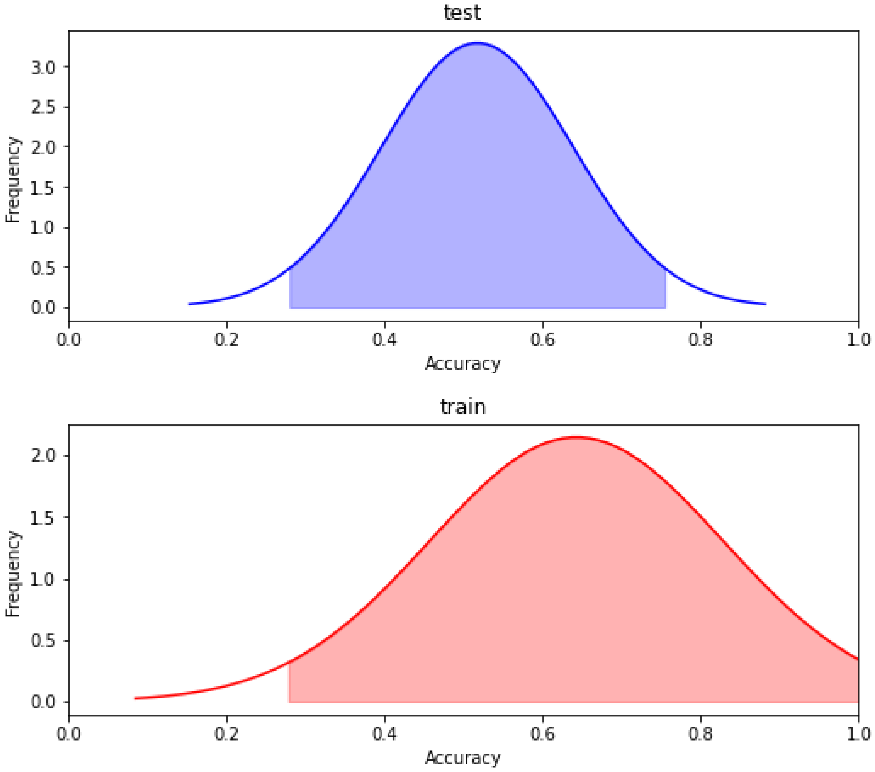
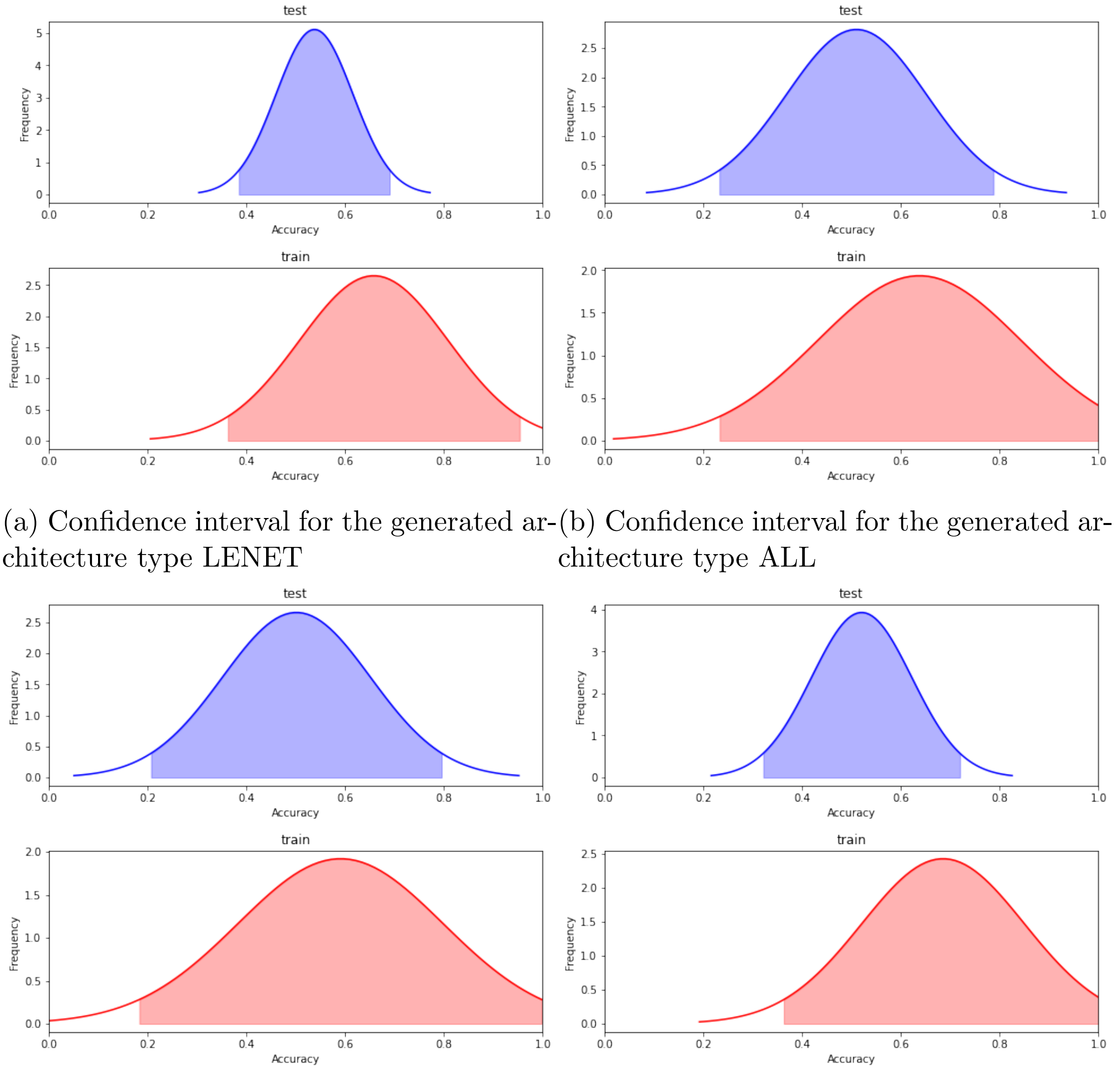


Figure 33: Confidence interval for the generator DenseNet

We also compute a confidence interval for the types of architecture (Figure 34) that we defined (i.e. LENET (34a), ALL(34b), DENSENET(34c), RESNET (34d)). We can notice that the confidence interval of LENET is closer to the mean than the others. We therefore believe that the LENET generated architectures are more stable than the others. Concerning the confidence intervals for ALL and DENSENET, we notice that the values are much more scattered, probably due to their complexity. Some architectures simply fail to converge (see Figure 30).



(a) Confidence interval for the generated architecture type LENET

(b) Confidence interval for the generated architecture type ALL

(c) Confidence interval for the generated architecture type DENSENET

(d) Confidence interval for the generated architecture type RESNET

Figure 34: Confidence interval for all the type of architecture genereted

So the current generator, based on the results seen previously, allows great variability in these constructions both from an architectural point of view (i.e. the arrangement of layers) and from a hyperparameter point of view (i.e. values). We can also increase this variability, for example, add new values for optimisation or loss functions that have not been taken into account in this work. We presented activation functions (See Section 5.2.3) like SeLu or ReLu but why not add eLu or another. We can also reduce this same variability. This flexibility of the generator has allowed it to catch up with state-of-the-art performance (see Figure 29) while others may provide worse performance than random. Nonetheless, they can remain useful to recover knowledge (i.e. new constraints) in order to specialise the generator and make it better.

9 Conclusion and Future work

9.1 Conclusion

The aim of our work is to explore the variability of several advanced convolutional neural network architectures (LeNet, ResNet, DenseNet). In particular, we provide generators that allow cross-architecture variations and help select hyperparameters including slot constraints.

In our work, we propose a methodology/workflow to model the variability of convolution neural network with a state machine. We modelled the architecture using a state machine and the hyperparameters using a feature model. We also structured the architecture and the hyperparameters in our DSL. We show how to build a cross-architecture generator for three popular architectures. Our generator is based on knowledge constraints. Each constraint imposed on the generator triggers improvements towards better results. For the evaluation of the generator, we compare it to the state of the art. Some generated architectures manage to match the state of the art, most stay slightly below and some simply could not converge.

We therefore have a generator capable of adjusting the variability (i.e. increase or decrease) both from the point of view of architectures and hyperparameters. However, the generator still builds architectures that cannot be used except to extract new knowledge or constraints.

In the following, we discuss different perspectives and some ideas that can help to improve the work we propose in this master thesis.

9.2 Future Work

The expressiveness of DSL has been improved throughout this work despite the fact that we can still ask ourselves the question of its expressiveness. We have also the current generator can be further improved either by adding a new architecture (SqueezeNet [13], Xception [3], AlexNet [20] ...) or by adding new constraints. We also noticed that some hyperparameters had requirements. For example the activation function SeLu (Section 5.2.3.2) requires an initialization function named LeCun Normal and used the alpha dropout rather than the dropout. We can further improve our generator, our DSL in parallel and see if these various improvements allow to outperform the state of the art.

Another possible improvement would be to add a spatial search algorithm [12]. The current generator creates an architecture based on constraints but in a random way. One possibility would be to investigate how it can work along Neural Architecture Search (NAS) techniques [21, 31, 5, 6, 2] research area. This domain emerged from various efforts to automate the architectural design process. An example of NAS that we have seen previously (Section 3.2) is the network morphism [30] used by Auto-Keras [15].

Our current workflow is manual and would be interesting to make it more automatic. For instance to extract a new constraint or knowledge of architectures with bad accuracy, we had to do trial and error to find where the error was. It would be interesting to use an algorithm able to find its errors then to integrate them in the generator, in other words, we aim at adapting program repair techniques for neural architectures.

List of Figures

Bibliography

[1] Md Zahangir Alom et al. "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches". In: (2018). URL: https://www.researchgate.net/publication/323570864_The_History_Began_from_AlexNet_A_Comprehensive_Survey_on_Deep_Learning_Approaches.

[2] Yi-Wei Chen et al. "On Robustness of Neural Architecture Search Under Label Noise". In: *Frontiers in Big Data* 3 (2020), p. 2. ISSN: 2624-909X. DOI: 10.3389/fdata.2020.00002. URL: https://www.frontiersin.org/article/10.3389/fdata.2020.00002.

[3] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: (2016).

[4] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2002.

[5] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. "Neural architecture search: A survey." In: *J. Mach. Learn. Res.* 20.55 (2019), pp. 1–21.

[6] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. "Simple And Efficient Architecture Search for Convolutional Neural Networks". In: (2017).

[7] Salah Ghamizi et al. "Automated Search for Configurations of Convolutional Neural Network Architectures". In: *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*. SPLC '19. Paris, France: Association for Computing Machinery, 2019, pp. 119–130. ISBN: 9781450371384. DOI: 10.1145/3336294.3336306. URL: https://doi.org/10.1145/3336294.3336306.

[8] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[9] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (2015). URL: https://arxiv.org/pdf/1512.03385.pdf.

[10] Christopher Henard et al. "PLEDGE: a product line editor and test generation tool". In: *17th International Software Product Line Conference co-located workshops, SPLC 2013 workshops, Tokyo, Japan - August 26 - 30, 2013*. 2013, pp. 126–129. DOI: 10.1145/2499777.2499778. URL: https://doi.org/10.1145/2499777.2499778.

[11] Gao Huang et al. "Densely Connected Convolutional Networks". In: (2016).

[12] Forrest Iandola. "Exploring the design space of deep convolutional neural networks at large scale". In: *arXiv preprint arXiv:1612.06519* (2016).

[13] Forrest N. Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size". In: (2016).

[14] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (2015).

[15] Haifeng Jin, Qingquan Song, and Xia Hu. "Auto-Keras: An Efficient Neural Architecture Search System". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2019, pp. 1946–1956.

[16] Mira Jose. *From Natural to Artificial Neural Computation*. Sandoval Francisco (Eds.), 1995.

[17] *Keras.io*. URL: https://keras.io/.

[18] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2014). URL: https://arxiv.org/abs/1412.6980.

[19] Günter Klambauer et al. "Self-Normalizing Neural Networks". In: (2017). URL: https://arxiv.org/abs/1706.02515.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: (2016).

[21] George Kyriakides and Konstantinos Margaritis. "An Introduction to Neural Architecture Search for Convolutional Networks". In: (2020).

[22] Yann Lecun et al. "Gradient-Based Learning Applied to Document Recognition". In: (1998).

[23] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-Specific Languages". In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892. URL: https://doi.org/10.1145/1118890.1118892.

[24] Klaus Pohl, Günter Böckle, and van der Linden Frank J. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.

[25] Ning Qian. “On the momentum term in gradient descent learning algorithms”. In: (1999).

[26] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: (2017). URL: https://arxiv.org/abs/1609.04747.

[27] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: (2014).

[28] *Tensorflow*. URL: https://www.tensorflow.org/.

[29] Andrew R. Webb and Keith D. Copsey. *Statistical Pattern Recognition, 3rd Edition*. Wiley, 2011.

[30] Tao Wei et al. “Network Morphism”. In: (2016).

[31] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. “A survey on neural architecture search”. In: *arXiv preprint arXiv:1905.01392* (2019).

[32] Matthew D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method”. In: (2012). URL: https://arxiv.org/abs/1212.5701.