NOVA IMS
Information Management School

MAA

Mestrado em Métodos Analíticos Avançados
Master Program in Advanced Analytics

Deep Semantic Learning Machine

Initial design and experiments

Konrad Piąstka

Dissertation presented as partial requirement for obtaining the Master of Science's degree in Advanced Analytics and Data Science

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

This thesis was prepared under the supervision of:

Mauro Castelli

Associate Professor

at Nova IMS

and

Ivo Carlos Pereira Gonçalves

Assistant Professor

at University of Coimbra

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

DEEP SEMATIC LEARNING MACHINE INITIAL DESIGN AND EXPERIMENTS

by

Konrad Piąstka

Dissertation presented as partial requirement for obtaining the Master of Science's degree in Advanced Analytics and Data Science

Supervisors: Mauro Castelli and Ivo Carlos Pereira Gonçalves

April 2021

ABSTRACT

Computer vision is an interdisciplinary scientific field that allows the digital world to interact with the real world. It is one of the fastest-growing and most important areas of data science. Applications are endless, given various tasks that can be solved thanks to the advances in the computer vision field. Examples of types of tasks that can be solved thanks to computer vision models are: image analysis, object detection, image transformation, and image generation. Having that many applications is vital for providing models with the best possible performance. Although many years have passed since backpropagation was invented, it is still the most commonly used approach of training neural networks. A satisfactory performance can be achieved with this approach, but is it the best it can get? A fixed topology of a neural network that needs to be defined before any training begins seems to be a significant limitation as the performance of a network is highly dependent on the topology. Since there are no studies that would precisely guide scientists on selecting a proper network structure, the ability to adjust a topology to a problem seems highly promising. Initial ideas of the evolution of neural networks that involve heuristic search methods have provided encouragingly good results for the various reinforcement learning task. This thesis presents the initial experiments on the usage of a similar approach to solve image classification tasks. The new model called Deep Semantic Learning Machine is introduced with a new mutation method specially designed to solve computer vision problems. Deep Semantic Learning Machine allows a topology to evolve from a small network and adjust to a given problem. The initial results are pretty promising, especially in a training dataset. However, in this thesis Deep Semantic Learning Machine was developed only as proof of a concept and further improvements to the approach can be made.

KEYWORDS

Neural networks, neuroevolution, genetic programming, convolution neural networks, image classification, heuristic search, network topologies

Contents

1. INTRODUCTION

1.1 BACKGROUND AND OVERVIEW

In this thesis, the main focus is on solving image classification tasks, but the idea remains the same for all the problems a neural network can solve. It is relatively easy to create an algorithm that will, with better or worse performance, provide predictions, but how to make sure that it is the best one? Different approaches to problem solving can be seen as optimization problem. The goal is to develop the best-performing algorithm for a given task. Every neural network has a topology, but how to determine whether it is a proper topology for a given problem? The same steps are followed in the standard process of solving classification problems with neural networks. First, the topology of neural networks has to be determined; then, the training of a given network is performed; and finally, the evaluation of its performance can be done. Unfortunately, a wrong decision made in the first step forces us to repeat the whole process. A new topology has to be selected, trained and it has to generate predictions that can be evaluated. No studies would give information on how to choose an optimal topology for any given task. The only approach that can be used is a trial and error method. However, as one might expect, it may be a cumbersome process; it can lead to a selection of a suboptimal topology and, consequently, to suboptimal results. Therefore, selecting an optimal topology can be seen as an optimization problem.

When it comes to complex optimization problems, the heuristic search method has provided satisfying results for various reinforcement learning tasks; thus, in this case, this method seems to be the best option to use. The first ideas for using that approach emerged in the 1990s when the focus was mainly on finding the proper topology for different control problems; however, the approach can also be a helpful way to find a topology to solve standard regression or classification problems. The critical question is: how to gain the most from evolving neural network topologies? Interesting studies have been proposed in "NeuroEvolution of Augmenting Topologies (NEAT)", a paper by K. Stanley and R. Miikkulainen (2002)[1] which exceeds the standard fix-topology approach on a variety of tasks. Although it is possible to produce encouraging results in some reinforcement learning tasks, there are also limitations. As mentioned in the original paper, NEAT only uses crossover operation. It is also helpful in GA for which the solution representation is fixed; however, with GP, it can lead to blobs – a significant increase in size does not significantly improve fitness.

In this thesis, a new algorithm for solving image classification problems is proposed. It is called Deep Semantic Learning Machine (DSLM). It is an extension to Semantic Learning Machine (SLM) proposed in 2019 for regression problem solving[2]. With DSLM, it is possible to solve classification problems as any number of neurons can be applied to the output layer, which assigns a classification to any number of classes. Additionally, to be able to solve image recognition problems, simpler neurons known from CNN architecture are implemented: Convolution Neuron, Pooling Neuron, and a concept of flattening layers. Thanks to those advances and a new mutation method, topology can be extended in both convolutional and non-convolutional parts of the neural network.

1.2 STRUCTURE OF THE THESIS

The next chapter presents a theoretical background for neural networks – from the invention of modelled artificial neurons through all the elements of the neural network up to different possible architectures. Chapter 3 contains details on convolutional neural network (CNN) as a topology that is most commonly used for solving image classification problems. Chapter 4 presents the fundamentals of heuristics search, the elements of each optimization problem and the optimization process. Chapter 5 extends the knowledge from the previous chapter to introduce a specific heuristic method – the genetic algorithms, as it is a base for what was used in DSLM. Chapter 6 continues to build on previous chapters to provide the logical link from Genetic Programming to SLM and Semantic Geometric Mutation. Finally, chapter 7 introduces DSLM and covers the architecture of the model and its functionality. In chapter 8, the results of initial experiments are presented over the different datasets and compared to the current state-of-the-art approaches. In chapter 9, the conclusions and limitations are described, and ideas for future works are given because the experiments are only initial, and the approach can be further improved.

2 NEURAL NETWORKS

2.1 INTRODUCTION

To start from the very beginning, Alan Turing must be introduced, who, along with few other data scientists, is considered the father of artificial intelligence[3]. He was a British mathematician who created "the Turing Machine" and "the Turing's test", an experiment that tries to define artificial intelligence. In his article "Computing Machinery and Intelligence"[4], Turing describes possible problems of artificial intelligence and suggests the definition of an "intelligent machine". In short, the machine can be considered intelligent if the person who experiments is unable to determine whether the discussion he or she is having is a human or a machine. In the article, he also suggests that instead of building an algorithm that can imitate the mind of a grown man, a simpler concept could be built ("child's mind") and trained (into "adult's mind"). It was an exceptionally accurate observation as it is what is happening today, 70 years after the article was published. A small architecture is built, which may be a reference to the Aristotelian "tabula rasa", then it is trained until it can understand patterns that exist in data. One of the hottest terms of 2020 the "artificial intelligence" was coined in Summer 1956 at Conference in Dartmouth by John McCarthy and Marvin Minsky[5] – two brilliant people, who are also considered to be the fathers of Artificial Intelligence (AI), created AI as a field at that conference. Both are the winners of the Turing Award for their contribution to the AI field. McCarthy mainly focused on inventing logic programming (LISP programming language), while Minsky created the first randomly wired neural network learning machine (SNARC – Stochastic Neural Analog Reinforcement Calculator neural net machine)[6] and wrote a book titled "Perceptrons: an introduction to computational geometry"[7]. The book's main topic is a perceptron, a type of AI introduced in the 1950-1960s. He attacked the work of Frank Rosenblatt by pointing out few major limitations of a perceptron, which created controversy as it discouraged further researches on neural networks for several years. The success of this book contributed to an "AI winter", a period in which funding and interest in general in AI were significantly reduced. It is an analogy to a "nuclear winter", which means a significant decrease in enthusiasm. It is a part of a hype cycle when after some time, after the emergence of new technologies, there is a so-called "winter" due to over-inflated promises and unrealistically high expectations of users and stakeholders. Fortunately, as it is with seasons in nature, if technology has enough potential it will renew the interest in the following years.

The phenomenon of an "AI winter" has been observed several times already. The most recent one lasted from the late 2000s to the early 2010s. It ended thanks to advances in both hardware

and algorithms that allowed achieving successful results. AI models started being useful in various industries. It is evident that for the past several years, the interest in AI has grown, not only in the academic field but in almost every sector of industry.

2.2 NEURON

Biological neurons were studied to a great extent. However, from the perspective of data science, the key observation was that a complex description of a real neuron (very complex in nature) could be significantly reduced to a much more straightforward representation. The simplification was about reducing the number of information processing rules which occur in a neuron. It can be abridged to just a few simple dependencies.

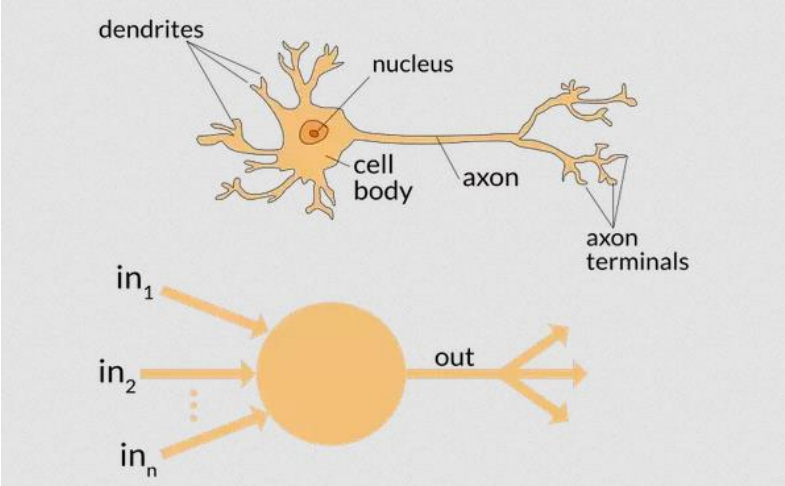


Figure 2.1: Comparison of biological and artificially created neuron

A simplified neuron – the one presented at the bottom of Figure 2.1, allows to recreate real networks as a computer implementation and preserve the remarkable and fruitful properties of a real neuron. The main feature of a biological neuron is conserved; it can have many input connections (dendrites) and only one output connection (the axon). Moreover, the primary function remains the same: processing input into output information using the "knowledge" it gains during the training phase. This operation comes down to a multiplication of input signals by weights of the neuron, then summed. However, it is the ability to gain "knowledge" by the training process that makes neural networks powerful. There are no components like a synapse, collaterals, and many other unnecessary forms in modelled neurons.

However, despite a heavy reduction in complexity, it is still possible to achieve complex and interesting behaviours. It makes you wonder how amazing our real biological neurons are together with the whole neural network. Once the signal is aggregated, an additional component called bias is added. As the value of bias also can be changed during the training process, it can be seen as an extra weight that does not come from any other neuron. In practice, it is often a dummy input neuron with a fixed weight that equals 1. Bias adds flexibility to the learning process. In short, without it, all of the activation functions have to traverse the origin of the coordinate system, which is a limitation to a training process. The last component of a basic neuron structure is the activation function. That component is the main distinction between linear networks like ADALINE (adaptive linear), which do not have the activation function, and other networks that have richer opportunities thanks to the activation function; for example, a multi-layer perceptron (MLP). The goal of an activation function, sometimes also called a transfer function, is to define the output of the given input.

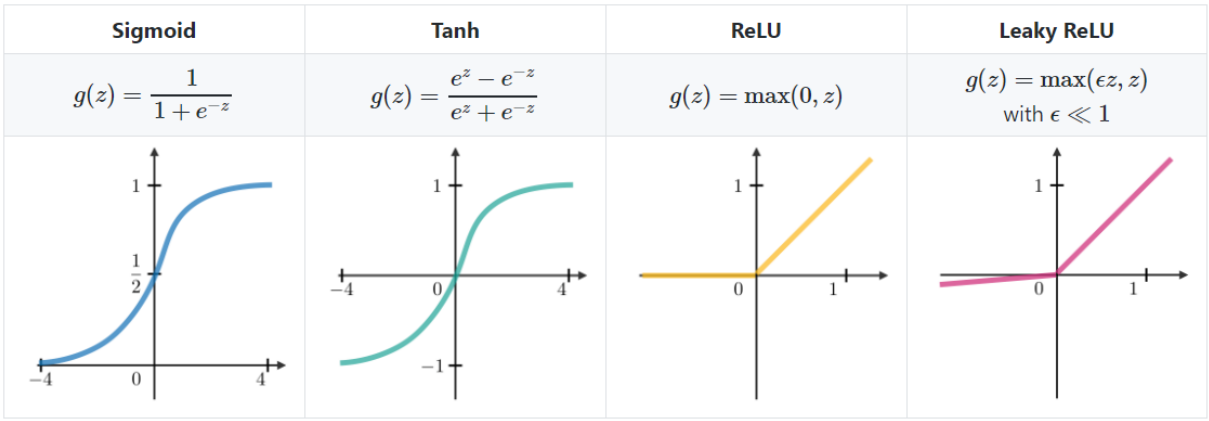


Figure 2.2: Most common activation functions

The name comes from its function in the neural network as it can activate or deactivate the neuron. Many functions can be used here, but to allow the network to reach its full potential (which is to solve nontrivial tasks using a small number of nodes) the nonlinear functions should be used. In figure 2.2, the most commonly used activation functions are presented. A linear activation function is generally used in the output layer of a regression task, but other functions have wider use as they are non-linear. With advances in deep learning, more complex models of neurons can be used, but they are still nowhere near the real ones!

2.3 PERCEPTRON

In 1958, Frank Rosenblatt invented the first neural network called Perceptron[8]. In data science, the perceptron is one of the algorithms for supervised learning to solve classification tasks. It is the simplest neural network that contains only one layer of hidden neurons (with at least one neuron in it) and can perform binary classification. The algorithm can decide which specific class a given observation (instance) belongs to, based on the input characteristics. Input data is usually a vector of real numbers that describe different features of a given observation. With only one layer it is a linear classifier. A classifier is making decisions based on linear predictors by multiplying weights and the input vector instance characteristics. Rosenblatt was very optimistic about the perceptron's potential. "The embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." – he writes in his statement reported by The New York Times. Unfortunately, despite the initial high potential and optimistic results, it was proven that a perceptron is unable to recognize more convoluted patterns.

\

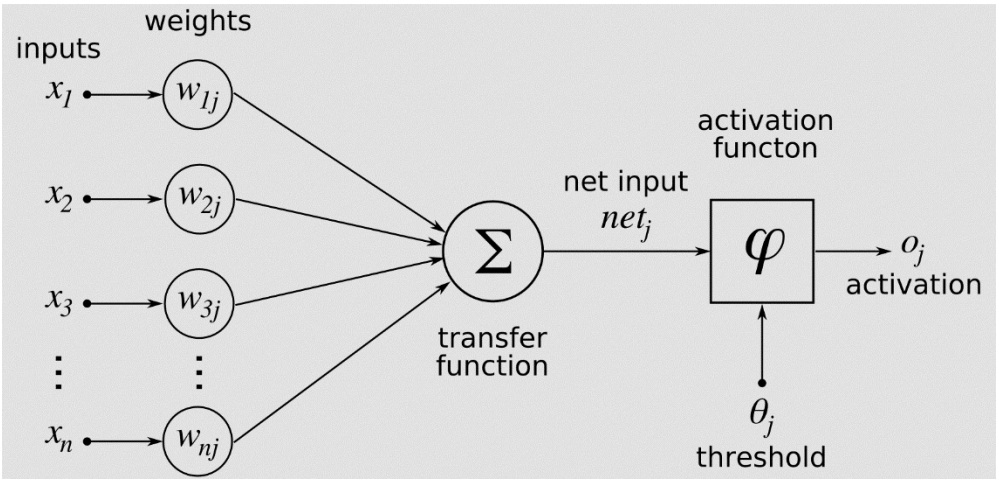


Figure 2.3: Schema of neuron with all components

In figure 2.3, a complete model of a neural network can be seen. It is more complex than the one in figure 1, but thanks to the activation function, a neuron gives the neural network the capability to discover a non-linear pattern in data. That was one of the most significant enhancements to the model of a neuron, which allows achieving significantly better performance with more complex problems.

2.4 NEURAL NETWORKS

An artificial neural network is a mathematical structure built from elements called neurons or nodes[9]. The inspiration for this concept was the nervous system of the human body. The connections between biological neurons are modelled as real numbers called weights. A positive value means that an excitatory connection and a negative number reflects inhibitory connections. Input information, also a vector of real numbers, is multiplied by weights and then summed. This operation is described as a linear combination. Often the activation function is being used to control the amplitude of the output. Artificial neural networks can be used in predictive modelling tasks for regression and classification problems, and their basic form is only capable of supervised learning. However, various architectures can be used to build a neural network to solve a whole range of different data science problems. Another important observation that helps overcome the limitations of perceptron was that hidden neurons could work together and create a more sophisticated concept called a layer. It is worth mentioning that the natural structure of a neural network is much more convoluted, but in simplified representation, it can be viewed as consecutive layers of neurons[10].

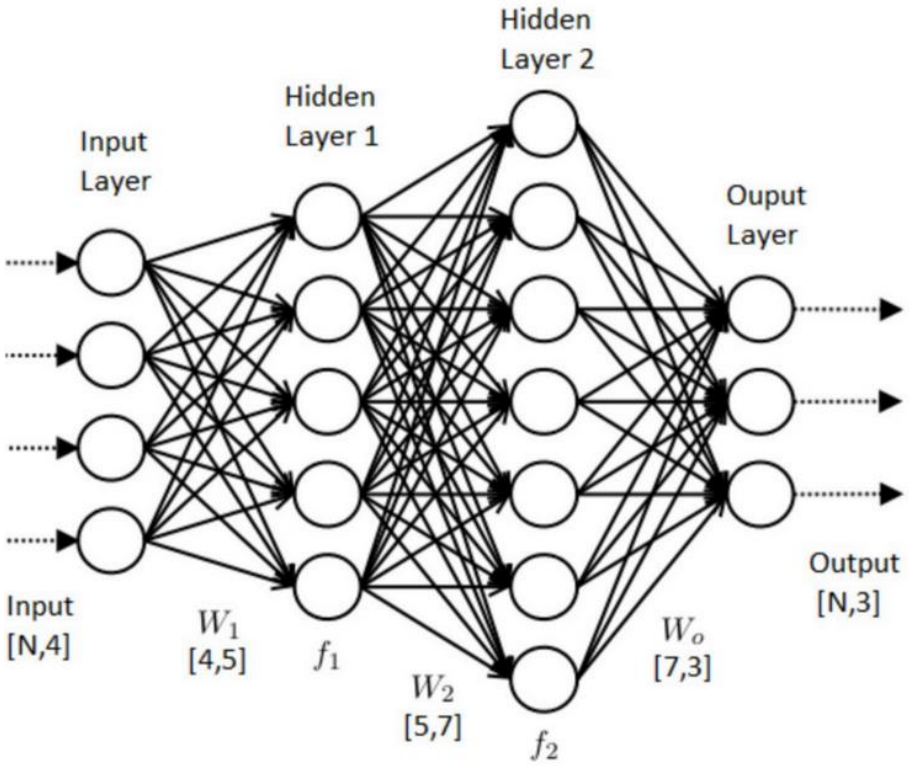


Figure 2.4: Model of simple feedforward dense artificial neural network

Such layered structure can be easily be built in a computer program. Nowadays, in one of the most popular deep learning frameworks in python "Keras", the concept of a layer is a key component. With this approach, there is no need to build each neuron separately as they work in the same way if they lay on the same layer, making building artificial neural networks much more straightforward. Users only need to specify how many neurons should be in each layer and input shape, and the rest is being taken care of by the framework. Other approaches also can be used, but most of the topologies of the neural networks have the same three kinds of layers. As shown in figure 2.4, the first layer in a network is the input layer, then it is followed by several hidden layers, and the last layer is the output layer. The connections between neurons play essential roles in how the neural network operates; it will be discussed in detail in the next section. However, it is vital to notice that connections occur only between subsequent layers in the most basic implementation. All neurons from one layer are connected with all neurons from the next layer. That is a fully connected structure called "dense". It is not obligatory to use such an approach, as the connections in a real brain are much more complex; however, simulating that in a computer program will exponentially increase the model complexity, making it extremely big and hard to train. This approach allows to spent minimum time to define the topology of a neural network. However, as one can imagine, it probably will very rarely happen to be an ideal structure. Looking for a perfect balance between complexity and the degree of mapping reality can take a long time. Another argument favouring using the default approach is the neural network's capability to decide which connections are more important than the others during the learning process. More important connections receive higher weight (higher in an absolute sense as the weight can be positive and negative) than the non-important ones, which can be set to zero.

2.5 How Neural Networks works

Each neuron has some kind of "knowledge" represented by a value of weights and bias, and it can process information. Although it looks as if a single neuron had quite limited capabilities; when it is a part of a well-designed network of neurons, it can solve some complex tasks but only when the whole network is working together. As it is usually impossible to pinpoint which neurons are responsible for which step of the process of providing the solution, they all contribute to producing the final output. To generate predictions, the neural network has to be trained; however, it is essential first to understand how already trained networks generate predictions that will make the understanding of a training process more manageable.

2.5.1 Feedforward part

In the input layer (see figure 2.5), information is only read, not processed yet. For example, to solve the problem of predicting the price of an estate, in the input layer, all the features of the given problem instance are read; it could be any feature like the size of the house in square meters, year of construction, information whether it is close to a city centre or not, and many others. In the next step, the input information is moved to hidden layers. They use their weights, bias, and activation function to process the information they received. After that, the initially transformed information is passed forward to the next layer until it reaches the output layer. The transformed information that is returned by the hidden neuron is tough to interpret by human reasoning, that is why neural networks are usually referred to as black-box algorithms. Although the math applied here is not complex, it is still tough to say which neurons are responsible for what. It is possible to try to interpret those intermediary results, and many advances have been made to improve that interpretation. However, for now, the official statement is that the intermediary outputs can not be clearly interpreted. Once the information reaches the output layer, it is finally possible to interpret the results as it is the final predicted value of the real estate in the example mentioned above. The example was made on the regression problem, but the process remains the same for classification tasks. The only difference is that in the regression task, the output is a single real value, and in classification, the binary vector is returned with one as an indication to which class a given observation should belong.

2.5.2 Error Function

Error function allows users to calculate the size of an error made by the predicting model[9]. Error is calculated as a difference between the expected and actual value for a given problem instance. It is a required step in training a neural network. Evaluation of the performance is crucial for the training process, especially if backpropagation is the selected training method. It allows to track the progress made by the network (figure 2.5) and what is most important, it gives the indication when the training should stop as the model begins to overfit. The mentioned optimization function is the minimization function, as the goal is to make the smallest possible error. The most common error functions are Mean Square Error or Mean Absolute Error for regression problems and Maximum Likelihood and Cross-Entropy for classification tasks.

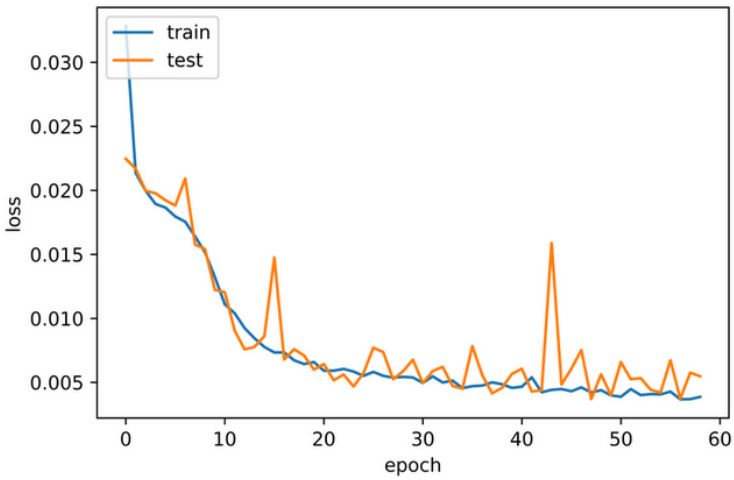


Figure 2.5: Visualization of how the error function is closing to zero as the training continuous

2.5.3 Backpropagation

Backpropagation is a training technique for neural networks introduced in 1986 by Rumelhart, Hinton and Williams in their work titled "Learning Representation by back-propagating errors". Backpropagation is an optimization method of updating neurons' weights for better performance of the network[10]. It is a concept that allows neurons to "learn from mistakes". It is the generalization of the "Delta Rule"[11]- the approach used in training of the Perceptron. The full name of the process is "backward propagation of error", which utilizes the concept of gradient descent. Given the network and the error function, the most common approach to teaching the network is calculating the gradient of the error, which helps update the weights of neurons that allow the network to make smaller mistakes for the same problem instance. That process assures that an algorithm is moving slowly but steadily towards better performance. The names come from the algorithm's ability to propagate an error back to the previous layer as it is impossible to calculate an error specifically for a given hidden neuron. It starts with calculating an error at the output layer. This step is very straightforward; given the error function, it is possible to calculate by how much the prediction made by the network is off from the expected result and how to update weights; however, that is not enough to have a well-trained network. An error has to be propagated to every layer to achieve an efficient training process. As it is impossible to determine by how much each hidden neuron was off with its output, the Delta Rule is not enough to train a Multi-layered Feedforward Network. For hidden neurons, the modification is done by propagating back the error from the next layer to the

previous one (an error is propagated from the last layer of the network to the first one). The error for hidden neurons is equal to the sum of the error from neurons from the subsequent layer. The formula to calculate the error is as follow

$$w_{ij} = w_{ij} + \Delta w_{ij} \qquad \Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

An essential drawback of the backpropagation is that it can get stuck in a local minimum. Although there are ways to avoid that, for example, using a proper learning rate or momentum, those are possible problems of which a creator of a network has to be aware. Another limitation is that once the network learns something, any additional learning leads to significant "forgetting" of the existing knowledge. The need to address those problems raises the question of whether it is the best approach to train neural networks, or maybe there are some better alternatives. Fortunately, although it is a vastly popular approach, it is not the only one that can be used. Those limitations provided additional motivation to develop an alternative approach that is used in SLM and DSLM.

2.5.4 How network's structure affects what a network can do?

It is essential to understand how different are neural architectures with their abilities to solve various tasks. The general rule is that the more complex task the more neurons the network should have. In theory, with more neurons and more hidden layers, the network should be smarter by discovering more complex patterns, but in practice, it is not always the case. It is tough to select an optimal topology of a network, especially for more complex tasks, which could be crucial in achieving the best performance. While with easy, almost trivial tasks, topology is not that important as the proper training is enough to achieve good performance (tests performed by Rosenblatt in 70' were so good that at the beginning no one believed in it). However, even with those tasks, a correctly selected topology can significantly reduce the training time and achieve better results. A simple artificial neural network with any topology sill might struggle with providing satisfactory results with very complex problems. There is a need for constructing an architecture with different types of neural networks. However, it remains true that once tasks get harder the suitable topology plays a much more important role in achieving a satisfactory performance. Unfortunately, with a traditional approach of constructing neural networks, you have to select a topology before the training begins. Any change requires one to start over, which can be time-consuming and computationally expensive.

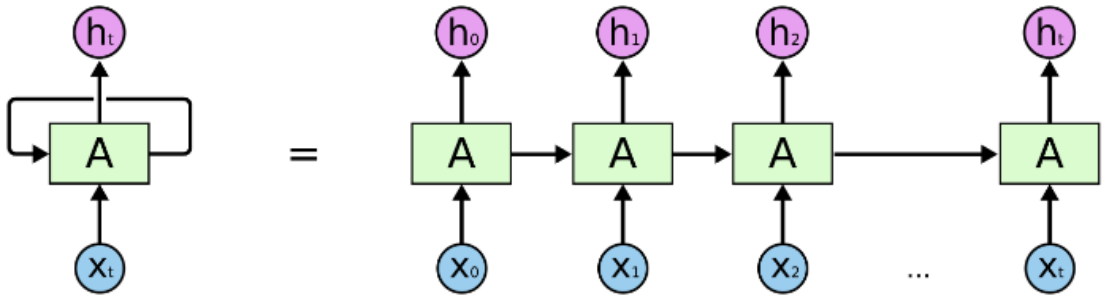
2.5.5 Types of neural network

Since the invention of a perceptron, the deep learning field has expanded greatly. Currently, there is a few major types of network architecture usage of which depend on a task it has to solve[12]. Those types of architecture differ in the use of different types of neurons. The main groups are:

a) Feedforward Neural Networks
b) Recurrent Neural Network
c) Convolutional Neural Network
d) Unsupervised Neural Network

Feedforward Neural Networks are of the primary type, and all components of that network are already covered. It is a simple architecture with an Input Layer, few Hidden Layers and an Output Layer. It is capable of solving both classification tasks and regression tasks. Unfortunately, it is unable to understand time-dependent data; for that, a Recurrent Neural network is recommended.

Recurrent Neural Networks are an extension of standard feed-forward neural networks; they have a new type of nodes – neurons with an additional component of recurrent connection.



An unrolled recurrent neural network.

Figure 2.6: Recurrent Neural Network Loop

The recurrent connection span is adjacent to time-steps. It allows the network to understand the concept of time. Recurrent neurons have a connection with themselves. Such nodes not only receive inputs from the previous layer but also from themselves, from the past. In figure 2.6, such a connection is visualized. On the right side, the unrolled version of the recurrent connection is presented. At the time *t*, the information is available to the neuron, namely: the

input from *t* observation and also a state of the neuron from *t-1*. Thanks to that structure, it can connect the new piece of information with knowledge from the past to detect time-depended patterns. The most famous recurrent networks are Long Short Term Memory (LSTM) introduced in 1997 by Hochreiter and Schmidhuber, and Gated Recurrent Unit (GRU) model introduced in 2014 by Kyunghun Cho et al. [13]

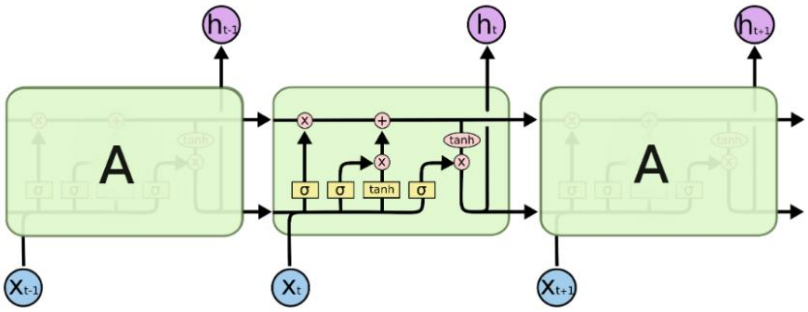


Figure 2.7: LSTM Memory Cell

In figure 2.7, the details of a recurrent neuron are presented. That figure should help understand what pieces of information are available to the neuron and how that information is being transformed. An LSTM architecture can overcome the problem of vanishing gradient by making the model capable of understanding the long and short-term dependencies.

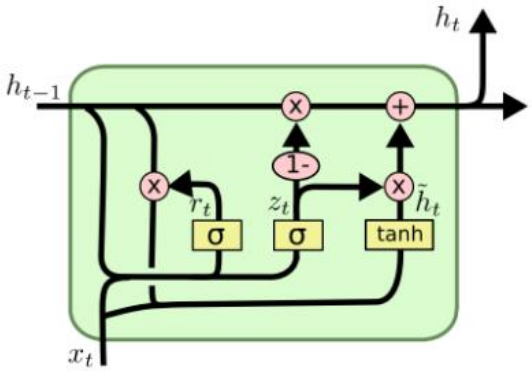


Fig 2.8: Gated Recurrent Unit Cell

In figure 2.8, the GRU architecture is presented. Although it is similar to the LSTM, GRU uses the hidden state instead of the cell state, and it only has two gates: the update and the reset gate. Compared to the LSTM, it has a lower number of parameters to set, making them easier to tune, yet it still possible to achieve the state of the art performance. Such networks can work on time series data by solving tasks like series forecasting (stock prices from historical data), speech recognition, and many other problems.

Unsupervised neural networks are a broad category of architectures. Those models do not require to have labelled data. They can detect patterns that exist in the data without supervision. Examples of those models are Autoencoders, Self-Organizing Maps, and recently very popular Generative Adversarial Networks. These topics are out of the scope of this work, so proper literature should be referred to.

Convolutional Neural Network

What is unique about the Convolutional Neural Network (CNN) is that they can learn higher-order features from the input data thanks to convolutions. They are especially good at solving more convoluted problems like image or sound recognition, object detection, and many others, thanks to the ability to understand visual and audio data[14]. A promising performance in visual recognition was a milestone in the field of autonomic vehicles or medical image analysis like MRI or X-Ray.

3 CONVOLUTIONAL NEURAL NETWORK

3.1 INTRODUCTION

The inspirations for CNN came from biology, the same as for neural networks. Researchers were trying to imitate the visual cortex of animals. As it is in nature, the cells in the visual cortex are reacting to the input only from a small subregion of what the eye can see. It is called the visual field. With many cells combined, the subregions are also joined to cover the entire receptive field. Those cells perform well utilizing the strong spatially local correlation in the kind of visual input the brain processes. The visual cortex acts as a filter over the input data. That is why, unlike other network architecture, the input does not need to have just one dimension. It usually has two, three, or even four dimensions.

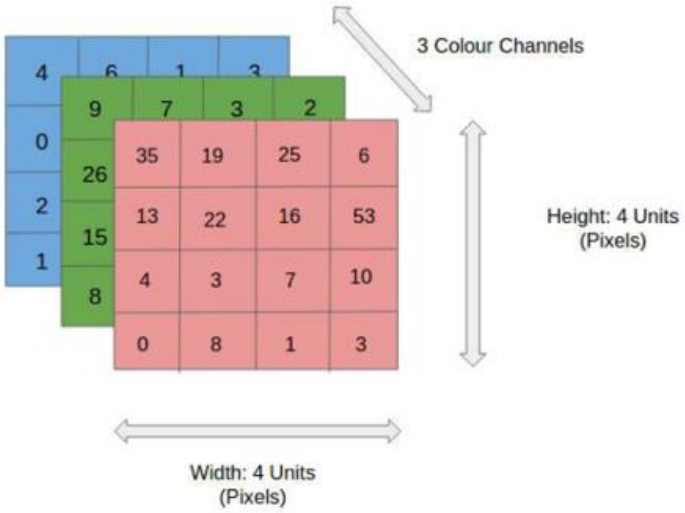


Figure 3.1: The cross-section of an input volume of size: 4 x 4 x 3.
It comprises the 3 Colour channel matrices of the input image

In figure 3.1, there is a decomposition of the three-dimensional input. Each pixel of an image can be described with the RBG colour model as an amount of red, green, and green light added together to reproduce any colour. One example of input data could be an image that is 32 pixels wide by 32 pixels in height with three RGB scale colour channels. That makes 3072 weights per neuron. That number has to be multiplied by many neurons in each layer and then again by, at least, a few layers, so the architectures do not become too big, which could significantly slow down the training process. If the more realistic size of the picture is considered, a network with billions of connections would be created.

That clearly shows how rapidly the size of the network can increase when it comes to image data and how difficult it would be to train such a network. That problem became a motivation for searching for a different way of solving such problems and developing an architecture that would be much smaller but still performs well.

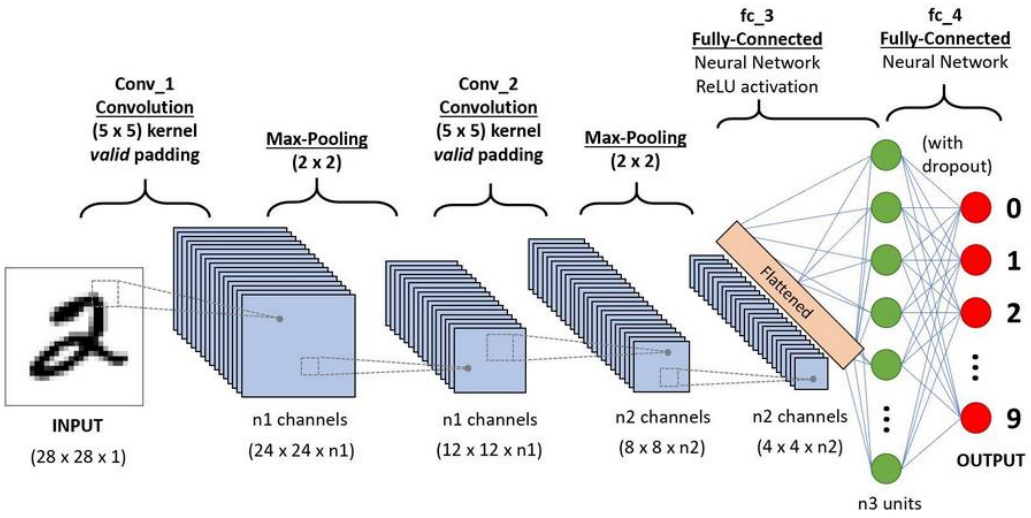


Figure 3.2: Simple CNN architecture

3.2 CNN Architecture

The convolutional network introduces two types of neurons. They were created specifically to work with multidimensional data: the convolution and the pooling neuron. In the CNN architecture, a flattening layer exists to connect multidimensional neurons with one-dimensional traditional hidden neurons; those neurons unravel the data and pass it to the fully connected hidden layer. The model still processes information sequentially, as it occurs in a traditional feedforward neuron network.

3.2.1 Convolutional Neuron

The name correctly indicates that it plays a crucial role in CNN architecture. To avoid having too big of architecture, every convolutional neuron is connected only to a small region of the input volume[13]. The dimensionality of this area is referred to as the captive field size of the convolutional neuron. Convolutional layers are focused on the use of learnable kernels. The kernels are pretty small in size, compared to the input data size, but they spread along the entirety of the input. When input data reaches the convolutional neurons, it convolves each filer across the special dimensionality of the input to create a 2D activation map.

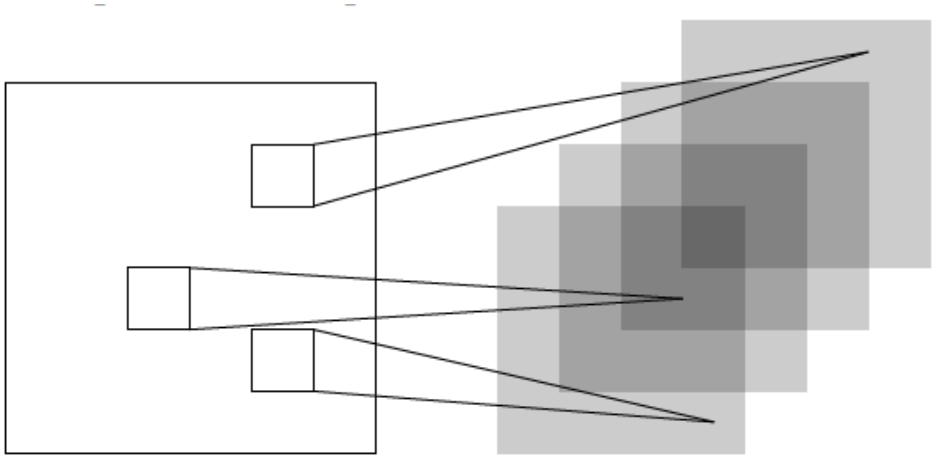


Figure 3.3: Illustration of a single convolutional layer

In figure 3.3, the example of the process is presented – their small subsection of the input that is processed at a given moment and is multiplied element-wise by a kernel. Results are summed to a single value output[15].

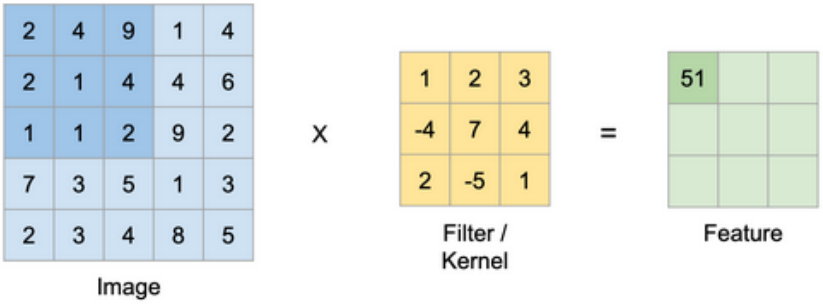


Figure 3.4: Example of the convolution process

That allows reducing the complexity of the model and the optimization of its output significantly. To create the convolution neuron few hyperparameters like the stride, the padding, and the size of a kernel have to be set. All previously mentioned parameters help control the size of the output of the neuron. It is possible to minimize the number of neurons used in the network significantly but it will also reduce the pattern recognition capabilities of the model. With stride parameters, it is possible to control the level of overlapping of the receptive field; for example, with stride equal to 1 results will be highly overlapped and have a higher value, which will produce the output of lower spatial dimensions. Zero-padding is the process of padding around the edges of the input, as presented in figure 3.5.

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 3.5: A zero-padded 4 x 4 matrix becomes a 6 x 6 matrix

It allows controlling how much information from around the borders will be lost to reduce the complexity of the output. To calculate the size of the output the following formula can be used:

$$\frac{(V - R) + 2Z}{S + 1}$$

Where:

- V stands for the input volume size (height x width x depth)
- R represents the size of the kernel
- Z is the amount of zero-padding
- S is a stride

3.2.2 Pooling Neuron

With real size pictures and reasonably set parameters, the model can still be enormous, so the further reduction of input sizes throughout the model is needed, yet still the most valuable information should be preserved. The goal of the pooling layer is to continue reducing the dimensional size of the representation, thus further reduce the computational complexity of the model[16]. The pooling layers operate over each activation map in the input image and scales its dimensionality using the aggregation function (like “max”, “min”, or “avg”). As shown in figure 3.6, the pooling operation happens over the input to produce an output reduced in size, yet preserving interesting patterns in the data. The max-pooling layer is most commonly used, as with min operation, the information tends to vanish over several pooling operations.

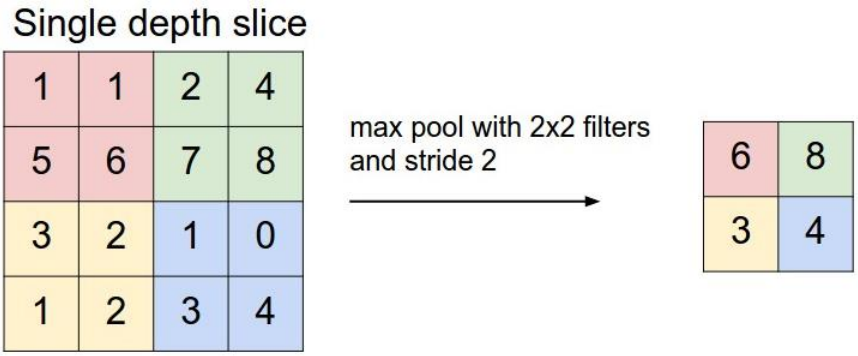


Figure 3.6: Max Pooling operation

The pooling layer also has the kernel, size and stride parameter and it can be controlled. With 2 x 2 kernel and stride equal to 2, it scales the activation map down to ¼ of the original size (weight and length-wise; the depth remains the same).

3.2.3 Flattening Layer

The goal of the flattening layer is to convert the three-dimensional input into a one-dimensional so standard feedforward layers could further process it. At this stage, there is no activation function being used, nor any calculations are being done. It is just a process of unravelling the information.

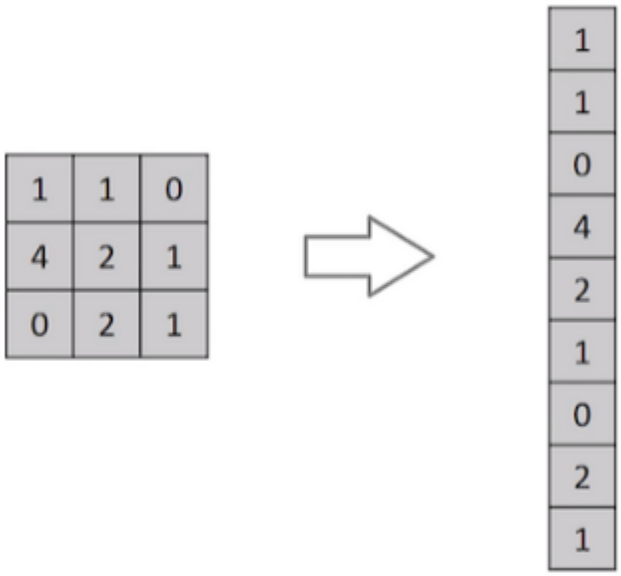


Figure 3.7: Flattening layer

3.2.4 Activation Functions in CNN

CNN also introduces new non-linear activation functions that perform very well on computer vision tasks. The most commonly used function is Rectified Linear Unit (ReLU). It is half restricted (from bottom) yielding 0 for any value of x less than 0. For values of x greater or equal 0 it is f(x) = x. The function and its derivative are both monotonic. The drawback of this function is that any negative values in the input data are immediately turned into 0, so all information from negative values is lost. In figure 3.8, the visualization of ReLU (left side) and Leky ReLu (right side) are presented.

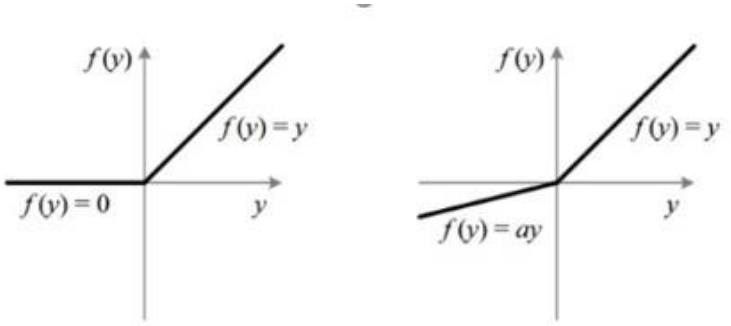


Figure 3.8: Activation Functions used in CNN

That was the motivation to introduce the leak. The leak means that the slope of the function for values of x below 0 is slightly positive which changes the range of possible output values from [0, +inf) to (-inf to +inf). Usually, the slope of a (See figure 3.8) is 0.01, but it can be any low positive value.

3.2.5 SoftMax activation function

It is a unique activation function that turns numbers (logits) into values that can be interpreted as a probability because all values after transformation are summed up to one. So the outputs of the output layer are a vector of values that can be interpreted as a probability of the data instance belonging to a given class[10]. The evaluation process takes the exponents of each output neuron and then normalizes the numbers by the sum of the exponents, which makes the entire output vector add up to one.

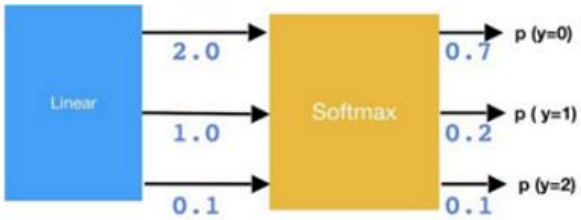


Figure 3.9 Exemplary process of SoftMax calculation

To calculate the output this formula is used:

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

Where:

- K is the number of classes
- j is a given class from K classes
- z is a vector of input to the Output layer

As shown in figure 3.8, after the SoftMax transformation, it is clear that the given data point most likely belongs to class 0. It makes the training process much more straightforward than reviving the binary output version, as it is impossible to assess how odd the predictions are. With the SoftMax activation function, it is advisable to use categorical cross-entropy as the loss function for multiclass classification problems. The SoftMax function is used in the well-known VGG-16 architecture (CNN that performed well in ImageNet competitions.)

4 OPTIMIZATION PROBLEMS

4.1 INTRODUCTION

A better and worse solution can be found for almost all problems. To classify the solution as good or worse, the cost of solving the problem or the benefits of the solution has to be quantifiable. Possible costs of the problem can be money or time which have to spend for a given solution, and in the same way, the benefits can be quantified as saved resources thanks to the implementation of a solution. Finding a solution to a trivial problem is easy, but it can be a challenging process with a complex one. However, suppose the difference in the number of resources that have to be used for solving a problem between a good and bad solution is significant. In that case, it is worth looking for an optimal solution. Optimization problems are very common in many disciplines and domains[17]. The usual goal is to find the optimal or near-optimal solution with respect to some objective, quantifiable goal. As the benefits of finding the optimal solution can be significant, a separate field of search algorithms was created. Many researchers dedicated their works towards finding an algorithm capable of finding the optimal solution to most complex problems. Although different approaches can be used to find the optimal solution, heuristic search has become very popular as it is possible to get promising results even to complex problems. The main classes of optimization problems are discrete problems; for example, assigning workers specific tasks to optimize efficiency. Another main type are continuous problems with an unlimited number of alternatives; optimal values for decision variables have to be determined, for example, allocation of money between research teams or time between tasks.

4.2 SOLUTION PROCESS

There are many possible solutions for optimization problems, and a researcher has to select only one. Choosing a particular, available solution will have some impact that can be measured with some quantifiable variable. The goal is to find either the minimum or a maximum value of a variable depending on the problem specification. Usually, not all of the theoretically possible solutions are valid; problem-solution is often bounded by some constraints like law, available budget, technical limitations. Therefore, the process of finding an optimal solution is twofold – one requires us to find the solution that either minimizes or maximizes some objective function, and the other ensures that the solution is feasible.

To sum up, optimization problems could be described with the following characteristics:

- Many different solutions are available
- Some limitations exist that reduce the number of available solutions
- Each decision has some effect that can be quantified
- Evaluation of the solution allows distinguishing better and worse solutions

4.3 PROBLEM CHARACTERISTICS

A problem instance is a concrete representation of a problem with a specific characteristic that makes it different from the rest. It comes from a broader, more general group of problems. A problem instance has only one search space, fitness function, and other attributes that make this problem tangible. Assigning a certain value to x can be denoted as a solution and all solutions together create a set of solutions X, where $x \in X$. An instance of a given problem can be described as pair (X, f), where X is a set of possible solutions $x \in X$ and $f: X \rightarrow R$ is an objective function that assigns a real number to every element x of the search space. A solution is valid if all constraints are satisfied.

An optimization problem is a set of problem instances where all of them share the same properties, and all of them are created similarly. Usually, a particular problem instance is considered to be an instance of a more general group of optimization problems. Although the same approach can be suitable for many problem instances, they should be treated separately. Target variables can be either discrete ($x \in Z^n$) or continuous ($x \in R^n$). All problems can be split into two categories: minimization or maximization problem to find an $x* \in X$ where:

$$f(x*) \geq f(x) \text{ for all } x \in X \text{ (maximization problem)}$$
$$f(x*) \leq f(x) \text{ for all } x \in X \text{ (minimization problem),}$$

x^* is a globally optimal solution for a given problem instance[17].

One of the most common examples in the search algorithms field is a travelling salesman problem (TSP). Given the list of cities and calculable distances between each pair of cities, the goal is to find the shortest possible road that makes it possible to visit all the towns once, and only once, and that leads back to the origin city[18]. This problem as an optimization task was formulated in 1930, but already over a hundred years earlier, it was mathematically formulated by Hamilton and Kirkman as a recreational puzzle; until today, it is an intensely studied problem in optimization.

Attributes of optimization problem:

a) Representation – blueprint for an individual
b) Search space – all possible solutions to the problem
c) Fitness function – objective function to evaluate a solution
d) Neighbourhood – locality from a solution

4.3.1 Representation

The representation of an individual can be a list of all the cities put in order that a travelling salesman should visit. An exemplar solution for the aforementioned problem could be [2, 3, 5, 1] as if the digits were ids of cities. In figure 4.1, exactly that representation is presented.

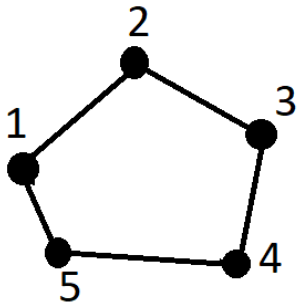


Figure 4.1: Traveling Salesmen representation

4.3.2 Search space

Search space refers to a collection of all possible solutions to a given problem and some notion of a distance that is calculated between potential solutions[22]. The search space size cannot always be calculated, and for a continuous optimization problem, there can be an infinite number of combinations. However, whenever it is possible to be calculated, it can hint at how difficult the problem is. Given the problem above, the search space size would be all possible combinations of arranging the five cities. The size of a search space can be calculated with this formula: ½ *(n-1)! there are 12 possible solutions for five cities.

4.3.3 Neighbourhood

For every problem instance, it is important to define the neighbourhood as it allows to determine which solutions are close or similar to each other. A neighbourhood can be defined as mapping function N(x), in which x∈ X assigns a set of solutions y ∈ X to each solution, as they are within close proximity to the solution x, those solutions can be named neighbours, and all other solutions are further away. Many different mapping functions can be used for every

problem instance, thus the neighbourhoods are a very subjective concept. Once the neighbourhood is defined, optimization algorithm can use it in a search for an optimal solution. Many algorithms look around for better solutions only within the neighbourhood. Given the unimodal search space, the global optimum will be reached within the finite number of iterations.

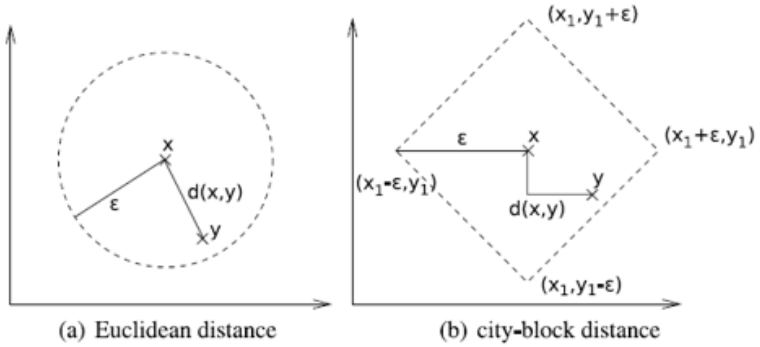


Figure 4.2: Examples of neighbourhood when different distance metric is selected

A more formal definition of local and global optimum will be defined in the fitness landscape section of this chapter. The neighbourhood can be either discrete or continuous depending on the category of a target value. An example of a neighbourhood mapping function can be the Euclidian function (see figure 4.2) for a continuous problem and discrete Hamming distance between bit representations (see figure 4.3) can be used.

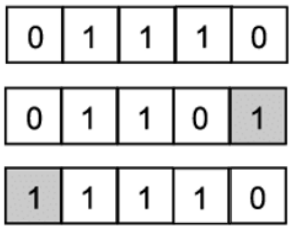


Figure 4.3: Bit representation neighbours

4.3.4 Fitness function

The fitness function is a particular case of objective function that is used to evaluate solutions. Usually represented as minimization or maximization function f(x) where x is a solution and f(x) is a real number. Given the aforementioned problem and assuming the Euclidian distance is used, the fitness function could be calculated as a sum of all distances that a travelling salesman has to go through.

4.3.5 Fitness Landscape

In evolutionary biology, the fitness landscape is used to visualize the relationship between every point in a search space and the fitness function value, which reflects reproductive success. A fitness landscape (X, f, d) of a given problem, an instance is a combination of search space with a set of solutions x ∈ X, fitness function that evaluates all solutions, and the distance measures *d*.

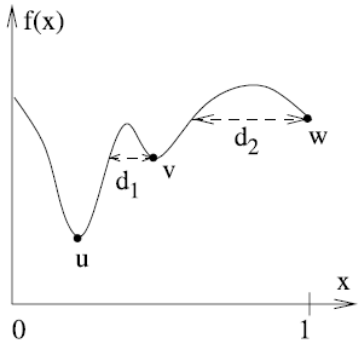


Figure 4.4: Dependence of the local optimum on the definition of N(x)

As it is presented in figure 4.4, u is a global optimum (global minimum for this problem) and v represents the local optimum, with a sufficiently small neighbourhood is the best solution (you cannot move in any direction within the neighbourhood to reach a better solution). It is true that with a neighbourhood with a size of a whole search space there is no longer problem with local optimum, but then it is just looking at each solution with brute force. That is why proper balance in neighbourhood size has to be preserved. The value of an objective function can be assigned to each solution, and all of those pairings can be mapped into a graph. In an optimization problem with not more than two parameters, it is possible to create a plot that allows us to understand how the fitness landscape looks like – whether it is a smooth slope with one peak (unimodal fitness landscape) or it is very rugged with many local optimums. Once the fitness landscape is plotted, it is easy to access the difficulty of a problem, but real-world problems usually have dozens or even hundreds of parameters that have to be set. Such problems can not be visualized, but in any number of dimensions, the principle remains the same, the more rugged the landscape, the more difficult it is to find the global optimum. As presented in figure 4.5, several local optima exist, and with too small neighbourhood, it can be difficult to reach the global optimum for that problem.

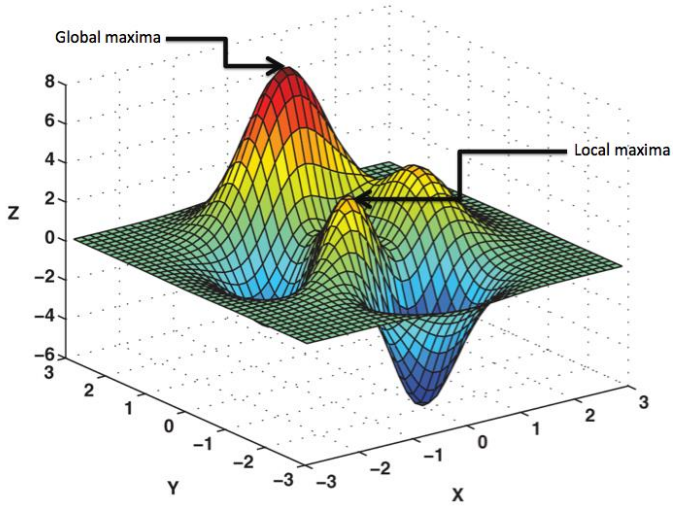


Figure 4.5: The visualization of fitness landscape for two-dimensional problem

4.3.6 Locality

The locality of a problem can be used to describe how the distance ($d(x, y)$ between any each pair of solutions $x, y \in X$ is correlated with the difference of the value of fitness function $|f(x) - f(y)|$. The locality for a given problem instance is high if a whole neighbourhood has a similar objective value and it is low even if for two close neighbours the difference in objective value is significant. The locality is highly correlated with problem difficulty; the higher the locality, the easier it is to find the optimal solution.

4.3.7 Heuristic techniques

It is a group of techniques designed to solve complex problems when the classical approach seems to be not feasible, or it will just be too slow. It is a separate field of computer science. With very complex problems, where number of possible solutions are close to infinity, finding the optimal solution with a standard brute force-like solution is not feasible. Nevertheless, algorithms that belong to the heuristic search group of methods are very efficient at finding a sufficiently good approximation of an optimal solution[20]. The goal of a heuristic search is to find a "good enough" solution within a reasonable timeframe to a problem at hand. With such an approach, there is a trade-off between completeness (is just one suitable solution needed or all solutions that fulfil the "good enough" criteria?), accuracy and precision, execution time, and optimality (a guarantee that found a solution is truly the best). An answer to the trade-off problem helps indicate which algorithm should be used.

5 GENETIC ALGORITHMS

5.1 INTRODUCTION

A genetic algorithm (GA) is a heuristic search technique to find the close approximation of the optimal solution to a search problem[21]. It is a particular class of the evolutionary computation algorithm that uses techniques inspired by biology. Darwin's theory of evolution was a core motivation to GA with the concept of "survival of the fittest" and operations like inheritance, crossover, mutation, and selection. In computer implementation, the simulation is created to mimic the existence of the population and its evolution throughout the iterations towards better solutions. With the complex optimization problem when many locally optimal solutions are expected and the problem has a high number of possible valid solutions, other approaches like straightforward brute-forced like search or less advanced heuristics search like hill-climbing are expected to fail to find global optimum in a feasible time frame. An additional benefit of using the Genetic Algorithm is that it is not limited by human creativity or biased by human expertise, often the solution found by GA can seem very odd and almost impossible to be created by a human, but it performs very well in a given problem.

5.2 HISTORY OF EVOLUTIONARY COMPUTATION

The history of Evolutionary Computation (EC) in computer science started in the 1950-60s when several computer scientists came up with the idea that Darwin's evolution theory can be used as an efficient optimization approach for computational and engineering problems. Darwin's main concepts about evolving population and surviving of the fittest and the natural genetic variation inspired by biology can be an effective search algorithm. John Holland invented genetic algorithms in the 1960s; after that, it was further developed by him and his team members at the University of Michigan.

The distinction between other advances in EC techniques and the approach used by Holland is that GA was not designed to solve specific problems but to study the phenomenon of adaptation as it is known from biology and to develop a process in which the mechanisms of the adaptation can be utilized in computer programs. Holland's idea revolves around the process of progressing from an old population, represented as chromosomes with, for example, a string of ones and zeros, to a new population by using the mechanisms of selection, a crossover of chromosomes between parents and the rare occurrence of mutation to introduce new structures of chromosomes. Chromosomes are built from genes, and each gene is an instance of an allele.

Holland's concept was a significant innovation with many possible exciting implications. What is convincing about the promising capabilities of GA is a schema theory which was also introduced by Holland (1975) and will be explained in the next section.

5.3 THE LIFE CYCLE OF GA :

Each phase of the process will be described in more detail in this chapter. However, first, it is essential to understand the process to grasp the fundamental concepts and understand how GA is capable of producing very promising results to many complex problems. The GA process starts with the initialization of a population; although many initialization methods can be used to create the initial population, the idea behind it all remains the same.

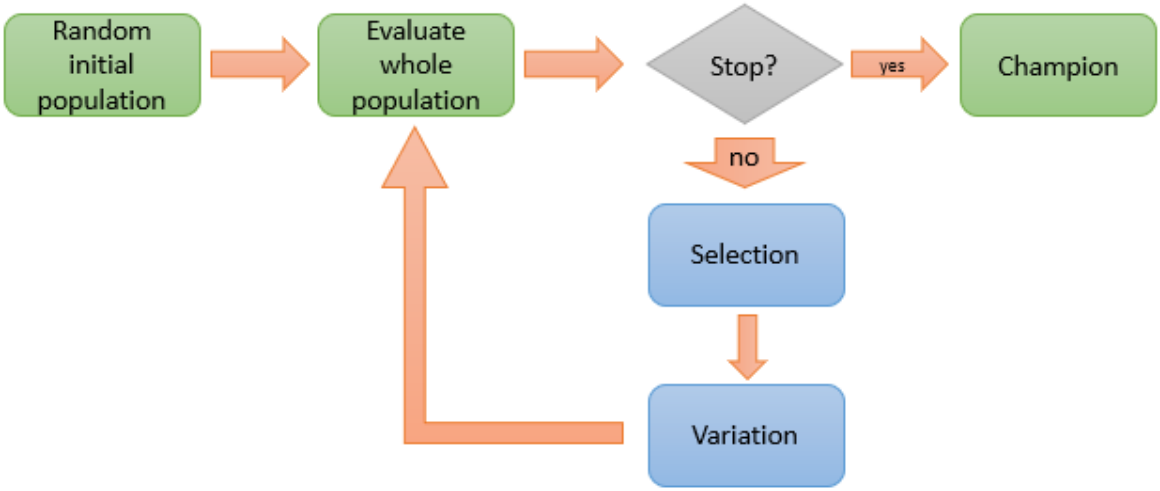


Figure 5.1: Genetic Algorithm cycle

First, different individuals are created at random to provide as much diversity in a population as it is possible. Once all the individuals are created up to the population limit, the process moves to the selection phase. Again, many different selection methods can be used at this stage, all of them have a common motivation to assign probabilities to each individual in a way that every individual has a non-zero chance of being selected; however, a more fitted one has a better chance at surviving. After the selection, the next phase is variations. The goal of the variation phase is to introduce new individuals to a population in search of a better solution. After the variation phase, the termination condition is checked, and either the selection and variation phase is repeated or the process ends.

Key terminology:

Chromosome – representation of a problem

Trait – possible features of an individual

Allele – possible settings of a trait

Gene – specific value of a trait

Locus – the position of a gene on the chromosome

Genotype – a combination of alleles that an individual has for a specific gene

Phenotype – a sum of an individual observable characteristics

5.3.1 Population

The population is a collection of valid solutions. The size of the population is a parameter that has to be set to conduct the search. Large population size is desirable; the more complex problem there is to solve, the bigger population should be used, after taking into account the computational capabilities of the machine.

5.3.2 Initialization

Initialization is the first step of a process and aims at creating the individuals for the first iteration of evolution. The goal of proper initialization is to create a diverse population rather than create individuals with the best initial objective values. In general, only when domain knowledge is available it can be a good idea to interfere at this stage, but usually, it is hard to provide more diversity in any other way than with a random process.

5.3.3 Selection

Selection is based on the phenotype, which in Genetic Algorithms directly represents the fitness of an individual. The goal of the selection is to choose a subset of the population to breed a population for the next generation. However, in general, the fitter individuals are more likely to be selected, but each solution should have a non-zero chance to be selected. Different selection methods can be used depending on how greedy or sophisticated the process should be. The most popular method is the roulette wheel, in which the chance of being selected is

proportionate to its phenotype. It is the easiest and the simplest method to implement and to use. The probability of selection for each individual can be described with this formula:

$$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$$

where: the probability *p(i)* of being selected for an individual, *i* is equal to a share of his fitness function *f(i)* in a sum of fitness for the whole population.

However, such an approach can suffer from premature convergence, which means all diversity that exists in a population is lost, as all individuals are very similar to each other and are locally the best possible solutions. That means the population is stuck in a local optimum. To avoid such problems, different selection methods could be used, such as tournament selection. In tournament selection, a subset of individuals is selected from a large population, and then those individuals compete against each other until only one remains. The amount of individuals chosen for a tournament is controlled by a parameter called "tournament size". All individuals have an equal chance of being selected for the tournament; however, the fittest participant wins the tournament. Such an approach gives control of selection pressure. With big tournament size, GA becomes very greedy during selection, with relatively small tournament size, it is less restrictive, leading to significantly slower improvement in fitness between generations.

5.3.4 Crossover

It is the most significant phase in GA. Two parents are selected to procreate. Parts of both parent's genotypes are mixed by exchanging a subset of their genes. That is how the offspring is created and added to the population. The most simple crossover is a one-point crossover (look at figure 5.2). A drawback of crossover is that it decreases the amount of diversity within a population.

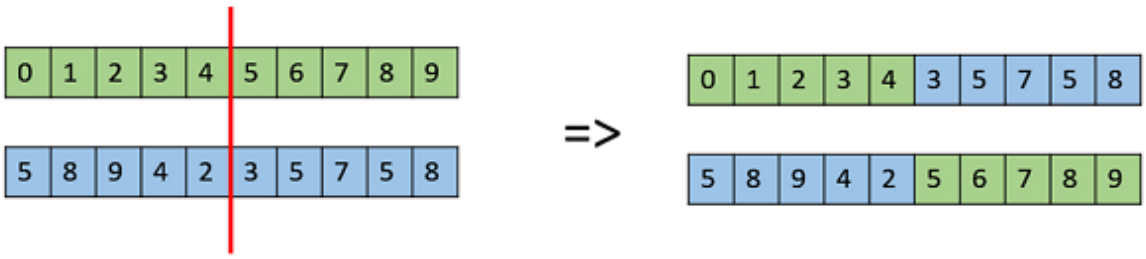


Figure 5.2: Visualization of a one-point crossover method

5.3.5 Mutation

The mutation is another crucial phase in a GA life cycle as the crossover cannot introduce new genes into the population. It is essential to provide some chances for mutation, which can introduce previously unseen structures of genes into individual chromosomes. As it is in nature, the mutation can have either positive or negative effect on an individual, so it cannot occur too often. Frequent mutation would destroy good structures that exist in a population; however, sometimes they are necessary to ensure that global optimum can be reached with basically any initial population. An example of a mutation approach can be a point mutation in which the iteration over the whole chromosome happens and with some low probability, each gene has a chance to be exchanged for any other within an allele (see figure 5.3).

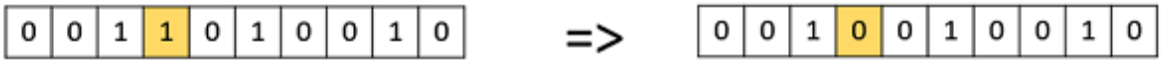


Figure 5.3: Visualization of point mutation

5.3.6 Terminations conditions

After selection and reproduction, the next step is to check whether the process should be stopped or the whole cycle repeated. The algorithm can terminate when the population has converged (GA is no longer able to produce significantly better offspring), or the limit of iteration is reached, which might mean, for example, that the allocated budget is reached.

5.3.7 Geometric operators

As GA can also work with continuous values, there is a need for special kinds of operators as one point crossover or point mutation can only work on discrete representation. In 2006, Moraglio and Poli proposed interesting concepts of geometric operators – geometric crossover coordinates of the offspring are calculated as the weighted average of the parents with weighs

between 0 and 1 whose sum is equal to 1. Interestingly, the offspring for sure will be no worse (further apart from Global Optimum) than the worst parent (see figure 5.4).

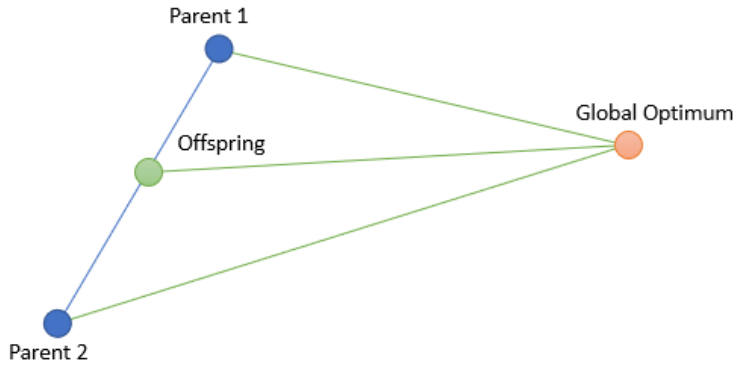


Figure 5.4: Crossover operation for continuous problem

An example of a geometric mutation can be a ball mutation, in which a radius or a box side length to mark as a neighbourhood is defined, and a random point from the neighbourhood is selected as mutated offspring (see figure 5.5).

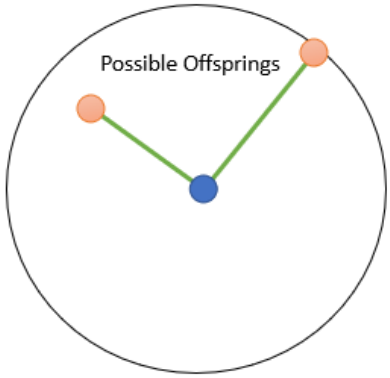


Figure 5.5: Ball mutation for continuous problem

5.3.8 Schemata Analysis

Schemata is a concept introduced in 1975 by Holland to model the ability of Genetic Algorithms to process similarities between binary targets variable[21]. When using l binary decision variables xi ∈ {0,1}, schema H = {h1, h2, h3} is a collection of symbols of length l here hi ∈ {0,1,∗}. ∗ denotes the "any" symbol, and it tells that a decision variable is not fixed. So, for example, with a 6-bits chromosome, the schema could be [*, *, 1, 0, 1, *]. The schema is a set of solutions whose representation match the schema at all locus, one of the representations of a solution that matches the schema can be [1, 0, 1, 0, 1, 0]. The schemata

approach allows treating a set of a solution as a single entity. With either 0 or 1 at a given locus means the given gene is fixed.

The order o(H) of schema H can be defined as the count of fixed genes in the schema chromosome, so in this example [*, *, 1, 0, 1, *] order o(H) = 3. The length δ(H) of a schema H can be defined as the distance between the two most outer fixed bits; for example, given the schema [*, 1, *, 0, 1, *] the length δ(H) = 4. The fitness fs(H) of a schema can be calculated as the average fitness of all solutions that match the schema. The number of solutions that can be instances of a schema H can be calculated as $2^{l-o(H)}$.

5.3.9 Building Blocks

Building Block is a concept introduced in 1989 by Goldberg (a PhD student of Holland) and it is based on Holland's schemata theory. In short, the building blocks can be defined as "highly fit, short-defining-length schemata". Building blocks (BB) can be defined as a solution to a subproblem that can be expressed by a schema. The building block has high fitness and its size is significantly smaller than the length l. A combination of BBs of low order can be efficiently used in GA to form high-quality solutions[23]. To compare it to nature, building blocks can be viewed a set of essential genes for the survival of individuals. It seems intuitive that some gens are more important and more suited for survival than the others, so preserving those BB in a population can lead to a healthier and stronger population. Building blocks also can be used to estimate the performance of an offspring after the recombination phase. When the building blocks are short, and of low order, the problem should be relatively easy to solve by Genetic Algorithms.

5.3.10 Limitations

The Genetic Algorithm's capabilities are limited to finding the fixed-length chromosome of individuals, which is a significant limitation when it is hard to say what the representation should look like. Fix-length representations do not readily support hierarchical optimization of tasks into subtasks. Therefore, programming concepts, like an iteration of recursion, cannot be used, limiting the capabilities of GA. In general, it significantly limits the a priori number of internal states of the program and undoubtedly limits what the program can learn.

6 BEYOND GA

6.1 INTRODUCTION

To overcome the limitation of the predecessor (Genetic Algorithms), Koza (1988) patented his invention of program evolution. Genetic Programming can be seen as an extension of GAs. The main difference between Genetic Algorithms and Genetic Programming is that individuals are no longer fixed size strings but, in fact, computer programs. Koza, who was also a PhD student of John Holland, is considered to be the father of Genetic Programming, thanks to the series of books written by him; however, the name GP was coined by Goldberg. Genetic Programming (GP) is another member of the family of evolutionary algorithms, in which usually unfit programs, starting from random, evolve into much better solutions over the iterations of recombination. GP is an evolutionary method that genetically breeds populations of computer programs to a particular problem[24]. A significant advantage over GA is that the computer programs are no longer limited by fixed-sized representation. Another benefit is that it can generate predictive models since any predictive model can be seen as an example of a computer program.

6.2 CONCEPT AND GP PROCESS

The individuals of a particular problem instance have a tree representation[19]; however other representations of GP programs can be used, the tree-based seems like the most straightforward approach (see figure 6.1). With the set of all possible structures that can be created through Genetic Programming, they can be described as all possible trees that can be built recursively from a set of function symbols $F = \{f1, f2, f3, \ldots, fn\}$ also called tree nodes and set of terminal symbols $T = \{t1, t2, t3, \ldots, tn\}$ also called tree leaves.

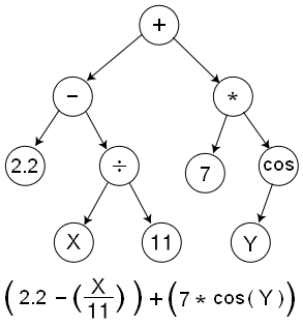


Figure 6.1: Example of a tree representation of GP

Terminals are features from the input data, and functions can be, for example, simple mathematical operations like addition or subtraction. Functions take a fixed amount of arguments, and there has to be a priori defined by assigning to each function its arity. Selection is based on phenotype, exactly as it was for GA, individuals better fitted to have a higher chance of mating, but individuals with low fitness have a non-zero probability of being selected as well. The same selection method as described for GA applies here as well, usually the selection with replacement is used. Variation is based on an individual's genotype. It is different than in GA as it takes the representation into account. Next generations are created by changing the structure of parents selected to be mated. The recombination phase consists of crossover and mutation. Crossover does not introduce new structures into a population and is conducted by mixing part of the parent's representation. Mutation of an offspring can be obtained by randomly modifying the genotype structure. As it was for GA, it can introduce new structures into a population, which helps maintain the population's diversity. It is vital to avoid premature convergence of a population and ensure a chance to reach the global optimum. The exact Schema theory described in the GA section has also been proven for GP by Poli and his co-workers. Genetic Programming is usually used to solve symbolic regression by using a set of primitive mathematical functions S (from the example in figure 6.1 F = {+, -, *,//, cos} and a set of terminal symbols T that contain real values variables and may also contain any set of numeric constants (from the example in figure 6.1 T= {x, 7, 2.2, y}). A GP individual can be expressed as a function that, for each observation x_i (input vector), returns the scalar value P(xi). Once the scalar vector is created, the error can be measured. With the error function, it is an optimization problem to minimize the error. The error can be defined as any distance (depending on metric selection) between the vector [P(X1), P(X2), …, P(Xn)] and target vector [t1, t2, t3]. For example, the mean Euclidian distance or root mean square error can be selected. The formula to calculate the error is as follow:

$$f(P) = \sqrt{\frac{\sum_{i=1}^{n}(P(\vec{x_i}) - t_i)^2}{n}}$$

As the symbolic regression task can be transformed into an optimization problem, it is possible to continue with the GP cycle that is similar to the GA process. Required parameters to be set to run GP algorithm are: population size, stopping criteria, initialization method, selection methods, crossover method and rate, mutation methods and rate, maximum tree depth. Several optional ones can also be used like: whether to use elitism or steady-state or not and many others.

6.2.1 Initialization

There are three most common initialization methods: the “grow” method, the “full” method, and “ramped half and half”, which is a combination of the previous two[19].

With the “grow” method, the trees are built with the algorithm:

1) A random function is selected to be a root of a tree.
2) From root departs *n* nodes, *n* is the arity of a function in a root. *N* nodes are selected with usually uniform probability from the set of functions and terminals.
3) If the terminal was selected, it is a stop of initialization for that branch, but if a function is selected, step 2 is repeated until initialization in all remaining branches terminates, or the tree reaches the depth equal to max tree depth minus one. After that, only terminals can be selected to create the remaining leaves in a tree.

“Full” initialization process is as follows:

1) A random function is selected to be a root of a tree.
2) From root departs *n* nodes, *n* is the arity of a function in a root. *N* nodes are selected with usually uniform probability only from the set of functions.
3) Once the depth of a tree is equal to the maximum depth parameter minus one, then all leaves are created as only terminals can be selected at this stage.

Trees build with the “grow” method are often of irregular shape and full initialization methods ensure that initial trees have a balanced and regular structure. “Ramped Half and Half” initialization methods were invented by J. Koza and the idea is to combine two previously describe initialization approaches as both have pros and cons. The main motivation to introduce this technique was to introduce more diverse individuals into the population as with aforementioned methods, very similar individuals can be created, and the population could converge too quickly before reaching the global optimum. In ramped half and half method, the population is divided into d groups, where d equals the maximum depth of a tree. For each tree depth, half of the group size is created with the full method and the second half with grow method.

6.2.2 Crossover

Standard crossover begins by selecting independently one random point in both parents. It can be seen as a similar process to one point mutation in GA, but in GP, the exchanged parts of representation do not need to be of equal size. The part of the parents that will be exchanged is a subtree with a root at the selected starting point. So the first offspring is created by removing the crossover fragment of T_1 from T_1 and inserting the respective subtree of T_2 at the crossover point of T_1 (see figure 6.2). It is worth mentioning that crossover can lead to rapid growth in the size of the programs. It is a phenomenon called bloat. Bloat is a progressive growth of the individuals without the corresponding improvement in fitness.

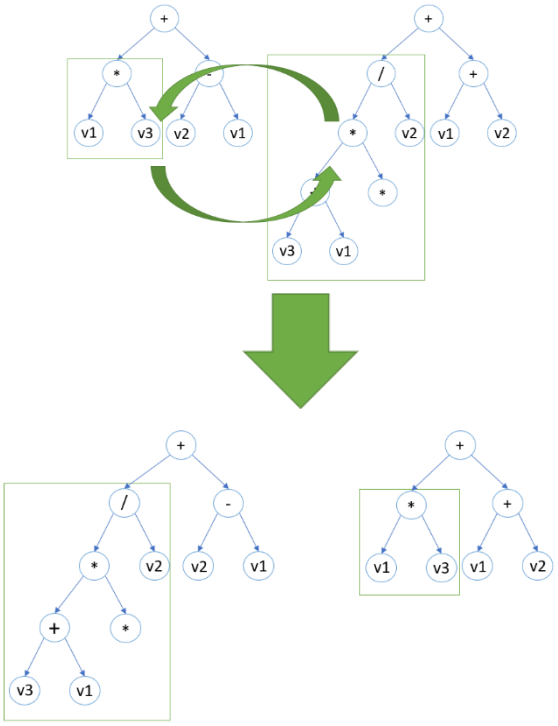


Figure 6.2: Crossover in Genetic Programming

6.2.3 Mutation

The mutation is asexual as it was in GA, as it operates on only one parental program. Mutation can happen at any node of the program with some probability (see figure 6.3). Such an approach is called a point mutation.

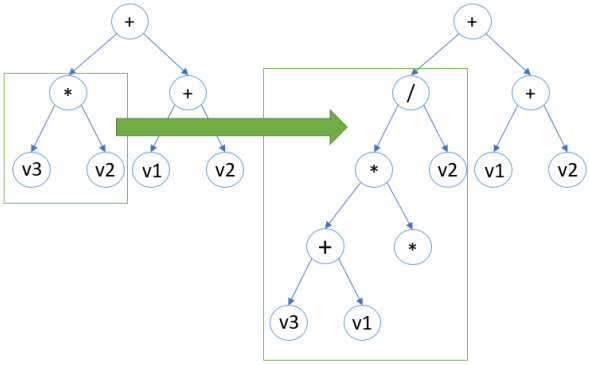


Figure 6.3 Mutation in Genetic Programming

Genetic Programming aims to allow the computer to generate a program that is capable of solving problems automatically. Although Genetic Programming was created to overcome the limitations of GA, GP is still not capable of fulfilling computer scientist dreams of being able to hit just one button and have the computer generate a program with a solution to any problem. The main limitations are bloats (a phenomenon of the rapid growth of programs without significant improvement in fitness), premature convergence. Moreover, GP is very prompt to overfit.

6.3 Geometric Semantic Genetic Programming

The next advance in evolutionary computing was achieved by introducing new concepts to Genetic Programming: semantics and semantics space[25]. Semantics is the vector of outputs of a program from the training data. Referring to the terminology known to data scientists: semantics is a vector of predicted values for training data. The goal was to invent transformation on the syntax of computer programs that the effect on their semantics will be known. As the semantics of a GP program is the vector of outputs on input data, the semantics space can be defined as mapping all possible semantics. Semantics space is composed of a vector of prefixed length of usually real numbers where the target is known, as it is in supervised learning problems.

The idea behind geometric operators in genetic programming is to have operators which will allow us to know the output a priori to the variation phase. That means that the effect of a mutation or a crossover on the offspring's semantics can be known and, more importantly, controlled. As it was with ball mutation in Genetic Algorithms, a user was defining viable mutation space within which the offspring would be created; the same is true with geometric operators as it is known that an offspring semantic will be within a predefined range. With those transformations, every optimization problem with a known target will have a unimodal

fitness landscape. The only task would be to match input data to training data, which is known as Distance Optimization with Known Optimum (DOKO). In 2012, Maraglio et al. proposed such transformation; although it still has drawbacks, it was a significant invention. Genetic Programming should achieve very promising results for many complex problems, especially on training data. If training data is a good representation of real-world data, it should achieve successful performance on unseen data as well.

6.3.1 Geometric Semantic Mutation

Given the parent computer program, the offspring will be created by adding the product of a difference between two random computer programs and mutation step (see figure 31). The mutation step can be defined as a function with a limited range of outputs; for example, a hyperbolic tangent function has a limited range of output values *(-1, 1)*, so the output of the mutation will be within a range *[-ms, ms]*.

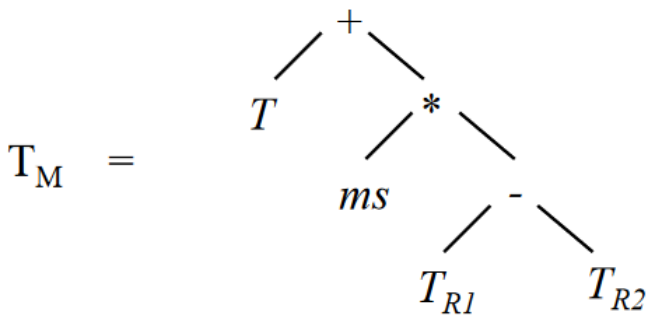


Figure 6.4 Example of a semantic mutation

With geometric semantic mutation, the whole semantic vector of the offspring can be seen as weak (as $T_{r1} - T_{r2}$ is centred at zero) perturbation of the corresponding semantic vector of the parent. An important note is that those two random trees are combined independently; their semantics does not affect each other. Therefore, the geometric semantic mutation is a linear combination of two programs.

6.3.2 Geometric Semantic Crossover

With two-parent computer programs T_1 and T_2, the output of a crossover will be the sum of a product of one of the parents T_1 and a random computer T_r program whose output between *[0, 1]* and product of the second parent and the difference between 1 and the random tree T_r

(see figure 6.5). The offspring generated with geometric semantic crossover has a semantic vector that is a linear combination of the semantics of the parent programs.

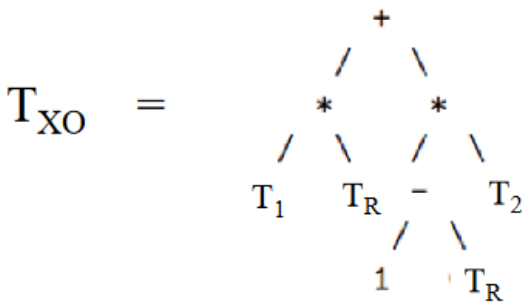


Figure 6.5: Example of a semantic crossover

6.3.3 Drawbacks of Geometric Semantic Operators

The main drawback of geometric semantic operators is that computer programs grow significantly and rapidly with every iteration. It is because of how they are constructed, these operators always breed offspring that are larger than their parents. Unfortunately, this can be a major limitation, but solutions to that problem are already proposed. For example, a new, more efficient implementation of Moragolio's operations was proposed that allows GP to be used on complex real-life problems[26].

6.4 Semantic Learning Machine

Knowing and understanding the fundamentals of evolutionary algorithms, the next advance in the field is a new, very promising invention: the Semantic Learning Machine. It is an algorithm that uses the concepts from EA within the deep learning field. As discussed in Chapter 2, finding the proper structure to solve a problem is a complicated task as there are no straightforward guidelines on how to construct an appropriate topology. Given that with a better-suited topology, better performance can be achieved, it is, in fact, an optimization problem. Thanks to the advances in the Evolutionary Algorithms field, it was possible to create a new approach to constructing efficient, tailor-made topology for neural networks. Although the concept of evolution of topology is not new; for example, a very popular NEAT[1] algorithm exists, and it can provide promising results in the reinforcement learning field; however, even that approach still can be improved. Semantic Learning Machines uses Geometric Semantic Genetic Programming concepts to replace backpropagation, which is the most common approach to train neural networks[27]. SLM introduces GSM-NN (geometric semantic mutation for neural networks), which works by taking an existing neural network and combining it linearly with a new randomly generated neural network[29].

Referring to the concepts of GSGP, the existing NN is a computer program selected to be a parent, and randomly created NN is an equivalent of a randomly created tree. Originally, SLM had a single neuron in the output layer with a linear activation function; however, the architecture was adjusted to work with ten classes output in this thesis. In this project, SLM, which was used as challenger models to Deep Semantic Learning Machine, was initialized with just one hidden layer with fully connected 100 hidden neurons. The learning step was calculated using Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (LBFGS). That algorithm is another improvement made to SLM compared to the original implementation where Optimized Learning Step (OLS) was calculated with Linear Regression. LBFGS algorithm is described in detail in Chapter 7.

6.4.1 SLM Mutation

The GSM-NN consists of two phases. The first phase is to initialize a random neural network and, in the second step, to combine it with the existing topology of a parent NN. As both networks share the same input and produce common output, both input and output layer of new NN does not have to be created, only hidden neurons are created (see figure 6.6).

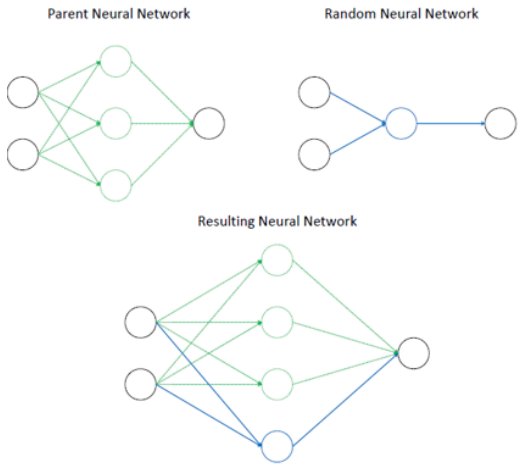


Figure 6.6: Example of a mutation process in SLM

The equivalent to the mutation step from GSGP is introduced in SLM, which is the learning step. It is a weight from the new neuron to the output neuron. It affects the amount of semantic variation that can be introduced for each use of mutation. This learning step can be either fixed (FLS) or optimally calculated (OLS). The input weights to the hidden neuron are randomly generated from any range.

SLM used in this project adds 50 hidden neurons at every iteration step, and the learning step for the mutated part is also calculated with the LBFGS algorithm. The idea behind the GSM-NN is to introduce the unimodal fitness landscape in the same way as in GSGP. It is possible to transform any supervised problem into a DOKO problem[29]. Since there are no local optima in a fitness landscape, the search for global optimum is very effective and efficient, especially for training data. However, the geometric semantic crossover could be formulated. The drawback known from GSGP applies to NN as well – the network would grow rapidly without respective improvement in performance.

6.4.2 SLM Life cycle

From the construction point of view, the SLM is, in essence, a geometric hill-climbing algorithm adjusted to work with neural networks, which is sufficient since there are no local optima in a semantic landscape.

The SLM works as follows:

1) N random initial neural networks are generated.
2) NN with the smallest error (B) is selected from the initial population.
3) Evolving phase until a termination condition is met:
    a. GSM-NN is applied to the current best NN by generating N new NN.
    b. B is updated by the best NN (with the smallest training error) from newly created NNs.
4) B is returned as it is the best performing NN, according to the training error.

Such an algorithm does not require to use backpropagation to avoid its drawbacks and limitations. The main disadvantage of backpropagation is that the search space for weights used in NN is not unimodal in relation to the training error, which means that the weights cannot be set to the global optimum with backpropagation as they will be stuck in a local optimum. Moreover, the computational costs can be high as the search for optimal weights is not effective compared to SLM, which operates on unimodal search space and performs only incremental evaluation, which can significantly speed up the process. Only the contribution of the added neuron has to be calculated as all other weights already have been calculated and they do not have to be re-evaluated.

SLM uses the population size parameter to control how many mutations are generated to select the best one that is added to the model at every iteration. A population with ten individuals was used in this project, and a fixed maximum number of iteration was used as the stopping criterium. Other stopping criteria like EDV or TIE could be considered [35]. However, since a clear cut-off point can be observed during the training in which the model is no longer improving; simply, a limited number of iteration was performed; thus, more advanced stopping criteria were not needed.

7 DEEP SEMANTIC LEARNING MACHINE

7.1 INTRODUCTION

After observing the promising outcome of experiments with Semantic Learning Machine, the clear next step was to improve and adapt SLM to other types of neural networks. The performance of a convolutional neural network greatly depends on the architecture of convolutional layers. Successful implementation of concepts of SLM to build complete CNN architecture was an appealing and thrilling challenge. Deep Semantic Learning Machine (DSLM) relaxes the limitation that SLM has. It is no longer limited to only hidden neurons but can also work with Convolution and Pooling neurons, Flattening layer and utilize Softmax as the activation function in the last layer. The complexity of that task and a need for a great level of flexibility made use of any commonly used framework to build deep learning models undesired. For example, the Keras framework forces users to create a whole layer at once, making the construction of models more straightforward but can also be considered a significant limitation. Thanks to the implementation of DSLM in pure NumPy, it is possible to construct a topology with a mixture of Convolutional and Pooling neurons on the same layer. Unfortunately, such a decision also has drawbacks; for example, implementing parallel execution to use all cores requires additional development. The major problem during the development of DSLM was RAM consumption during execution, which skyrocketed already in the initialization phase. To overcome the problem, many advanced performance optimization techniques were made. Also, additional effort was made, like deleting semantics during the training process, as after contributing to the predictions, it was no longer required for the further learning process. In the end, DSLM is working relatively fast; however, a machine with at least 32 GB of RAM is needed to build models that are presented in the final results. Even more is required to achieve better results as the population size cannot be bigger than three for the CIFAR10 dataset, which is not enough to explore all available concepts that can be created at each iteration.

7.2 DSLM LIFE CYCLE

The output of a DSLM is a standalone convolutional neural network that can be saved and used to generate predictions. To create such output, a whole search process has to be performed. The DSLM works as follows:

1) N random initial convolutional neural networks are generated.
2) CNN with the smallest error is selected from the initial population.
3) Evolving phase until a termination condition is met:
    a. GSM-CNN is applied to the current best CNN (B) by generating N new CNN.
    b. B is updated by the best CNN (with the smallest training error) from the newly created CNN.
4) B is returned as it is the best performing CNN, according to the training error.

The following section goes into the details of each step of the DSLM; however, several parameters must be defined before DSLM can be explained. The essential ones are the population's size, which algorithm to calculate the learning step will be used, termination condition, and loss function – how to calculate the error. In case of the loss function in image classification problems, cross-entropy is a popular choice.

7.2.1 Initialization

It starts with the pre-processing of input data (i.e. Label Binarizer is used to create one-vs-all labels), detecting how many output classes CNN should expect, and other validation steps to endure the proper operation of the DSLM. As the next step, the initialization of a population is performed. Plenty of different parameters are used to control the initialization process. The most important ones are the range of convolutional and hidden layers to be created in CNN, the number of neurons per layer for both convolutional (CP) and non-convolutional part (NCP) of the network, the probability of convolutional and pooling neuron created at each location, activation function for both CP and NCP. After the initialization of each CNN, semantics are calculated, and a learning step is applied. The LBFGS algorithm is used to calculate the learning step. Parameter stings used for experiments in this work are described in Chapter 4. The best network (that is the one with the smallest error) is selected to be mutated from the initial population.

7.2.2 Mutation

Different approaches to the mutation that allow CNN topology to grow can be applied in DSLM; however, as in this thesis, only initial experiments are performed, one straightforward mutation method is implemented. Geometric Semantic Mutation for Convolutional Neural Networks (GSM-CNN) creates at least one new neuron at each layer in the CP part; then, the results are propagated up to the output layer where the new "knowledge" is added to final predictions (see figure 7.1). The addition is made with the learning step as it was done in the initialization. Again, in the mutation phase, several vital parameters have to be defined, mainly the range of new neurons that can be created at each layer in both CP and NCP layers. The learning step is calculated again with the LBFGS algorithm; the algorithm is described in more detail in the next section.

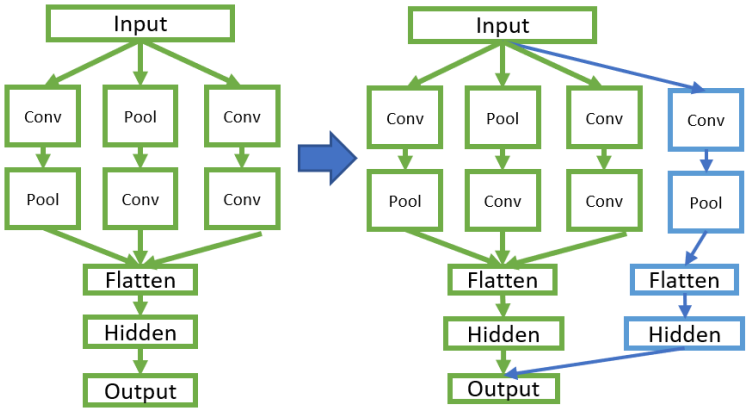


Figure 7.1: GSM-CNN example

7.2.3 LBFGS

Broyden-Fletcher-Goldfarb-Shanno algorithm is an approach used to solve optimization problems. The unconstrained nonlinear optimization problem can be solved iteratively. The idea is to determine the direction of the descent by preconditioning the gradient using curvature information[30]. It is done by improving approximation to the Hessian matrix of loss function gradually. The values are obtained from approximate gradient evaluations via a generalized secant method. The advantage of this approach is that inversion of a matrix is not needed; this reduces the computational complexity from $O(n^3)$ to $O(n^2)$. In DSLM, the limited-memory version of the Broyden-Fletcher-Goldfarb-Shanno algorithm is used. LBFGS is well suited for problems with large numbers of variables, more than a thousand. In DSLM, LBFGS is used to optimize weights to the output layer. It is working with cross-entropy loss. It is a quite popular algorithm for weight optimization in machine learning. It is available for Multi-layered Perceptron in the Scikit-learn package.

8 BENCHMARKING

This section of the thesis is dedicated to the results of initial experiments performed with DSLM. First, the dataset used for experiments is described; next, an overview of parameter setting is presented. In section 8.3, the results of DSLM parameter testing are described, and after that, the final performance of DSLM is presented. In section 8.5, the performance of challenger models can be found. In the end, the DSLM performance is compared with other state-of-the-art algorithms. The benchmarking is performed on CIFAR-10 and MNIST datasets. The challenger models are:

1) Standard CNN
2) VGG16
3) SLM

8.1 BENCHMARK PROBLEMS

Tests on different data sets should be performed as it is impossible to have an algorithm that will outperform all other models on all datasets. Ideally, a wide range of different tasks should be selected. However, since the training process is very computationally expensive, only two datasets were selected to give an overview of models performance in this thesis. Those datasets have a nice variety that allows testing performance on both grey and colour scale images and both actual pictures and images of handwritten objects.

The first dataset is “MNIST”, created by Yann LeCun and Crinna Cortes. It consists of 60 thousand training images of handwritten digits from 0 to 9 and 10 thousand testing examples. Each picture is a 28 x 28 pixels grayscale image (see figure 8.1). It is a very popular dataset used to validate the performance for many different results by many researchers.

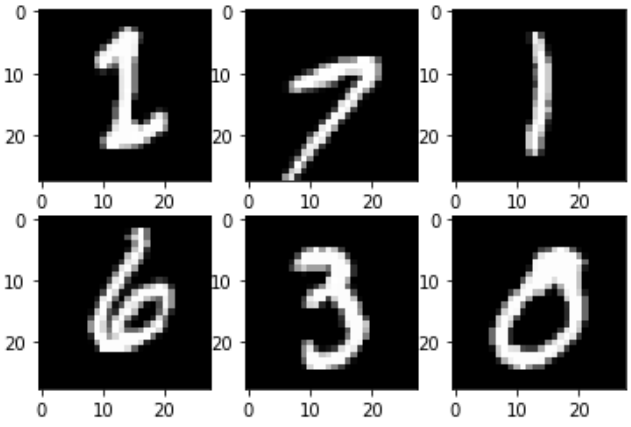


Figure 8.1: Examples of MNIST dataset

The second dataset is CIFAR-10, a labelled subset of the 80 million images dataset created by A. Krizhevsky, V. Nair and G. Hinton. The dataset consists of 50 thousand training images in 10 classes and 10 thousand testing examples. Each class has six thousand images; the dataset classes are the following: aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. The classes are entirely mutually exclusive as there is no overlap between any classes. Each picture is a 32 x 32 pixels colour image (see figure 8.2).

Figure 8.2: Example of CIFAR dataset

8.2 DSLM Parameter Settings

This section gives an overview of the hyperparameter settings that were used to train DSLM. As the training process is very computationally expensive, only the results of the initial experiments are presented. Tests for an impact of different parameter settings were performed; however, a thorough hyperparameter optimization was not performed, which would improve the model performance. DSLM has many parameters that need to be set; that is why only those that affect the model performance in a significant way are presented below. The left table contains parameter setting for the MNIST dataset and the right table for the CIFAR10 dataset.

Model Parameters	Values
Population size	10
Initial hidden Layers	2
Initial cnn layers	2
Range of initial CNN neurons per layer	Min 3; Max 5
Mutation Method	GSM-CNN
Mutation range of cnn neurons	Min 1; Max 2
Mutation range of hidden neurons	Min 10; Max 50
Weights distirbution	Unimodal
Evaluation Metric	Cross Entropy
Learning step algorithm	LBFGS
Probability of convolutional neuron	0.7
Use sparse connection	False
Activation function for CP	Relu
Activation function for NCP	Sigmoid
Stride in Pooling Neuron	[1, 2]
Stride in Convolutional Neuron	[1, 2]
Kernel size	[3, 5]
Pooling size	[2, 3, 4]

Model Parameters	Values
Population size	3
Initial hidden Layers	3
Initial cnn layers	3
Range of initial CNN neurons per layer	Min 3; Max 5
Mutation Method	GSM-CNN
Mutation range of cnn neurons	Min 1; Max 2
Mutation range of hidden neurons	Min 30; Max 50
Weights distirbution	Unimodal
Evaluation Metric	Cross Entropy
Learning step algorithm	LBFGS
Probability of convolutional neuron	0.7
Use sparse connection	False
Activation function for CP	Relu
Activation function for NCP	Sigmoid
Stride in Pooling Neuron	[1, 2]
Stride in Convolutional Neuron	[1, 2]
Kernel size	[3, 5]
Pooling size	[2, 3, 4]

Table 1 Parameter settings used in DSLM experiments

8.3 EXPERIMENTS WITH PARAMETER SETTING

In this section, the results of a few parameter setting are presented. Experiments were performed to investigate the impact of different settings on DSLM performance. All experiments are done on the CIFAR10 dataset, as this problem is more challenging to solve than the MNIST problem. That makes the impact of parameters more clear. All tests were performed over seven seeds, and Welch's t-test was used to calculate statistical significance. DSLM was training for ten iterations as in this phase, the fastest improvements happen, and the effects of poorly chosen settings are already visible.

8.3.1 Welch's t-test

To test the impact of given parameter setting, the unequal variances t-test on two independent samples was performed. Welch's t-test assumes that the test statistics that are compared are normally distributed. The null and alternative hypothesis is constructed to perform the test. The null hypothesis is a default hypothesis that a value of measured quantity will equal zero, the test measures whether there is a significant difference across samples[33]. A high p-value means that the null hypothesis cannot be rejected (there is not enough evidence to claim that samples are significantly different). When the p-value is low, it is possible to reject the null hypothesis of equal scores. In this thesis, the test from the SciPy package was used[34]. Proposed hypothesis are as follow:

$$H_0: \mu_1 = \mu_2$$

$$H_1: \mu_1 \neq \mu_2$$

The formula to calculate t-statistics:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{s_p\sqrt{\frac{2}{n}}} \quad \text{, where} \quad s_p = \sqrt{\frac{s_{X_1}^2 + s_{X_2}^2}{2}}$$

8.3.2 Impact of a number of convolutional layers

At this point, the impact assessment of a different number of convolutional layers was performed. The tests were executed by running DSLM with two and four convolutional layers. The theory suggests that two convolutional layers are not enough to detect more complex patterns that exist in data. On the other hand, more convolutional layers can lead to a significant reduction in picture size, leading to a loss of important information from the pictures.

	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	Seed 6	Seed 7	**Avg Perf**
2 Conv	36.682%	38.048%	33.194%	35.18%	36.83%	35.19%	36.18%	**35.85%**
4 Conv	37.264%	36.558%	35.756%	38.45%	37.05%	36.76%	37.45%	**37.04%**

Table 2 Assessing the impact of the number of convolutional layers in DSLM

```
conv2 = [36.382,38.048,33.194,35.18,36.83,35.19,36.18]
conv4 = [37.264,36.558,35.756,38.45,37.05,36.756,37.45]
print(stats.ttest_ind(conv2,conv4, equal_var = False))
```

```
Ttest_indResult(statistic=-1.7924597438759469, pvalue=0.1057246609171968)
```

P-Value indicates that, with a 90% confidence level, the null hypothesis of equal means should be rejected. The alternative hypothesis that means are not equal should be accepted. Having four convolutional layers allows to achieve better results than with just two convolutional layers, which is aligned with observations made over the years of advances in CNN architectures, that deeper architecture should outperform a wider one. The results are aligned with the intuition that deeper models can perform better as they can discover more complex nonlinear patterns in data.

8.3.3 Impact of a population size

Here the impact of population size on the DSLM performance is presented. As the theory suggests, a bigger population size helps find the best model at each iteration. With a very high level of randomness that DSLM introduces, there can be almost endless possible mutations that can be created at each iteration, so bigger populations should outperform smaller ones.

Performance	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	Seed 6	Seed 7	Avg Perf
Pop size 1	35.94%	34.95%	36.06%	36.07%	35.59%	36.3%	35.6%	**35.78%**
Pop size 3	38.40%	38.65%	37.85%	37.89%	38.05%	37.9%	37.15%	**37.98%**

Table 3 Assessing impact of population size in DSLM

```
indiv1 = [35.94,34.95,36.06,36.07,35.59,36.3,35.6]
indiv3 = [38.40,38.65,37.85,37.89,38.05,37.9,37.15]
print(stats.ttest_ind(indiv1,indiv3, equal_var = False))
```

```
Ttest_indResult(statistic=-8.889183435410994, pvalue=1.2839692000130822e-06)
```

The results are unequivocal that the size of a population plays a crucial role in DSLM performance. A much bigger population size should be considered as it was also observed on the MNIST dataset where DSLM with a population size equal to 10 was used, and the bigger population was, the better results were obtained. The amount of randomness in constructing the DSLM topology is visible in this test. A bigger population improves the performance of a model as both very good and bad individuals can be created through the mutation on each iteration. Such a low p-value means that the null hypothesis should be rejected and the alternative accepted.

8.3.4 Impact of number of neurons created in CP during mutation

Testing the effect of the number of neurons created at each iteration with GSM-CNN mutation. Every new neuron in CP can play an essential role in detecting interesting patterns in data; therefore, better results should be achieved with neurons. However, the topology that grows too much tends to overfit.

	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	Seed 6	Seed 7	**Avg perf**
1 neurons	37.03%	37.16%	37.05%	35.11%	37.55%	36.65%	36.9%	**36.77%**
3 neurons	36.57%	34.40%	35.22%	36.00%	37.82%	37.3%	36.4%	**36.24%**

Table 4 Assessing the impact of the number of new neurons per mutation in DSLM

```
neuron1 = [37.03,37.16,37.05,35.11,37.55,36.65,36.9]
neuron3 = [36.57,34.40,35.22,36.00,37.82,37.3,36.4]
print(stats.ttest_ind(neuron1,neuron3, equal_var = False))
```

```
Ttest_indResult(statistic=1.0024492253709367, pvalue=0.33871235508944275)
```

The results seem counterintuitive as there is no significant difference between adding one or three convolutional neurons per layer. There is no basis to reject the null hypothesis that the average performances between those two approaches are not equal. It seems that the critical factor in mutation is to correctly calculate the learning step and add new information to an

existing topology. Adding more neurons at every layer does not reflect a bigger improvement in performance. Since more neurons in every mutation consume more and more RAM, the leaner approach is preferable.

8.4 DSLM Performance

In this section, the final performances of DSLM are presented. Tests were performed over three seeds with Max Generation as a stopping criterium. With a clear cut point at which the model stopped improving and overfit, more advanced stopping criteria were not needed. It is important to note that due to high RAM consumption, it was impossible to validate the training process for the CIFAR10 dataset as loading additional 10k pictures every few iterations lead to usage of an additional significant amount of memory. Hence, it is likely to run out of memory even with a 32 GB RAM machine. The performance on unseen data was only performed at the end of the training process when DSLM was no longer needed; only the final best performing CNN can generate predictions.

8.4.1 Performance on MNIST dataset

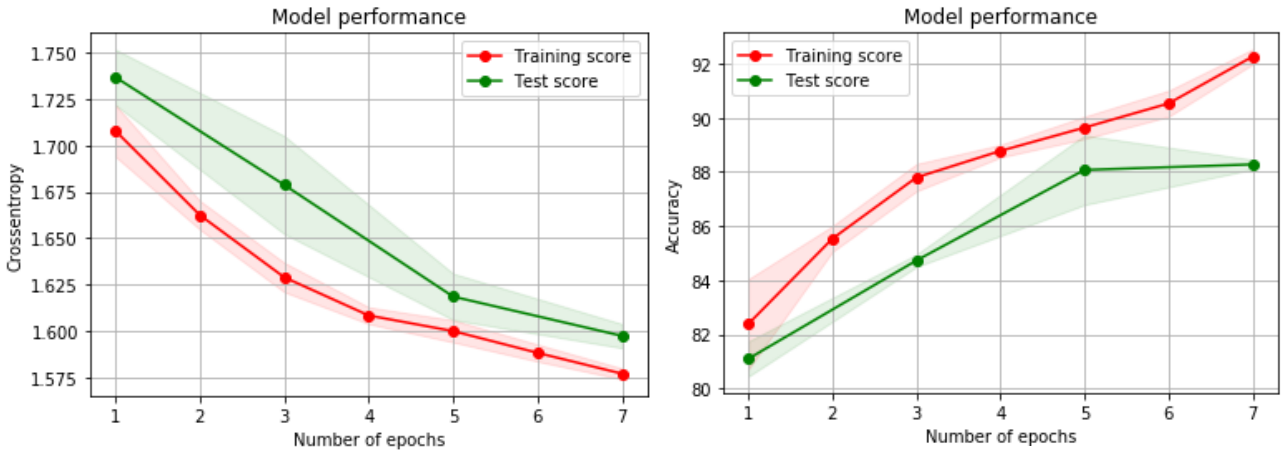


Figure 8.3: Performance of DSLM on MNIST dataset

After just seven iterations, DSLM on the MNIST dataset reached its potential. DSLM was able to achieve 88% accuracy and reduce the cross-entropy to 1.6. The training score rapidly improves with every iteration, and it can achieve above 92% accuracy. However, the validation score does not improve further with more iterations, and it starts to perform even worse, which is a clear indication of overfitting.

8.4.2 Performance on CIFAR10 dataset

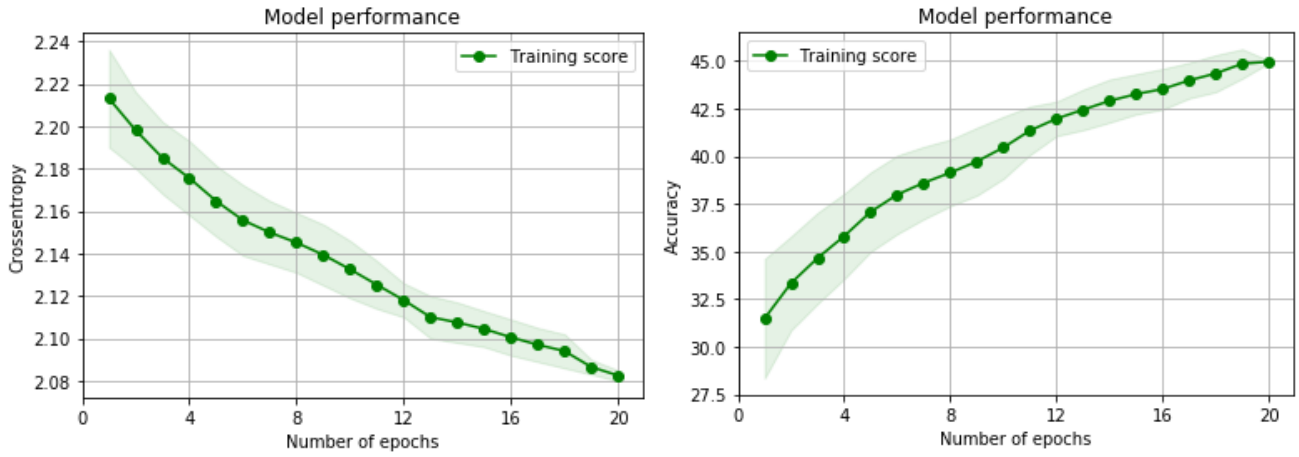


Figure 8.4: Performance of DSLM on CIFAR10 dataset

Final accuracy achieved by DSLM after 20 iterations were on average 43% and CE on average 2.10. As shown in figure 39, the training score curve flattens out, which makes further training ineffective. Moreover, topologies were getting to be very big, yet DSLM was still able to evaluate only a tiny fraction of possibilities.

8.4.3 Final model

The architecture of CNN created by DSLM differed with every run; however, it is possible to create a rough description of how the final CNN architecture will look like by averaging the topology of final models gathered over the three seeds on which the training was performed.

Layer	# of neurons
Sensors	3072
1 CP layer	60
2 CP layer	68
3 CP layer	53
1 NCP Layer	1170
2 NCP Layer	1160
3 NCP Layer	1120
Output	10

Layer	# of neurons
Sensors	782
1 CP layer	13
2 CP layer	12
1 NCP Layer	385
2 NCP Layer	370
Output	10

Table 5 Average final CNN topology found by DSLM

8.5 CHALLENGER MODELS PERFORMANCE

The standard CNN architecture is a sequential model built with the Keras framework. The model was trained with backpropagation and weights optimizer. Another challenger model is an original VGG16 proposed by K. Simonyan and A. Zisserman from Oxford [31]. The main difference between the standard CNN architecture and VGG16 is that several convolutional layers are followed by a pooling layer instead of a standard approach where the layers are interwoven, one convolutional after one pooling layer. VGG16 architecture was constructed to recognize 1000 classes; so to adjust the model to work with ten classes, the transfer learning was used in the last layer to adjust the model to provide proper predictions. The SLM was used to compare it to the DSLM approach to determine whether the advances made to the original architecture helped solve image classification problems. Each of the models was trained to solve a specific problem, and the results of the training and validation score are presented below. As it was presented in a famous paper presented by David Wolpert [32], there is no reason to prefer a specific model to outperform any other model if absolutely no assumptions regarding datasets are made. Simply, a glance at the dependent and independent variable allows data scientists to make some assumptions to the degree of what category of models to select; it is impossible to select the best performing model. For that reason, it is essential to compare several models on several different datasets to provide valid benchmarks. However, image classification tasks are one of the most computationally expensive tasks, so only two datasets were selected.

8.5.1 Challnagers models settings

The training was performed over three seeds as the level of randomness plays a small role in training a neural network with static architecture, as is the case for models built with the Keras framework. The SLM also provided very stable predictions, even though only three seeds were considered. As can be observed in the plots below, the confidence intervals are very narrow, which indicates that the results are very stable over different seeds. The architectures of the models are available in appendix A (for CNN) and B (for VGG16). The training was performed with batch size 512. For MNIST, data set of six epochs are considered to be enough as both the training and the testing curve flattens out; training for ten epochs was performed with CIFAR10. Training for few more epochs could be considered a standard for CNN as it still has some learning capabilities; however, the differences between all models are so clear that further training was unnecessary. Hyper-parameter tuning was not conducted here because achieving

the best possible results for the challenger models are not the goal of this thesis. The purpose of the challengers model is to provide a reasonable benchmark.

8.5.2 Performance of VGG16 on MNIST dataset

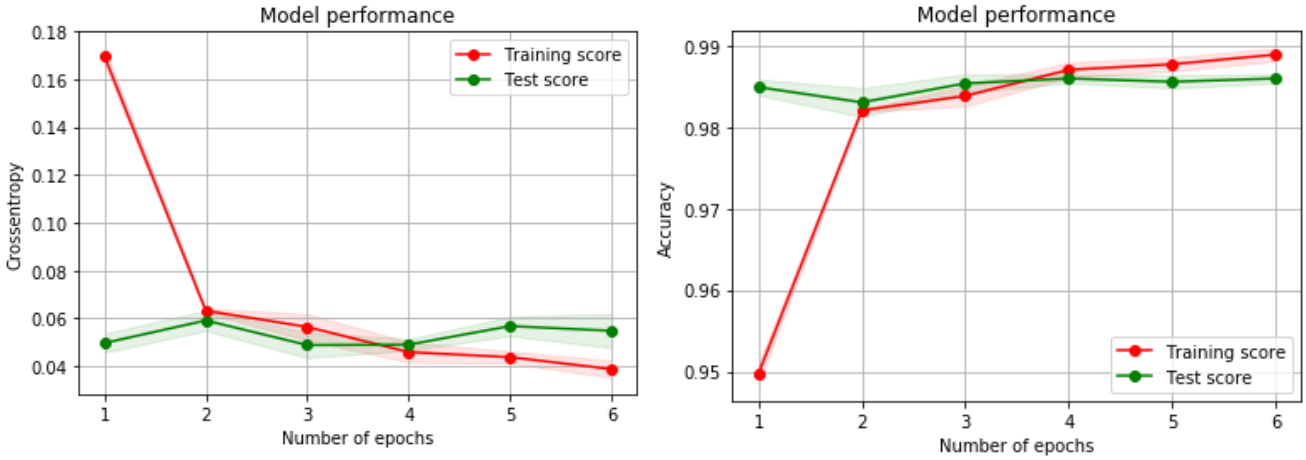


Figure 8.5: Performance of VGG16 on MNIST dataset

Although originally VGG16 was developed for image classification tasks of 1000 classes based on 48x48 colour pixels pictures, it was still able to perform exceptionally well. After the second epoch, the accuracy was above 98%, and after five epochs, the final results were around 98.5%. Although the pre-trained model was used, the last layer had to be retuned to be able to work with ten classes. There were only 100k trainable parameters in this setup, but the task was still relatively time-consuming as, on average, it took around 105 minutes.

8.5.3 Performance of VGG16 on CIFAR10 dataset

Figure 8.6: Performance of VGG16 on CIFAR10 dataset

The training process on the CIFAR10 dataset was conducted over ten epochs, and it is considered to be enough as the test score curve flattens out relatively quickly. VGG16 achieved

75% accuracy on unseen testing examples, which is an excellent result. However, the training process took over 150 min which can be seen as a very long process considered that only the last layer is tuned and all other weights in the model are frozen.

8.5.4 Performance of standard CNN on MNIST dataset

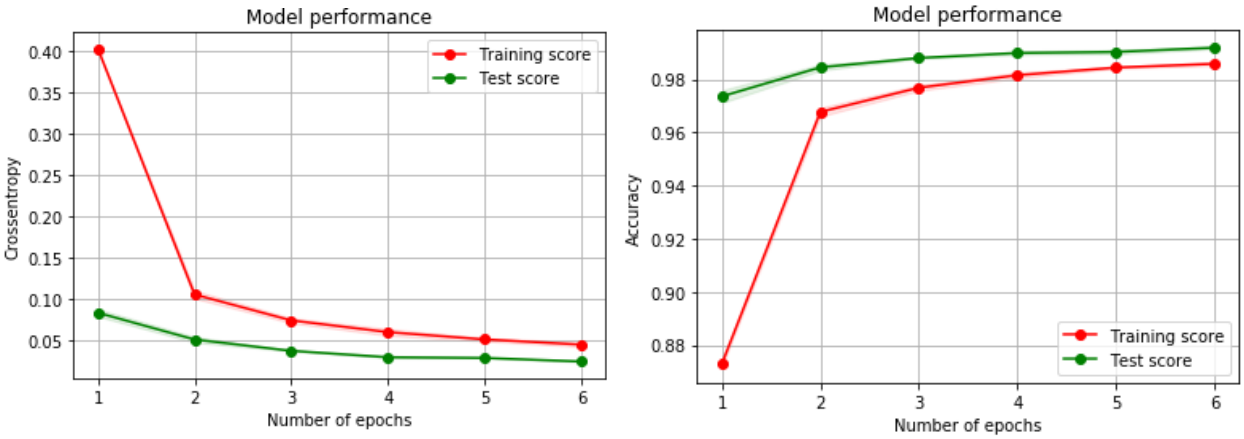


Figure 8.7: Performance of standard CNN on MNIST dataset

The performance of CNN on the MNIST dataset is almost perfect. The test score reached an astonishing 99%. Moreover, the test score is still above the training score, which indicates that it still could be improved; however, as it presents the best performance over all models, further training was unnecessary. The training process is very time-consuming as the model has around one million trainable parameters. The training process took over 140 min.

8.5.5 Performance of standard CNN on CIFAR10 dataset

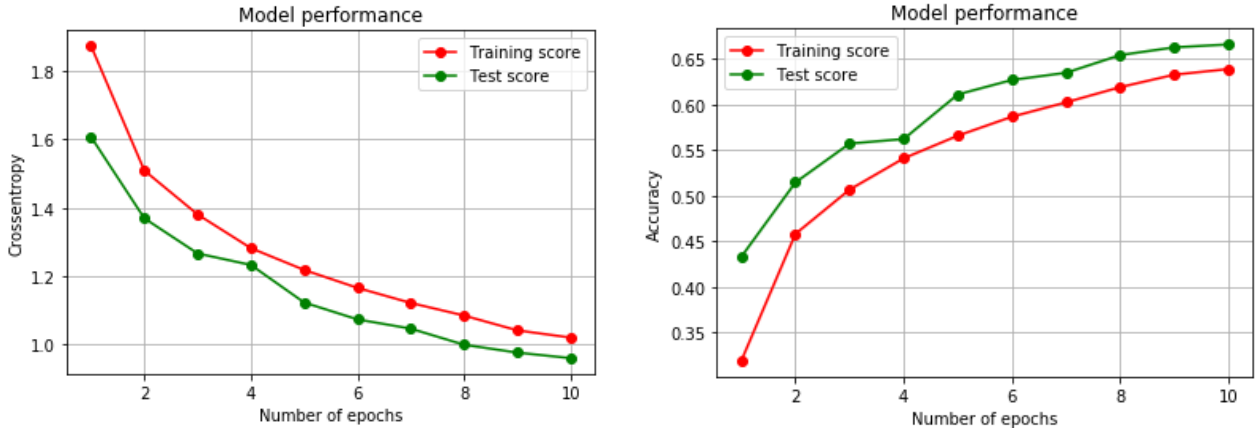


Figure 8.8: Performance of standard CNN on CIFAR10 dataset

The results of CNN indicate that, although the test score curve starts to flatten out, the final performance could be slightly better. However, achieving the best results on CNN is out of

scope for this project as the purpose of the challenger models is only to provide a benchmark for DSLM. After ten epochs of training, the test results were around 65%, which is a positive result; however, statistically significantly lower than achieved by VGG16. Training around one million parameters in CNN is, again, very time-consuming. The training process took over 215 minutes.

8.5.6 Performance of SLM on MNIST dataset

Figure 8.9: Performance of SLM on MNIST dataset

Although SLM architecture consisted of a single hidden layer, the performance it achieved was surprisingly exceptional. It is the only model that, in theory, was not explicitly created to solve image classification problems. However, it is essential to note that the MNIST task is relatively easy as the picture consists of only 728 greyscale pixels. However, over 92% accuracy on testing data can be considered an excellent result. Moreover, the training process was much shorter than CNN and VGG16, as it took just slightly over 26 minutes.

8.5.7 Performance of SLM on CIFAR10 dataset

Figure 8.10: Performance of SLM on CIFAR10 dataset

As the image classification task on the CIFAR10 dataset is much more complex, it reflects the achieved results, which are significantly lower. However, the 42% accuracy on testing data is again a surprisingly positive result considering that the model does not pose the neurons designed to solve image classification tasks. Moreover, the training process is relatively short – it took less than 84 minutes.

8.6 DSLM VS CHALLENGERS COMPARISONS

In this section, the final performance of DSLM is compared to all other models to evaluate the relative performance with the state-of-the-art approaches of solving image classification tasks. The comparison was made based on cross-entropy values and each model's training time, as it can play a crucial role when the minimal time to market factor is needed.

8.6.1 Cross-Entropy on MNIST and CIFAR10 dataset

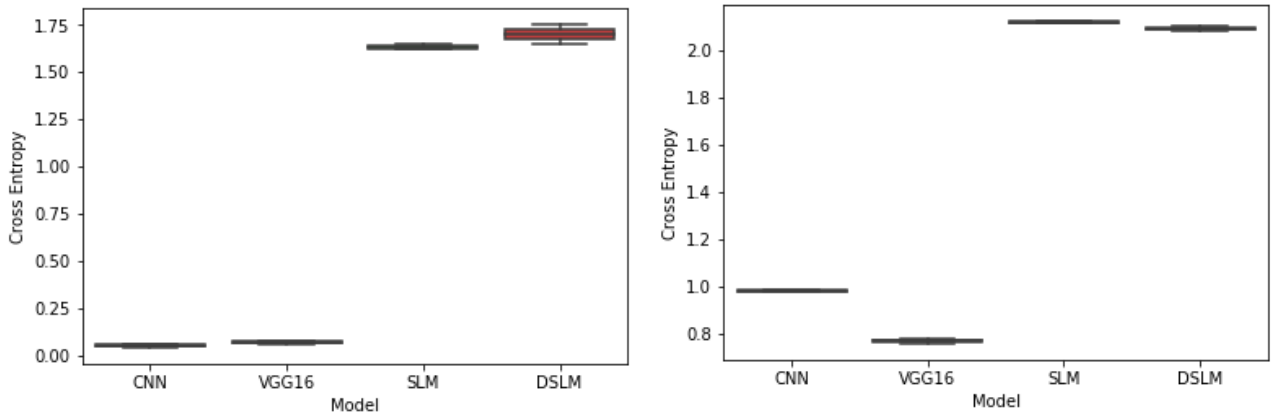


Figure 8.11: Performance of all models on MNIST and CIFAR10 dataset

Unfortunately, both standard CNN and VGG16 outperform both SLM and DSLM. On CIFAR10, VGG16 can provide the best predictions. It is worth noting that DSLM can provide better results than SLM on the CIFAR10 dataset. Although the results are not as promising as expected, the potential of DSLM is visible.

8.6.2 Training Time on MNIST vs CIFAR10

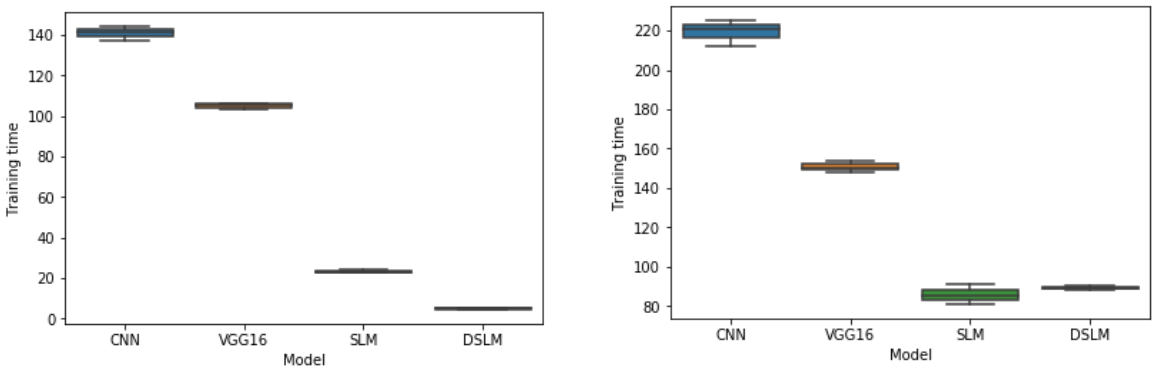


Figure 8.12 Timing results of all models on MNIST and CIFAR10 dataset

Considering the training time, it is indisputably clear that incremental evaluation can significantly shorten the time of training of the neural networks as it takes over 2 hours to train standard CNN on the MNIST dataset and over 3.5 hours to train on the CIFAR10 dataset, which is several times longer than to train SLM and DSLM. It is also important to notice how efficient the implementation of DSLM is – with hundreds of neurons in CP and thousands of neurons in NCP, and the training time is not significantly longer than it is for SLM.

9 CONCLUSIONS & LIMITATIONS

9.1 CONCLUSIONS

Initial experiments performed with DSLM show great potential and indicate that further exploration of this concept can be beneficial. Although the predictions made with CNN architecture found by DSLM are not as accurate as they are for the state-of-the-art approaches, DSLM was able to create interesting convolutional concepts capable of adequately detecting patterns that exist in data. This work should be seen as proof of the concept; the results achieved in initial experiments prove that this theoretical idea works in practice as well. However, an additional amount of work still has to be done. The settings used in DSLM should enable a user to build the correct, efficient concepts which are proven to be working well on various datasets without introducing a high level of randomness in both initialization and mutations processes which make it much more difficult to construct efficient building blocks. With such a limitation of population size, DSLM did not get enough chances to explore the search space well enough to provide as good predictions as, for example, VGG16 architecture. In theory, this is one of the possible outputs of DSLM, so the final performance cannot be worse, but it should be even better with infinite resources available. However, a high level of randomness is necessary to ensure the possibility of reaching the global optimum. The probabilities of different settings could be skewed to preferred concepts proven to work well for image classification problems. For example, after years of research on CNN, it is a widely accepted concept that CNN topologies should be deeper than wider and that kernel sizes bigger than three are not working too well on complex problems. Such knowledge should be used to skew the random process towards a known concept. An interesting concept which should also be considered is weights optimization, or a change of initializing weights process as sampling from a uniform distribution is not the ideal approach. The problem appears in DSLM, and it did not exist in SLM, as only one hidden layer exists there, and the connection between the hidden layer and the output layer was optimized with the LBFGS algorithm, so it was able to provide reasonable predictions. However, with a much deeper topology, the optimization of the learning step only between the last two layers is not enough. Training CNN and looking into the distribution of the weights in neurons can give a better idea about how the weights should be created.

9.2 LIMITATIONS & FUTURE WORK

The main problem that DSLM experience is RAM consumption which limits the size of a population that can be used, which plays a crucial role in building proper topology as each new set of neurons should be added carefully to prevent rapid growth of the network, which leads to overfitting. Another mutation method could be considered as, for example, the mutation does not have to start from the first convolutional layer, but it can begin at any layer. Another idea is that new neurons can utilize knowledge from existing neurons, so the connections between existing neurons and new ones could provide better generalization ability of a model. Moreover, some experiments should be done to limit the dataset size because a further reduction in RAM consumption is required; perhaps 10 or 20 thousands of images can be enough if a whole dataset has to be considered at once. From the perspective of implementation, the parallelization of execution can further speed up the training process.

10 BIBLIOGRAPHY

[1] Kenneth O. Stanley, Risto Miikkulainen *"Evolving Neural Networks through Augmenting Topologies"* Evolutionary Computation Volume: 10, Issue: 2, June 2002, pages 99 – 127

[2] Paulo Lapa, Ivo Gonçalves, Leonardo Rundo, Mauro Castelli *"Semantic Learning Machine Improves the CNN-Based Detection of Prostate Cancer in Non-Contrast-Enhanced MRI"* Evolutionary Computation Conference 2019, GECCO '19 pages 1837–1845

[3] Steven Homer, Alan L. Selman *"Computability and Complexity Theory",* 2001

[4] Alan M. Turing "*Computing Machinery and Intelligence*" Mind, Volume LIX, Issue 236, October 1950, pages 433–460

[5] Pamela McCorduck "*Machines Who Think (2nd ed.)"* 2004

[6] Daniel Crevier "*AI: The Tumultuous Search for Artificial Intelligence*" 1993

[7] Marvin Minsky, Seymour Papert *"Perceptrons: an introduction to computational geometry"* 1969

[8] Frank Rosenblatt "*The Perceptron: A Perceiving and Recognizing Automaton"* Report 85-460-1, Cornell Aeronautical Laboratory, Buffalo, New York, 1957

[9] Ian Goodfellow et al. "*Deep Learning (Adaptive Computation and Machine Learning series)*" 2016

[10] Ryszard Tadeusiewicz et al. *„Odkrywanie właściwości sieci neuronowych"* 2007

[11] Marian P. Kazmierkowski *"Neural Networks and Fuzzy Logic Control in Power Electronics"* 2002

[12] Anukrati Mehta *"A Comprehensive Guide to Types of Neural Networks" https://www.digitalvidya.com/blog/types-of-neural-networks/* 2019

[13] Christopher Olah "*Understanding LSTM Networks*" *http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015*

[14] Keiron O'Shea, Ryan Nash "*An Introduction to Convolutional Neural Networks*" CoRR,abs/1511.08458, 2015

[15] David Stutz "*Understanding Convolutional Neural Networks*" 2014

[16] Jianxin Wu "*Introduction to Convolutional Neural Networks*" 2016

[17] Franz Rothlauf "*Design of Modern Heuristics*" Natural Computing Series 2011

[18] Michael Hahsler, Kurt Hornik, *"TSP– Infrastructure for the Traveling Salesperson Problem"* Journal of Statistical Software, Volume 23, Issue 2, 2007, pages 1 - 21

[19] Riccardo Poli, William B. Langdon, Nicholas F. McPhee "*A Field Guide to Genetic Programming"* 2008

[20] Judea Pearl "*Heuristics: intelligent search strategies for computer problem solving*" 1984

[21] John H. Holland *"Adaptation in Natural and Artificial Systems"* 1975

[22] Mitchell Melanie "*An Introduction to Genetic Algorithms*" 1998

[23] David E. Goldberg "*Genetic Algorithms in Search, Optimization, and Machine Learning*" 1989

[24] John Koza "*Genetic Programming: On the Programming of Computers by Means of Natural Selection"* Vol. 1. MIT press, 1992

[25] Alberto Moraglio, Krzysztof Krawiec, Colin Johnson *"Geometric Semantic Genetic Programming"* 2012

[26] Leonardo Vanneschi, Mauro Castelli, Luca Manzoni, Sara Silva *"A new implementation of geometric semantic GP and its application to problems in pharmacokinetics"* European Conference on Genetic Programming, 2013, pages 205–216

[27] Ivo Gonçalves, Sara Silva, Carlos Fonseca *"Semantic Learning Machine: A Feedforward Neural Network Construction Algorithm Inspired by Geometric Semantic Genetic Programming"* Progress in Artificial Intelligence: 17th Portuguese Conference on Artificial Intelligence, EPIA, 2015, pages 280-285

[28] Ivo Gonçalves *"An exploration of generalization and overfitting in genetic programming: standard and geometric semantic approaches"* Doctoral dissertation, Universidade de Coimbra 2017

[29] Paulo Lapa, Ivo Gonçalves, Leonardo Rundo, Mauro Castelli *"Semantic Learning Machine Improves the CNN-Based Detection of Prostate Cancer in Non-Contrast-Enhanced MRI"* GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion, July 2019, pages 1837–1845

[30] John Dennis, Robert Schnabel *"Numerical Methods for Unconstrained Optimization and Nonlinear Equations"* (Classics in Applied Mathematics, 16) Soc for Industrial & Applied Math, 1996

[31] Karen Simonyan, Andrew Zisserman *"Very deep convolutional networks for large scale image recognition"* in International Conference on Learning Representations (ICLR), 2015

[32] David H. Wolpert, W.G. Macready *"No free lunch theorems for optimization"* IEEE Trans. on Evolutionary Computation, vol. 1, no. 1, 1997, pages 67-82

[33] B. L. Welch *"The Generalization of `Student's' Problem When Several Different Population Variances Are Involved."* Biometrika, Volume 34, Issue 1-2, January 1947, pages 28–35

[34] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html

[35] Ivo Gonçalves, Sara Silva, Carlos Fonseca *"On the generalization abilityof geometric semantic genetic programming"* in Proceedings of the 18th European conference on Genetic Programming (EuroGP), 2015, pages 41–52

LIST OF FIGURES

LIST OF TABLES

11 APPENDIX

A AN ARCHITECTURE OF STANDARD CONVOLUTIONAL NEURAL NETWORK

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
activation_2 (Activation)    (None, 16, 16, 64)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_3 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_4 (Dense)              (None, 256)               1048832
_________________________________________________________________
activation_3 (Activation)    (None, 256)               0
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                2570
_________________________________________________________________
activation_4 (Activation)    (None, 10)                0
=================================================================
Total params: 1,070,794
Trainable params: 1,070,794
Non-trainable params: 0
_________________________________________________________________
Train on 50000 samples, validate on 10000 samples
```

B AN ARCHITECTURE OF VGG16

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 48, 48, 3)	0
block1_conv1 (Conv2D)	(None, 48, 48, 64)	1792
block1_conv2 (Conv2D)	(None, 48, 48, 64)	36928
block1_pool (MaxPooling2D)	(None, 24, 24, 64)	0
block2_conv1 (Conv2D)	(None, 24, 24, 128)	73856
block2_conv2 (Conv2D)	(None, 24, 24, 128)	147584
block2_pool (MaxPooling2D)	(None, 12, 12, 128)	0
block3_conv1 (Conv2D)	(None, 12, 12, 256)	295168
block3_conv2 (Conv2D)	(None, 12, 12, 256)	590080
block3_conv3 (Conv2D)	(None, 12, 12, 256)	590080
block3_pool (MaxPooling2D)	(None, 6, 6, 256)	0
global_average_pooling2d_1 (	(None, 256)	0
batch_normalization_1 (Batch	(None, 256)	1024
dense_1 (Dense)	(None, 256)	65792
dense_2 (Dense)	(None, 128)	32896
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1290

Total params: 1,836,490
Trainable params: 100,490
Non-trainable params: 1,736,000

C PROFILER OF ONE OF DSLM EXECUTION PROCESS (EXECUTION 50+ SEC)

Ordered by: cumulativ time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.006	0.006	5976.539	5976.539	common.py:21(fit_and_predict)
1	0.004	0.004	5425.168	5425.168	deep_semantic_learning_machine_app.py:108(fit)
29	0.007	0.000	5150.602	177.607	deep_semantic_learning_machine_app.py:247(_apply_mutation_on_sample)
58	1.075	0.019	5149.956	88.792	mutation.py:19(simple_conv_mutation)
127	52.489	0.413	3980.263	31.341	conv_neuron.py:77(calculate)
127	24.773	0.195	3840.712	30.242	conv_neuron.py:48(convolv)
127	206.034	1.622	2253.174	17.742	arraypad.py:964(pad)
278	1653.168	5.947	1949.098	7.011	conv_neuron.py:106(convolv)
8211	1836.397	0.224	1836.397	0.224	{built-in method numpy.concatenate}
508	21.587	0.042	1133.013	2.230	arraypad.py:136(_append_const)
508	0.018	0.000	998.882	1.966	arraypad.py:102(_do_append)
508	20.455	0.040	797.260	1.569	arraypad.py:107(_prepend_const)
151	9.374	0.062	729.343	4.830	conv_neuron.py:82(calculate2)
151	0.459	0.003	689.585	4.567	conv_neuron.py:60(convolv2)
508	0.005	0.000	674.887	1.329	arraypad.py:97(_do_prepend)
1	0.001	0.001	551.308	551.308	slm_classifier.py:288(predict)
1	0.000	0.000	548.612	548.612	base_slm.py:362(predict)
1	0.000	0.000	548.612	548.612	conv_neural_network.py:185(generate_predictions)
1	0.000	0.000	548.502	548.502	conv_neural_network.py:88(calculate_semantics)
232600	520.921	0.002	520.921	0.002	{method 'reduce' of 'numpy.ul objects}
1	19.243	19.243	500.480	500.480	conv_neural_network.py:317(calculate_conv_semantics)
34007	486.154	0.014	486.199	0.014	{built-in method numpy.array}
203984	1.104	0.000	476.675	0.002	fromnumeric.py:69(_wrapreduction)
5332	0.013	0.000	377.181	0.071	hidden_neuron.py:84(calculate_semantics)
5332	365.178	0.068	372.330	0.070	hidden_neuron.py:58(_calculate_weighted_input)
153507	0.741	0.000	298.133	0.002	fromnumeric.py:1966(sum)
116	0.008	0.000	276.134	2.380	deep_semantic_learning_machine\mutation.py:240(<listcomp>)
2	0.010	0.005	253.037	126.519	deep_semantic_learning_machine\initialiation.py:72(init_conv_standard)
47	4.099	0.087	249.064	5.299	conv_neural_network_components\pool_neuron.py:75(calculate)
2	0.000	0.000	236.218	118.109	conv_neural_network_builder.py:31(generate_new_conv_neural_network)
60	0.003	0.000	234.044	3.901	semantic_learning_machine\lbfgs.py:23(fit)
60	0.011	0.000	234.042	3.901	semantic_learning_machine\lbfgs.py:51(_fit_inner)
60	0.030	0.000	234.004	3.900	lbfgsb.py:49(fmin_l_bfgs_b)
60	0.801	0.013	233.970	3.899	lbfgsb.py:211(_minimize_lbfgsb)
6995	0.031	0.000	232.515	0.033	lbfgsb.py:284(func_and_grad)
47	0.317	0.007	232.160	4.940	pool_neuron.py:49(pool)
6995	0.023	0.000	232.121	0.033	optimize.py:298(function_wrapper)
6995	0.043	0.000	232.098	0.033	optimize.py:61(__call__)
6995	0.036	0.000	232.026	0.033	lbfgs.py:149(_loss_grad_lbfgs)
6995	18.308	0.003	231.668	0.033	lbfgs.py:106(_backprop)
99	44.729	0.452	224.519	2.268	pool_neuron.py:104(pool)
58	1.071	0.018	218.753	3.772	mutation.py:301(mutation_lbfgs_new_neurons)
1016	0.005	0.000	214.460	0.211	numeric.py:293(full)
1016	211.889	0.209	211.889	0.209	{built-in method numpy.copyto}
42935	0.048	0.000	179.552	0.004	pooling_functions.py:19(calculate_pool_output)
42935	0.047	0.000	179.486	0.004	pooling_functions.py:4(calculate_max)
42995	0.112	0.000	179.457	0.004	fromnumeric.py:2397(amax)
5709	0.011	0.000	137.262	0.024	activation_function.py:24(calculate_output)
377	131.683	0.349	131.683	0.349	activation_function.py:19(calculate_relu)
6995	14.744	0.002	129.632	0.019	lbfgs.py:84(_forward_pass)
52	2.974	0.057	123.271	2.371	pool_neuron.py:81(calculate2)
52	0.086	0.002	118.008	2.269	pool_neuron.py:58(pool2)
278	0.003	0.000	117.446	0.422	conv_neuron.py:73(_calculate_output)
7056	56.320	0.008	88.749	0.013	neural_network_base.py:79(softmax)
8401	58.530	0.007	58.530	0.007	{method 'copy' of 'numpy.nd objects}
4	0.001	0.000	53.384	13.346	conv_neural_network_builder.py:147(<listcomp>)
6995	46.814	0.007	51.666	0.007	_base.py:198(log_loss)