ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΙΑΤΡΙΚΗ

ΒΑΘΙΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΤΕΧΝΗΤΩΝ ΝΕΥΡΩΝΙΚΩΝ ΔΙΚΤΥΩΝ ΚΑΙ ΣΥΓΧΡΩΝΕΣ ΕΦΑΡΜΟΓΕΣ

Διαμαντής Δημήτριος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
Επιβλέπων
Ιακωβίδης Δημήτριος

Λαμία, 06/12 έτος 2017

UNIVERCITY OF THESSALY

SCHOOL OF SCIENCE

INFORMATICS AND COMPUTATIONAL BIOMEDICINE

DEEP ARTIFICIAL NEURAL NETWORK ARCHITECTURES AND MODERN APPLICATIONS

Diamantis Dimitrios

Master Thesis
Supervisor
Iakovidis Dimitris

Lamia

06/12/2017

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΔΙΑΤΜΗΜΑΤΙΚΟ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΠΡΟΓΡΑΜΜΑ
ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΙΑΤΡΙΚΗ
ΚΑΤΕΥΘΥΝΣΗ

«ΥΠΟΛΟΓΙΣΤΙΚΗ ΙΑΤΡΙΚΗ ΚΑΙ ΒΙΟΛΟΓΙΑ»

ΒΑΘΙΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΤΕΧΝΗΤΩΝ ΝΕΥΡΩΝΙΚΩΝ ΔΙΚΤΥΩΝ ΚΑΙ ΣΥΓΧΡΩΝΕΣ ΕΦΑΡΜΟΓΕΣ

Διαμαντής Δημήτριος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
Επιβλέπων
Ιακωβίδης Δημήτριος

Λαμία, 06/12 έτος 2017

«Υπεύθυνη Δήλωση μη λογοκλοπής και ανάληψης προσωπικής ευθύνης»

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, και γνωρίζοντας τις συνέπειες της λογοκλοπής, δηλώνω υπεύθυνα και ενυπογράφως ότι η παρούσα εργασία με τίτλο «Βαθιές αρχιτεκτονικές τεχνητών νευρωνικών δικτύων και σύγχρονες εφαρμογές» αποτελεί προϊόν αυστηρά προσωπικής εργασίας και όλες οι πηγές από τις οποίες χρησιμοποίησα δεδομένα, ιδέες, φράσεις, προτάσεις ή λέξεις, είτε επακριβώς (όπως υπάρχουν στο πρωτότυπο ή μεταφρασμένες) είτε με παράφραση, έχουν δηλωθεί κατάλληλα και ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο ΔΗΛΩΝ
Διαμαντής Δημήτριος

Ημερομηνία
06/12/2017

Υπογραφή

ΒΑΘΙΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΤΕΧΝΗΤΩΝ ΝΕΥΡΩΝΙΚΩΝ ΔΙΚΤΥΩΝ ΚΑΙ ΣΥΓΧΡΩΝΕΣ ΕΦΑΡΜΟΓΕΣ

Διαμαντής Δημήτριος

Τριμελής Επιτροπή:

Ιακωβίδης Δημήτριος

Δελήμπασης Κωνσταντίνος

Πλαγιανάκος Βασίλειος

Abstract

The importance of artificial neural networks in today's artificial intelligence and more specifically in the field of computer vision has been demonstrated by their remarkable performance in a variety of applications. This thesis presents a survey in the field of Artificial Neural Networks (ANN) with focus on deep learning. It investigates state of the art architectures and modern applications. A detailed analysis of a special kind of ANN architecture, called "Convolutional Neural Networks" (CNNs) is included. CNNs have been widely used to tackle problems in computer vision with remarkable results. The usage and the results of CNN architectures are investigated on various computer vision problems, including facial makeup presence detection and the problem of abnormality detection in the human gastrointestinal tract, by analyzing capsule endoscopy images.

Acknowledgments

This thesis would not have been possible without the support given to me by my advisor and mentor Dr. Dimitris Iakovidis Associate Professor, who inspired me to dive into the subject of Artificial Neural Networks and in general the field of computer vision. His inspiring work contributed drastically in my professional and academic carrier. I would also like to thank Mr. Michael Vasilakakis for his support and know-how contribution in the general field of computer vision.

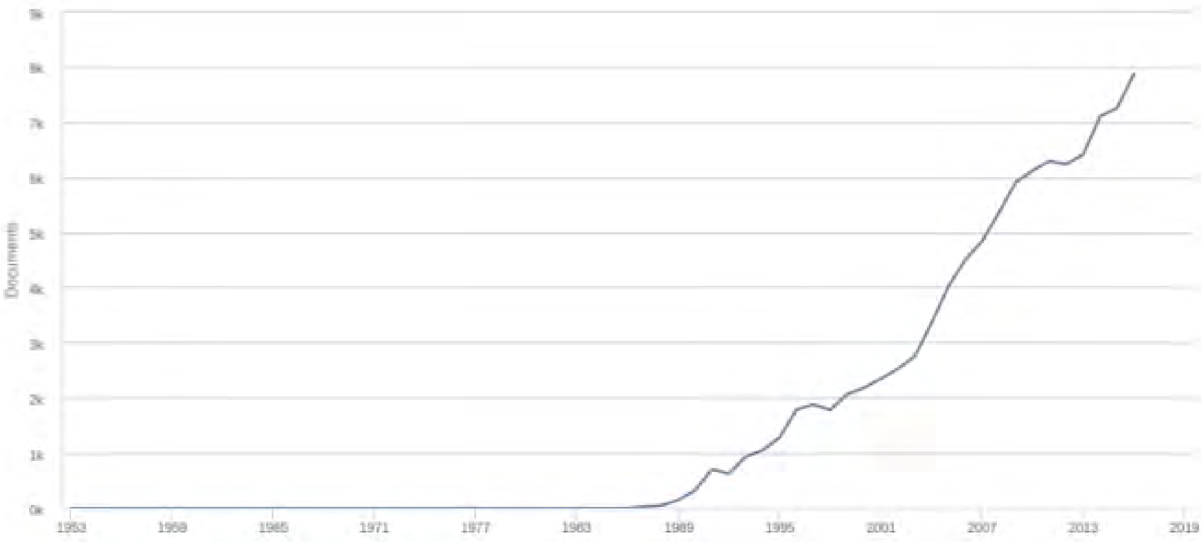
Contents

1. Introduction

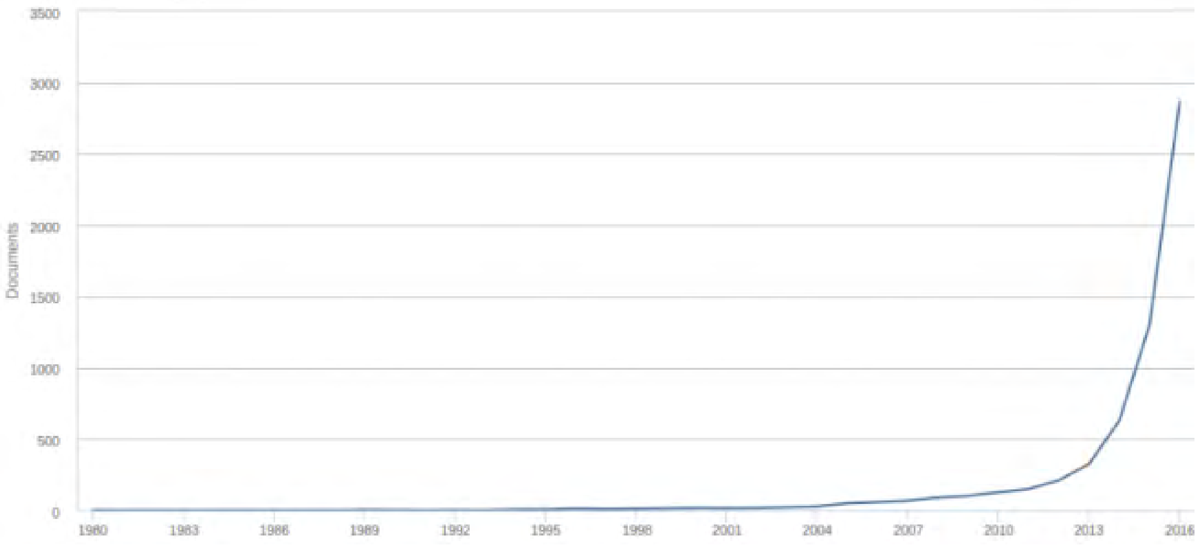
This thesis is about the concept of Artificial Neural Networks which is a subject of artificial intelligence in computer science. More specifically, it aims to explore the areas of machine learning via the usage of deep neural networks by reviewing the existing work that has been done over the years in the field and by investigating novel approaches and applications. This thesis presents applications experiments that have been done by applying the acquired knowledge in the field of computer vision and more specifically in abnormality detection in images that were obtained by capsule endoscopy, along with novel results from facial makeup detection.

1.1. History

Even before the invention of computers, humans where always fascinated by the idea of creating a system that will present some short of intelligence. Stepping forward into that direction artificial neural networks where invented as early as the mid 50s [1]. The idea behind them was to mimic the biological neurons and the connections that appear between them in the human brain.



(a)



(b)

Figure 1 Historical overview of the number of articles published for “Artificial Neural Networks” on the top and “Deep Learning” on the bottom.

The initial attempts were successful yet limited by the low computational power of early computers combined with the lack of powerful algorithms to train such networks. Almost 30 years later, with the discovery of more robust training algorithms and the increase in computation power, shifted the interest back to the field (Fig. 1). As the computational power continued to increase, researchers started to exploit the power of deeper, more complex networks, which lead to the creation of the concept of “deep learning” which effectively exploits the power of deep neural networks in order to automatically capture the structural representation of the data with which it is trained. This resembles closer the generic way that the biological brain works, which contradicts the classic application specific feature extraction process that other methodologies are following.

1.2. Aims and Contributions

This thesis aims to provide:

- An introduction to the concepts of artificial neural networks along with a historical analysis
- Present methods of training deep neural networks
- Discuss the problems of training in deep neural networks
- Investigate various deep neural network architectures

- A novel application for facial makeup discovery using deep convolutional neural networks
- A novel Peephole Fully Convolutional Neural Network (PFCNN), which is a deep fully convolutional neural network architecture designed, but not limited, to deal with the problem of abnormality detection in gastrointestinal images. Experiments shown the proposed architecture is able to achieve state-of-the-art results.

1.3. Thesis Structure

The rest of this thesis consists of 7 sections. Section 2 contains a brief introduction of the concept of artificial neural networks, along with a historical overview and inspiration that guided to the creation of them. Some basic concepts are also presented aiming to help the reader understand the sections that follow. Section 3 explores the field of feed forward neural networks, various architectures that have been proposed and successfully used over the years, such as the Multilayer Perceptron Networks (MLPs). The section contains an extensive overview of the existing training methodologies and more specifically the backpropagation algorithm along with modern variations of it that can be found in the modern bibliography. Section 4 is dedicated to the challenges that are faced in the training process of a neural network and presents various practices that are followed in order to overcome them. Section 5 focuses on a special type of neural network, called Convolutional Neural Networks (CNN), which has attracted the attention of research in the last years, mainly because of its performance in the field of Computer Vision (CV). A chronological review of the most important modern deep convolutional neural network architectures is included along with their contributions that each one brought to the field. In the sixth section an overview of other type of network architectures is included such as the autoencoders, which have been used extensively in the field of CV, and Self Organized Maps (SOMs). The last section of the thesis is dedicated in the presentation of applications and results that were obtained by applying the knowledge acquired from the previous sections. More specifically, a novel approach of facial makeup discovery is presented that outperforms the current state of the art methodologies in the field. Secondly results obtained from various experiments that were performed by applying convolutional neural networks, and more specifically a novel architecture named PFCNN, in the field of anomaly detection in images obtained by the usage of endoscopic capsule in the human intestines, are presented and compared with the previous state-of-the-art approach.

2. Artificial Neural Networks

This section is providing a brief introduction in the concept of Artificial Neural Networks (ANNs) their usage and where their source of inspiration came from.

2.1. Motivation

It is well known that the human brain contains billions of neurons (Fig. 2) that are connected between each other via synapses and that these neurons are acting together in parallel and are responsible for our logic.

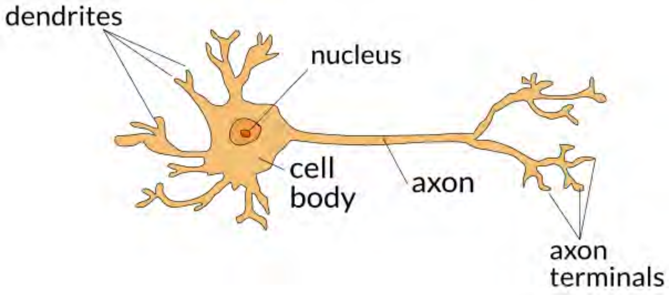


Figure 2 A stripped down human brain neuron. Dendrites reassemble the input of the cell body (neuron), nucleus the computational unit and axon the output of the neuron.

Artificial Neural Networks (ANNs) are inspired from that model and in fact are trying to mimic the way that the human brain works. Unfortunately until today, a fully detailed picture of how human brain works is missing. With that said, ANNs are a simplified simulation of the human brain neurons and their synapses connected into a graph. Each artificial neuron, called perceptron, is a tiny computational unit, which receives input signals and if a certain input threshold is reached, it is activated.

2.2. The Perceptron

The basic computational unit trying to mimic the human brain neuron is called, perceptron. The term perceptron along with the theory behind it was developed by [1] in the late 50’s with basic source of inspiration coming from the earlier work of [2]. As it can be noticed in (Fig. 3) a perceptron is a direct translation of a biological neuron into a computational unit. Dendrites have been replaced with weights W_i the nucleus with an input and a weight sum which, known as transfer function and the axon with a function φ known as an activation function.

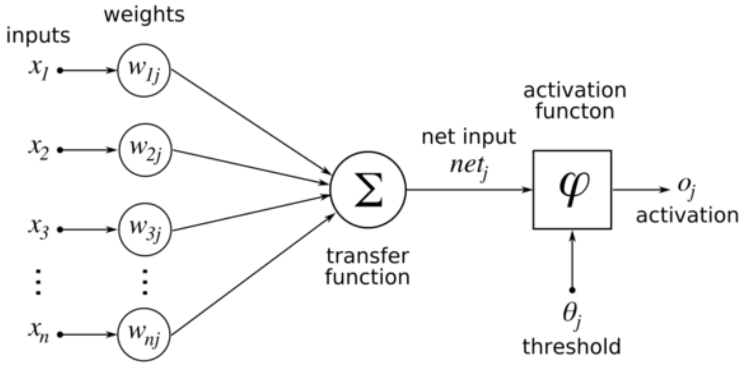


Figure 3 Generic illustration of a basic perceptron

Perceptron essentially computes the weighted sum of the inputs passed by an activation function which produces a signal if and only if a threshold value is reached. Thus a perceptron can be expressed as:

$$y = \varphi(\textstyle\sum_i x_i\, w_i) \tag{1}$$

By examining further the above equation it becomes clear that a single perceptron is a basic linear binary classifier (Fig. 4.a) of the form:

$$f(x) = \begin{cases} 1, if\ w \cdot x + b > 0 \\ 0, otherwise \end{cases} \tag{2}$$

Where $w \cdot x$ is the dot product between input x and weight w matrices respectively. The b term stands for bias and is a parameter, independent from the input, introduced to help the decision boundary, move away from the origin. That, rather small change, in the expression of a perceptron, has an important impact as, a neuron with large bias, can easily be activated, or in other words output 1, while, on the other hand, a large negative bias, can affect the perceptron, making it extremely hard to get activated. The bias term can be thought as the parameter that controls the ease of a perceptron to produce 1

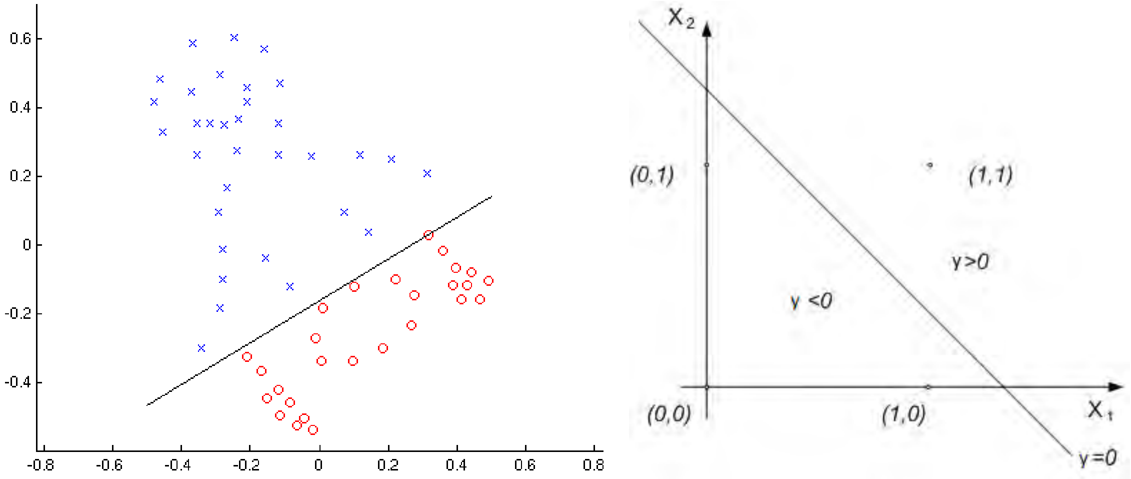


Figure 4 On the left, a generic linear classifier with form $\boldsymbol{f(x) = w \cdot x + b}$. On the right, the NAND gate input space

To illustrate the effectiveness and the simplicity of the above model, it is helpful to perform a basic example of modeling a typical NAND gate (Fig. 4.b). The gates have two inputs and one output and the behavior of it can be seen in Table 1.

Table 1 The NAND gate behavior

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

This elementary logic function can be expressed as a perceptron with two inputs and thus two weights, a bias and a threshold such that:

$$f(x_1, x_2) = \begin{cases} 1, if\ -2 \cdot x_1 \cdot -2 \cdot x_2 + 3 > 0 \\ 0, otherwise \end{cases} \tag{3}$$

where $w_1 = w_2 = -2, b = 3$.

This is a very simple example; however, the following question is raised: what can be done to model a more complex expression, like the XOR gate; or in general any function that is not, linearly separable? The following chapters illustrate how this can be done by combining multiple perceptrons forming a network and, more importantly, how to adjust the weights of each perceptron targeting a specific output.

2.3. Perceptron Weight Calculation

As seen in the previous example, it is possible to tune a perceptron to perform a binary classification by adjusting the weights and the threshold of the system. The question that arises is how to automatically compute those parameters. It is possible to be done by performing a brute force methodology, by trial and error of random weights at random thresholds, but that would be a resource wasteful idea without a guarantee backing it up. Thus the need for an automated way of computing those parameters arises.

There are many ways that have been proposed in order to train a neural network, but in this section a relatively simple one is presented which focuses on this specific example, yet it presents the basic principles behind the process which are used in the sections that follow, where an in-depth analysis is done for most commonly used ones.

In order to compute the weights and the threshold defined in (3) an iterative method is presented that adjusts these parameters in each iteration based on the output of the perceptron [3]. This process is called “training” of the network and requires a number of examples called “training dataset” which are presented to the network on each iteration. At each presentation small changes are applied to the parameters according to the “training rule”. It is important to note here that the process described above is called “supervised learning” as the network learns from a training dataset containing the inputs of the model along with the desired output.

Let w_i be the weights of the network, and x_i the input parameters. As training dataset let T defined as a set $T = \{t_1, t_2 \dots t_i\}$ containing example vectors $t_i = \{x_1, x_2 \dots x_i, y\}$ where y the desired output. To simplify the computations the weights combined with bias and the inputs of the model can be combined as a matrix W and X respectively (4).

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_i \\ b \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_i \end{bmatrix} \tag{4}$$

The output of the neuron z (5) is computed by computing the dot product between the two matrices of (4)

$$z = w \cdot x \tag{5}$$

To train the model the weight vector W is adjusted according to the distance of the output z compared to the desirable output y presented into the training vector t_i according to (6)

$$w^{'} = w \pm a \cdot t \tag{6}$$

where a a small number in range $0 < a < 1$ which is called “learning rate”. If the new output z is approaching the desired y then the term $a \cdot y$ will be positive and thus $w^{'} = w + a \cdot y$, on the other hand, if the desired y is moving away then $w^{'} = w - a \cdot y$. That leads to a learning rule that accounts the output of the network combined with the desired output which is defined in (7)

$$w^{'} = w + a(y - z) \cdot w \tag{7}$$

The (7) can now be rewritten as a desired weight change $\varDelta w = w^{'} - w$ and thus

$$\varDelta w = a(y - z) \cdot t \tag{8}$$

This equation is called the “perceptron rule” which was introduced in [1] and was historically the first used to train logical unit models as the one illustrated in previous section. The perceptron rule can be expressed algorithmically as:

```
while  != $y$:
    foreach $t_i$ in $T$:
        $z$ = evaluate model using $t_i$
        if  != $y$:
            $w$ = $w'$ according to (7)
        end if
    end foreach
end while
```

Variations of the above converge theorem was later introduced by [3] and later on in [4]. The question that arises here is how this theorem can be generalized to multiple perceptrons; in fact that was the main reason behind the [3] in which the obstacles were presented in detail. Furthermore this obstacle placed the neuroscience research into a halt for more than 15 years, as the power of the neural networks was put into question.

2.4. Activation Function

The activation function is an important part of a perceptron as it defines the way that the function behaves based on different input. Without activation functions the network would not be able to learn more complex, non-linear functional mapping between the input and the response variable. Their purpose is to translate an input signal of a neuron in a network to an output signal, usually within a specified range of values.

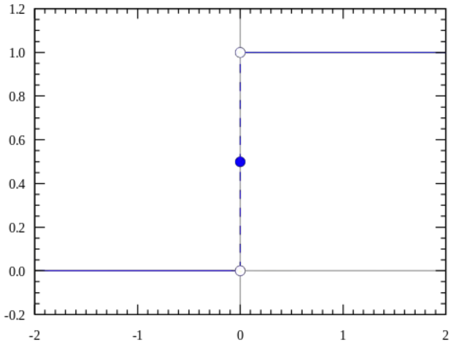
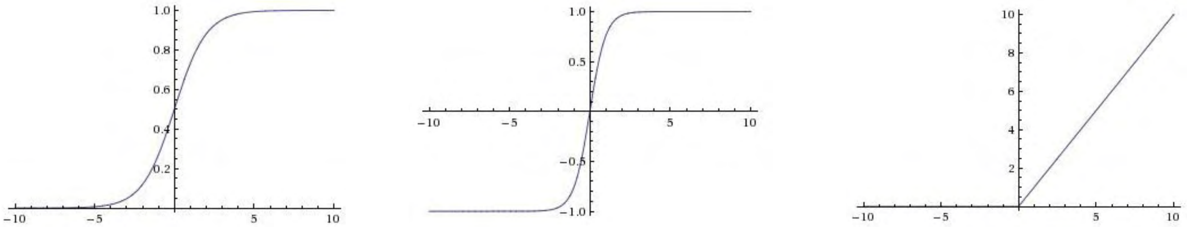


Figure 5 Step function

In the previous example the step function (Fig. 5) was utilized which outputs 1 only if the input is greater than 0, otherwise it outputs zero. There are number of issues with this function, with the most important one that, a small change in the input weights alter drastically the output of the neuron. A more desirable function would be one that has the property to change according to the magnitude of change on weights of the unit. In fact there are many functions that fulfill that property with the most commonly used ones listed below:



(a) Logistic function
$a(x) = 1/\left(1 + e^{-\beta x}\right)$

(b) Tanh function
$$a(x) = \frac{1 - e^{-2\beta x}}{1 + e^{-2\beta x}}$$

(c) ReLU function
$a(x) = max(0, x)$

Figure 6 Common activation functions

(Fig. 6.a) presents the logistic non-linear function which squashes the real number to the range between 0 and 1. The major advantage of this function is that it follows a smooth change to the output according to the provided input. On the other side, the squashing nature of it, makes it prone to computational mathematical loss as small real numbers, between 0 and 1, are produced. It is worth noting that when $\beta \rightarrow \infty$ the logistic function becomes same as the step function (Fig. 5).

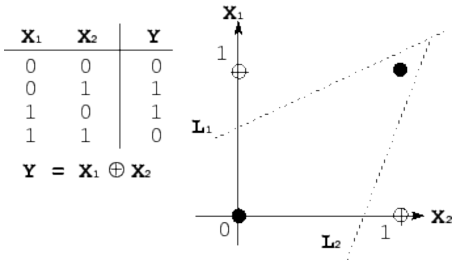
The tanh function illustrated in (Fig. 6.b) squashes real numbers between the range of -1 and 1 and effectively is used to overcome the issue of non zero-centered values. On the other hand it suffers from the issue of saturation while training which is examined the following sections. When the parameter $\beta \rightarrow \infty$ the function behaves same as the sign function. Logistic and tanh functions are both belong to a group of functions known as Sigmoid functions and that is the reason why neurons with this type of activation functions are commonly named “sigmoid neurons”.

The Rectified Linear Unit (ReLU), is an extremely simple to compute function compared to the previous ones, that became popular the last few years in the field of Artificial Neural Networks. In (Fig. 6.c) the function outputs 0 when $x < 0$ and then x when $x \geq 0$ is presented. It has the desirable properties of a smooth change according to the input while and it doesn’t saturate easily while training. Unfortunately, at present, there are not enough mathematical evidence to prove that this is the case; a well adopted heuristic argument is that the function doesn’t saturate in the limit of large x unlike the common sigmoid-like perceptron, which helps the training process to continue.

It is important to note here, that using the term “perceptron” to name the neurons that are using activation functions different than the step function comes to contradiction with the original source of the term. In fact a neuron that uses a sigmoid-like function is called “sigmoid neuron” and in general a neuron can be named after the activation function that uses. In bibliography though, the term perceptron has been used extensively to name any sort of artificial neuron and that might create confusion. For historical reasons, the neurons are named by the well accepted naming convention “perceptron” and use the term “sigmoid” or “ReLU” neuron only when it is appropriate to note the activation function that has been used.

2.5. Neural Networks

It has been shown that an artificial neuron, or perceptron, can be used as a simple, linear, binary classifier. Unfortunately when it comes to more complex function estimation a single neuron is unable to the respective problem. A rather simple, yet accurate example is the modeling of the XOR logical function. The XOR operator is a non-linear function which computes the eXclusive OR operation. The non-linearity of the function can be seen bellow (Fig. 7) along with the truth table of it.



X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

$Y = X_1 \oplus X_2$

Figure 7 The XOR function with the truth table on the left and the plot on the right.

As presented, there is no single line able separate the two dimensional space. For that reason the need of extending the previous, primitive single-neural model, arises in order to accumulate the power of more than one binary classifiers; and thus neurons. Bellow (Fig. 8) illustrates a more complex model containing two input units x_1 and x_2, with each one of them been connected to two neurons z_1 and z_2 and finally a single neuron Y, which represents the output of the neural network.

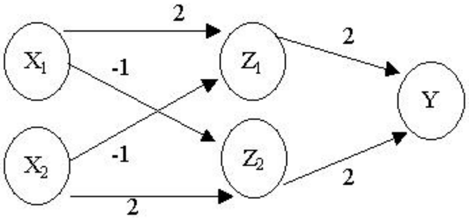


Figure 8 The XOR function with the truth table on the left and the plot on the right.

The x_1 and x_2 parameters can be thought as fixed neurons that they have no input and they produce always a constant value and thus that explains the reason of using the term units, instead of neurons as it is more appropriate and thus that is the reason the term "units" is used instead of "neurons" for the representation of them. It is also noticeable that all neurons are connected with each other in a "feed-forward", "fully-connected" way. The "feed-forward" term means that the input is always traveling forward throughout the neurons until they reach the output neuron while the "fully-connected" term stands for the fact that all neurons are connected between each other. A detailed exploration of what those terms mean along with examination of alternative neuron schemas is presented throughout the sections.

The fully connectivity of the neurons employed by the above three-neuron network, introduces six weight parameters for the model, along with three biases, which sum up to nine in total "free-parameters" that need to be fine tuned in order for the network to compute an acceptable estimation of the desired function. With that in mind a conclusive evidence is validated that, the number of "free-parameters" introduced into the system is increasing rapidly and thus, the complexity of the overall system increases. For the above example, it is easy to compute these weights by hand, but when it comes to larger networks, with some of them containing thousands, or even million parameters, the need of an automated parameter computation quickly becomes necessary.

3. Feed Forward Networks

The neural network architectures that are examined in this section are called "Feed Forward Networks". The name of these networks came from their property of the signal flow which is forwarded in one direction, following the neural connections from the input towards the output, without any backward connections. Another important property of this neural arrangement is the synchronous behavior of their signal propagation, meaning that their signal is propagated by one neuron at a time interval, without introduction of any kind of "delays" or "accumulation" of signals, which comes into contradiction with the human brain whom, neurons are connected in a much more complex way, allowing them to activate, asynchronously. Even with that in mind, this simplified approach of modeling the biological neurons, has proved extremely effective solving both classification and regression problems [5], [6].

3.1. Multilayer Perceptrons

The first neural architecture that is examined in this section is called "Multilayer Perceptron" (MLP). This architecture is effectively an extension of the perceptron theory, with the difference that instead of using one or two layers of perceptrons, one for the input and one for the output of the network, it incorporates one or many layers in between which are called hidden layers (Fig. 8). The name "hidden" is due to the fact that these layers are not visible in the input or output of the network and thus, they can be thought as a black box.

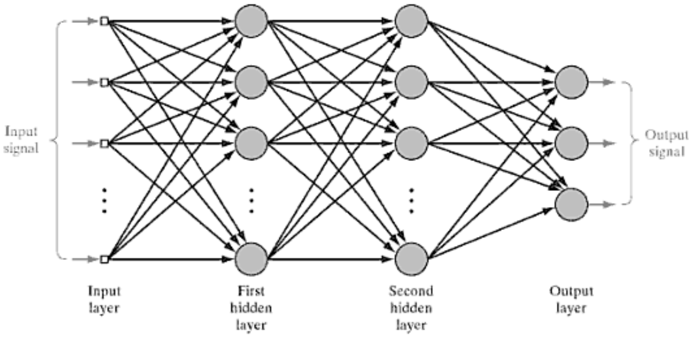


Figure 9 MLP network with two hidden layers of m neurons and three output neurons.

Another characteristic of the typical MLP networks is that they are fully connected, which means that every neuron is connected with all the neurons of the layer that follows (Fig. 9). This means that an MLP with 100 inputs, 2 hidden layers of 50 and 10 neurons respectively and 2 output neurons will result into $100 \cdot 50 + 50 \cdot 10 + 10 \cdot 2 = 5520$ weights and 62 biases to be optimized.

It worth noting that following the Turing Machine [7] and the fact that using perceptrons a neural network can simulate any logical TLU, it has been shown by [5] and [6] that a neural network of just one hidden layer, is able to simulate any kind of continuous function. This is also known as universal approximation theorem. That means that no matter the continuous function that is desirable to be computed, there is a neural network able to simulate it. This universality can also be extended to discontinues functions but only if the nature of the problem can accept a good approximated solution.

The question that has to be answered is, what are the reasons for the usage of more than one hidden layer in network architecture, as that comes in contradiction with the universal approximation theorem. In practice, the same principles with software development apply here; abstraction. Limiting an architecture to only one hidden layer, is resulting into an increase number of neurons, which introduce more free parameters to the system and thus, increase the overall complexity. To get around this issue, more hidden layers can be introduced, each sharing the simulation load with the others and reduce the number of weights and biases that have to be computed.

The introduction of multiple layers into the architecture raises a new type of hyper-parameter that needs to be computed, the number of hidden layers. Unfortunately there is no mathematical way of computing the number of hidden layers that are needed for each type of architecture or how many neurons each of the layer will have. Instead, researchers have developed heuristics methods which basically rely on train – validate – test approach. What that means is that a subset of the training data T is used as validation data while training the model. If the model does not converge on the validation data, then additional hidden layers can be added or removed and also alter the number of neurons of each layer [8][9].

3.2. Training of Neural Networks

In order to continue the analysis of the MLPs is important to answer the question of how to compute the weights and the biases of the model.

There are many training algorithms that are available today in bibliography, which mainly fall into two categories, genetic algorithms, and in general Evolutionary Algorithms [10][11],and back-propagation based.

3.3. Gradient Descent Optimization Algorithms

There are many training algorithms which belong to this category and an examination of the most popular ones starting from the least complicated is presented. Their main goal is to minimize the error of the output of the network while adjusting the weights and the biases of the model. To understand that, it helps to introduce a visual example on three dimensional space of an upside down cliff containing a ball (Fig. 10) where the goal is to guide the ball to reach the lowest point of the cliff called, global minima.

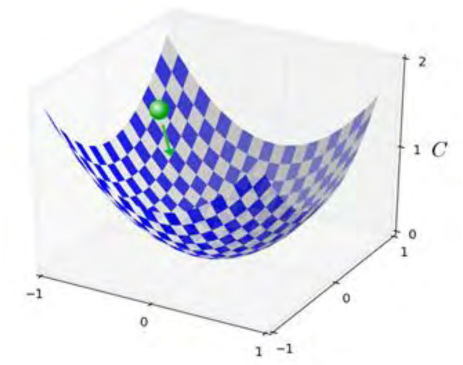


Figure 10 Visual example of an upside-down cliff and a ball following down-hill direction.

3.3.1. Loss Function

An important tool that allows the estimation of the performance of the output of an artificial neural network is called loss function. This function is used to quantify the error produced by the network compared to the expected value. In bibliography this function is commonly referred as “loss function”, “cost function” or “optimization function” and is that function output that is going to be minimized by adjusting the weights of the neurons. A loss function is defined as a function dependant on four parameters (9), where x_i, y_i the input and output values of the network.

$$C(w, b, x_i, y_i) \tag{9}$$

A commonly used loss function is called mean square error (MSE), which is also known as "quadratic function" and it is defined as follow

$$C(w, b, x, y) = \frac{1}{2n} \sum_x |y(x) - a|^2 \tag{10}$$

Here the w and the b parameters represent the weights and the biases of the network respectively, $y(x)$ the expected output and a the actual output of the network, while the n parameter represents the total number of examples presented to the model. By examining the function in more detail, it can be observed that the sum of the errors is always positive, yet it approaches $C(w, b) \cong 0$ when $y(x) \cong a$. Based on those observations this function can be used in order to minimize the difference between the output of the network and the desired value that is desired to estimate, thus and the name "optimization function".

There are many loss functions used in bibliography, such as the square root and the cross entropy loss function, which are presented in detail along with the examination of different analytical training algorithms together with the pros and cons of them throughout the rest of the sections.

3.4. Gradient Descent

A relatively simple yet powerful algorithm that has been used to train artificial neural network is called gradient descent and is one of the first introduced in the field. Gradient descent is an iterative algorithm that given an optimization function C, tries to minimize it in each iteration by altering the parameters of the function in small steps.

Lets imagine a function f with n parameters that is minimized based on C. Let Δx_i a small change in direction of the function that is applied to x_i original directions. Following the principles of calculus the change of C is obtained as (11) which has as a goal to obtain a negative ΔC and thus minimize the f. To do so, the need to find a way of altering the parameters Δx_1 and Δx_2, arises.

$$\Delta C \cong \frac{\vartheta C}{\vartheta x_1} \Delta x_1 + \cdots + \frac{\vartheta C}{\vartheta x_i} \Delta x_i \tag{11}$$

Lets define Δx as a transposed matrix of the variables and thus $\Delta x = (\Delta x_1, \dots, \Delta x_i)^T$ along with the gradient of C as (12)

$$\nabla C = \left(\frac{\vartheta C}{\vartheta x_1}, \dots, \frac{\vartheta C}{\vartheta x_i}\right)^T \tag{12}$$

The equation (11) can now be rewritten with respect of Δx and ∇C as (13)

$$\Delta C \cong \nabla C \cdot \Delta x \tag{13}$$

Based on (13) the goal of minimizing the ΔC become possible by the introduction of a parameter n called learning rate (14) which is a small positive parameter.

$$\Delta x = -n\nabla C \tag{14}$$

Following (13) ΔC can be written as (15)

$$\Delta C \cong -n\nabla C^2 \tag{15}$$

and because ∇C^2 will always be positive, $\Delta C \leq 0$ is guaranteed. The update now of the x vector can now be expressed as (16) which if repeated for certain amount of iterations, the parameters of the function f will reach a global minimum.

$$x \rightarrow x' = x - n\nabla C \tag{16}$$

3.4.1. Gradient Descent in Neural Networks

The gradient descent technique can be utilized in the field of neural networks directly by trying to estimate the best parameters on which the loss function C is dependent on (Fig. 11). These free parameters are the weights and biases of the network and thus applying them in (16) the equations (17) and (18) are obtained respectively.

$$w \rightarrow w' = w - n\frac{\vartheta C}{\vartheta w} \tag{17}$$

$$b \rightarrow b' = b - n\nabla C = b - \frac{\vartheta C}{\vartheta b} \tag{18}$$

The challenges of applying gradient descent on large number of parameters are profound as, the goal of it is to minimize the quadratic cost function C (9) and thus in order to compute the gradient ∇C the need to compute the ∇C_x is unavoidable for each training example as the function is ultimately an average computed by $C_x = \frac{|y(x)-a|^2}{2}$ and thus $\nabla C = \frac{1}{n}\sum_x \nabla C_x$. The last computation is expensive especially in terms of time for large number of training examples and thus the learning, or in other words the minimization process of C, will occur slowly.

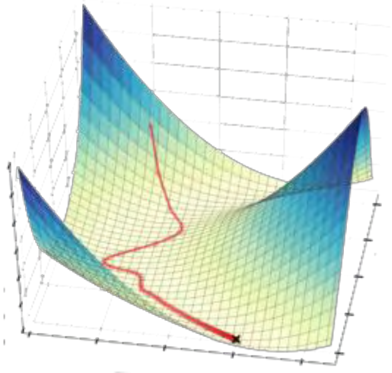


Figure 11 Visual example of loss function minimization with the value (red line) been minimized over time until it reaches the global minima (black ×) of the function.

3.4.2. Stochastic Gradient Descent

To overcome the slow learning limitation of the normal gradient descent procedure, stochastic gradient descent (SGD) was introduced. This variation works by randomly picking small number m training examples, which is known as "mini-batch" and computing the gradient descent based on them. In other words the gradient descent is estimated instead of fully computed over the whole number of training input and thus the process is speeded up significantly. If the number of training examples is enough, the estimation will be close to the original gradient and thus converge faster.

Let mini-batch $M = \{X_1, X_2 \dots, X_m\}$ of training dataset T, where $M \subseteq T$, gradient descent can be estimated by averaging the ∇C_{X_j} and thus (19) which confirms that the computation of can be obtained by a randomly chosen subset of T (20)

$$\nabla C = \frac{\sum_x \nabla C_x}{n} \cong \frac{\sum_j^m \nabla C_{X_j}}{m} \tag{19}$$

$$\nabla C \cong \frac{1}{m}\sum_j^m \nabla C_{X_j} \tag{20}$$

Using (19) in the field of neural networks an estimation of ∇C_w and ∇C_b as (21) and (22) respectively is feasible,

$$w \rightarrow w^{'} = w - \frac{n}{m}\sum_j \frac{\vartheta C X_j}{\vartheta w} \qquad (21)$$

$$b \rightarrow b^{'} = b - \frac{n}{m}\sum_j \frac{\vartheta C X_j}{\vartheta b} \qquad (22)$$

where the $\sum_j$ are the sums over all the examples presented in the mini-batch M. This can be applied over all the training examples in order to compute the changes to the weights and biases of the network. Doing so is called "epoch" which represent a full iteration over all the training examples presented to the model. Repeating the same process over and over again the loss function is minimized.

The nature of stochastic gradient descent, which is based on estimation of the gradient, leads to a more noisy convergence compared to the classic gradient descent approach (Fig 12). This might lead to a wrong assumption that a model does not perform well based on the chosen hyper-parameters. Based on that, monitoring of the network behavior should be done with patience. It worth mentioning at this stage that there is a special kind of stochastic gradient descent; the one with a mini-batch of size 1. This special case can be used in "on-line" learning approach (which is practiced when only one training example is present, typically provided by a data-stream), where for each training example a gradient descent update is performed. Unfortunately that does not always converge and the learning might end up in a local minima.

Another important advantage of this procedure is that the learning is feasible without prior knowledge of the whole training dataset. That gives the advantage of automatically generating training examples while training and thus minimizes the storage and memory exhaustion of the computational unit that the algorithm runs on.

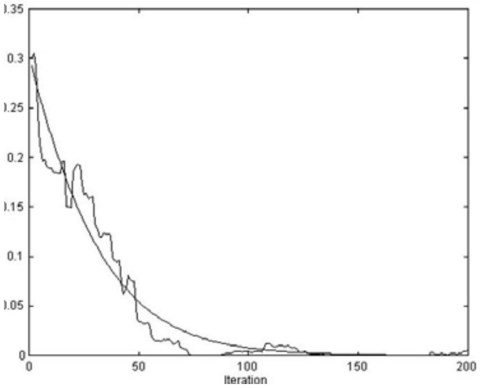


Figure 12 Representation of GD (smooth line) versus SGD (noisy line) converge.

3.4.3. Mini-Batch Gradient Descent

Mini-batch gradient descent (MBGD) is relatively similar to SGD, as it still relies on mini-batches in order to shrink the time spend for the computation of the gradient, but instead of updating the weights and biases of the model of every epoch, it updates on every mini-batch. That reduces the fluctuation that is present on the original SGD graph and thus combines the best out of both GD and SGD approaches.

It is important to note here that because it performs the update on every mini-batch completion, it leverages new techniques of matrix multiplication on the Graphical Processing Unit (GPU) of a computer without having the need to transfer the result of each mini-batch back to RAM which involves time spent in CPU, this fact allows faster parameter computation and thus faster training.

Batch sizes are heuristically chosen and usually fell between 50 and 250 training examples, yet that's purely dependant on the nature of the problem. With that said, a common approach of batch-size selection is the trial and error or the usage of Genetic Algorithms for parameter optimization.

Unfortunately this algorithm does not always guarantee good convergence as it raises multiple difficulties on learning rate selection. As expected a really low learning rate might result into slow learning while a large one can introduce big fluctuation and thus unstable learning.

For this reason there have been developed ad-hoc solutions like, learning rate scheduling [12][13] which can adjust the learning rate based on the fluctuation of the cost function. As an example when the fluctuation is small, the learning rate can be increased while on the opposite, decrease. These rules unfortunately have to be defined in advance and thus are not able to adapt the dataset idiomorphic characteristics [14].

3.4.4. Newton's Optimization

A different approach of minimizing a loss function C, is by following Newton's optimization method [15]. Let $C(w)$ be the loss function that is subject to be minimized, with $w =$

$\{w_1, w_2 \dots w_n\}$ the corresponding weights of a network. Following the Taylor's theorem, an approximation of the loss function can be expressed as (23)

$$C(w + \Delta w) = C(w) + \sum_j \frac{\vartheta C}{\vartheta X w_j} \Delta w + \frac{1}{2} \sum_{jk} \Delta w_j \frac{\vartheta^2 C}{\vartheta w_j \vartheta w_k} + \cdots \qquad (23)$$

Rewriting (23) with respect of gradient vector ∇C, (24) is obtained

$$C(w + \Delta w) = C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w + \cdots \qquad (24)$$

Where H represents the Hessian matrix, with j, k_{th}entries set as $\frac{\vartheta^2 C}{\vartheta w_j \vartheta w_k}$. The C can now be estimated by just computing the (25)

$$C(w + \Delta w) \cong C(w) + \nabla C \cdot \Delta w + \frac{1}{2} \Delta w^T H \Delta w \qquad (25)$$

Now utilizing calculus the (24) can be minimized by applying (25)

$$\Delta w = -H^{-1} \nabla C \qquad (26)$$

Following (26) an estimation that there will be a significant decrease in C can be obtained by applying (27)

$$w \rightarrow w^{'} = w - H^{-1} \nabla C \qquad (27)$$

(26) can also be extended for the bias terms as (28)

$$b \rightarrow b^{'} = b - H^{-1} \nabla C \qquad (28)$$

Applying that iteratively an algorithm can be formed that minimizes the loss function$C$ known as Newton's optimization algorithm which can be expressed in the following iterative steps:

1) Choose a starting point w
2) Compute $w^{'}$ by applying (26)
3) Update $w^{'}$ to new $w^{''}$ by replying (26) and thus $w^{''} = w^{'} - H^{'^{-1}} \nabla' C$

The above approximation can be further optimized by minimizing the step of change on the second step by introducing a learning rate parameter n and thus (26) can be rewritten to (29)

$$\Delta w = -n H^{-1} \nabla C \qquad (29)$$

Theoretically the above technique can lead to quicker converge than the standard gradient descent approach. Unfortunately in practice the usage of second order derivatives in impractical and relatively expensive as on each step it requires the calculation of all first order derivatives of the model. In a neural network, where the number of free parameters are thousands if not millions, makes the technique not computation wise, feasible. As an example imagine a network with 10^5 free parameters. The Hessian matrix [16] itself it must be $10^5 \cdot 10^5 = 10^{10}$ total entries, which is unreasonable in terms of both memory and computation consummation. For the these reasons this methodology is not used in practice and thus more advanced algorithms have been developed.

3.4.5. Stochastic Gradient Descent with Momentum

One problematic characteristic of the original SGD algorithm is when local minima create steep curves in one dimension of the optimization function [17]. That steepness affects the algorithm in such a way, which leads to hesitant movement towards the local minima slope (Fig. 13).

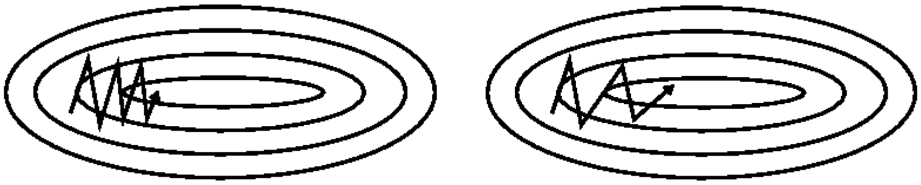


Figure 13 A) Vanilla SGD B) SGD with momentum [18]

To deal with this, momentum parameter has been introduced [19] to the gradient computation. That parameter, which in fact affects the velocity of the minimization rate, mimics closely the Hessian based technique, but without having to compute second order derivatives and thus, the computation complexity is reduced. To understand how momentum parameter m affects the conversion speed a redefinition of the update of the weights and biases from SGD equations is needed.

By introducing a term $\boldsymbol{v}$ which stands for velocity and the change of it as (30)

$$v \to v^{'} = mv - n\nabla C \tag{30}$$

where m a constant momentum and n the learning rate the update rule of $w \to w'$ can be expressed as (31)

$$w \to w^{'} = w + v' \tag{31}$$

The m parameter value ranges between 0 and 1. When $m = 1$ from (30) is understandable that v will build up on each iteration and thus the speed of change will become quickly, really high and thus, fluctuation on learning will occur. On the other hand when $m = 0$ velocity will not build up and (28) will become the original gradient descent algorithm. Thus the m parameter controls the accumulation of speed of change towards the direction of the global minimum.

3.4.6. Nesterov Accelerated Gradient

One of the problems that the variation of gradient descent, momentum, suffers is that even if the velocity of the estimated converge varies; the direction of it is not accelerated and thus, might end up in local minima. To overcome this limitation Nesterov Accelerated Gradient (NAG) [20] introduced.

The approach of NAG is instead of using the current v for the w parameter update, to alter (30) to (32). This is done as an attempt to approximately look into the future direction of the gradient ant thus instead of blindly searching for the global minima, the gradient direction is forced to move violently to the correct direction (Fig. 14).

$$v \rightarrow v^{'} = mv - n\nabla C(w - mv) \tag{32}$$

Compared to the previous algorithms, momentum gradient descent initially moves towards the minima with a small step (small blue line) and then, the accumulated velocity pushes for big “jump” ahead (long blue line). NAG (green line), initially makes a big jump towards the direction of the previous gradient (brown line) and then measures the gradient where it ends up in order to make the correction (red line).

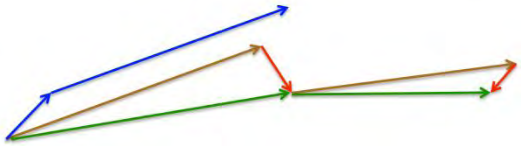


Figure 14 Momentum gradient descent (blue line) versus NAG (green line).
Brown vector shows the jumps while red vector the change in direction of NAG. [21]

3.4.7. Adagrad

An addition to the original gradient based optimization algorithms is Adagrad [22] . This algorithm performs learning rate adaptation based on the current state of free parameters. Larger updates are done on the infrequent used parameters while smaller ones to the ones are frequently used which, in this context it is well suited for sparse data.

To incorporate Adagrad algorithm into the classical SGD approach an alternation of the symbols used for the updates of each free parameter is needed, as now, every update has a different learning rate.

Let $w_{t,i}$ be the free parameter that has to be updated were i the index of the weight and t the time step. Let also $\nabla C\left(w_{t,i}\right)$ be the gradient of i parameter at time step t. Thus the update can be expressed as (33)

$$w_{t,i} \to w_{t+1,i} = w_{t,i} - n\nabla C\left(w_{t,i}\right) \tag{33}$$

Using Adagrad the learning rate will be adjust for every next time step such as (34)

$$w_{t,i} \to w_{t+1,i} = w_{t,i} - \frac{n}{D_t}\nabla C\left(w_{t,i}\right) \tag{34}$$

where $D_t = \sqrt{G_t + \varepsilon}$,ε is a smooth term usually set to $1e - 8$ and G_t a diagonal matrix with values the sum of squires of the gradients of all the previous time steps.
Vectorizing the above equations an element-wise matrix multiplication can be performed between the G_t and the $\nabla C(w_t)$(32), where $\odot$ is the element wise matrix vector multiplication.

$$w_t \to w_{t+1} = w_t - \frac{n}{D_t} \odot \nabla C(w_t) \tag{35}$$

An example usage of it which reported great benefits over the traditional SGD algorithm was the one used by Google[23] to recognize cats on YouTube videos. Another successful example that was based on Adagrad was [24] which used the algorithm for the purpose of training a neural network for the “Glove word embeddings”, where naturally frequent words require much smaller updates than the infrequent ones.

An important gain from the Adagrad algorithm is the fact that are not rely on manually setting the learning rate n; a small learning rate can be set initially, usually $\mathrm{n} = 0.001$, and thus, the need for trial and error approaches for optimization of this parameter, is no longer needed. On the other hand Adagrad suffers from exploding sums, as every added term is positive which leads to gradually vanishing learning rate and thus the learning process is set on halt.

3.4.8. Adadelta

To solve the problem that Adagrad presents regarding the exploding sums which lead into vanishing learning rate Adadelta [25] was proposed. Adadelta took a relatively simple approach of fixing the issue by using only fixed size w of accumulated gradients. The accumulated gradients are stored in an efficient way by recursively define them as a decaying average of all the previously computed squared gradients.

Let $E[\nabla C(w)^2]_t$ be the running average at the time step t and γ a momentum like parameter which can be expressed as (36) where γ value is usually 0.9.

$$E[\nabla C(w)^2]_t = \gamma E[\nabla C(w)^2]_{t-1} + (1-\gamma)\nabla C(w)_t^2 \tag{36}$$

The classical SGD can now be rewritten as (37) and (38)

$$\Delta w_t = \frac{-n}{D_t} \odot \nabla C(w_t) \tag{37}$$

$$w_{t+1} = w_t + \Delta w_t \tag{38}$$

where $D_t = \sqrt{G_t + \varepsilon}$ similar to Adagrad but now, G_t is the decaying average vector over the previously computed squared gradients $E[\nabla C(w)^2]_t$which results into (39)

$$\Delta w_t = \frac{-n}{\sqrt{E[\nabla C(w)^2]_t + \varepsilon}} \odot \nabla C(w_t) \tag{39}$$

The parameter $E[\nabla C(w)^2]_t$ can be replaced with root mean square error criteria (RMS) of the gradient. Based on the (39) it can be seen that the need for an initial learning rate is eliminated as the update rule can be computed by (40) and (41).

$$\Delta w_t = \frac{-RMS[\Delta w]_{t-1}}{RMS[\nabla C(w)]_t} \nabla C(w_t) \tag{40}$$

$$w_{t+1} = w_t + \Delta w_t \tag{41}$$

That update rule change relies on the fact that $E[\Delta w^2]_t = \gamma E[\nabla C(w)^2]_{t-1} + (1-\gamma)\Delta w_t^2$ and thus the RMS parameter updates are (42)

$$RMS[\Delta w]_t = \sqrt{E[\Delta w^2]_t + \varepsilon} \tag{42}$$

as $RMS[\Delta w]_t$ is unknown and the approximation of it can be achieved by computing the RMS until $RMS[\Delta w]_{t-1}$.

3.4.9. RMSprop

A rather successful yet unpublished methodology which reminds Adadelta and was developed to solve the same problem of vanishing learning rate, learning rate is called RMSProp [21]. The update rule is computed by (43) and (44).

$$E[\nabla C(w)^2]_t = 0.9E[\nabla C(w)^2]_{t-1} + 0.1\nabla C(w)_t^2 \tag{43}$$

$$w_{t+1} = w_t - \frac{n}{\sqrt{E[\nabla C(w)]_t^2 + \varepsilon}} \nabla C(w)_t^2 \tag{44}$$

The author suggests an initial learning rate of $n = 0.001$ and $\gamma = 0.9$. The difference between the Adadelta is that RMSProp using exponentially decaying average of the squared gradients, divides the learning rate. That also imposes that the algorithm depends on the learning rate that is set which is the main difference in comparison to Adadelta which, eliminates the learning rate factor from the update rule.

3.4.10. Adam

Another method to compute gradient descent with adaptive learning is called Adaptive Moment Estimation (Adam) [26]. The methodology was inspired by Adadelta and RMSProp keeping the exponentially decaying gradient average factor v_t (46) and extending it by incorporating an exponentially decaying average of the previous gradients m_t (45), resembling the momentum methodology. These two vectors are estimates of the first moment (mean) and second moment (uncentered variance) of gradients, respectively.

$$m_t = p_1 m_{t-1} + (1 - p_1)\nabla C(w)_t \tag{45}$$

$$v_t = p_2 v_{t-1} + (1 - p_2)\nabla C(w)_t^2 \tag{46}$$

where p_1, p_2 are the decay rates. The authors suggest an initial value of $p_1 = 0.9$ and $p_2 = 0.999$. The update rule can now be rewritten as (47).

$$w_{t+1} = w_t - \frac{n}{\sqrt{v_t} + 10^{-8}} m_t \tag{47}$$

3.4.11. AdaMax

A variation of the Adam algorithm also proposed in the same publication by the authors is called AdaMax [26]. Adam’s update rule scales the gradient inversely proportionally to the l_2 norm of the previous gradients as the rule contains the v_{t-1} term and current gradient $|\nabla C(w)_t|^2$. Leveraging that fact the authors proposed an alternative in which, the l_2 norm was extended to l_∞ norm. That has been accomplished by rewriting the (46) as (48) where n is the norm factor.

$$v_t = p_1^n v_{t-1} + (1 - p_1^n)|\nabla C(w)_t|^n \tag{48}$$

A large number in n parameter results into numerically unstable computations and thus a usual range is either l_1 or l_2. A special case of infinite n also results into stable behavior where (48) can be now rewritten as (49), with the exception that now the vector is detonated as u_t, in favor of not mixing the two equations.

$$u_t = p_2^\infty v_{t-1} + (1 - p_2^\infty)|\nabla C(w)_t|^\infty = max(p_2 v_{t-1}, |\nabla C(w)_t|) \tag{49}$$

By applying the (49) to the update rule of Adam (50) is obtained

$$w_{t+1} = w_t - \frac{n}{u_t} m_t \tag{50}$$

As the (49) relies on u_t relies on max operation, it is not recommended to bias towards the zero in oppose to the original Adam and thus that explains why there is not computation for the bias correction of u_t. Authors suggest $n = 0.002$ and decay rates p_1, p_2 to be 0.9 and 0.999 respectively.

3.4.12. Nadam

An incorporation of Adam algorithm, which combines RMSprop with momentum, with Nesterov Acceleration Gradient (NAG) is called Nesterov-accelerated Adaptive Moment Estimation [27]. For the incorporation of Adam, m_t vector (45) has to be modified into (51) and thus the update rule is translated into (52), where γ a momentum like parameter.

$$m_t = \gamma m_{t-1} + n\nabla C(w_t - \gamma m_{t-1}) \tag{51}$$

$$w_{t+1} = w_t - m_t \tag{52}$$

Nadam proposed an alternation to NAG computation, where instead of computing the momentum step twice, to apply the momentum look-ahead directly on the vector update and thus (51) and (52) can be rewritten as (53) and (54) respectively.

$$m_t = \gamma m_{t-1} + n\nabla C(w_t) \tag{53}$$

$$w_{t+1} = w_t - \left(\gamma m_t + n\nabla C(w_t)\right) \tag{54}$$

With that alternation, instead of using the previous m_{t-1} for the update, it relies only on the current m_t momentum vector**.**

For the incorporation of the Nesterov momentum to Adam, the replacements of original algorithm are similar as with NAG (55), (56).

$$m_t = p_1 m_{t-1} + (1 - p_1)\nabla C(w)_t \tag{55}$$

$$w_{t+1} = w_t - \frac{n}{\sqrt{v_t}+\varepsilon}\left(\frac{p_1 m_{t-1} + (1-p_1)\nabla C(w)_t}{1-p_1^t}\right) \tag{56}$$

Where p_1 the decay rate and $\frac{p_1 m_{t-1}}{1-p_1^t}$ is a bias corrected estimation of the momentum vector from the previous step, which is the estimation of current momentum vector m_t. Replacing the (56) with that (57) is obtained which is the final update rule of Nadam.

$$w_{t+1} = w_t - \frac{n}{\sqrt{v_t}+\varepsilon}\left(p_1 m_t \frac{(1-p_1)\nabla C(w)_t}{1-p_1^t}\right) \tag{57}$$

3.4.13. Conclusion on Optimization Functions

In the previous sections I presented the most used optimization algorithms that are available today. Yet the subject of finding better performing, in respect of both time and performance, algorithms is still ongoing and thus a lot of effort has been put towards this field in the last years. Concluding, in (Fig. 15) a visual representation of the paths which the presented algorithms are following in order to reach the global minimum of function, illustrated in 2D surface.

What is noticeable in this illustration is that, SGD does not reach the global minima, while the rest, although they might follow different directions, they succeed finding it. NAG and Momentum are actually falling completely off track, and later on, correct themselves, violently to the correct direction. This is due to the momentum incorporation in their update rule. It worth noting here that NAG look-ahead estimation is able to quickly alter the direction towards the negative slope.

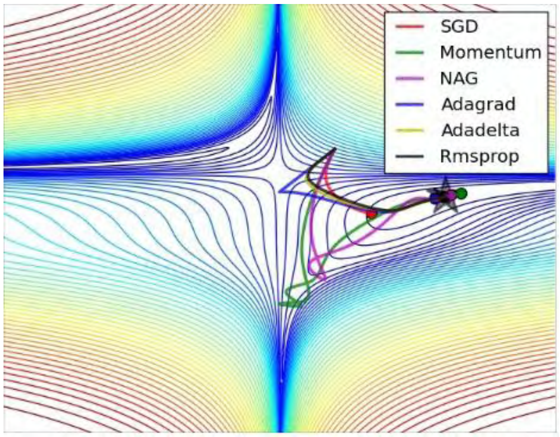


Figure 15 Visual representation of the algorithms [18]

Summarizing, Adam and RMSProp are the most advanced optimizers that are showing good performance in both computational complexity and speed [26]. Yet, interesting in recent publications, vanilla stochastic gradient descent with learning rate scheduling is dominating the field, although it takes significant more time to converge and run the risk of getting stack in local minima.

3.5. Error Backpropagation

In the previous sections, numerous ways to optimize a function were presented; the incorporation of them on how to train an artificial neural network has yet to be examined. The most well known methodology to train, or in other words, calculate the optimal weights and biases of a multilayer neural network is called backpropagation. This algorithm solves the problem of applying changes to the free-parameters of the network, with respect of the output error. It was originally introduced in mid 70s but did not get enough attention until the work of [28] which illustrated many neural network architectures where backpropagation works much faster than the earlier training approaches.

Let w_{jk}^{l} be the weight on k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron on l^{th} (Fig. 16). Similarly let b_j^l, a_j^l be the bias and the activation function of the j^{th} neuron in the l^{th} layer, respectively. An activation function of a specific neuron can now be expressed as (58) which sums all the neurons k of the layer $l-1$. To increase the computational performance of the equation (58) a rewrite can be performed as (59) which utilizes matrix multiplication operations

$$a_j^l = \sigma\left(\sum_k w_{jk}^l \; a_j^{l-1} + b_j^l\right) \tag{58}$$

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right) \tag{59}$$

where a^lnotates a matrix of the activation function outputs of the l^{th} layer, w^l and b^l matrices with all the weights and biases of the neurons in l^{th} layer and a^{l-1} a matrix with the activation function output of each neuron of the previous layer.

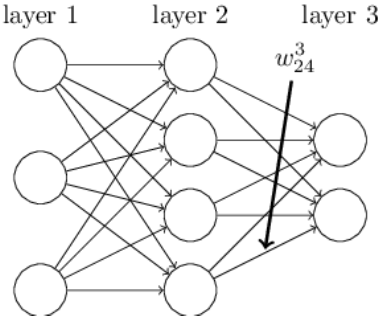


Figure 16 Weight vector w_{24}^3 detonating connection from 4th node of the second layer to the 3rd layer [29]

While applying the equation (59) on a forward pass of the network to get the output of the last layer, $l-1$ intermediate activation function matrices are computed. These matrices, which are denoted as z^l, are worth to be kept as they are used to speed up the next steps of the backpropagation algorithm.

3.5.1. Loss Function Characteristics

Backpropagation has as a target to compute the partial derivatives $\frac{\vartheta C}{\vartheta w}, \frac{\vartheta C}{\vartheta b}$ of the cost function C with respect of all the weights and bias of the network. In order to do that, backpropagation relies on two characteristics of the cost function. The first characteristic is that C can be expressed as a an average sum of all C_t of individual all training examples and thus $C = \frac{1}{n}\sum_x C_x$. The reason of this is that backpropagation is able to compute the $\frac{\vartheta C_t}{\vartheta w}, \frac{\vartheta C_t}{\vartheta b}$ and later or average them in order to compute the actual$\frac{\vartheta C}{\vartheta w}, \frac{\vartheta C}{\vartheta b}$. The second characteristic is to be able to express the cost function as a function of the output of each neuron of the last layer and thus be able to express C as $C = C\left(a^l\right)$.

As an example the quadratic cost function, where for a single training example t can be written as (60)

$$C = \frac{1}{2}(|y - a^L|)^2 = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2 \tag{60}$$

where y is the desired output of the training example t and thus it demonstrates a function which relies on the output of the activations.

3.5.2. Fundamental Equations of Backpropagation Algorithm

Backpropagation algorithm, ultimately tries to compute the $\frac{\vartheta C}{\vartheta w_{jk}^{l}}, \frac{\vartheta C}{\vartheta b_j^l}$ based on the error of the cost function C. In order to compute that an intermediate quantity is introduced, called "error of the neuron j^{th} neuron in the l^{th} layer" notated as δ_j^l. The goal is to compute the δ_j^l matrices using backpropagation and later then associate them with $\frac{\vartheta C}{\vartheta w_{jk}^{l}}$ and $\frac{\vartheta C}{\vartheta b_j^l}$ partial derivatives.

The error $\delta_j^l$is defined as a small noise introduced when the input passes throwout the neurons of the network in the feed-forward pass of the input signals. That small noise can be defined as $\Delta z_j^l$to the input weights of the neuron, which instead of producing an output of $\sigma\left(z_j^l\right)$, it produces $\sigma\left(z_j^l + \Delta z_j^l\right)$. This small change propagates all way to the output neurons of the network and affects the overall performance of the network, creating an overall effect that can be expressed as $\frac{\vartheta C}{\vartheta z_j^l}\Delta z_j^l$. The goal now is to compute the Δz_j^lthat reduces the overall loss function Coutput and thus when the quantity $\frac{\vartheta C}{\vartheta z_j^l}$ is large can lower down the cost by choosing a$\Delta z_j^l$with opposite sign. On the other hand when the $\frac{\vartheta C}{\vartheta z_j^l}$ is close to zero, the Δz_j^lhas also to be close to zero, which in other words it means that the neuron is already optimized. Based on the above the quantity δ_j^l can be defined as (61)

$$\delta_j^l \equiv \frac{\vartheta C}{\vartheta z_j^l}, \delta^l \equiv \frac{\vartheta C}{\vartheta z^l} \tag{61}$$

where δ^l is the error of the l^{th} layer in a vectorized form.

In the output layer of a neural network the components δ^l can be computed as (62)

$$\delta_j^L \equiv \frac{\vartheta C}{\vartheta a_j^L} \sigma'\left(z_j^L\right) \tag{62}$$

The first part of this expression, i.e. $\frac{\vartheta C}{\vartheta a_j^L}$, encapsulates the rate of change of the cost function in respect of the activation function output of the j^{th}neuron of the last layer, where the last part $\sigma'\left(z_j^L\right)$, measures the rate of change of the activation function σat z_j^L. Computationally the equation (62) is relatively easy to calculate. The only additional overhead is to the computation of $\sigma'\left(z_j^L\right)$. As an example by using the quadratic cost function the computation is as simple as computing (63).

$$\frac{\vartheta C}{\vartheta a_j^L} = a_j^L - y_j \tag{63}$$

Equation (62) is expressing the rate of change of each neuron and thus to speed up the computations a rewrite is needed to the matrix-based equivalent (64), where $\nabla_a C$ is a matrix whom components are the partial derivatives $\frac{\vartheta C}{\vartheta a_j^L}$

$$\delta^L = \nabla_a C \sigma'(z^L) \tag{64}$$

The second equation that backpropagation relies on, computes the error δ^lin terms of the error in the next layer and can be expressed as (65)

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right) \tag{65}$$

where $\left(w^{l+1}\right)^T$is the transposed weight matrix of the $(l+1)^{th}$layer. Breaking down the components of (65) it can be seen that first component $\left(w^{l+1}\right)^T$multiplied by the error $\delta^{(l+1)}$ expresses that the error is passed backward through the network and thus expresses an error of the l^{th} layer. The second component, relying on Hadamard product[1] $\odot \sigma'(z^l)$, expresses the same thing but for the activation function and thus expresses a backward error propagation through the activation function of the l^{th}layer. By combining the (62) and (65) the error δ^l is computed for any layer of the network.

[1] Hadamard product: binary operation between two matrices A, B of the same dimension which produces a matrix of the same dimensions such as $A \odot B = A_{i,j} \cdot B_{j,i}$, where $A_{i,j}$ the element at row i and column j of the matrix A.

The third equation that backpropagation relies on has to do with the rate of change of the cost function with respect to any bias of the network and can be expressed as (66) which can be equivalently rewritten in its matrix form (67).

$$\frac{\vartheta C}{\vartheta b_j^l} = \delta_j^l \tag{66}$$

$$\frac{\vartheta C}{\vartheta b} = \delta \tag{67}$$

Based on (67) and because (62) and (65) is becoming clear how to compute exactly the error δ^l and thus the quantity $\frac{\vartheta C}{\vartheta b}$ is already known by the previous steps. It is noticeable here that the δ parameter of (67) is the error of the same neuron as the *b* bias term.

The last equation that backpropagation makes use of is (68), which computes the rate of change of the cost function with respect of any weight in neural network from which, it is already known how to compute the error $\delta^l$and the a^{l-1}.

$$\frac{\vartheta C}{\vartheta w} = a_{\text{in}}\delta_{\text{out}} \tag{68}$$

In other words the above equation expresses the quantity of $\frac{\vartheta C}{\vartheta w} = a_{\text{in}}\delta_{\text{out}}$ from which the a_{in} is the activation of the neural input to the weight $w$and δ_{out} is the error of the neural output with weights w. One important characteristic of the a_{in}parameter in the partial derivative computation is that if the weight output from low-activation neurons learn slowly as if $a_{\text{in}} \approx 0$ then the gradient $\frac{\vartheta C}{\vartheta w}$ will also be small. Another useful insight from the (62) is that if $\sigma(z^l) \approx 0$ or $\sigma(z^l) \approx 1$ then the $\sigma'(z^l)$ is approximately zero as the sigmoid function behaves flat on regions close to zero or one. As a consequence of that it is becoming clear that a weight in the final layer will learn slowly if the output neuron value is too high or too low which in bibliography is called “saturated neuron” as the weights have stopped learning. Same applies for the biases of the output neurons. By following the second equation of backpropagation (65) the above can be extended for all the neurons of the network.

3.5.3. The Backpropagation Algorithm

Backpropagation can be expressed algorithmically by following the bellow 5 steps

1) Input t: Set the activations a^1 for the input layer

2) Feedforward phase: for each layer l in $\{2,3,\ldots,L\}$ compute the $z^l = w^l a^{(l-1)} + b^l$ and $a^l = \sigma(z^l)$

3) Output error δ^L: Compute the matrix $\delta^L = \nabla_a C \odot \sigma'(z^L)$

4) Backpropagate the error through the network: for each layer l in $\{L-1, L-2, \ldots, 2\}$ compute $\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l)$

5) Output: Compute the $\frac{\vartheta C}{\vartheta w} = a_{\text{in}} \delta_{\text{out}}$ and $\frac{\vartheta C}{\vartheta b_j^l} = \delta_j^l$ which are the gradient of the cost function C

By iterating the above steps it is visible that the error is backpropagated through the network and hens, where the name of the algorithm comes from. It seems ambiguous that the algorithm starts in reverse as it contradicts the normal expectations, but that comes from the fact that in order to compute the overall error of the entire network which is a result of previous errors accumulated by the neurons of each layer, a feed-forward phase need to be performed first, and then as a result of that it is becoming possible to apply the chain rule, from mathematical calculus, working backwards throughout the previous layers in order to obtain the final expression.

The same algorithm can be used in mini-batch gradient descent by following the steps outlined below.

1) Input: training examples

2) For each training example t: set the input activations $a^{t,1}$

   2.1) Feedforward phase: For each l in $\{2,3,\ldots,L\}$ compute $z^{t,l} = w^l a^{(t,l-1)} + b^l$ and $a^{t,l} = \sigma(z^{t,l})$

   2.2) Output error $\delta^{t,L}$: Compute the matrix $\delta^{t,L} = \nabla_a C \odot \sigma'(z^L)$

   2.3) Backpropagate the error through the network: for each layer l in $\{L-1, L-2, \ldots, 2\}$ compute $\delta^{t,l} = \left((w^{l+1})^T \delta^{t,l+1}\right) \odot \sigma'(z^{t,l})$

3) Gradient descent: For each layer l in $\{L-1, L-2, \ldots, 2\}$ update the weights according to the rule $w^l \rightarrow w^l - n/m \sum_t \delta^{t,l}$ $\alpha^{\mathrm{t,l-1}}$ and the biases $b^l \rightarrow b^l - n/m \sum_t \delta^{t,l}$

To implement the above as a stochastic gradient descent, the only change that needs to be applied is to wrap the steps into an outer loop that selects a subset of the training examples and train with that in multiple epochs.

3.6. The Loss Function

Loss function plays a major role in the training process of a neural network as backpropagation relies on it in order to estimate the change in the weights and biases of the network. Choosing the right loss function might improve the performance of the training process significantly. In previous sections I presented a simple cost function called “Root Mean Square Error” (RMS). This cost function works well for small neural networks, however as the network becomes deeper, the problem of “slow learning” is becoming unavoidable. To understand why this is happening, it helps to consider the case of simple case scenario where the desired output of a single sigmoid neuron is zero when the input is one (69).

$$\mathrm{C} = \frac{(\mathrm{y}-\alpha)^2}{2} \tag{69}$$

The partial derivatives of (69) are computed by the equations (70) and (71) for the corresponding weights and biases respectively:

$$\frac{\vartheta C}{\vartheta w} = (a-y)\sigma^{'}(z)t = a\sigma^{'}(z) \tag{70}$$

$$\frac{\vartheta C}{\vartheta b} = (a-y)\sigma^{'}(z)t = a\sigma^{'}(z) \tag{71}$$

Considering now that the sigmoid neuron output is getting nearly flat in the areas close to 1 the corresponding $\sigma^{'}$ will be really small and thus, the learning process will slow down as the derivatives (70) and (71) will also be very small.

To overcome the above limitation a different loss function can be utilized called “cross-entropy cost function” (72).

$$\mathrm{C} = -\frac{1}{n}\sum_t (yln(a) + (1-y)\ln(1-a)) \tag{72}$$

This loss function has a characteristic that is never negative as both of the summing units are always negative. Secondly if the output of the neuron is close to the desired output, then the cross-entropy will be close to zero and thus if fulfills the desired properties of a cost function demanded by backpropagation algorithm. Furthermore, in case of a sigmoid neuron it does not rely on the $\boldsymbol{\sigma}'$ which is the route of the learning slowdown problem of the quadratic cost function. To prove this a computation of the partial derivatives of the cost function follows (73)

$$\frac{\vartheta C}{\vartheta w_i} = -\frac{1}{n}\sum_t\left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\vartheta \sigma}{\vartheta w_i} =-\frac{1}{n}\sum_t\left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)\,t_i \tag{73}$$

Expanding the (68) and because of the fact that a sigmoid function is calculated as $\sigma(z) = 1/(1+e^{-z})$ and thus $\sigma'(z) = \sigma(z)(1-\sigma(z))$, the (72) can be rewritten as (74) and similarly into (75) for the biases

$$\frac{\vartheta C}{\vartheta w_i} = \frac{1}{n}\sum_t t_j\left(\sigma(z) - y\right) \tag{74}$$

$$\frac{\vartheta C}{\vartheta b} = \frac{1}{n}\sum_t\left(\sigma(z) - y\right) \tag{75}$$

3.7. Genetic Algorithms

A genetic algorithm is trying to solve optimization problems by simulating the biological evolution. It focuses on the genetic population based reproduction, mutation and recombination with the goal of optimal generation selection.

To follow the approach that genetic algorithm requires, the definition of five components is needed

1) A way to encode the solutions of the problem into chromosomes
2) An evaluation function which accepts chromosome results and returns ratings based on how well they perform
3) To define operations which are applied to the parents when they reproduce. The most commonly used are the crossover and the mutation
4) A way to initialize the population of chromosomes
5) The hyper-parameters of the above

The above components can be encoded into an algorithm that performs as follows:

1) Initialize the population based on a selected initialization function
2) Evaluate the initial population and compute the ratings
3) While stopping criteria are not met:
    a. Stochastically or best performing member of population are picked as parents
    b. Crossover or mutate the parents and produce children that are forming a generation
    c. Evaluate the generation and incorporate it into the population while removing the least performing parents

The ultimate goal of this approach is to find the optimal hyper-parameters of a neural network, which includes but not limited, to the number of hidden layers, the number of neurons of each layer and can be extended with incorporation of other methodologies to compute the weights and bias of the neurons.

4. Neural network challenges

This section describes general challenges that are faced on both designing and training of neural networks. More specifically, an initial introduction to the problem of model overfitting and ways to overcome it, the initialization of the weights of the network and how these can be calibrated in such a way, that can improve the training performance of the entire model and, lastly focus on neural network training issue called vanishing and exploding gradient descent which affects deep neural network architectures and challenges the scientific community until today.

4.1. Overfitting

A general problem that all models with large number of free parameters are exposed is called overfitting. This problem lies on the fact that the model becomes really good on predicting data that has already been trained with, but behaves poorly on generalization, or in other words is unable to recognize patterns that are not similar to the patterns learned from the training data.

4.1.1. Early-stopping

In the process of training a neural network the training dataset is used to evaluate the performance of the model by computing the results of the cost function. This works well, yet it is prone to overfitting as the model free parameters are calibrated based on that dataset. Furthermore another problematic characteristic of this approach is that the hyper parameters of the model are chosen based on the performance of the training dataset. In order to evaluate the model more accurately on every epoch, a validation dataset can be introduced. This dataset is formed by splitting the original training dataset into two smaller ones; one which is used for validation and another one which is used for training.

By using these datasets the evaluation criteria of each epoch depend on the accuracy of the validation dataset, according to a process called early stopping [30]. The accuracy of each epoch on validation dataset is evaluated when it starts to decrease while the accuracy of the training dataset increases training is terminated (Fig. 17). When stochastic gradient descent algorithm or equivalent is used, early stopping decision can become harder to be applied, as on every epoch there might be contradictive accuracy changes between the two datasets. A good practice is to allow a fixed number of epochs to be applied and then, only if the early stopping criteria are met to stop the training process.

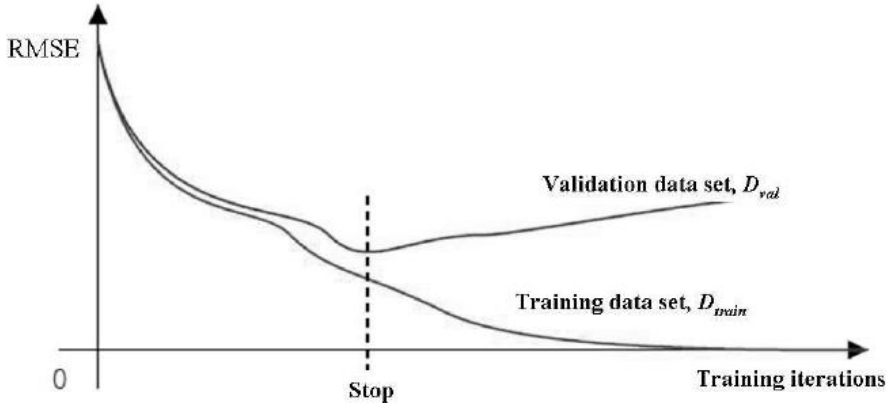


Figure 17 Validation and training dataset early stopping point [31]

Another approach is to keep a history of every epoch in a form of free-parameter snapshot, in order to find the optimal accuracy between validation and training dataset, yet this approach requires many resources and thus is avoided.

4.1.2. Dropout

A rather interesting solution to control the overfitting of neural networks that was recently introduced is called “dropout layer” [32]. This technique is usually applied in deep neural networks and in practice is randomly switching off neurons and its connections. An example of this can be seen in (Fig. 18) where temporarily a fixed percentage of neurons are disabled. By repeating this process over every epoch, the neurons are getting trained in a way that resembles using multiple neural network architectures and thus, they learn to generalize better. Furthermore the complexity of using multiple neural network architectures and then choosing the best is minimized as with this technique, the trials are based on every epoch which would have been computed on each network individually. This technique was used successfully in [33] were it was described as a technique that reduces the complex co-adaptations of neurons.

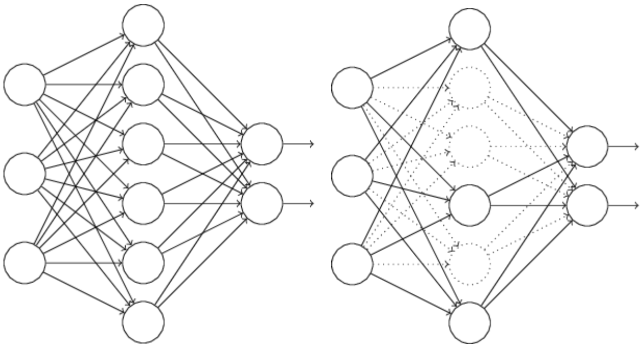


Figure 18 On the left a neural network before dropout and on the right a neural network after dropout process [29]

4.1.3. Weight Decay

Weight decay is a regularization technique which is also known as L2 regularization. In this approach the idea lies on the modification of the cost function by incorporating an additional term called "regularization term".

Rewriting the original cross-entropy loss function (72) with the additional regularization term as (76).

$$\mathrm{C} = -\frac{1}{n}\sum_t (yln(a) + (1-y)\ln(1-a)) + \frac{\lambda}{2n}\sum_w w^2 \tag{76}$$

The first term of (76) is the original cross-entropy loss function, while on the second one the sum of the squares of all the weights was added. The scaling factor $\frac{\lambda}{2n}$, where λ>0 and n the number of training examples, was added in order to regularize the parameter. This modification can also be applied to different loss function, as an example the quadratic cost function which can be re-written as (77).

$$C = \frac{1}{2}(|y - a^L|)^2 = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2 + \frac{\lambda}{2n}\sum_w w^2 \tag{77}$$

The general approach of applying L2 regularization to any cost function can be expressed as (78) where C_0 is the original cost function.

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2 \tag{78}$$

This addition to the loss function, forces the network to learn small weights while larger weights are allowed only if they result in a considerable change of the original cost function value. The term that regulates the weight scale balance is λ where a small value swifts the attention to minimize the original cost function C_0 while a larger value swifts the preference to the small weights.

To understand how the regularization factor λ helps to overcome the overfitting issue that appear in neural networks training, a simple example is presented utilizing the stochastic gradient descent algorithm. The partial derivatives of gradient descent can be computed with the incorporation of the regularization factor which can be expressed as (79) and (80).

$$\frac{\vartheta C}{\vartheta w} = \frac{\vartheta C_0}{\vartheta w} + \frac{\lambda}{n} w \tag{79}$$

$$\frac{\vartheta C}{\vartheta b} = \frac{\vartheta C_0}{\vartheta b} \tag{80}$$

The computation of partial derivatives of bias are unchanged as the regularization term is only applied to the computation of the partial derivatives of the weights of the network and thus the update rule of backpropagation can be re-written as (81) and (82). The only difference of the original gradient descent update rule is the rescale of the weights, which is also called weight decay.

$$w \rightarrow w^{'} = w - \left(1 - \frac{n\lambda}{n}\right) w - n \frac{\vartheta C_0}{\vartheta w} \tag{81}$$

$$b \rightarrow b^{'} = b - n \frac{\vartheta C_0}{\vartheta b} \tag{82}$$

Based on (81) and (82) the stochastic gradient descent update rule can be expressed as (83) and (84) where, as the gradient descent approach remains unchanged. The term C_t is the unregularized cost of each training example of each mini-batch as computed by the original backpropagation algorithm.

.

$$w \rightarrow w^{'} = w - \left(1 - \frac{n\lambda}{n}\right) w - \frac{n}{m} \sum_t \frac{\vartheta C_t}{\vartheta w} \tag{83}$$

$$b \rightarrow b^{'} = b - \frac{n}{m} \sum_t \frac{\vartheta C_t}{\vartheta b} \tag{84}$$

4.1.3.1. Example Of Weight Decay

To evaluate the performance change of a typical MLP neural network architecture, the usage of MNIST [34] dataset and benchmark is incorporated. This dataset was constructed by rescaling and normalizing a subset of 70.000 handwritten digit images from NIST dataset. This subset has be divided into 60.000 training and 10.000 test examples and is widely used as a performance evaluation tool in the image recognition sector.

The neural network architecture that was used to evaluate the performance benefits of L2 regularization practice, consists by an input layer of 4096 neurons, as the images in the dataset are scaled to 64x64 size and are grayscale, a 30 sigmoid neurons fully connected hidden layer and an output layer of 10 neurons, one for each class presented in the dataset. For both unregularized and regularized backpropagation algorithm the learning rate was set to 0.5 and a mini-batch of size 10 was used with cross-entropy loss function. For the regularized version a weight decay term was set to $\lambda = 0.1$. Evaluating both networks with a muted random function, seeded with a constant value to eliminate performance changes on randomization factor, a value of 82.27% accuracy was obtained from the unregulated network and 87.1% to the regulated one, which is considered noticeable, as the only change on the unregulated network architecture was the weight decay factor. Another noticeable difference was the converging speed improvement of the regulated network, which steadily grew the accuracy onwards to 400 epochs, while the first had started overfitting from the 280 epoch and then kept fluctuating between 82% accuracy (Fig.19).

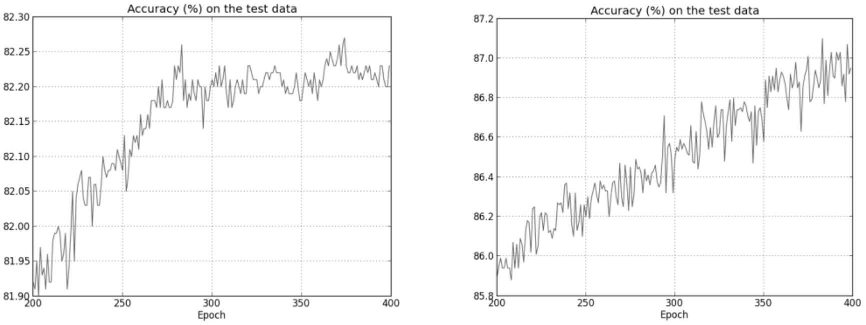


Figure 19 Accuracy graph over epochs on both (left) unregulated and (right) regulated networks [29]

4.1.4. L1 Regularization

This form of regularization works on the same principles as the L2 regularization with the difference of the regularization term of the cost function, which in this case, the term represents the absolute sum of the weights of the network multiplied by the regularization factor λ and can be written as (85)

$$C = C_0 + \frac{\lambda}{n} \sum_w abs(w) \tag{85}$$

Using calculus the partial derivative of (85) with respect of the weights of the network can be written as (86), in which the $sgn(w)$ represents the sign of the weight w. The update rule of the regularized model can be expressed as (80) and (81).

$$\frac{\vartheta C}{\vartheta w} = \frac{\vartheta C_0}{\vartheta w} + \frac{\lambda}{n} sgn(w) \tag{86}$$

$$w \rightarrow w' = w\left(1 - \frac{n\lambda}{n}\right) - n\frac{\vartheta C_0}{\vartheta w} \tag{87}$$

$$b \rightarrow b' = b - n\frac{\vartheta C_0}{\vartheta b} \tag{88}$$

While L1 and L2 regularization have as an effect to alter the weights of the network, the first shrinks the weights by a constant amount towards zero while on the second the shrinkage is proportional to the weight. As a result, when the magnitude of $|w|$ is large the L1 regularized network shrinks the weight less than L2. In contrast when the magnitude of the weight is small L1 regularization will affect more the weight than an L2. Concluding L1 regularized networks tend to focus the weights on a relatively smaller number of high importance neural connections, while the rest are lead towards zero. While the difference between the two regularization forms is definite, it is still unclear which performs the best and thus both of them are widely used in literature.

4.1.5. Training Dataset Expansion

A common practice that is employed by researchers while training neural networks, is to increase artificially the training dataset of the network. The reason behind this lies on the fact that, in order to train a model with thousand or million parameters is relatively hard, by using a small dataset, as that often results into overfitting those parameters to that limit set of examples. The artificial expansion of the dataset can be performed systematically if the dataset allows it. An example that is commonly used in the field of computer vision and more specifically in image classification is to artificially rotate, scale and distort the training examples while keeping the same class. That gives the model the chance to adjust the free parameters accordingly and thus expanding the generalization capabilities of the whole network. An example taken from affNIST [35] dataset is illustrated in (Fig. 20) which contains images artificially transformed from the original MNIST dataset. Using this dataset which expands the training examples of original MNIST dataset the training accuracy of the model reached 96.74% [29], i.e., is an increase of 9.64% is achieved which illustrates the performance gains that can be obtained by the artificial pre-processing of the training dataset.

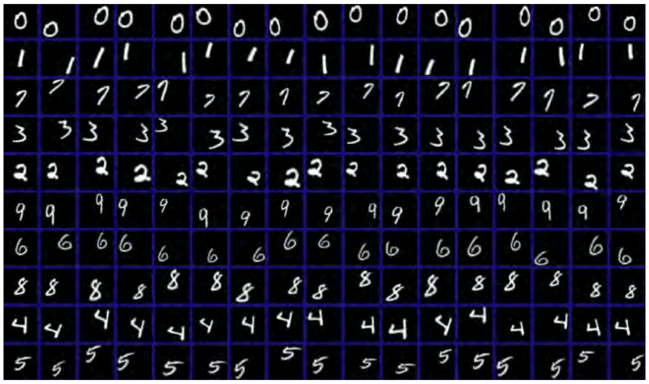


Figure 20 Grid of 10x17 artificially transformed 40x40 digits of the first column and their 16 variations from affNIST dataset [35].

4.1.6. Training Data Pre-Processing

An interesting approach that can be applied to the original training dataset is to perform a principal component analysis (PCA) [36]. In order to improve the generalization capabilities of the model. PCA has been traditionally used in statistical and signal analysis in order to transform multi-dimensional data that are possibly correlated, into linearly uncorrelated variables which are called “principal components”. This idea has been successfully used in numerous studies including [37] where this pre-processing methodology was used in order to eliminate the problem of correlated information that exist in the real-world training data, and thus remove the hyperspace overlap that exist between them.

4.1.6.1. PCA and ZCA Whitening

PCA and Zero Component Analysis (ZCA) whitening are common image and audio pre-processing methodologies that are applied to decorrelate information. In order to understand how whitening works a simple example of applying PCA to raw data is presented.

Let us consider as training examples, grayscale images with pixel values $x_{i,j} \in \Re^{256}$. In there images, adjacent pixels are highly correlated as they usually correspond to the same object and thus utilizing PCA the input space can be reduced with very little approximation error. Let $X = \{x^{(1)}, x^{(2)}, x^{(3)}, \dots, x^{(m)}\}$ be a dataset with input dimensions $n = 2$, thus $x_{i,j} \in \Re^2$ (Fig. 20).The chosen dimensionality was peaked to simplify the visualization of the process.

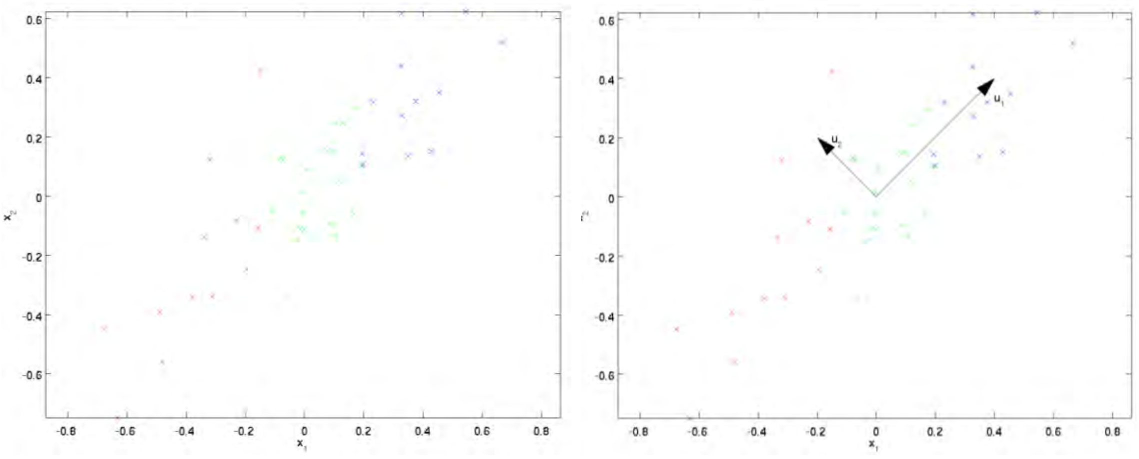


Figure 21 Left) Pre-processed zero mean dataset visualization before PCA. Right) Two principal directions of variation [38]

Applying PCA to dataset X a lower-dimensionality subspace can be found, on which the data can be projected. In (Fig. 21) u_1 is the primary principal direction of variation and u_2 the secondary direction of variation of the data which means that the data varies much more in u_1 direction than u_2. The principals are found using (89) where Σ is the covariance matrix for the zero mean x dataset. u_1 and u_2 can be obtained by the order of appearance of the eigenvectors of covariance matrix Σ. Stacking the eigenvectors in columns results in the creation of matrix U (90), in which the order of appearance of each eigenvector corresponds to the largest eigenvalue $(\lambda_1, \lambda_2, \dots, \lambda_n)$.

$$\Sigma = \frac{1}{m}\sum_{i=0}^{m}\left(x^{(i)}\right)\left(x^{(i)}\right)^T \qquad (89)$$

$$\mathrm{U} = \begin{bmatrix} | & | & | & | \\ u_1 & u_2 & \dots & u_n \\ | & | & | & | \end{bmatrix} \tag{90}$$

The magnitude of the projection of x onto the vector u_n can be computed as $u_n^T x$ where u_n^T is the transposed eigenvector u_n. To represent the data x in eigenvector basis, a rotation has to be applied which is computed by (91).

$$\mathrm{x_{rot}} = \begin{bmatrix} u_1^T x \\ \dots \\ u_n^T x \end{bmatrix} = \mathrm{U^T x} \tag{91}$$

In (Fig. 22) the visualization of the entire dataset transformed by $\mathrm{x_{rot}^{(i)}} = U^T x^{(i)}$, which is the training dataset transformed by u_1, u_2 basis. A noticeable property of the U matrix is that it satisfies $U^T U = UU^T = I$; which is also called orthogonal matrix. This can be used in order to rotate back the x_{rot} vectors to x by computing $\mathrm{x = Ux_{rot}}$ as $\mathrm{Ux_{rot}} = UU^T \mathrm{x} = \mathrm{x}$.

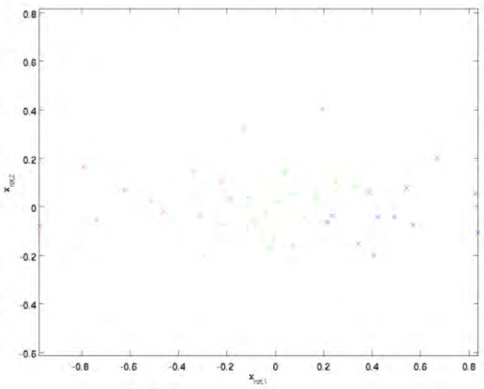


Figure 22 $\mathbf{x_{rot}}$ dataset visualization [38]

To reduce the data dimensionality using PCA, the principles of the matrix U can be used. More specifically if $x \in \mathfrak{R}^n$ and aiming to reduce the dimensionality of it into k where $k < n$ and thus transform x to $x' \in \mathfrak{R}^k$ then the first k components of the U matrix can be kept. This is possible as the order of values in x_{rot} vector are in descending order and thus if the initial components of x vector are largely correlated, the first values of x_{rot} vector will be significantly larger than the later ones and thus an estimation of the original vector can be obtained by zeroing the values that are close to zero. That can be expressed as (92):

$$x^{'} = \begin{bmatrix} x_{rot,1} \\ x_{rot,2} \\ \cdots \\ x_{rot,k} \\ 0 \end{bmatrix} \approx \begin{bmatrix} x_{rot,1} \\ x_{rot,2} \\ \cdots \\ x_{rot,k} \\ x_{rot,n} \end{bmatrix} x_{rot} \tag{92}$$

Applying (92) to the two dimensional example and by choosing $k = 1$ a one-dimensional representation of the original x vector (Fig. 23.a) is obtained.

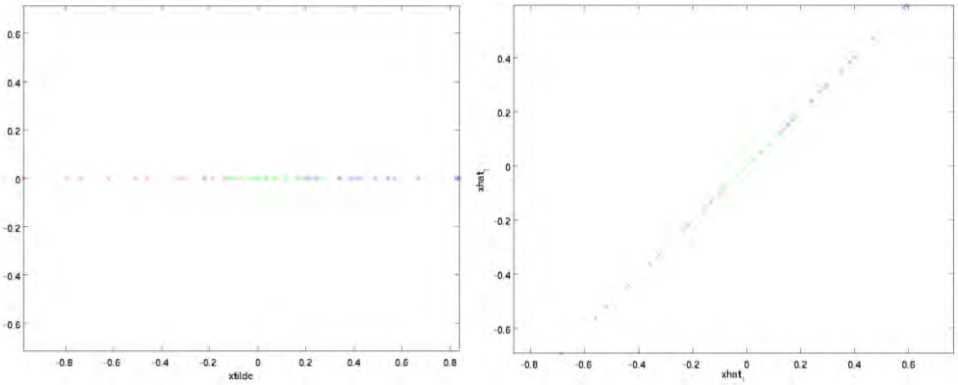


Figure 23 On the left, $\mathbf{x}'$ visualization. On the right, $\hat{\mathbf{x}}$ visualization [38]

In order to utilize the k first principles and obtain an approximation of the original vector x, x' vector has to be transformed by utilizing the U matrix (93). The visualization $\hat{x}$ vector can be seen in (Fig. 23.b).

$$\hat{x} = U \begin{bmatrix} x'_1 \\ x'_2 \\ \ldots \\ x'_k \\ 0 \end{bmatrix} = \sum_{i=0}^{k} u_i x'_i \tag{93}$$

When applying PCA to a dataset, the k hyper-parameter has to be chosen in such a way that the minimum information loss with the maximum dimensional decrease will occur. Setting $k = n$ there will be no information loss and thus all of the variation of the original data will be retained. Setting $k = 0$ zero retainsion will be obtained and thus all the variation will be lost. If $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of the covariance matrix Σ then the percentage, γ, of variation retained by keeping k principals can be obtained by (94). A common heuristic approach in case of applying PCA in image pre-processing is to set $\gamma \geq 0.99$ yet that strongly depends on the application as in some cases the nature of the problem allows higher information loss in favor of higher dimensionality reduction which in return will increase the overall performance in terms of time that has to be spent while training a model.

$$\gamma = \frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j} \tag{94}$$

To apply PCA on a dataset, each of the features has to meet certain criteria and more precisely, a common pre-processing step is to normalize the values of the features so they can have a mean close to zero. This is not something that can be applied in most of the cases when it comes to images. Natural images, or in other words, images that are obtained in the wild, have a property called "stationarity" which means that statistically the pixels in one part of the image should be the same as the others. That means that if the raw pixel of the images is used as a feature set, then the nature of them already comply with the variance normalization criteria set by PCA. That holds true for other kind of features such as audio, or text. The only normalization that has to be done in such data is mean normalization which can be achieved utilizing (95) and (96).

$$\mu^{(i)} = \frac{1}{n} \sum_{j=1}^{n} x_j^{(i)} \tag{95}$$

$$x_j^{(i)} = x_j^{(i)} - \mu^{(i)} \tag{96}$$

A common data-preprocessing similar to PCA is called "whitening" or as it sometimes referred in the literature "sphering", aims to reduce the redundancy appearing in raw image data, considering that adjacent pixels are highly correlated. This can be applied by utilizing the same procedure as PCA. The covariance matrix in natural images closely resembles a diagonal matrix and thus, to make each input feature have a unit variance, a rescaling of the original $\mathrm{x}_{\mathrm{rot},\mathrm{i}}$ can be done by dividing it with the square root of the corresponding eigenvalue value (97), where ε is a small constant, usually of the order of 10^{-5}, which is used in order to protect the $\mathrm{x}'_{\mathrm{rot},\mathrm{i}}$ from numerical explosion, as λ_i might take relatively small values and thus the division produce large numbers. Following the same principles of PCA and combining it with whitening, which in fact means keeping k principles from the resulted $\mathrm{x}'_{\mathrm{rot}}$ vector.

$$\mathrm{x}'_{\mathrm{rot},\mathrm{i}} = \frac{\mathrm{x}_{\mathrm{rot},\mathrm{i}}}{\sqrt{\lambda_i + \varepsilon}} \tag{97}$$

Another approach of transforming the data to have covariance identity I is called ZCA whitening [39]. Utilizing any orthogonal matrix R then $Rx_{PCA_{white}}$ will also have identity covariance. In ZCA whitening $R = U$ and thus $x_{ZCA_{white}} = Ux_{PCA_{white}}$. Unlike PCA whitening, when ZCA whitening is applied, usually the whole x'_{rot} vector is kept and thus does not apply any dimensional reduction. It has also been shown that, out of any R rotational matrix choices, the ZCA approach causes $x_{ZCA_{white}}$ to be as close as possible to the original x vector and thus, has the minimum information loss. An illustration of PCA whitening without dimensionality reduction on grayscale images can be seen in (Fig. 24), where image patches are PCA whitened by using 116 out of 144 principle components and preserving 99% of the variance achieving less correlated features. It is worth noting that the whitened features have the same variance, which is beneficial as a pre-processing step in most classification algorithm cases as that way they are classified easier.

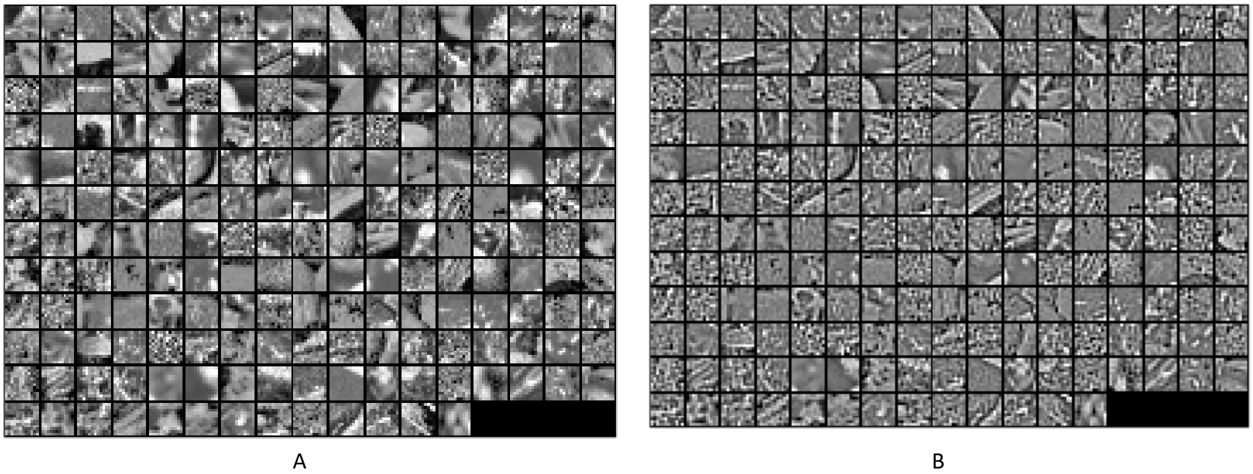


Figure 24 (A) Image patches before and after (B) PCA whitening [40].

4.1.7. Weight Initialization

An important factor that has to be taken into consideration when training any kind of artificial neural network is the initialization of the weights and biases of the model. This is important as random weights might degrade the performance of the entire training process or even stop it entirely. This can occur when weights or biases have zero value and thus the neurons are already saturated before the training process.

A common practice is to use random independent values that follow the Gaussian distribution and they are normalized to have 0 mean and 1 standard deviation. Let z be the sum of all the weights and biases of a single neuron and thus (98).

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{98}$$

That means that z is a sum of random Gaussian variables an extra bias term. Assuming that half of x_i will be turned off, or in other words they will be set to 0, and the rest are activated, if the number of weights connected to a single neuron is large, for example 500 weights, z will have a value of 501 considering 500 weights and 1 bias. The standard deviation of z will be $\sqrt{501} \cong 22.4$ with zero mean (Fig 25) which shows that the $abs(z)$ will be a large value and thus $z \gg 1 \; or \; z \ll -1$. In case of a sigmoid neuron, this will lead to a pre-saturated neuron as $\sigma(z)$ will be either strongly 0 or 1. As backpropagation works by applying small weights in order to explore the surface of the cost function of the network, the changes will have little to no effect to the neuron which leads to slow learning and thus damages the whole training process.

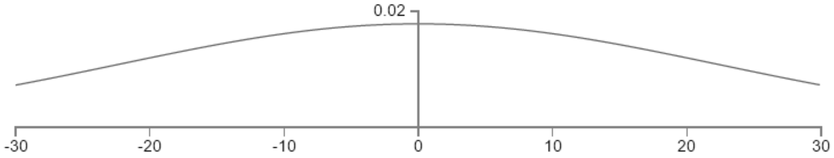


Figure 25 Gaussian distribution of the values of a 500 weight and 1 bias neuron [29]

In order to prevent this to occur, an alternative approach is to keep using Gaussian normalized initial free parameters with mean zero, but unlike the previous approach, the standard deviation will be around $1/\sqrt{n}$ where n the number of weights connected to the neuron. That causes the Gaussian to be squashed down and thus have much less possibility for the neuron to be saturated upon initialization. By following that in the previous example, the standard deviation of z will be $\sqrt{2/3} = 1.22$ with a much sharper Gaussian distribution (Fig. 26).

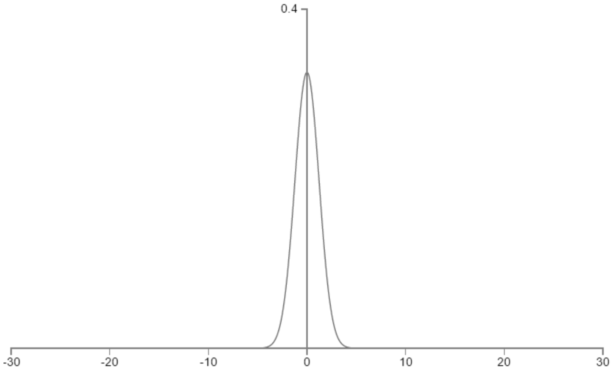


Figure 26 Gaussian distribution with standard deviation 1.22 and mean zero [29]

To demonstrate the impact of free-parameter initialization in the training process of a model, a neural network has been constructed and trained upon MNIST dataset. The architecture the network was 784 input neurons, 1 hidden layer with 30 sigmoid neurons and an output layer of 10 neurons, one for each possible digit contained into the dataset. The cost function was regularized using L2 regularization with $\lambda = 5$. For the stochastic gradient descent the chosen learning rate was $n = 0.1$ with batch size $m = 10$. The network was trained using both initialization approaches to illustrate the difference between the two approaches (Fig. 27). The classification accuracy in both cases was $\cong 96.4\%$ but following the regulated weight initialization had an impact in converge speed as the accuracy reached its pick within the first 10 epochs, yet on the unregulated version, the stabilization of accuracy occurred only after 28 epochs.

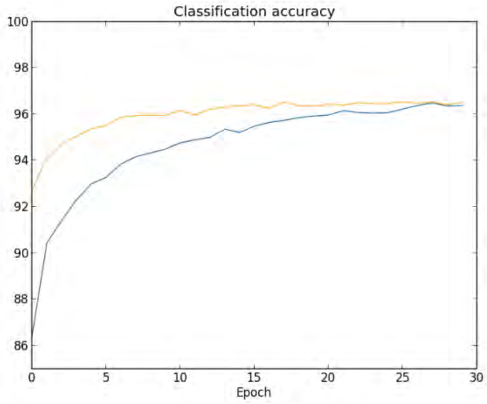


Figure 27 Visualization of the training classification accuracy in 30 epochs. Blue line the unregulated initialization while orange line the regulated approach [29]

4.1.8. Hyperparameter Selection

In the neural network context, hyper-parameters are parameters that have to be decided prior the training process of the model. These parameters, are dependent on the different set of algorithms that are used while training which includes, the learning rate n, the regularization parameter λ, the batch size m in case of stochastic gradient descent like algorithms. Unfortunately for these parameters, there is no definite answer to what values they should get and thus, most of the researches are following heuristic approaches. The same rule applies to the rest hyper-parameters like the number of hidden layers that the network should have, the number of neurons on each hidden layer and even the activation function of the neurons. In most of the cases the most effective way is the trial and error, yet there are some basic principles that can be followed as short paths.

Firstly to speed up the learning process a common practice is to keep the training dataset small thus on each epoch, the accuracy can be monitored and if the model accuracy does not follow an upwards direction alter the learning rate accordingly. Depending on the gradient descent algorithm that is used, the learning rate and the regularization parameter are usually provided by the creator of the algorithm, yet if that is not the case, a safe approach is to follow a learning rate reduction by a factor of 10 on each trial. As an example a usual starting point for learning rate, can be $n = 10^{-1}$, if the accuracy is unstable, then a lower learning rate should be applied. On the other hand, if the classification accuracy grows steadily but slowly, learning rate should be increased, usually again at the same rate as the reduction. To determine the number of epochs that the model should be used, it is a good practice to follow early stopping approach and thus as a consequence avoid overfitting too. For the regularization parameter, a common approach is first to start the training without regularization at all and thus $\lambda = 0$, after learning rate adjustments, the regularization parameter can be increased by a steady factor of 10, following the same principles with learning rate, yet instead of starting from high values, usually the initial value is as low as $\lambda = 10^{-5}$.

Numerous automated techniques have been proposed to help on hyper-parameter selection for neural networks. A common approach is the “grid search” which systematically searches through grid in hyper-parameter space to find the optimal values. A review of existing algorithms along with practical ways to implement them can be found in [41]. Another rather interesting Bayesian optimization approach of parameter selection has been proposed in [42] in which a learning algorithm’s generalization performance is modeled as a sample from a Gaussian process.

5. Convolutional Neural Networks

Deep learning is a class of machine learning algorithms which use multiple cascaded neural layers for feature extraction. It can be used in supervised and unsupervised learning where multiple levels of feature are extracted. The levels are forming a hierarchy on which higher level features are extracted from lower level features that are up in the hierarchy. The training algorithm that is used for these kinds of neural nets is usually gradient descent based back propagation.

In recent years a lot of attention has been drawn to a special type of deep neural network architecture called Convolution Neural Networks (CNNs). CNN is a type of multi-layer feed-forward network architecture that at its core, contains at least one layer of neurons that perform a special kind of operation known as “convolution”. They are inspired by the natural biological process of animal visual cortex, in which neurons are individually responding to small regions of the visual field. That has been initially discovered by examining the visual system of cats and monkeys from the biologists Hubel and Wiesel in 1950s and 1960s. Later on in [43] it has been identified that there are two basic types of visual cells in the brains. The first type is called “single cells” whose output is maximized by edges with particular orientation within their receptive field (the portion of the visual image that the cell is able to view). The second type of cells referred as “complex cells” has a relatively larger receptive field than the first ones and their output is insensitive to the exact position of the edges presented into that field.

Inspired by the discoveries of [43], in early 1980s there was an adoption of this in the field of neural networks [44] with the name “Neocognitron”. The difference with the previous neural network architectures was that the neurons did not require to share the same trainable weights which means that instead of relying to a fully connected architecture, Neocognitron was able utilize neurons similarly to how they work in the visual context of animals. Unfortunately due to the lack of computer resources that were required to train such a network, the idea was halted and no further research was done on it for more than a decade. In 1998, when computation resources increased the initial design was improved [34] with the introduction of a 7 layer Convolutional neural network trained to recognize digits from images with size 32x32, applied in banking industry. In 2003 [45] a more generalized approach was introduced and simplified [46] opening the path to the scientific community to leverage the power of these kind of networks in the field of computer vision.

One of the most well-known CNN architectures was introduced in 2012. It is known by the name “AlexNet” [33] which was trained, using Graphical Processing Units (GPUs) to speed up the training process utilizing a framework provided by NVIDIA known as Compute Unified Device Architecture (CUDA) [47]. The model was trained using a dataset known as “ImageNet Large Scale Visual Recognition Challenge” (ILSVRC) [48], which is an annual competition started in 2010. This competition is based on classification accuracy upon a dataset containing a large number of images from the wild, following the principles set by PASCAL VOC challenge [49].

AlexNet was trained on a dataset from LSVRC-2010 which contains 1.3 million high resolution images from 1000 classes and managed to achieve 39.7% and 18.9% error rates on top-1 and top-5 scales outperforming by far previous state of the art approaches. AlexNet consists of 5 convolution layers, some of which were followed by max-pooling and normalization layers, and they have two fully connected layers at the end followed by one output softmax layer of 1000 neurons. The resulting network of the above architecture had more than 60 million free-parameters and 500.000 neurons which was considerably big for the time that was created. In Fig. 28 an illustration of the above architecture is shown.

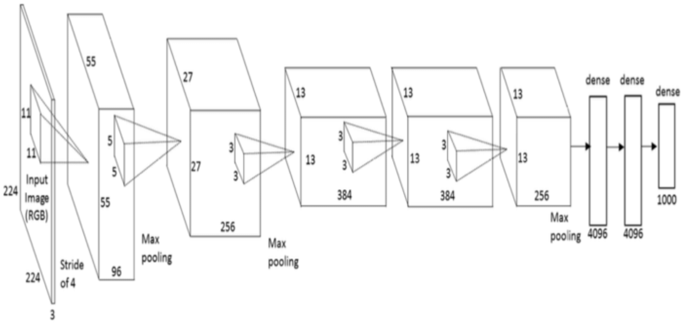


Figure 28 AlexNet architecture [50]

5.1. Convolution Layer

In a CNN architecture the most crucial layer that is used is called convolution layer. This is the essential part of every CNN based architecture which is inspired by the biological visual cortex. More specifically, a convolution layer is the same as any other hidden layer of a conventional MLP architecture, with one distinct difference lying behind the neuron connections. While in a typical MLP architecture all neurons of the previous layer will be connected with all the neurons of the current layer, in a convolution layer, each neuron is connected with a specific set of neurons from the previous layer, which is called “receptive field”.

The local receptive field, which can be seen in Fig. 29, is responsible for the topological specific weights, or features, that CNNs are known for. Each neuron of a convolution layer is connected to a specific set of neurons from its previous layer. When it comes to a 2D convolution layer, the receptive field is defined as a rectangular set of neurons, with a size of $w \times h$, where w is the width and h is the height of the rectangular area. Each neuron is connected to the next rectangular set of neurons until the entire surface of neurons from the previous layer is covered. These rectangular regions, can, but not always, overlap with each other. The distance by which the rectangular region (filter) shifts on the input volume is called is called stride s.

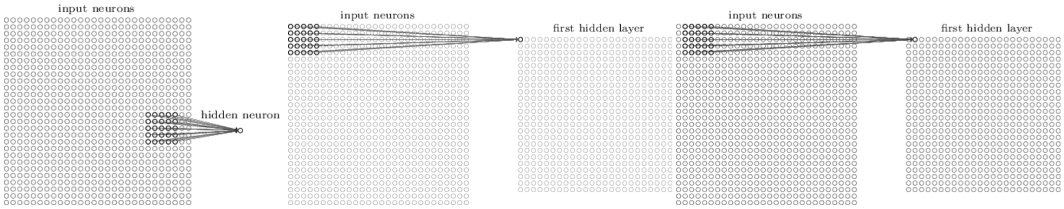


Figure 29 Left: Receptive field demonstration for a neuron. Middle and Right: Overlapping receptive fields with stride 1 [29]

Another important parameter that most convolution layer architectures have is called padding p. This parameter aroused from the fact that in some cases, the size of the rectangular area does not fit precisely with the size of the input layer. For that reason blank input signals, or pixels in case of images, are added around the original input layer (Fig 30).

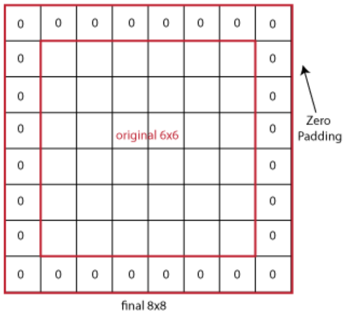


Figure 30 [51] Zero padding $p = 1$ of around an input image of 6x6 with resulting image 8x8.

Convolution operation is extended across the depth of the input volume. A common example of that is when the input of the network is in fact an RGB image, where each channel can be considered as a separate depth $\boldsymbol{d}$ and thus creating a cube of width and height equal to the dimension of the image and depth equal to the number of channels. In case of input with depth, the receptive field of neurons, which is also known as “filter”, are running across the whole width, height and the depth of the input layer. That results into the creation of multiple depth filters, in the convolution layer. Each filter has now the same topological location of extraction, yet it might differ vastly from the others as the actual input, belongs to different depth.

By known these parameters the calculation of the total number of neurons along with the total number of weights that a convolution layer will have, is possible. Summarizing a convolution layer has:

- Input volume of size $W \times H \times D$ where W is its width, H is its height and D is its depth
- A set of hyper parameters defined as
  - K which is the number of filters that extracts
  - F known as spatial extent or filter size
  - S which is the stride or gap on which the filters are extracted
  - P is the padding that can be placed in case of input volume size mismatch
- An output volume of
  - Width calculated as

$$W^{'} = (W - F + 2P)/(S + 1) \tag{99}$$

  - Height calculated as

$$H^{'} = (H - F + 2P)/(S + 1) \tag{100}$$

  - Depth K

The resulting number of weights W_{filter} of each filter can be calculated using equation (101), and the total number of weights W_{total} by (102).

$$W_{filter} = F \cdot F \cdot D \tag{101}$$

$$W_{total} = (W_{filter} \cdot K) \tag{102}$$

The total number of free parameters can be calculated by adding the number of biases for each extracted filter which is $W_{total} + K$. A simple example is presented.

- Input volume: RGB image of size $32 \times 32 \times 3$
- Filter size: $5 \times 5 \times 3$
- For stride $S = 1$ and padding $P = 0$ the resulting activation map size is:
    - $28 \times 28 \times 3$ as (92) and (93) results into $(32 - 5 + 2 \times 0) / 1 + 1$
- Each neuron has 75 weights + 1 bias, which is a result of applying (94) and thus
- If there are 6 filters of this size the resulting activation maps have 450 weights and 6 biases

A special case of convolution layer can be obtained by setting the filter size to be equal to 1, no zero padding and a stride of 1. That leads effectively to dimensionality reductionality. As an example let input volume of $200 \times 200 \times 50$, by using 20 filters of size $1 \times 1 \times 20$ the resulting volume can be reduced to $200 \times 200 \times 20$. The most common use case of this practice is as a replacement in the final layers of a CNN architecture, where fully connected layers are more likely to be seen. Another use case of the above technique has been seen in GoogLeNet [52] which is presented in Section 5.7.5.

5.2. Pooling Layer

A common practice that is applied in CNN architectures is the usage of pooling layers. A pooling layer acts as a summarization of an input volume. The units of this layer have a receptive field, following the same principles with any convolution layer, yet instead of computing convolution operation, depending on the pooling operation, summarize the receptive field of each unit into a single scalar value. That means that a pooling layer has as hyper-parameters the width, height and depth of the receptive field, stride and padding. The most common pooling operations applied in CNN architectures are max-pooling, L2-norm pooling and average pooling.

Max-pooling has been used extensively in many network architectures including [33], [52] and [53]. The basic operation applied in a max-pooling layer is illustrated in (Fig. 31), where each unit selects the maximum value of each rectangular region applying the argmax operation (103).

It is important to note here, that the pooling operation is applied on each depth of the input volume and thus the resulting filters have the same depth as the original input volume.

$$V' = argmax(x_{input}\ ) \tag{103}$$

Average pooling follows the same principles as max-pooling, yet, instead of selecting the max value of the unit receptive field, it computes the average value (104) from the values of the rectangular region that is applied.

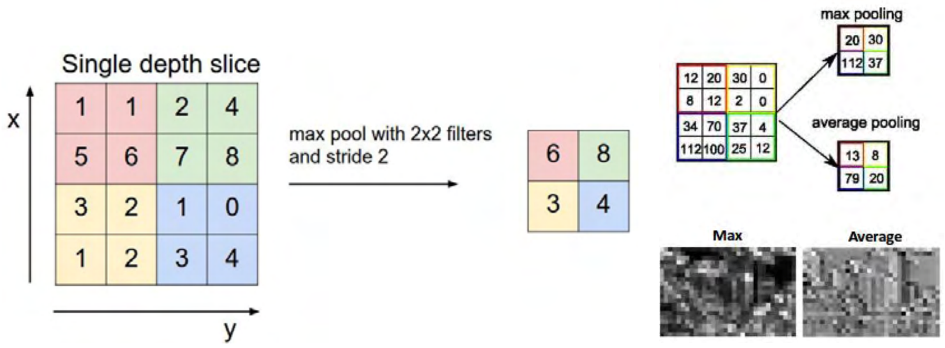


Figure 31 Left. Max-Pooling with size 2x2 and stride 2 [54]; Right. Average versus Max pooling [55]

$$V' = \frac{\sum_{i=1}^{k} x_{input}}{k} \tag{104}$$

The most common pooling size that is used in the literature is 2x2 with a stride of 2 which effectively down samples the input volume by discarding 75% of the activations. Higher pooling sizes would result into larger information loss that, most of the time, it is not desirable. As a general rule, max-pooling keeps the most important features of the activation map, such us edges, while the average pooling acts like a low pass filter and thus extracts a smoothed-out representation of the original activation map. Because of the above, in most CNN architectures max-pooling is preferred as it helps the model to recognize edges and thus increases the overall performance of the network.

The most profound benefit of pooling operation is the free-parameter reduction and thus lower computation cost while training a model. Another benefit, which comes with the reduction of spatial size of the model, is that pooling reduces overfitting. On the other hand it has been shown that the benefits of pooling layer usage can be obtained by replacing them with larger filter size of convolution layers [56] which reduces the overall implementation and computation complexity that is introduced by intercepting different layer types in a CNN architecture. It has been also shown that discarding completely pooling layers can be found important in training good generative models such as variational autoencoders (VAEs) [57] or generative adversarial networks (GANs) [58].

5.3. Normalization Layer

A layer type that has been shown in CNN architectures but has been fallen out of practice as their contribution to the overall performance of the model has been seen minimal is the normalization layer. There are many normalization operations that have been proposed in literature, such as Local Response Normalization (LRN), Mean Variance Normalization (MVN) and Batch Normalization (BN), which are inspired by the biological normalizations of signals that happen in brain neurons and has been successfully used in [33].

LRN performs a “lateral inhibition” by normalizing over input regions. That has been shown to be useful when ReLU activation is used in the layer that is to be normalized and it is because ReLU neurons have unbound activations and thus, aiming to detect frequency features with large response. That means that if normalization is done around local neighborhood of an explicit neuron, it becomes more sensitive as compared to its neighbors. Unfortunately this operation will also discriminate the responses that are uniformly large in any given local neighborhood and thus if all the values are large the normalization of those values will diminish all of them. The goal of the LRN is to encourage some kind of inhibition and boost the neurons with relatively large activations [33]. The normalization can be done either to a specific channel or depth of the previous layer, or it can be extended across all depth. In both cases the size of normalization filter follows the same principles as any other layer and thus it can be configured based on the size of the receptive field that normalization is desired. MVN is working similarly to LRN layer; yet they perform different type of normalization. MVN layer normalizes the input volume values so they will have 0 mean and a variance of 1.

5.4. Fully Connected Layer

In CNN architectures it is common to use fully connected layers as the last layers of the network. This is because, convolution layers are used to exploit the local associations between the input signals yet and not to decide if a set signals is strong enough to be considered significant. That means by adding fully connected layers after the convolution layers the network is able to “look” at the big picture of all the filters that have been extracted from the input volume and thus perform classification of it.

5.5. The Last Layer of a CNN Architecture

As the last layer of a CNN architecture it is relatively common to choose a fully connected layer in which the number of neurons are matching the number of classes that the dataset has. In other words the last layer can also be called “classification layer” as it is the final layer from which the network will output the results of the feed-forward process through the previous layers. It is also common to use different activation functions for the neurons of this layer in comparison with the previous layers. In most cases and especially to CNN architectures the usage of softmax or SVM activation functions is used.

5.5.1. Multi-Class Support Vector Machine Loss

The Multi-class Support Vector Machine (SVM) [59] loss has been used extensively in the field of neural networks. The SVM loss is set so that SVM wants to correct the class of each training example to have a score higher than the rest, incorrect, classes. The margin Δ between the correct and the incorrect classes is fixed. Let x_i and y_i a training example and its class respectively. The score function receives as a parameter the training example and computes vector $f(x_i, W)$ of a class scores s. Thus the score of the above example for the class j will be $s_j = f(x_i, W)_j$. Formalizing the above, the multi-class SVM loss for the i^{th} training example can be expressed as (105), where the threshold at 0 of the max operation is called “hinge loss”. It is also common to use an L2 variation of multi-class SVM or “squared hinge loss” where the max function is squared, which has as a result the larger penalization of violating margins.

$$L_i = \sum_{j\,!=y_i} \max(0, \mathrm{s_j} - \mathrm{s_y} + \Delta) \tag{105}$$

The formal approach of applying the L1 and L2 multi-class SVM loss, has an issue when it comes to practical implementations. The issue is based on the uniqueness of the weights that are used and it appears in some cases where these weights are qualified for all classes presented in the model and thus $\boldsymbol{L_i = 0}$ for every $\boldsymbol{i}$. That raises the need to regularize the function in order to encode a preference towards certain sets of weights $\boldsymbol{W}$ over the others. This can be done by introducing a regularization penalty $\boldsymbol{R(W)}$. There are many ways to do that, but the most common use-case is the L2 norm that has as property the discourization of large weights utilizing an element-wise quadratic penalty over all parameters (106).

$$R(w) = \sum_k \sum_l W_{k,l}^2 \tag{106}$$

Combining (106) [59] over the multi-class SVM loss (105) can now be rewritten into (107). Notice that the regularization function is based purely on weights and not the data. The impact of the regularization can be controlled by the hyper-parameter $\boldsymbol{\lambda}$. The hyper-parameter Δ can be safely set to 1 and that is because the parameter is there to create a difference between each class and not to quantify it; with that said, Δ can be any positive value.

$$L = \frac{1}{N} \sum_i \sum_{j\,!=y_i} [\max(0, \mathrm{f}(\mathrm{x_i}, \mathrm{W})_\mathrm{j} - \mathrm{f}(\mathrm{x_i}, \mathrm{W})_{\mathrm{y_j}} + \Delta)] + \lambda \sum_k \sum_l W_{k,l}^2 = \frac{1}{N} \sum_i L_i + \lambda\, R(w) \tag{107}$$

It is important to note here that there are other forms to apply SVM into multiple classes, which includes but not limited the “One Vs All” (OVA) SVM which trains an independent binary SVM for each class versus all the other classes. Another, less common, approach is the “All Vs All” (AVA) SVM. An interesting comparison OVA approach can be seen in [60] which compares multiple multi-class SVM approaches including AVA and the presented approach along.

5.5.2. Softmax Classifier

Softmax classifier is another commonly used classifier which has a different loss function than SVM. While Multi-class SVM treats the outputs $f(x_i, W)$ as scores, which are uncalibrated and thus difficult to interpret, Softmax maps the output into intuitive probabilities. That is, Softmax output can be read as probabilities of how much an input maps to each of the existing classes. While the function mapping $f(x_i, W) = Wx_i$ remains unchanged, the scores can now be interpreted as the unormalized log probabilities for each class and thus replace the hinge loss with a cross-entropy loss of form (108) where f_j is the j^{th} element of the class vector scores f.

$$L_i = -log\left(\frac{e^{f(y_i)}}{\sum_j e^{f_j}}\right) = -f(y_i) + log(\sum_j e^{f_j}) \tag{108}$$

The softmax function can be expressed as a function that takes an input z, $z \in R$ and squeezes the values between 0 and 1 with a sum equal to 1 (109).

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{109}$$

To understand how Softmax function works it is important to note that the cross-entropy between a true probability distribution p and the estimated distribution q is defined as (110) and thus the softmax classifier is minimizing the corss-entropy between the estimated class probabilities and the real distribution. That means that the interpretation of the distribution is focused on the correct class estimation and thus, in an optimal case, the probability vector p which contains the probabilities of an example to belong to all the available classes, will contain 1 only to the correct class and zero to the rest. Furthermore as the cross-entropy can be written in terms of entropy and the Kullback-Leibler (KL), divergence as $H(p,q) = H(p) + D(p||q)$ and the entropy of the data p is zero this is equivalent to minimizing KL divergence between the two distributions. In other words the cross-entropy objective is to accumulate all the mass of the probability distributions to the correct class.

$$H(p,q) = -\sum_k p(x) \log\big(q(x)\big) \tag{110}$$

In Fig. 32 a comparison between SVM and Softmax output is presented. A hypothetical weight vector W is used with an input vector x_i and bias b. On one side, SVM treats the output as unormalized scores of x_i vector with, highest score focused on the correct class. On the other hand Softmax treats the output scores as unormalized probabilities of x_i vector belonging to all the available classes and encourages the normalized log probability of the correct class to have the maximum value. In both cases the selected class has the highest value, even though the numbers are not comparable.

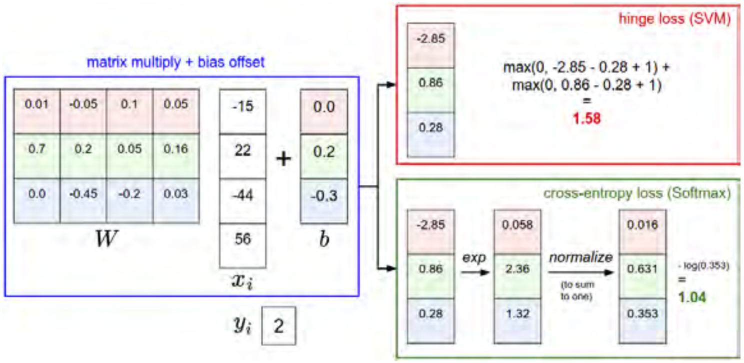


Figure 32 Softmax versus SVM output comparison. [61]

5.6. Parameter sharing

A rather important aspect on which CNN differ from conventional neural network architectures, like classical MLPs, is the way free parameters are computed. Each neuron is connected with a receptive field, on which weights and biases need to be computed, yet instead of using different parameters for each of those receptive fields that are formed in the convolution layer, the parameters are shared between them. That is, the weights and bias of all the filters, with the same size, on a convolution layer are having the same weights. That can be expressed as (111) where σ is the activation function of a neuron, b the shared bias and $w_{l,m}$ the array of $w \times h$ shared weights. The $a_{j+l,k+m}$ detonates the activation function output of the previous layer neuron at position $j + l, k + m$.

$$\sigma(b + \sum_{l=0}^{w} \sum_{m=0}^{h} w_{l,m} a_{j+l,k+m}) \tag{111}$$

Equation (103) is also known as mathematical convolution and is the one that gives the name to this type of networks. That is why in literature, sometimes the equation is also written as (112) in which $a^{l,m}$ is the set of output activations of feature map m in layer l, a^{l-1} the set of input from the previous layer and the $\cdot$ the convolution operation between the input and the shared weights of the feature map.

$$a^{l,m} = \sigma(b + w \cdot a^{l-1}) \tag{112}$$

The parameter sharing introduces a concept where, the same filter is computed and reused across all the input volume. In that way the same feature detector is used across all the image and thus that introduces a translation invariance between the input volume and the convolution layer. In that way all neurons in a convolution layer, are looking for the same feature and thus, if the same feature is present in whichever part of the layer, the neurons will get activated. For that reason it is common in literature to call the map between input volume and convolution layer, as “feature map”. To increase the feature detection capabilities of a convolutional layer, more feature maps can be added and thus, forming multiple layers of feature detectors. A visualization of feature maps detected by a convolutional layer with 20 feature maps trained with MNIST dataset can be seen in Fig. 33.

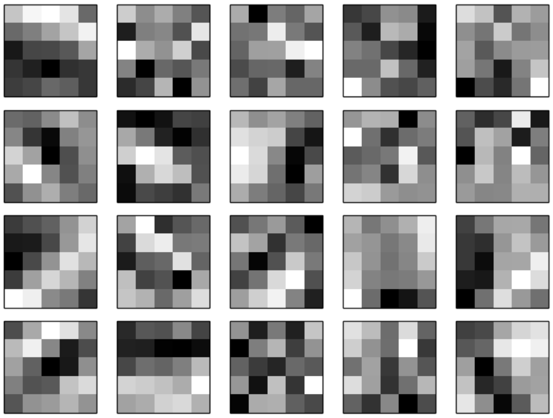


Figure 33 20 feature maps formed by training a convolutional layer with MNIST dataset [29]

It is clear that the feature maps have a spatial structure with lighter and darker regions, sensitive to corners. This resembles a lot the traditional approaches of feature extraction methodologies, like Gabor filters etc; yet the extraction of those features are relatively different as they are learned based on the training dataset and they do not follow a specific mathematical procedure. It is important to note here that the parameter sharing approach is extended across the whole depth of the input volume. Another important benefit of the parameter sharing approach in the feature map formation is that there is a considerably large decrease in number of free parameters required to compute in each layer, compared to traditional fully connected layers. This decrease makes the entire training process considerably faster and thus allows the network to be trained with relatively high number of inputs, like raw image pixels.

5.7. Fully Convolutional Neural Networks

The large number of free parameters that are introduced by the addition of fully connected layers in the end of classic CNN architecture affects drastically the computation complexity in both the training and testing process of any CNN architecture. Furthermore other layers such us, pooling, introduce complexity to the overall design of any CNN architecture. For these reasons in recent years, a more simplified approach has been adopted [62], [63] with name Fully Convolutional Neural Networks (FCNNs). At its core FCNN architecture is composed only by convolutional layers. The main different with the classic CNN architecture is that it learns filters everywhere; even in the decision-making layers at the end of the network are filters. FCNN is learning representation and making decisions based on local spatial input while appending a fully connected layer at the end of it enables the network to learn using the global information and the spatial arrangement of the input is lost.

5.8. Case Studies

There are many CNN architectures that have been proposed the last years that are dealing specifically with image and audio data**.** In this section a presentation of the most interesting architectures is presented along with their unique features that each one has to offer and how it changed the way CNN architectures are formed.

5.8.1. LeNet

LeNet-5 [34] architecture, which consists of 5 layers, was the first CNN architecture presented in literature that featured multiple convolution layers, combined with pooling and fully connected layers as the final layers of the architecture.

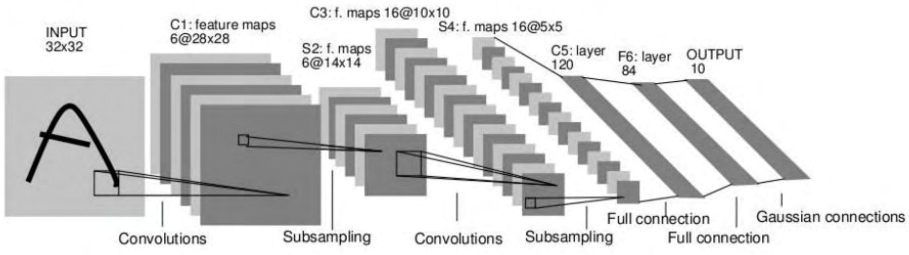


Figure 34 LeNet-5 architecture overview [34]

In Fig. 34 an outline of the architecture is presented. More specifically the architecture features an input layer for grayscale raw image data of 32x32 dimensions which means that the input feature vector size is 1024. The input layer is followed by a convolution layer with 6 feature maps of 28x28 filters followed by a max-pooling layer of 14x14 pooling filters. The second convolution layer has as an input volume the filters extracted from the previous pooling layer and extracts 16 feature maps of 10x10 filters. On this a second sub-sampling layer is added with filter size 5x5 followed by 2 fully connected layer and an output layer of 10 neurons, one for each possible class of the dataset. The architecture was trained upon MNIST dataset and it was successfully used in banking industry. It is worth mentioning that, at that time the usage of sigmoid activation functions was used as rectified linear units were not available before [33], yet as the task was relatively simple compared to newer datasets, it did not affect the classification performance. This architecture marks the initiation point where the newer approaches, like the one presented in section 4 with name “AlexNet” was based on.

5.8.2. AlexNet

This architecture has been presented in detail in previous sections, yet it is worth mentioning here that even if it was based on LeNet-5 initial architecture, it improved the initial model at many points. The introduction of rectified linear units (ReLU) as activation functions for the neurons of the network along with the introduction of local response normalization layers reduced drastically the overfitting issues of the initial model allowing the entire architecture to go deeper and thus improve the predictive power of the entire model. Furthermore the experiments done by the authors helped to understand the importance of smaller filter sizes for the feature maps. Finally, in this architecture the introduction of dropout layer in the last fully connected layers to reduce overfitting was adopted.

5.8.3. VGGNet

An architecture with codename “VGGNet” [64] was proposed in 2014 aiming to explore how deep a CNN architecture can grow. The architecture was based on simplicity, extracting relatively small filters of size 3x3. The authors proposed two variations of the architecture, one featuring 16 layer deep and another featuring 19 (Fig. 35). In both cases authors found challenging the training process, specifically regarding convergence on the deeper networks, so in order to make training easier they initially trained smaller versions of the network. The network was trained in ILSVRC-2012-val dataset achieving a top-5 error of 7.5% and on ILSVRC-2012-test with top-5 error of 7.4%.

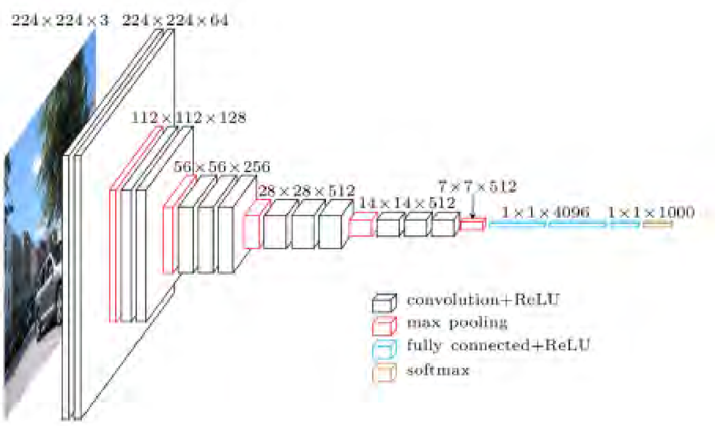
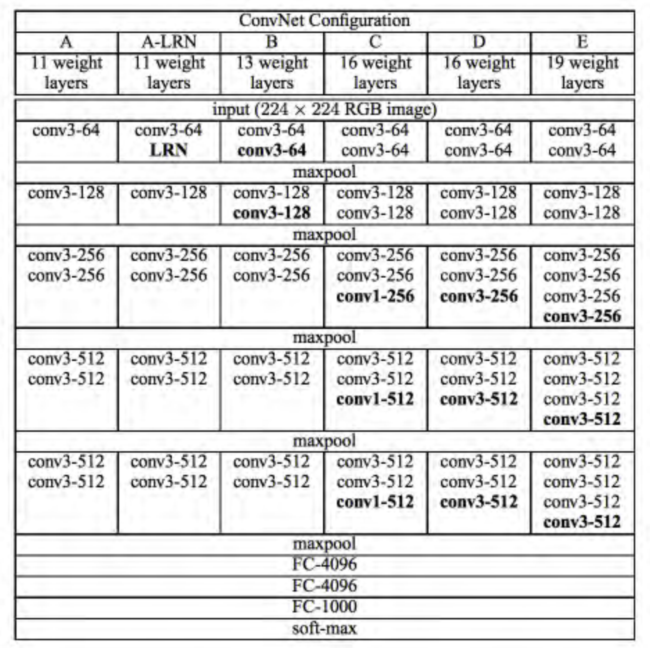


Figure 35 VGGNet-19 architecture [64]

One of the problems of VGGNet architecture is the vast number of free parameters that have to be trained, as in the 16 layer architecture, the network contains 138million parameters which made the training process a relatively computational and resource demanding operation, prone to overfitting.

The networks that were trained by the authors, can be seen in (Fig. 36). Each time a smaller network was converging, the authors introduced a larger one, utilizing the trained ones as initializations. This process is also called “pre-training”. While this process is logically correct, it is also relatively time consuming task as it requires an entire network to be trained, before it can serve as an initialization for a deeper network. Nowadays, pre-training, in most cases, is no longer used and instead, other methodologies are utilized, such as [65] which introduced the concept of Parametric Rectified Linear Unit (PReLU) which is a generalization of classic ReLU improving the model fitting with nearly zero extra computational cost and little overfitting risk. This work also introduced a robust initialization method, which is based on the investigation of variance of the responses in each rectified layer that particularly considers the rectified nonlinearities enabling them to train extensively deeper rectified models.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 36 Variations of VGGNet architecture from smaller (A) to larger (E) [64]

5.8.4. ZFNet

This architecture known as "ZFNet" [66] was based on the initial "AlexNet" architecture, keeping the same number of convolution and pooling layers while tweaking the hyperparameters of them and more specifically expanding the size of the middle convolution layers (Fig. 37). That made the network able to achieve a top-5 error rate of 14.8% in the ILSVRC 2013 competition marking it as the winner. It is worth noting that although ZFNet follows a really similar architecture with its predecessor of it, AlexNet, it was trained with only 1.3 million images while AlexNet with 15 million. The first tweak of the original architecture was the alternation of the first convolution layer filter sizes which changed from 11x11 pixel to 7x7. The reasoning behind this decision was that, smaller filter sizes, especially in the initial convolution layer, allow the network to retain a lot of original pixel information from the input volume. The original 11x11 proved to opt-out a lot of relevant information and thus that is one of the reasons that ZFNet was able to be trained with much smaller dataset. Following this principle, the rest of the layers are following an increased filter size, while the architecture moves deeper, providing more abstract features to the final layers. While the winning of competition was a major achievement, the main goal of the work was to understand the way that convolution neural networks work and more specifically provide a way to visualize the feature maps of the convolution layers.

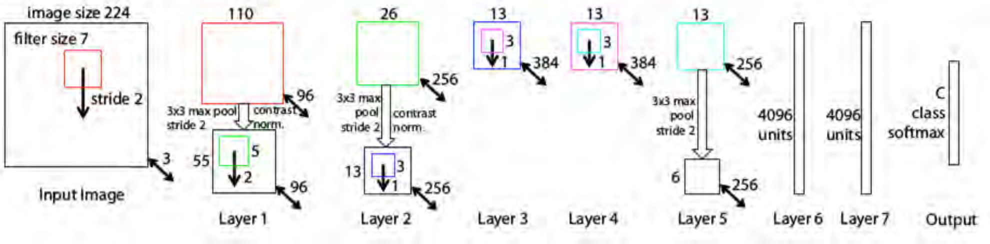


Figure 37 ZFNet architecture break down [67]

In particular ZFNet showed that the power of CNN architectures come from the existence of large datasets combined with large computational resources, usually, GPUs. Yet by that time, there was no clear understanding of how these networks work. Their contribution to the field was to provide a way to visualize the weights / feature maps extracted by convolution layers of the network and thus increase the understanding of what features actually the layer learns. The technique used to create these feature map visualizations is called "DeConvNet", as the goal of the technique is to act the opposite way than a convolution layer and thus, visualize the feature maps in pixels.

The idea of DeConvNet is to attach a deconvolution layer right after every convolution layer of a trained network. Then an image is fed into the network by performing a forward pass, just like in any other CNN. To examine the features that a feature map has learned in the $\boldsymbol{n^{th}}$ layer, the activations of that map are held while, the rest of feature maps are set to zero. Then the feature map is passed through the deconvolution layer which has the same features as the original CNN. The input vector then passes through a series of unpooling, rectify and filter operations, one for each preceding layer until the input space match the input volume. The result of the operation for the first two layers on ZFNet model are illustrated in (Fig. 38). More particularly it can be seen that the initial convolution layer learns more specific features about the images, like colors that reassemble close the original input volume, while moving to deeper layers the network learn more abstract features like corners.

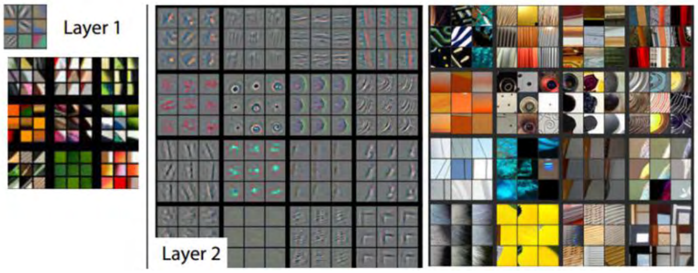


Figure 38 Deconvolution operation of the first and second layer of ZFNet model using as input the images in the right [67]

ZFNet and more specifically the deconvolution operation, gave a tool an important tool to CNN researchers for effective understanding and debugging a CNN architectures, by providing a visualization technique of the artifacts that a model learns.

5.8.5. GoogLeNet

In ILSVRC-2014 competition a remarkably different approach was introduced to the field of CNN architectures with name "GoogLeNet" [52] which is also known as the "Inception" model. This model features 22 layers (Fig. 39) and it was the winner of the competition with top-5 error rate of 6.7%. While VGGNet was proposed in the same competition and followed the traditional approach of stacking convolution and pooling layers with target of increase the depth and thus the performance of network, Inception module followed a different approach. The authors emphasized on the issues that traditional stacking approaches suffer, i.e., they require an increased amount of computational resources, both in memory and power usage along with the fact of increasing chances of model overfitting while number of free parameters increase.

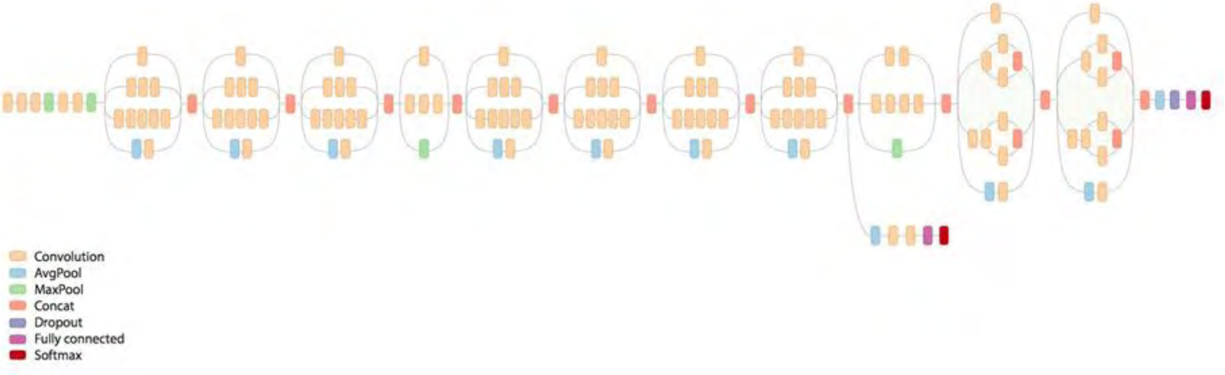


Figure 39 GoogLeNet architecture visualization [52]

To deal with these problems, the Inception module was introduced. As it can be observed in Fig. 39, the first change on the traditional CNN architecture, is that there are parts of the network that are activated in parallel. A closer view of this is illustrated in Fig. 40. In this approach, each Inception module has the chance to perform both convolution and pooling operations of multiple filter sizes, instead of having to decide just one operation per layer. Later on, each of the mini-layers output is concatenated, creating a new input volume for the next Inception module layer.

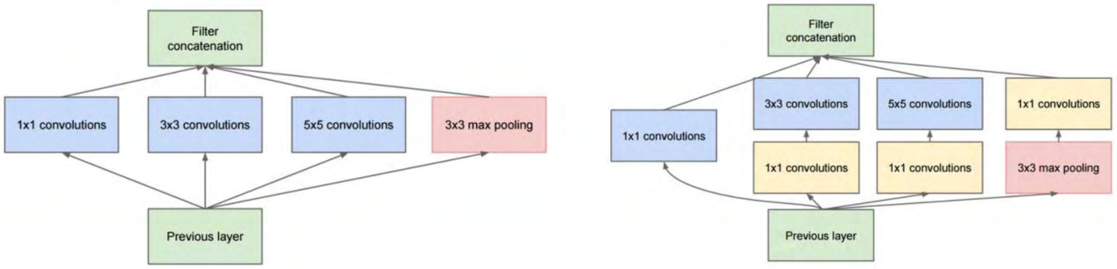


Figure 40 Left: general idea of the Inception module
Right: fully featured Inception module featuring parallel processing architecture [52]

While the original approach give many advantages over the traditional one, suffers from the problem that the output volume of a standard Inception module layer is huge. To deal with this problem, an introduction of 1x1 convolution layers was added to the original idea. These convolution layers were added right before the 3x3 and 5x5 convolution layers and right after the 3x3 max-pooling layer, serving as a dimensionality reduction mechanism. The dimensionality reduction provided by 1x1 convolution layer was similar to any pooling operation, but instead of trying to deal with the width and height of the input volume, this is dealing with the depth reduction of it. The main goal of this idea was to allow each Inception module layer to extract both high grained details about the input volume, by using small filter sizes and on the other hand, combine them, with more abstract details extracted by feature maps with larger filter size as 5x5.

GoogLeNet contains in total 9 Inception module layers, with all of them containing over 100 layers. By not using any fully connected layer in the end of the CNN architecture, GoogLeNet managed to reduce the number of free parameters drastically, having 12 times lower number than the original AlexNet architecture. To fight overfitting, the authors introduced max-pooling operations as a spatial size reduction mechanism along with ReLU layers after each convolution layer that helped improve the nonlinearity of the network. The replacement for the last fully connected layer was done by an average pooling layer, which reduced the input volume from 7x7x1024 to 1x1x1024. Concluding, GoogLeNet and in particular the Inception module, was the first architecture proposal showing that CNN architectures do not have to be a collection of sequential stacked up layers.

5.8.6. ResNet

In ILSVRC-2015 another, radical CNN architecture was introduced with codename "ResNet" [53]. This approach was the winner of the competition with a remarkably low top-5 error rate of 3.6%, considerably lower than the pre-accentors and for the first time, outperforming the human top-5 error rate which is between 5 and 10%. The idea was based on the introduction of residual blocks, stacked up together in a top-down approach forming a deep convolution network of 152 layers (Fig. 41).

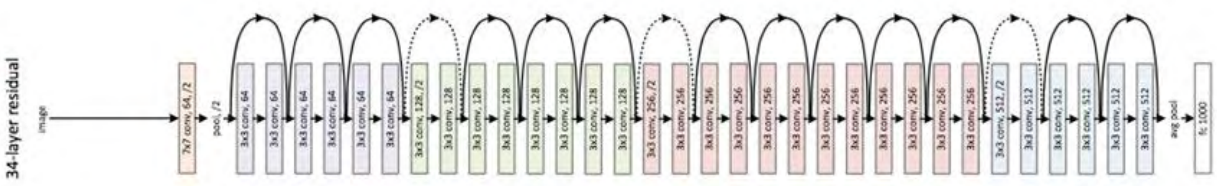


Figure 41 A residual network architecture of 34 layers [53]

Residual block relies on the idea of having an input volume $\boldsymbol{x}$ feed forwarded through a series of convolution – ReLU – convolution layers. Let this pass-through function be $H(x) = F(x) + x$. In a traditional CNN approach $H(x)$ will be equal to $F(x)$ while in residual block, the $F(x)$ output is combined with the original input vector x. That change resembles a slight change to the original input volume which comes into contrast to the traditional approach as there the output volume $H(x)$ resembles a completely new vector. Another reason for the effectiveness of the residual block approach is that in backpropagation backward pass, the gradient flows easier through the deep graph, compared to traditional approach, as the operations performed are mainly additions (Fig. 42).

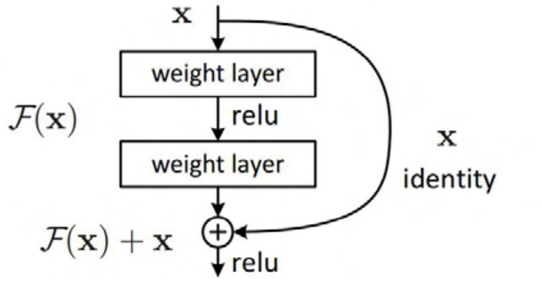

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 42 Stripped down, residual block operation [53]

This approach showed that CNN architectures can become very deep and yet trained effectively, without suffering from overfitting. Interestingly ResNet input volume was compressed by a factor of 4 from the second residual layer, and thus downscaled from 224x224 original input volume size to 56x56. The network was able to be trained in two to three weeks utilizing 8 GPU enabled machines. Finally the authors noted that naïve increase of the layers can lead to lower performance, as their original approach was to train a network of 1202 layers, yet the results were poor, compared to the proposed ResNet configuration, mainly because of overfitting due to increased number of free parameters.

5.8.7. Region Based CNN Architectures

One of the domains that computer vision is dealing with since the begging of the sector, is the problem of object detection. That is, initially detect if an object is present in an input volume and then to locate it. So far the previous CNN architectures were dealing with the first part, or in other words the detection of presence of an object in an input volume. Region Based CNNs and more specifically R-CNN [68] in 2013 and later on Fast R-CNN [69] and Faster R-CNNs [70] in 2015, were introduced to tackle both of the problems of object detection, with the last one achieving real-time object detection rates.

The original approach of R-CNN can be split into two different components; initially to propose a region for the object in the image and then the classification step. For the region proposal step the authors used Selective Search [71] while they note that any class agnostic region proposal method can fit into the model. Selective Search approach produces 2000 different regions from an input image that are most likely to contain and object. After the region extractions the sliced images are resized in order to be fed into a pre-trained CNN network, which in the author case was the AlexNet model. That network was used as a feature extractor, extracting a feature vector for each of the proposed regions. These feature vectors were then used to train a set of linear SVM classifiers who performed the final classification process. Finally the vectors were also fed into a bounding box regressor to obtain the most accurate positions for the bounding boxes of the objects into the final image. To combine similar overlapping bounding boxes a non-maxima suppression were used. These steps of the process are illustrated in (Fig. 43).

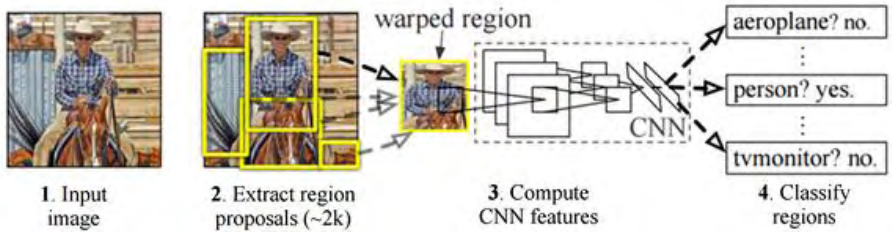


Figure 43 Steps involved into R-CNN architecture [68]

Fast R-CNN was an alternation of the original design aiming to solve three main issues. On the initial R-CNN design training was a computational expensive and slow process that involved multiple stages, mainly CNN training to SVMs and then to bounding box regressors. Fast R-CNN solved that issue by sharing the computation of convolution layers between different proposals and swapping the order of generating region proposals and running the CNN. In this approach the first step was to feed the image into the CNN model, then extract region proposal features from the last convolution layer and finally fully connected layers, classification and bounded box regressor layers were attached as final layers of the network (Fig 44). This significantly reduced the complexity of the previous architecture and increased the speed towards real-time object detection.

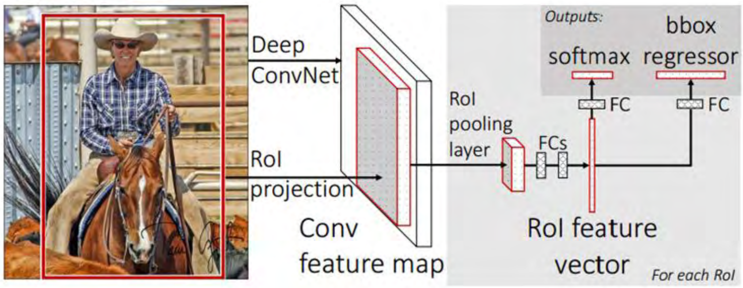


Figure 44 Fast R-CNN architecture [68]

The last work of the same authors, with name Faster R-CNNs was introduced the same year, 2015, with a target to provide a general purpose real-time object detector. The changes were not radical yet significantly reduced the computation complexity of the pre-accentors proposals. Mainly the change was the addition of a region proposal network (RPN) after the last convolution layer of the network. This insertion gave the ability to the model to just look at the last convolution feature map and produce region proposals from that; the same pipeline of Fast R-CNN is used as the final steps of the architecture (Fig. 45). Faster R-CNNs were a ground breaking approach into the field of object detection and have now become the standard.

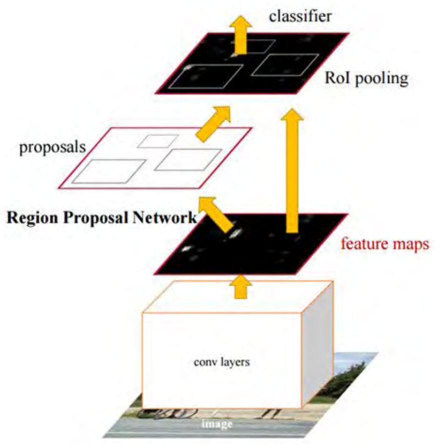


Figure 45 Faster R-CNN architecture [68]

5.8.8. Generative Adversarial Networks

Generative Adversarial Networks (GAN) [72] are a special type of networks that are dealing with prediction error maximization of trained CNNs, i.e. to "fool" a trained CNN. The goal of these networks is, by using one image classified as A by a CNN, to alter it as less as possible, yet enough to affect the prediction of a model (Fig. 46). There creation was inspired by the work of [73] which was dealing with the fact that convolution neural networks, even if they are performing relatively well in the field of computer vision, they actually "see" differently than the biological vision. They tend to rely on simple, spatial, not heavily correlated features which can easily lead to miss-classifications that, by the human brain have no logical explanation.

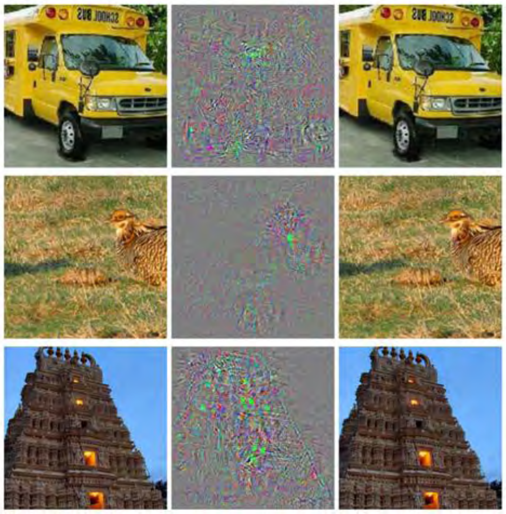


Figure 46 Left: images that are correctly classified. Right: Slightly altered examples that lead to misclassification Middle: visual representation of the distortion between the left and right examples [72]

GANs are working by combining two models, a generative model and a discriminative model. The first has the role to generate slight changes to the input image in order to trick a CNN and misclassify the input volume. On the other hand, the second, tries to detect the changes and classify if the input volume is natural, authentic image, or it is artificially generated. The goal of this procedure is to train the discriminative model by using the images generated by the generative model; as the training process continues, both models are trained until they reach the point discriminative model is unable to detect any change on the generated image compared to the original one. Ultimate goal of the procedure is that the final model, is improved in such a way, that is aware of the internal structural representation of the data and thus can be used as a reliable feature extractor in a CNN architecture. As a sub-product of this procedure, the models are able to create artificial images that are hardly recognized as unnatural [74].

5.8.9. Generating Image Descriptions

An interesting combination of CNN architecture combined with a bidirectional RNN was presented in 2014 in [75]. The goal of this work was to train a model that is able to perform object detection along with natural language descriptions for each detect object into an input volume (Fig. 47).

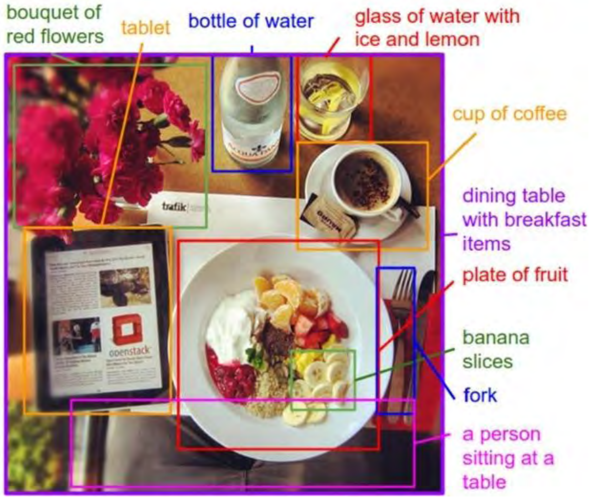

bouquet of red flowers
tablet
bottle of water
glass of water with ice and lemon
cup of coffee
dining table with breakfast items
plate of fruit
banana slices
fork
a person sitting at a table

Figure 47 Detected objects into a natural image, along with natural language descriptions for each region [75]

The framework can be broken down into two parts; the "alignment model" and the "generation model". The alignment model has as goal to align the visual data of the image and the textual data, by accepting as an input an image and a descriptive natural language sentence of it. This process is handled by [76] which is trained on compatible and incompatible image and sentence pairs. The first step of the process is to process the images, without the sentences, using a pre-trained R-CNN, which in the case of [75] was trained with ImageNet data. The results of the processing, where scored and the top 19 regions and the image were represented into a 500 dimensional space feature vectors. To extract information about the natural language part of the same images, a bidirectional recurrent neural network is used which serves as an illustration of the information about the context of the words in the given sentences. Lastly computing the inner product of both extracted features the framework computes the similarity between them.

The generation model, receives as an input the dataset created by the alignment model process and outputs descriptions for the given images. A CNN is used in this process with the final, softmax layer replaced with the input neurons of an RNN which serves as a probability distribution extraction from the different words in a sentence.

5.8.10. Spatial Transformer Networks

An interesting type of network created in 2015 with the name "Spatial Transformation Networks" [77] or SPNs has as a goal to create images that are easier to be classified in later phases and thus serves as a pre-processing stage of the input volume of a CNN. The two goals of that pre-processing are the pose normalization and spatial attention, with the second meaning to bring attention of the image towards the correct object which is blended into a crowded environment.

Traditionally in CNNs the layer that was used to increase spatial invariance of the model, was the max-pooling layer as it had the ability to maximize the importance of a specific response of an input volume, by removing the weakest responses and thus the increasing the importance of the relative position towards other feature locations. The proposed methodology, with name "spatial transformer layer", is dynamic in such a way, that it is producing different behaviors, depending on the input volume.

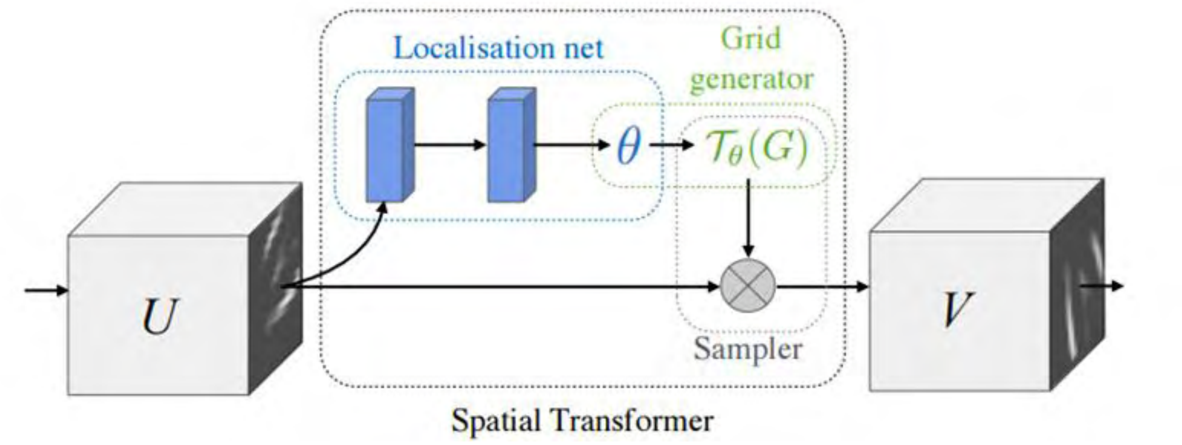


Figure 48 Spatial transformer layer input signal flow broken down into three components [77]

The spatial transformer layer (SPL) can be broken down into three components (Fig. 48). The first component is a localization network with an input volume, the original input and as output, the spatial transformation that has to be applied. The parameters for an affine transformation, $\boldsymbol{\theta}$, can be a six-dimensional vector. The second component deals with the creation of a sampling grid that is a result of wrapping the regular grid with the affine transformation created by the previous localization layer. Lastly, the final component is called "sampler" and its whole purpose is to perform a wrapping of the input feature map. The SPL can be added as any other layer in a traditional CNN architecture and helps the network learn how to transform feature maps (Fig.

49) in a way that minimizes the cost function during the training process. The importance of this approach is that, instead of following the approach of altering completely a CNN architecture, by extending the depth of the layers to increase the accuracy, it chooses a different and simpler way of applying simple existing knowledge, and in this case affine transformation, to the data itself and more importantly as just a drop-in layer that can be reused to any CNN architecture.

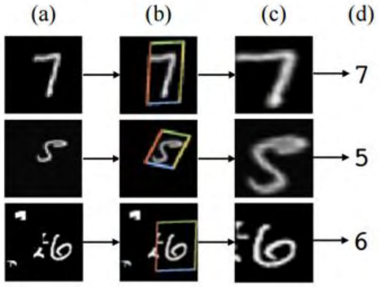


Figure 49 Spatial transformer layer process, applied to MNIST dataset [77]

6. Other Types of Neural Networks

Beyond the traditional feed-forward neural network architectures that are based on supervised learning in order to perform a classification challenge, there are other kind of networks which is briefly discussed in the following sections. The detailed analysis of each of those individual network architectures is beyond the scope of this work, yet a general overview of the most important methodologies is inspiring.

6.1. Autoencoders

A type of network that is trained to copy its input volume to its output is called autoencoder. The architecture it has at least one hidden layer whose parameters are trained in such a way that is representing the input volume. The architecture can be broken down into two individual components; the encoder $f(x) = h$ and a decoder $g(h) = x$. The $f(x)$ function tries to encode an input volume x into a symbolic, representation of h mapped onto the weights and biases of the hidden layers, while the decoder $g(h)$ receives as a parameter the encoded representation h and translates it back into x. The goal of this procedure is to train the autoencoder in such a way, that the mapping will not be direct and thus allow a certain number of error to be implied into the original trained vector, let x' and the decoding function be able to recall the original x (105). The generic autoencoder model is illustrated in (Fig. 50).

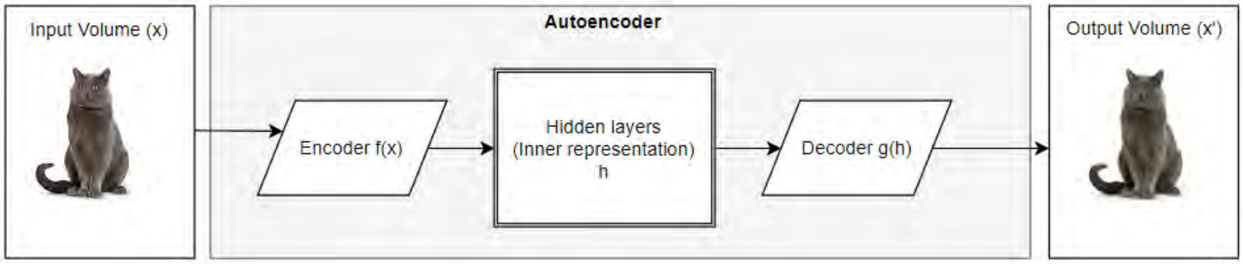


Figure 50 A generic autoencoder model

Because of this need, autoencoders are restricted in ways that will not allow them to fully copy the input volume on which they are trained and thus, they are forced to learn the most important aspects of their input which makes them a great tool to be used as a feature extractor. Autoencoders are not a new idea as they have been proposed originally in 1987 [78] and since then they have been improved and generalized [79][80]. They can be trained with the same procedures like any other feed-forward network, typical mini-batch gradient descent and backpropagation, yet there have been proposed other approaches like recirculation [81] in which the activations of the network is compared with the original input. This process is more biologically plausible than backpropagation approach, yet technically much more computation intense process and thus, is not commonly seen in machine learning applications. Auto encoders can be broken down into two main categories; the undercomplete and regularized autoencoders.

$$g(f(x^{'})) = x \tag{105}$$

6.1.1. Undercomplete Autoencoders

The first autoencoder that was proposed was called undercomplete autoencoder. They operate by using an encoder of input vector x and mapping it into a smaller dimensional space h finally recalling it with a decoder of same size as the encoder input size. Generally an autoencoder whose inner hidden layers neurons are smaller, dimensionally, than the input vector is called undercomplete. The learning process is happening by minimizing a loss function, such as root mean square loss, which is penalizing the output depending on how close is to the input vector.

An undercomplete autoencoder with a linear decoder and as a loss function it uses the mean squared error, and it learns to span the same dimensional space as a typical PCA because PCA is restricted to a linear map. That is because the model is trained to perform a copy task of the input to output volume and as a byproduct the network has learned the principles of the subspace. On the other hand an autoencoder with a non-linear encoder and decoder, can learn much more than the classical PCA approach. In order to do that the autoencoder must be configured in advanced with a smaller capacity than the one needed to fully represent the input volume, in order to limit the model and force it to store the most important information of the input vector. Same as any neural network, theoretically a nonlinear autoencoder with at least one hidden layer and enough neurons is able to represent any type of input, yet in practice this is computationally exhausting and thus more advanced architectures have been proposed, such as deep autoencoders or autoencoders that are not dependant on their inner storage / number of neurons to be lower than the input dimensional space.

6.1.2. Regularized Autoencoders

In order to deal with the problem of overcompletion that undercomplete autoencoders suffer when their inner hidden layer is big enough to allow it to learn the input dimensional space and thus let the model to not extract any useful features from the input space, regularized autoencoders have been proposed. In practice regularized autoencoders have as basic principle, not to limit the capacity of the model, but regularize the loss function to penalize overcomplete behavior.

6.1.2.1. Sparse Autoencoders

Sparse autoencoders are a family of regularized autoencoders whose regularization function relies on a penalizing function $D(h)$ on the hidden layers h in addition to the typical reconstruction error and thus, the loss function can be adjusted into (113). Their usage is typically found as a feature extractor for other classification applications.

$$L\left(x, g\big(f(x)\big)\right) + D(h) \tag{113}$$

There is no clear probabilistic interpretation of this regularization expression compared to others, such as weight decay found in other regularization functions. Weight decay, is dependent on the previous state of the weights and thus can be thought as a regularized maximum likelihood corresponding to maximization of a probability $p(\theta|x)$, which is equivalently interpreted to maximum $log(p(x|\theta)) + log(\theta)$. Therefore the term $log\big(p(x|\theta)\big)$ is usually the data log-likelihood and log-prior over parameters the $log(p(\theta))$ term which incorporates the preference over particular values of θ. On the other hand, sparse autoencoder loss function regularization is utilizing the data themselves and thus by definition not dependant on previous acquired knowledge.

The entire sparse autoencoder can be seen as approximating maximum likelihood training of a generative model. Let h be the latent and x the input parameters of the model with an explicit joint distribution $p_{model}(x,h) = p_{model}(h)p_{model}(x|h)$, where $p_{model}(h)$ is the prior distribution of the model over latent parameters, which represents the model prior-knowledge of x input volume. The log-likelihood can now be expressed as (114) which is similar to sparse coding generative models, yet it uses the h as the output of the parametric encoder (115) rather than a result of an optimization that expresses the most likely h.

$$log\big(p_{model}(x)\big) = log \sum_h p_{model}(h,x) \tag{114}$$

$$log\big(p_{model}\,(h,x)\big) = log\big(p_{model}\,(h)\big) + log\big(p_{model}\,(x|h)\big) \tag{115}$$

The term $log\big(p_{model}\,(h)\big)$, of the equation (115) can be sparsity inducing such as the Laplacian prior $p_{model}\,(h_i) = \frac{\lambda}{2}e^{-\lambda|h_i|}$, which corresponds to an absolute value sparsity penalty which yields (116) and (117) where c a constant value dependant only on the value of the hyperparameter λ .

$$D(h) = \lambda \sum_i h_i \tag{116}$$

$$-log\big(p_{model}\,(h)\big) = \sum_i (\lambda|h_i| - log(\tfrac{\lambda}{2})) = D(h) + c \tag{117}$$

From the above the sparsity penalty introduction is just a consequence of the model distribution over its latent parameters and thus is not a regularization term. As a result training these type of autoencoders is a way of training a generative model and thus the features learned by them have the useful feature of latent parameters that describe the input at great extend. In early work of sparse autoencoders [82],[83], various sparsity forms have been explored, proposing connection between the sparsity penalty and the $log(Z)$ term arising from maximum likelihood applied on undirected probability model $p(x) = \frac{1}{Z}p'(x)$ which is based on the idea of minimizing the $log(Z)$ regularize a probabilistic model from having high probabilities everywhere. This connectivity leads to the fact that applying sparsity on an autoencoder prevents it from low reconstruction error everywhere. Another way to achieve actual zeros in h space, for sparse and also denoising autoencoders has been introduced in [84] which exploit the properties of rectified linear units (ReLUs) in the h layers of the model. That is because ReLUs act as a prior that enforces an absolute value penalty, zero and thus it provides an indirect control to the average number of zeros into the representation of the encoded data.

6.1.2.2. Denoising Autoencoders

Denoising autoencoders (DAE) are similar to sparse autoencoders with difference on penalizing function, in which instead the computing the penalty D, they rely on the computation of the reconstruction error term and the addition of it in the cost function. Based on that the cost function can be expressed as (118), where x' is a noisy representation of the original x input vector and thus the goal of a DAE model to correct this noise and not just to copy the input volume. It has been shown in [85] and [86], that this process forces the encoder f and the decoder g to learn the structural representation of the input volume.

$$L(x, g(f(x^{'}))) \tag{118}$$

DAE process works by introducing a corruption process to the input volume x such as $C(x'|x)$ which can be seen as a conditional distribution over the corrupted sampled features of the vector x. The DAE model can learn the reconstruction distribution $p_{reconstruction}\ (x|x^{'})$ by been trained with pairs $(x, x^{'})$ by following a three-step procedure. Initially a sample is extracted from the training x vector. Secondly a new sample, this time corrupted x', is extracted from $C(x^{'}\,|\,x = x)$. The two previous steps are creating a training pair $(x, x^{'})$ which is used by the DAE in order to estimate the reconstruction probability distribution (119) where h is the output of the encoder $f(x^{'})$ and $p_g(h)$ the decoder output of the model.

$$p_{reconstruction}(x|x^{'}) = p_g(x|h) \tag{119}$$

As a learning algorithm for the above procedure is relatively common to follow a gradient descent based algorithm, such as mini-batch gradient descent on the negative log-likelihood $-log(p_g(x|h))$.

An alternative to maximum likelihood is the “score matching” proposed in [87] which provides an estimator of probability distributions which is a product of encouraging a model to have the same score as the data distribution at every training point x and thus the score is a particular gradient field $\nabla_x log(p(x))$. Learning the gradient field of $log(p_{data})$ is equivalent of learning the structural representation of the p_{data} .

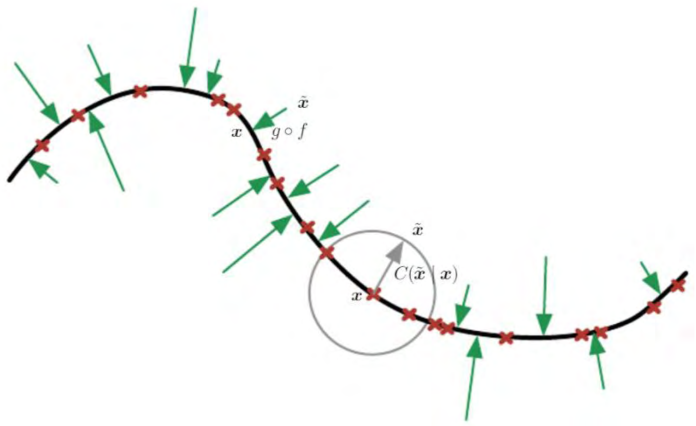


Figure 51 Denoising Autoencoder training to map a $(\boldsymbol{x}, \boldsymbol{x}')$ pair [88]

An important property of DAE models lying on the fact that their training criterion with the conditional Gaussian $p(x|h)$, makes the model to learn a vector field $g\big(f(x)\big) - x$ capable to estimate the score of the data distribution. An illustration of a DAE learning procedure can be seen visually in (Fig. 51) in which, a DAE is trained to denoise a x' vector back to its original form x. The red crosses which are lying on top of the low-dimensional manifold black line are representing the training examples x. Corruption process of x into x' is illustrated with a gray circle and the green arrows are representing the dynamics appearing in the model while it is trained to learn the vector field $g\big(f(x)\big) - x$.

A specific type of autoencoder which has sigmoid neurons in its hidden units and linear neurons to the reconstruction layer and trained with Gaussian noise and mean square error as reconstruction cost function is equivalent to training a specific kind of undirected probabilistic model known as RBF with Gaussian visible units [89] which is a model that provides explicit probabilities of type $p_{model}\ (x; \theta)$. By trying an RBF model using a procedure called “denoising score matching” introduced by [90], it has been shown that the resulting learning algorithm is equivalent to denoising training in the corresponding autoencoder. When the noise level is minimized to be close to zero and the training examples are approaching infinity the consistency of the model is restored, in comparison to a fixed noise level, in which regularized score matching is not a consistent estimator and thus recovers a blurry version of the desired distribution. It has also been shown that other connections between autoencoders and RBMs exists, such as if the score matching is applied into an RBM model, then the cost function yields an identical reconstruction error combined with a regularized term constructive penalty of a contractive autoencoder (CAE) [91]. Yet another similarity shown in [92], is that a constructive divergence training of RBMs is obtained by an autoencoder gradient.

An interesting property of a continuous-valued training vector x has found by [85] in which has been shown that the denoising Gaussian corruption and reconstruction distribution are constructing a score estimator for any generic encoder and decoder parameterization. More specifically a generic encoder and decoder model can be used to estimate score when it is trained with squared error criterion (120) and input corruption (121) where σ^2 is the noise variance.

$$\left|g\left(f(x')\right) - x\right|^2 \tag{120}$$

$$C(x' = x'|x) = N(x'; \mu = x;\ \Sigma = \sigma^2 I \tag{121}$$

In Fig. 52 an illustration of the above training procedure outcome is shown. The 1-D curved manifold surrounding the vector space learned by a DAE is depicted as the data concentrates in a 2-D space. The arrows are proportional to the reconstruction $g\left(f(x)\right)$ minus the input vector of the model, and are pointing towards the highest estimated probability distribution. The smaller the arrows, the closer are to a local or global maxima or minima of the estimated density function. On the other hand when the arrows length is long, it means that the probability can be increased by moving in the direction of them as the length is expressing the norm of the reconstruction error.

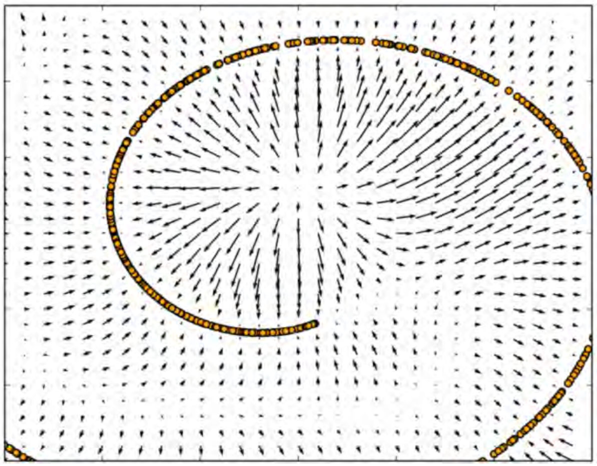


Figure 52 Vector field learned by DAE on a 2-D space [88]

Unfortunately there is no guarantee that the $g\big(f(x)\big) - x$ is corresponding to the gradient of any function or either the score and thus early results [89], where dependant to a specific parameterization of the model whom $g\big(f(x)\big) - x$ was obtained by derivative of a different function. Only lately and more specifically in 2015 [93], a generalized version of [89] results was found by identifying a family of swallow autoencoders whom reconstruction minus the input vector corresponds to a score for all the members of that family. It is worth noting that DAE family autoencoders can also be used beyond the field of probability distribution representation and more specifically can be used as a generative model in which samples can be drawn by the output distribution.

6.1.2.3. Penalized Derivatives Regularization

Autoencoders can also be regularized by following a strategy that utilizes a penalty D similar to sparse autoencoders, yet in different form (122). This procedure forces the model to learn a function resilient to slight changes of the input vector x and thus as the penalty is applied only to the training vectors, it leads to a learning of features that capture the probability distribution of the training examples. An autoencoder model that is based on this regularization approach is called contractive autoencoder (CAE) and has theoretical connections to the previously described DAEs, probabilistic modeling and manifold learning.

$$D(h,x) = \lambda \sum_i |\nabla_x h_i| \qquad (122)$$

6.1.2.4. Contractive Autoencoders

Contractive Autoencoders or CAE, are another family of regularized autoencoders that were introduced by the work of [91] in which an alternative penalizing, derivative based function was introduced (123) in which $h = f(x)$. This function is the squared Frobenius norm of the Jacobian matrix [94], which contain the partial derivatives of the encoder $f(x)$ function.

$$D(h) = \lambda \left|\frac{\vartheta f(x)}{\vartheta x}\right|^2 \qquad (123)$$

Then name “contractive” of CAE model arises from the fact that its training procedure forces the encoders input to be wrapped to a smaller neighborhood of output points as it is trained to resist fluctuations of its input volume. The work of [85] shown that there is a connection between contractive and denoising autoencoders as when a small Gaussian input noise is introduced to the second, the outcome is a denoising reconstruction error equivalent to the contractive penalty of a reconstruction function of a CAE that maps the input volume x to $r = g(f(x))$. That, even if the goal of the two encoders is different, with the first trying to make the reconstruction function resilient to small alternations of the input volume and the second trying to make the feature extraction function resilient to alternations of the input vectors, the outcome is the same.

The contractive capability of CAE model is limited only to local neighborhood of the input volume as globally, two different points, x and x' will be mapped to a different $f(x)$ and $f(x')$ respectively. To shrink the Jacobian matrix in a sigmoid model, is to make the sigmoid units saturate either to 0 either to 1. The outcome of this encouragement is that the autoencoder maps the input volume to extreme values and thus the sigmoid output can be interpreted as binary code. As a byproduct of the above, the CAE also maps the h throughout the space of the inner-storage hypercube.

In CAE, the balance of two forces that is enforced to any regularized autoencoder is the balance between the reconstruction error and the contractive penalty $D(h)$; the balance between those two forces is achieving an autoencoder whom derivatives $\frac{\vartheta f(x)}{\vartheta x}$ are tiny.

A comparison between CAE and PCA tangent vectors of the manifolds estimated between the two approaches is illustrated in (Fig. 53) on an image from CIFAR-10 dataset. It is clearly seen that although PCA is able to compute local accusations of the input volume, CAE is performing much better. That is because the CAE approach is based on hidden layer(s) whose volume is large enough to span across larger input space and thus exploit the inner connections of the input volume.

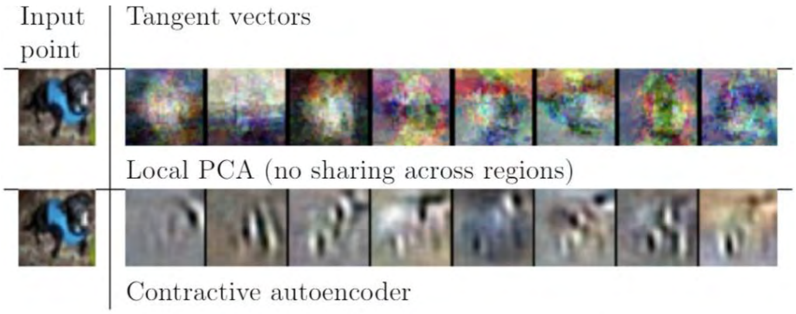


Figure 53 Visualization of CAE and PCA tangent vectors [88]

An issue of CAE model is that although contractive penalty is easy to compute in case of a singular hidden layer architecture, it becomes computationally exhausting to deeper models. One solution that was utilized by the authors of the model is to train each hidden layer individually and then stack them all together in order to form a deeper autoencoder. Unfortunately the tradeoff of this approach is that the result is not the same as training the entire encoder at once as the Jacobian penalty when it is computed across all the layers and a singular one is different; yet this approach is capable of preserving most of the desirable properties of the standard approach. Another issue of CAE model has to do with the penalty regularization, where when it is not regularized the model can fail to obtain any useful feature from the input volume. The regularization that is proposed by the authors was to multiply the input volume with a small constant $\boldsymbol{\varepsilon}$ and divide the output of the decoder with the same value. When $\boldsymbol{\varepsilon}$ is close to 0 the contractive penalty is also approaching 0 without having to learn anything about the input distribution, while the decoder maintains a perfect reconstruction. To achieve this, encoder and decoder layer weights have to be bonded. To do so, both layers are normal neural network layers consisted by affine transformation followed by an element-wise nonlinearity to impose a straight forward way to set the weight matrix of the decoder to be the transposed matrix of the weights of the encoder.

6.1.2.5. Hybrid Autoencoder Architectures

A hybrid encoder known as “Predictive Sparse Decomposition” (PSD) was introduced in 2008 by the work of [ref] (Kavukcuoglu et al., 2008) which combines the sparse coding with parametric autoencoders. The idea is to train a parametric encoder $f(x)$ and decoder $g(h)$ to the output of iterative inference and has successfully used for unsupervised image, video (Kavukcuoglu et al., 2009, 2010; Jarrett et al., 2009; Farabet et al., 2011) and audio (Henaff et al., 2011) feature extraction. The training procedure minimizes the equation (124) where γ and λ are hyperparameters of the minimization function following the same principles as sparse coding as the minimization of the function relies on the balancing of two forces, the hidden layer parameters and the model parameters .

$$|x - g(h)|^2 + \lambda|h| + \gamma|h - f(x)|^2 \tag{124}$$

In applications that use PSD, interactive optimization methodology is only used during the training phase and the parametric encoder is used to compute the features when model is deployed as the evaluation of the encoder is much less computationally intense process in comparison to the inferring h with gradient descent. Another property of PSD model is that it can be used as a layer in any deep neural network, such as CNNs, and be trained following different criteria than the original approach presented here, as the encoder function is differentiable.

6.2. Self-Organizing Maps

Self-Organizing Map (SOM) [95] or Kohonen map is a type neural network that is trained using unsupervised learning and is able to approximate high-dimensional input space by reducing it into, typically two dimensional space in order to group it together into clusters (Fig. 54). That is why the SOM networks are used typically when the discovery of data similarities in a dataset is desirable while the dimensional space does not allow a clear suppuration.

More specifically SOM networks are usually two dimensional rectangular arrangement of neurons that each hold a connection to each input value. Once the model is fed with the input vector, the network is starting the training process of self organizing the input according to the training rule. The weight vector of the neuron that is closest to the current state, is becoming the winning neuron. During the training process, the input values are getting adjusted in order to preserve the neighborhood relationships that exist in the input volume. The data similarity is based on the distance of the weight vectors and the input volume computed usually with a basic distance function, such as the classic Euclidian distance or Cosine distance.

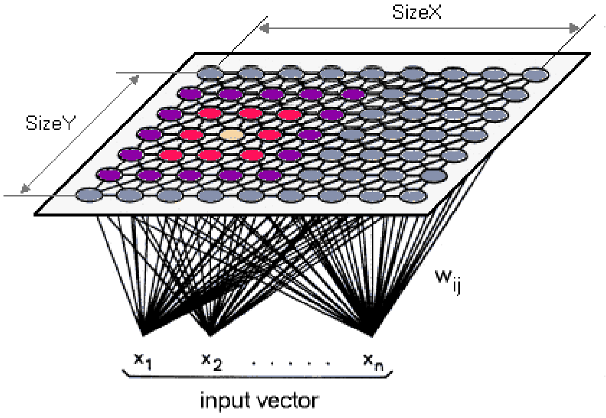


Figure 54 Example SOM network neuron arrangement [96]

The algorithm of SOM network training process is based on the fact that every vector in the training dataset is competing between each other in order to become the winning representation. Initially all the weights of the map are getting a default, random value which usually follows the Gaussian distribution. After the initialization of the weights, a random vector from the training dataset is selected and it is compared with the rest of the dataset in order to find which one is best representing the sample. Each weight vector is neighbor with the rest of the weight vectors that are close to it and the weight that is chosen is rewarded by increasing its probability, along with the neighborhood to be more likely to be picked in the random selection process of the next iteration. This process is repeated multiple times, usually a few thousand times until a threshold is reached.

The entire process can be summarized into 6 steps:

1. Weight initialization
2. Randomly pick a vector from the training set
3. Compare each node with the selected vector and choose the best performing one
4. Neighborhood of the winning vector is calculated while the neighbors of it are decreasing over each iteration
5. Allocate higher probability to the winning vector and its neighbors for the next iteration. The closer neighbors are experiencing higher weight changes
6. Repeat from step 2, until the maximum number of iteration is reached.

An example of the resulting map after the training process is illustrated in (Fig. 55) where the clusters are shown with light white color and the borders of each cluster via darker color. The darker the color between the white groups, the higher distance and thus the further distinctable are the clusters between each other.

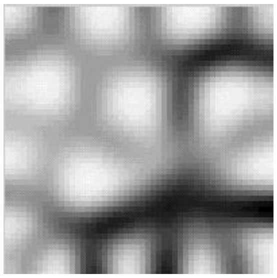


Figure 55 Visualization of SOM output [97]

6.3. Recurrent Neural Networks

Recurrent Neural Network (RNN) is a type of neural network that unlike feed-forward networks their arrangement contains loops between the neurons of each layer. That means that in a single, feed-forward pass of the input, the neurons of each layer, might have connections that are coming as input from neurons that are from any layer of the network. More generally, the connection arrangement of the entire network allows the neurons to contain cycles (Fig. 56). That gives the network the ability to exhibit dynamic temporal behavior that acts as memory. This ability gives them significant advantage over traditional feed-forward architectures on problems that are related to sequential data, such us, speech recognition [98][99], text auto correction[100] and more.

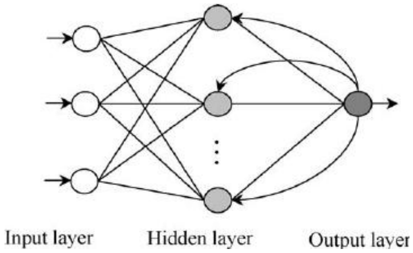


Figure 56 RNN architecture example with the hidden layer receiving input from the previous and output layer [101]

6.3.1. Long Short-Term Memory Neural Networks

A common recurrent neural network architecture that is used to classify, process and predict time-series data, is called Long Short-Term Memory (LSTM) [102] and was proposed in 1997 and since then they have been used in numerous applications, including text compression, unsegmented connected handwritten recognition [103] which was used to win the handwritten completion of ICDAR [104] in 2009. LSTMs are widely used today in modern applications, such as speech recognition in Android smartphones and typing speed encashment such as the iPhone "QuickType" technology.

In LSTM architecture a network is constructed using LSTM units along with normal neurons. LSTM units are a special kind of neurons that have the ability to remember their output for long or short period of time. This is possible by not using any activation function within their recurrent neurons and as an effect, their values are not vanishing when they are trained using the backpropagation algorithm.

In a typical RNN architecture, multiple simple chain modules are linked together in order to form a chain. An example of that can be seen in (Fig. 58).

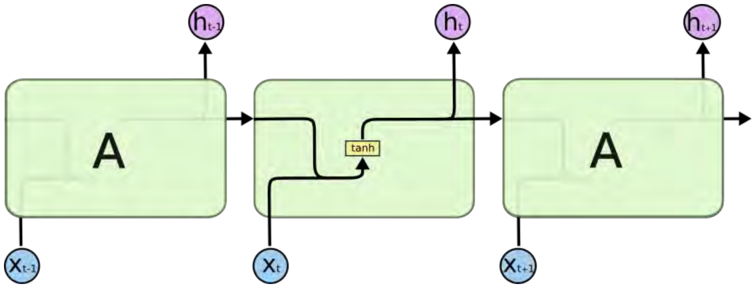


Figure 58 A traditional unrolled recurrent neuron, with activation $\boldsymbol{A}$, current state $\boldsymbol{h_t}$ and input $\boldsymbol{x_t}$[105]

The same chain-like principle applies also to LSTM architectures, with the difference in the LSTM units, whom instead of using simple activation function such as sigmoid function, they are utilizing four layers that the input must pass until it reaches the output of the unit. This can be seen in (Fig. 59) where, the layers are notated with yellow color and pair-wise operations between the vectors with pink cycles. The core idea of the LSTM unit is the single straight line that is passing throughout the unit and carries the previous state, altered slightly with pair-wise operations of the current input value. This gives the power to the unit to alter the value of the state on demand, by adjusting the affect of the pair-wise operations which are controlled by a structure, called “gate”. A gate has the ability to optionally let information pass through and is made by a sigmoid function and a multiplication operation on the current signal. The sigmoid function, ranging from 0 to 1 gives the power to the gate to control how much of the information will pass through and thus, a zero value would detonate, to block the entire signal while a value of one, would allow the whole signal to pass through. Three of these gates are used in a typical LSTM unit in order to gain full control over the signal that is passing through the unit and thus regulate the state.

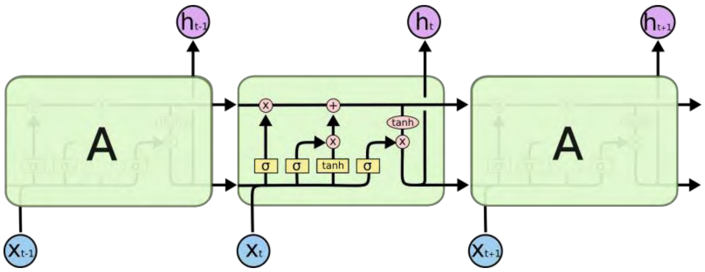


Figure 59 An LSTM unit utilizing four different layers that input must pass until it reaches the output [105]

The input signal of an LSTM unit is passing through four steps until it reaches the output of the unit. Initially the signal is passing through the first gate which decides what information are going to be left out from the unit state. It looks at the previous state h_{t-1} and the current input x_t and outputs a number between zero and one for each of the numbers of the cell previous state C_{t-1} (128)

$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f) \tag{128}$$

The second step is a combination of two functions that control what information will be stored in the current state of the unit. The first function (129), decides which values are going to get updated while the second (130) creates a vector of candidate values $C_t^{'}$ that might be added to the state.

$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i) \tag{129}$$

$$C_t^{'} = tanh(w_C[h_{t-1}, x_t] + b_C \tag{130}$$

The third stage that the signal is passing is dealing with the update of the previous state C_{t-1} into the new C_t state of the unit. This is done by multiplying the old state by the previously computed value of f_t in order to forget what information should be left out and then adding the value of $i_t \cdot C_t^{'}$ which were computed by the previous step (131).

$$C_t = f_t \cdot C_{t-1} + i_t \cdot C_t^{'} \tag{131}$$

The final output of the unit is passing through a sigmoid layer which decides what information from the current state will be output (132). The cell state passes through a tanh gate to align the values in range 1 and -1 in order to be multiplied by the output of the sigmoid operation and thus, output only the values that are decided to pass through (133). The steps are visually summarized in (Fig. 60).

$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o) \tag{132}$$

$$h_t = o_t \cdot tanh(C_t) \tag{133}$$

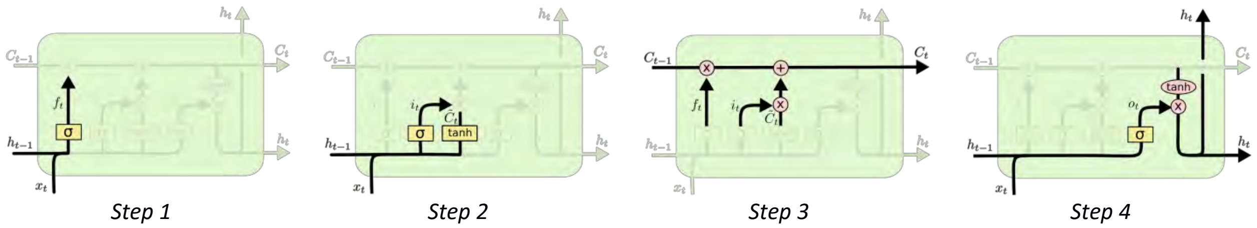


Figure 60 Step by step signal passing throughout an LSTM unit [105]

There have been numerous variations of the above typical LSTM architecture proposed throughout the years. One of the most popular ones is the addition of "peephole connections" to the original model which was introduced in [106] which adds connections to all the gated layers with the current state of the unit which can be expressed as (134), (135) and (136).

$$f_t = \sigma(w_f[C_{t-1}, h_{t-1}, x_t] + b_f) \tag{134}$$

$$i_t = \sigma(w_i[C_{t-1}, h_{t-1}, x_t] + b_i) \tag{135}$$

$$o_t = \sigma(w_o[C_t, h_{t-1}, x_t] + b_o) \tag{136}$$

Another variation employs a collaboration between the forget and input gates, step 1 and step 2 respectively. That gives the ability to the LSTM unit to forget only if something new is needed to be added. The collaboration is done by introducing a new gate between the two steps that subtracts the output of f_t with 1 and thus (131) can be rewritten as (137)

$$C_t = f_t \cdot C_{t-1} + (1 - f_t) \cdot C_t^{'} \tag{137}$$

Lastly in 2014 a new type of LSTM unit was introduced [107] that alters entirely the inner architecture of the unit. It was named as Gated Recurrent Unit (GRU) and combines forget and input units into a single gate, called "update gate". Another alternation is the merge of the cell state C_t and the hidden state h_t. The resulting model, illustrated in (Fig. 61), is a lot simpler and computationally efficient compared to the original LSTM unit.

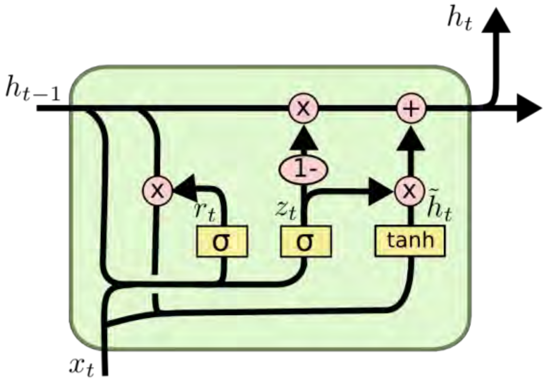


Figure 61 Visual representation of a GRU [105]

The four steps involved into the signal passing of a GRU unit are expressed in (138), (139), (140) and (141)

$$z_t = \sigma(w_z[h_{t-1}, x_t]) \tag{138}$$

$$r_t = \sigma(w_r[h_{t-1}, x_t]) \tag{139}$$

$$h_t^{'} = tanh(w[r_t \cdot h_{t-1}, x_t]) \tag{140}$$

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot h_t^{'} \tag{141}$$

6.3.2. Bidirectional Recurrent Neural Networks

To increase the capacity of information that an RNN is capable to encode, Bidirectional Recurrent Neural Networks (BRNNs) [108] was introduced. The idea behind this special kind of RNN architecture is to connect two hidden layers of opposite directions to the same output and like that create a bidirectional connection with the output layer, which can leverage the information from the previous and next states of the model (Fig. 62). That is done by connecting the neurons of the layers in two directions; one for positive time direction which resembles the forward state and one for the negative time direction to handle the opposite states. This makes the architecture capable of handling problems that require context of the input, such as, handwritten recognition where the overall performance of the model can benefit by knowing the letters before and after the current letter.

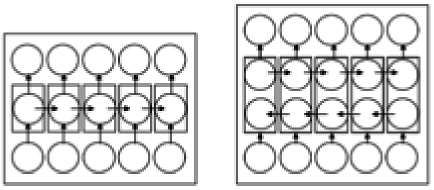


Figure 62 On the left, a typical RNN architecture. On the right, a BRNN architecture with two hidden layers [108]

Training of such architecture requires an alternation to the typical backpropagation algorithm were, initially the forward pass is done firstly on the positive and negative direction layers and secondly to the output layer. The modification for the backward pass requires initially passing from output layer, then the forward states and lastly the backward states. That is done because the weights of the input and output layers of network cannot be updated at once.

There have been numerous successful applications that leverage the power of BRNN models in fields such as, text translation[109], on-line handwritten text recognition [110], speech recognition[111],[112] combined with LSTM network and in bioinformatics sector for protein structure prediction [113],[114].

7. Experiments and Results

This section presents two case studies; the first one investigated the development of a deep convolutional neural network for robust facial makeup detection and the second a novel deep fully convolutional neural network for abnormality detection in images obtained from human gastrointestinal tract.

7.1. Deep Learning on Robust Facial Makeup Detection

Facial makeup is a well-accepted cosmetic among the female population [115]. A less-known aspect of makeup is that its use can raise security issues. It has the ability to effectively alter the overall appearance of a human face, resulting in a performance degradation of face authentication algorithms. To cope with this problem, facial makeup detection algorithms have been proposed so that the presence of makeup is detected early in a face authentication process. Only a few studies have addressed automated facial makeup detection. For this purpose, in [116] hand crafted local shape, color and texture features were used to represent various facial Regions of Interest (ROIs). Shape was encoded by means of GIST features (capturing the 'gist' of a scene) [117], color was encoded by means of up to third-order central moments, and for texture encoding the well-known Local Binary Pattern (LBP) histograms was used [118]. AdaBoost [119] and Support Vector Machine (SVM) [120] classifiers were used for the discrimination of the faces with makeup based on these features. The experiments were based on two benchmark datasets, namely the YouTube MakeUp (YMU) [121], and the Makeup In the "Wild" (MIW) datasets [116]. A face authentication method tolerant in makeup changes has been proposed in [122]. That method incorporates a makeup detection system based on facial features that are similar to those used in [116]. They include skin color tone and smoothness, quantified by means of color moments; skin texture, quantified by means of LBP histograms; and facial highlights, quantified by means of chromaticities.

In this work a deep learning approach is proposed for makeup detection based on Convolutional Neural Network (CNN) architecture. To the best of our knowledge this is the first deep learning-based facial makeup detection methodology. Its main advantages over relevant state-of-the-art methodologies include generality, independence from ROI detection algorithms, and minimal annotation requirements, in the sense that it does not require any detailed, pixel-level annotation of the images used for training. It requires only a weak, semantic annotation of these images, indicating whether the depicted face has makeup or not.

Deep learning and especially CNNs have become extremely popular over the last decade mainly because they can be trained without the need of handcrafted features. That property gives them the power of learning image features by examples instead of relying on prior knowledge from an expert. Their remarkable performance in classification of real-world images has been demonstrated in a variety of applications, especially involving large scale image databases [48].

7.1.1. Network Architecture and Training Process

Inspired by the deep neural network architecture presented in [64] a smaller scale CNN was constructed which is illustrated in (Fig. 63). This network consists of 4 learning layers, three convolutional and one fully connected, all of them followed by intermediate layers that contribute to the overall classification performance increase.

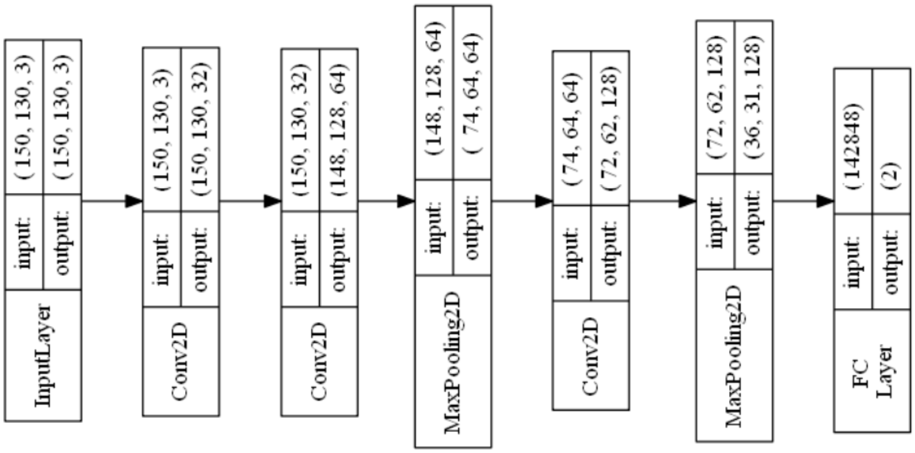


Figure 63 The proposed CNN architecture. Each layer is illustrated as a block with the number of inputs on the top left and number of outputs on top right. The width w, height h and depth d of the input and output volume of each layer is illustrated on top of each block in a vectorial form ($\boldsymbol{w}, \boldsymbol{h}, \boldsymbol{d}$).

The training of the CNN architecture was based on the mini-batch Stochastic Gradient Decent (SGD) algorithm [123]. This approach differs from the traditional error back-propagation, in that, instead of using all training examples for the estimation of the gradient in every iteration, it uses several randomly selected small batches of examples. This helps the generalization of the network while it favors a smaller memory footprint, which is important especially when the input samples are whole images [13]. Training involves three parameters namely: learning rate n, which controls the magnitude of bias and weight changes on each training epoch; weight decay λ, which is multiplied on the weights after each update, preventing the weights from becoming too large; momentum a, which adds a small percentage of the previous weight value to the next update. In order to cope with possible overfitting of the network to the training data the early stopping strategy was applied that stops the training process when the error in the validation dataset is minimized.

7.1.2. Results and Comparison

Experiments were performed to assess the capability of the proposed CNN architecture to classify facial images into two classes corresponding to faces with and without makeup. The experimental procedure is based on the datasets used in [116]. For CNN training and validation the YMU dataset (Fig. 64) was utilized which contains 604 faces of 302 women before and after makeup, while for testing MIW dataset was used which contains 154 women faces before and after makeup. In both datasets the images are 130×150 pixels in size with faces that are considered faces in the wild. The cropped faces where extracted using a Viola-Jones framework and cropped using automatically generated bounded boxes [124].

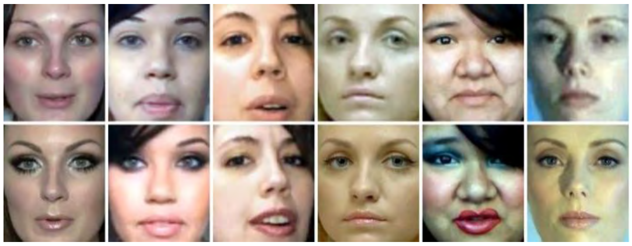


Figure 64 Sample images from YMU dataset. Six models before and after makeup is depicted. The dataset is diverse incorporating various ethnicities and poses of the face.

It should be noted that part of the images contained in both datasets are not ideal in the sense that some faces that are not properly facing the camera or parts of them are covered by hair.

The YMU dataset was randomly shuffled and split into two non-overlapping subsets, an 80% subset used for CNN training (483 images), and a 20% subset used for CNN validation (121 images). The average number of positive and negative labels were equally distributed, resulting into 242 positive (with makeup) and 241 negative (without makeup) labels in the training subset, and 60 positive and 61 negative labels in the validation subset.

For the construction of the network the popular TensorFlow framework [125] and the utility library of Keras [126] were utilized. The CUDA toolkit [127] was used in order to speed up the training process in order to perform the computations on an NVIDIA GTX-960 Graphical Processing Unit (GPU), with 1024 CUDA cores, 2GB of RAM and clock speed of 1127MHz. The overall training time for the 380,000 neurons of the CNN was 4.8 minutes and the testing time was 3.6 seconds. The parameters of the SGD used for training include a learning rate of $n = 0.01$, a standard constant decay of $\lambda = 10^{-6}$ and momentum $\alpha = 0.9$. The root mean squared logarithmic was used as a loss function (142) where n is the total number of observations of each training iteration, p_i is the prediction, and α_i is the actual response for sample i.

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(p_i + 1) - \log(\alpha_i + 1))^2} \tag{142}$$

In addition, a batch size of 32 samples was used in each mini-batch training iteration, which led to a slight improvement in the classification accuracy compared to larger batch sizes. Max pooling of size 2×2 was chosen with zero padding and a stride of 2, which shown to be a good size as it produces maximum summarization while favoring minimum information loss of the visible units. Pooling layers were placed only after the second and the third convolutional layers (Fig. 65). The use of pooling layers after every convolutional layer, which is the usual approach in CNN design, resulted in a lower accuracy (by 8.2%). On the contrary, an increase in accuracy was obtained using a dropout of 25% in the fully connected layer which increased the performance of the network by 6.8%.

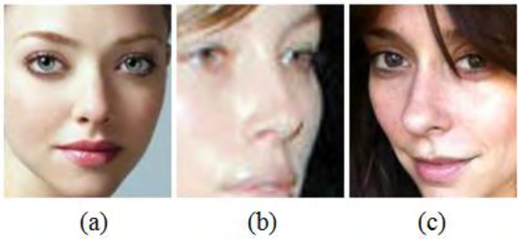


Figure 65 Misclassified images from MIW dataset. (a) Face wearing makeup classified as not wearing makeup (false negatives). (b)-(c) Faces not wearing makeup classified as wearing makeup (false positives).

The results obtained after training of the CNN on the YMU dataset and testing on the MIW dataset are summarized in Table 2, in comparison to the results presented in [116] using the same dataset. Overall, the Area Under the Receiver Operating Characteristic (ROC) curve

(AUC) is 99.26%. The best classification accuracy achieved for a 0.6 decision threshold in the output of the CNN is 98.05%. The true and false positive rates for this threshold are 98.7% and 97.4% respectively. These results indicate the robustness of the proposed methodology and its comparative advantage over the state of the art, which is achieved without the use of handcrafted features.

The faces that the best performing CNN architecture failed to classify correctly are illustrated in (Fig. 66 The face in (Fig. 66.a) was classified as not wearing makeup, possibly because of the smoothness characterizing its facial skin. The misclassification of the faces that were not wearing makeup was possibly due to different facial poses along with the poor quality of the images. Also, the face in (Fig. 66.c) includes hair that hides a part of its facial characteristics.

Table 2. Comparative results in terms of classification accuracy on the MIW test set.

	CNN (proposed)	**SVM** [116]	**AdaBoost** [116]
AUC	**99.26%**	98.63%	98.5%
Classification accuracy	**98.05%**	95.45%	92.21%

7.2. Peephole Fully Convolutional Neural Network

In this thesis a novel CNN architecture is proposed, named Peephole Fully Convolutional Neural Network (PFCNN), created in order to deal with the problem of large number of free parameters that deep neural networks tend to suffer. Furthermore because of the small number of free parameters it is able to be trained with a relatively small number of training examples in comparison with the previous architectures, such the ones proposed by [33] and [64].

7.2.1. Network Architecture

The architecture of PFCNN is inspired by the work of [106] which introduced a peephole connection in LSTM networks to give the gates a chance to react based on previous states of the model. The multi-scale design is inspired by GoogLeNet architecture [52] were high and abstract features are combined to produce a more complete description about the input volume. The architecture is based in principles of Fully Convolutional Network were the fully connected layers that are typically used as the last classification layers of a network architecture is omitted.

This reduces the number of free parameters that the network has to be trained with and thus allows the model to be trained well on datasets where the number of training examples is limited.

The proposed architecture is based on two characteristic components:

1. A Large-Medium-Small Convolutional Block (LMSCB)
2. A Peephole Connection

Large Medium Small Convolutional Block (Fig. 66) is a small convolutional neural network that is formed by five convolution layers. The first convolution layer performs a 1x1 convolution operation with N number of filters to the input space. The second, third and fourth perform convolution operation to the output of the first convolution layer, with the same number of filters, yet with different sizes. The second performs a convolution operation with filter size 8x8 in order to detect large features from the input space, while the third and fourth perform the same operation but with filter size 4x4 and 2x2 to detect smaller features that would not have been preserved by the larger convolution. The output features maps are concatenated and convolved by the fifth and last convolution layer which has a filter size 1x1. All the convolution layers have Parametric ReLU (PReLU) activations followed by a Batch Normalization[128].

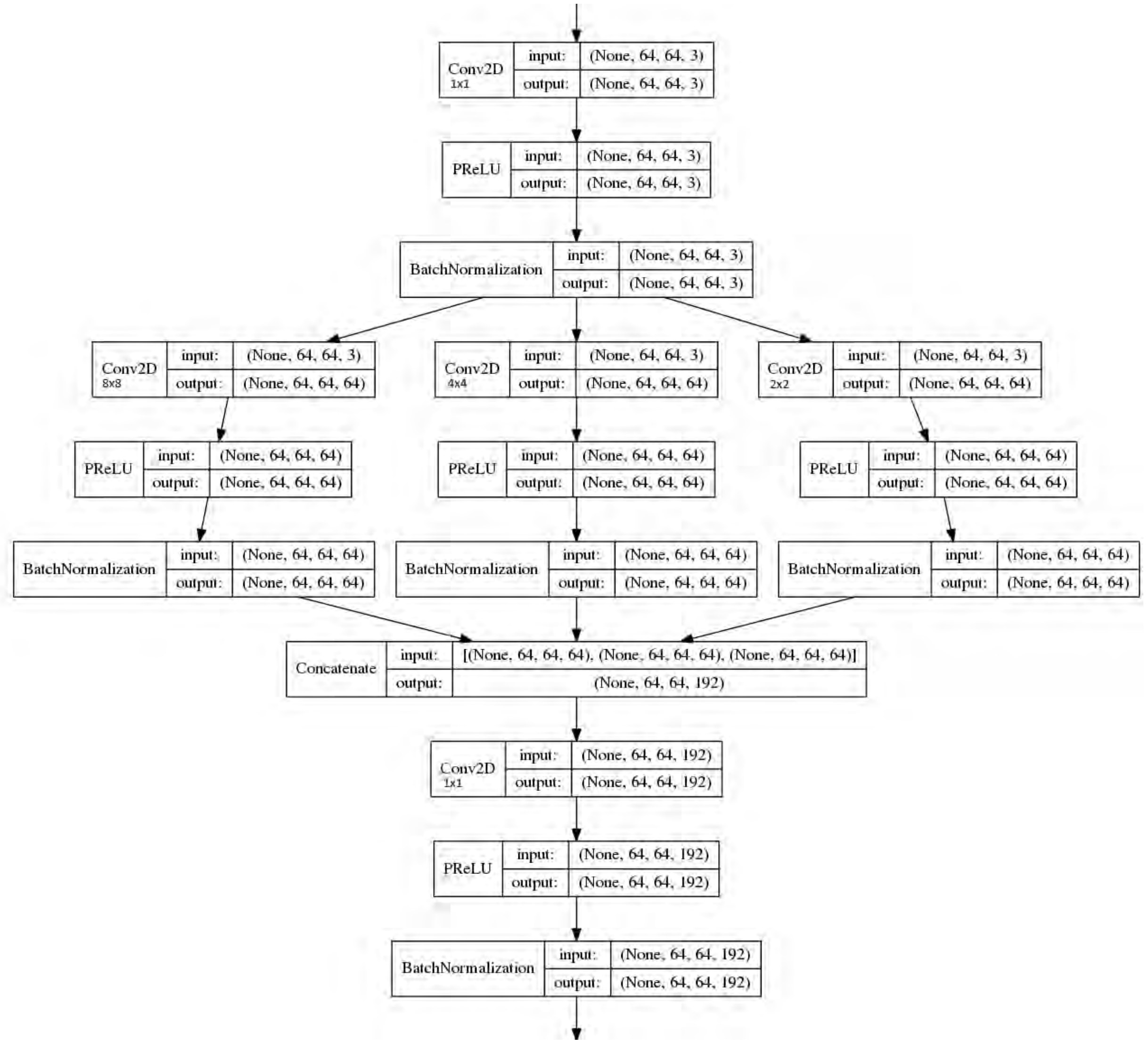

Conv2D 1x1 | input: (None, 64, 64, 3) | output: (None, 64, 64, 3)
PReLU | input: (None, 64, 64, 3) | output: (None, 64, 64, 3)
BatchNormalization | input: (None, 64, 64, 3) | output: (None, 64, 64, 3)
Conv2D 8x8 | input: (None, 64, 64, 3) | output: (None, 64, 64, 64)
Conv2D 4x4 | input: (None, 64, 64, 3) | output: (None, 64, 64, 64)
Conv2D 2x2 | input: (None, 64, 64, 3) | output: (None, 64, 64, 64)
PReLU | input: (None, 64, 64, 64) | output: (None, 64, 64, 64)
PReLU | input: (None, 64, 64, 64) | output: (None, 64, 64, 64)
PReLU | input: (None, 64, 64, 64) | output: (None, 64, 64, 64)
BatchNormalization | input: (None, 64, 64, 64) | output: (None, 64, 64, 64)
BatchNormalization | input: (None, 64, 64, 64) | output: (None, 64, 64, 64)
BatchNormalization | input: (None, 64, 64, 64) | output: (None, 64, 64, 64)
Concatenate | input: [(None, 64, 64, 64), (None, 64, 64, 64), (None, 64, 64, 64)] | output: (None, 64, 64, 192)
Conv2D 1x1 | input: (None, 64, 64, 192) | output: (None, 64, 64, 192)
PReLU | input: (None, 64, 64, 192) | output: (None, 64, 64, 192)
BatchNormalization | input: (None, 64, 64, 192) | output: (None, 64, 64, 192)

Figure 66 The LMSCB module

The inclusion of the last batch normalization helps the layer that follows it by normalizing the output of the previous layer so that it has mean close to 0 and standard deviation close to 1. This copes with the problem of carefully choosing a relatively small learning rate, as without it, the distribution of the inputs of each layer is changing during training as the parameters of the previous layers are adjusted. The inclusion of batch normalization let the architecture to omit entirely the usage of the commonly used Dropout layer, as the entire architecture did not show to suffer from overfitting. Furthermore, because of the higher learning rate the entire architecture is able to converge much faster, in comparison to the traditional approaches. PReLU was chosen over the traditional ReLU activation function, as it proved by [65] that the addition of parameter a (alpha) when the output of the activation is lower than 0 and is adjusted through gradient descent, to the original function helped to solve the saturation problem of original ReLU function (Fig. 67).

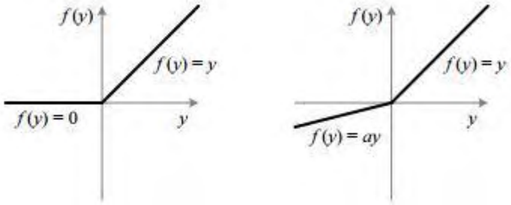


Figure 67 On the left, the ReLU activation function and on the right, the modified PReLU activation function.[65]

The second aspect that PFCNN architecture introduces is the addition of peephole connections after every LSCB module (Fig. 68). The introduction is done by connecting the input of the previous LSCB module with the output of it using the addition operation followed by an 1x1 convolution layer and Batch Normalization. This operation increases the significance of the features that have been detected by the LSCB module and also preserves the ones that have been left by the module. Peephole-like connections have also been seen in ResNet architecture [53], yet their role there is significantly different as each peephole-like connection aims to perform a small changes to the input volume by the addition of the previous convolution-ReLU-convolution operation of the residual block. The LSCB module output, after the transformation of it by the peephole connection is followed by a pooling operation. The pooling is done by a convolution layer of size 2x2 and stride 2 with PReLU activations which are normalized with a batch normalization. The usage of a convolution layer instead of a traditional max pooling was to simplify the overall network architecture, as max pooling layer can be replaced by a convolution layer of appropriate size and stride without affecting the overall predictive power of the model [56] (Fig. 69).

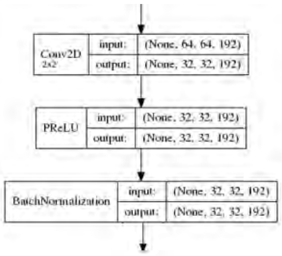

Conv2D 2x2
input: (None, 64, 64, 192)
output: (None, 32, 32, 192)
PReLU
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
BatchNormalization
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)

Figure 69 Pooling layer of PFCNN

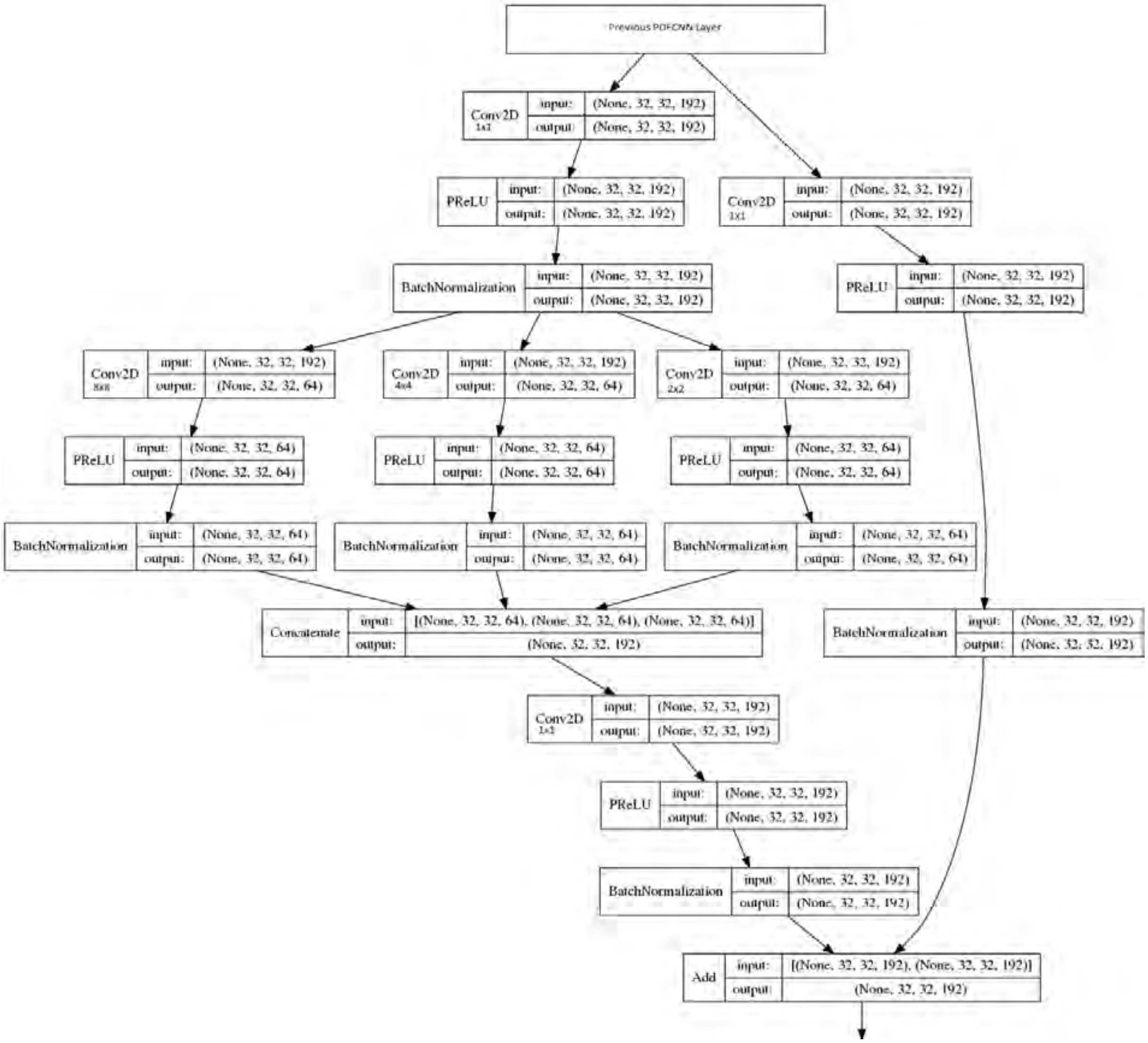

Previous POFCNN Layer
Conv2D 1x1
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
PReLU
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
Conv2D 1x1
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
BatchNormalization
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
PReLU
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
Conv2D 8x8
input: (None, 32, 32, 192)
output: (None, 32, 32, 64)
Conv2D 4x4
input: (None, 32, 32, 192)
output: (None, 32, 32, 64)
Conv2D 2x2
input: (None, 32, 32, 192)
output: (None, 32, 32, 64)
PReLU
input: (None, 32, 32, 64)
output: (None, 32, 32, 64)
PReLU
input: (None, 32, 32, 64)
output: (None, 32, 32, 64)
PReLU
input: (None, 32, 32, 64)
output: (None, 32, 32, 64)
BatchNormalization
input: (None, 32, 32, 64)
output: (None, 32, 32, 64)
BatchNormalization
input: (None, 32, 32, 64)
output: (None, 32, 32, 64)
BatchNormalization
input: (None, 32, 32, 64)
output: (None, 32, 32, 64)
Concatenate
input: [(None, 32, 32, 64), (None, 32, 32, 64), (None, 32, 32, 64)]
output: (None, 32, 32, 192)
BatchNormalization
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
Conv2D 1x1
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
PReLU
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
BatchNormalization
input: (None, 32, 32, 192)
output: (None, 32, 32, 192)
Add
input: [(None, 32, 32, 192), (None, 32, 32, 192)]
output: (None, 32, 32, 192)

Figure 68 The LSMCB module with Peephole connection to the previous input

In the experiments that follow, the architecture utilized five LSCB modules with four of them utilizing the peephole connection. From the datasets that was used in the experiments, the reduction of number of LSCB modules shown a decrease is accuracy of the trained model while the addition of more modules did not prove to be beneficial as the accuracy remained similar if not the same.

7.2.2. Evaluation On MICCAI 2015 Gastroscopy Challenge

MICCAI 2015 Gastroscopy Challenge was a challenge held by MICCAI conference in 2015 where on the same dataset, multiple participants competed having as a goal to automatically classify normal and abnormal gastroscopic images[129]. The gastroscopy challenge dataset was derived from a total of 10000 images obtained from 544 healthy volunteers and 519 volunteers having various abnormalities, such as cancer, bleeding and gastritis. The image had originally 768×576 pixels resolution and cropped to 489×409 pixels in order to be anonymized [130]. For the purpose of the challenge a subset of 698 images from 137 volunteers was selected (Fig. 70). The dataset then was split into two balanced subsets; one for training with 465 images and one for test containing 233 images.

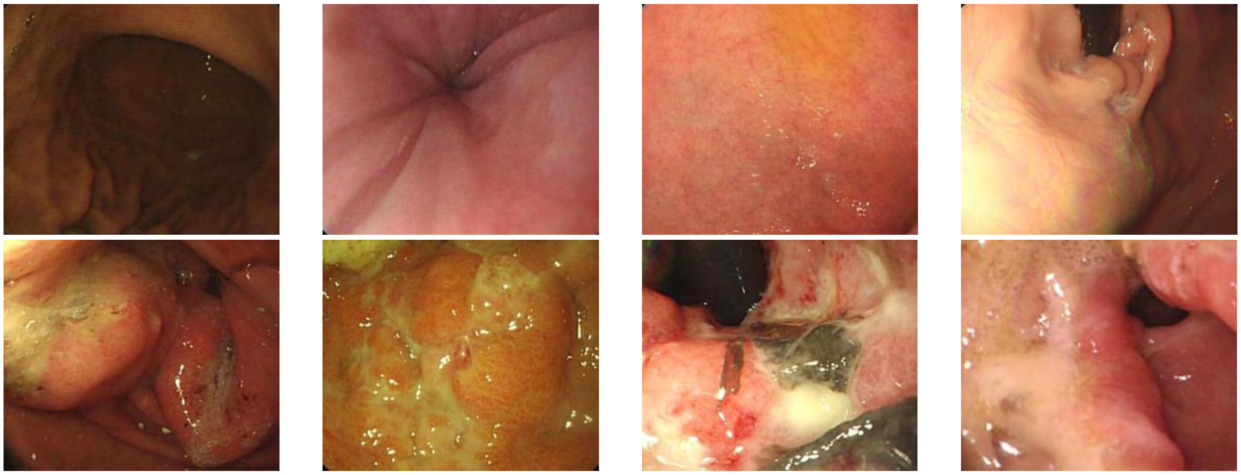


Figure 70 MICCAI 2015 Gastroscopy Challenge, sample images. First row contain normal images and the second images with abnormalities

The proposed PFCNN architecture was trained and evaluated on the same dataset provided by the challenge, using only semantically annotated images, which were resized to 63×53. No additional data pre-processing was required to utilize the proposed architecture. The network was trained using RMSProp optimizer with initial learning rate $n = 0.01$ and fuzz factor $\epsilon = 1e - 8$. A fully connected layer was used as the last layer of the network using Softmax activation acting as the classification layer of the model. The network was trained for 2000 epochs with mini-batch of size 32 samples on an NVIDIA GTX-960 Graphical Processing Unit (GPU), with 1024 CUDA cores, 2GB of RAM and clock speed of 1127MHz. The entire training process took 2 hours as the network had only 9 million parameters to be trained. The model, was evaluated on the test dataset provided by the challenge and achieved an Area Under Curve (AUC) of 98.68% (Fig. 71).

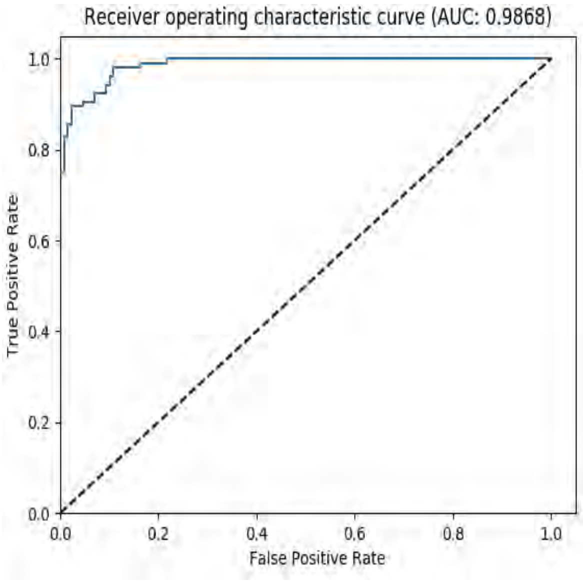


Figure 71 Receiver operation characteristic curve (ROC) with AUC 98.68%.

7.2.3. Evaluation On Wireless Capsule Endoscopy KID Dataset

Wireless Capsule Endoscopy WCE is a non-invasive way of capturing images from the gastrointestinal tract using a swallowable camera (Fig. 72) which has the size of a typical pill. While this procedure offers a lot of benefits over traditional invasive methods, such as surgery, it suffers from the lack of an automated methodology of examining the resulting video. Because of this, a lot of manual human effort is required which typically requires an individual to spend 45 to 90 minutes to review the entire video which is prone to human error as the individual becomes tired over the time.

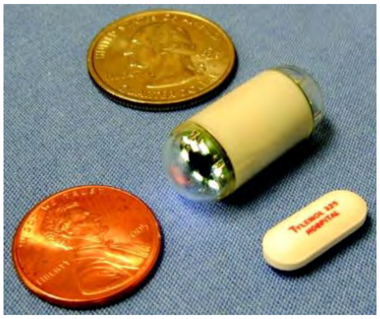


Figure 72 A typical WCE swallowable pill

KID dataset [131] is a public, open access database of semantically annotated WCE images and videos (Fig 73). The 2731 images contained in the dataset have resolution of 360×360 pixels and containing 1778 non pathogenic images and 953 images of various pathogenic conditions that are found to the entire gastrointestinal tract, such us vascular bleeding, polyps and inflammatory conditions.

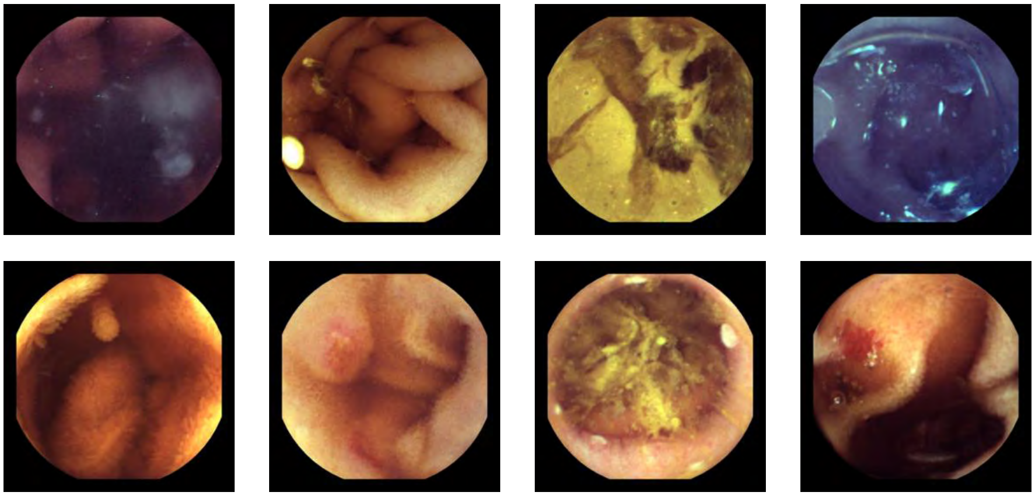


Figure 73 KID Dataset, sample images. First row contain normal images and the second images with abnormalities

In order to utilize the KID dataset to train and test PFCNN performance, the images were cropped to 320×320 to remove the excess black border that surround the actual picture. Furthermore the images were downscaled to 64×64 pixels to speed up the training process of the network, as 10-fold cross validation procedure that followed to evaluate the performance of the trained networks. On each fold of the cross validation procedure 10% of the original dataset was excluded in order to be used as a test dataset. It is important to note here that on each iteration a different subset was picked to assure the quality of the experiment. The PFCNN configuration that was used was the same as with the one presented in section 7.2.1, in order to evaluate the generality of the entire architecture. The average Receiver Operation Characteristic (ROC) curve obtained by the evaluation of the model had an Area Under Curve (AUC) of 90.07% (Fig. 74).

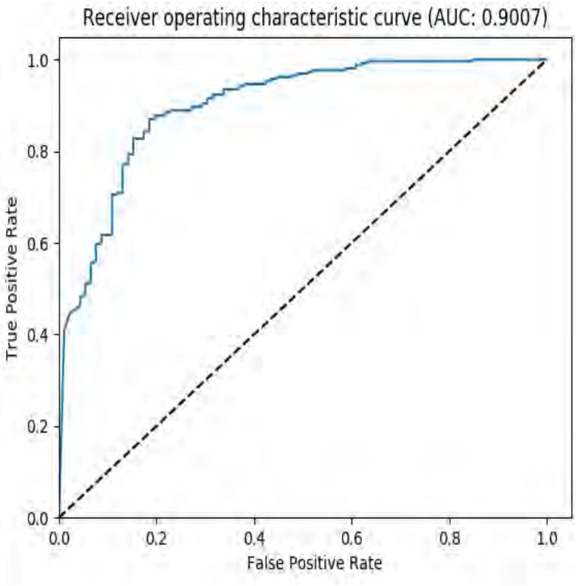


Figure 74 Receiver Operation Characteristic curve (ROC) with AUC 90.07% obtained by training and testing the PFCNN on KID dataset.

Using the same architecture on the same dataset but trained only to classify images that contain vascular bleeding, yield a much higher AUC (98.77%) and classification accuracy (94.51%), which shows that the model, if trained with images that contain the same abnormalities is able to extract better and specialized feature maps.

7.2.4. PFCNN Architecture Comparison With The Existing State-Of-The-Art Methodology

For the purpose of evaluating the performance of the purposed architecture, the experiments described in sections 7.2.1 and 7.2.2 were compared with the previous state of the art methodology presented in [132] which was also tested on the same datasets but used a manual feature extraction procedure, were color-based features were extracted from patches around salient points that were detected the images. The extracted features were then used in order to create a visual vocabulary for the image that was used to form a Bag of Words (BoW) model, which is a widely used method to model generic categories in classification and recognition problems.

A comparison of the results can be seen in Table 3 were the superiority of the proposed architecture on both AUC and classification accuracy, over the previous state-of-the-art methodology is presented.

Table 3. Comparative results between PFCNN and BoW-Based Weakly Supervised [132] approach

	PFCNN	BoW-Based Weakly Supervised [132]
AUC– MICCAI 2015	**98.68%**	94.60%
ACCURACY – MICCAI 2015	**93.14%**	89.20%
AUC – KID	**90.07%**	80.20%
ACCURACY – KID	**89.11%**	76.80%

It is important to note, that the datasets used to train the PFCNN model were not preprocessed more than resizing and cropping the input images. It is expected that applying more advanced pre-processing approaches, such as ZCA whitening and automatic augmentation, such us rotation, flipping along with higher resolution images can yield much better results. The purpose of the experiments focused on evaluating the effectiveness and generality of the model, rather optimizing it to deal with the specific datasets that was trained.

7.2.5. Evaluation On EndoVis 2017 GIANA – Polyp Detection In Colonoscopy Videos Dataset

Endoscopic Vision Challenge 2017 [133] is a challenge held by MICCAI Conference 2017 [134] on which the proposed PFCNN architecture was evaluated on a sub-challenge with name Gastrointestinal Image ANAlysis (GIANA). The GIANA challenge consists of three sub-challenges, one for polyp detection, one for polyp segmentation and one for angiodysplasia detection and segmentation. PFCNN was trained and evaluated on the polyp detection dataset provided by the polyp detection sub-challenge.

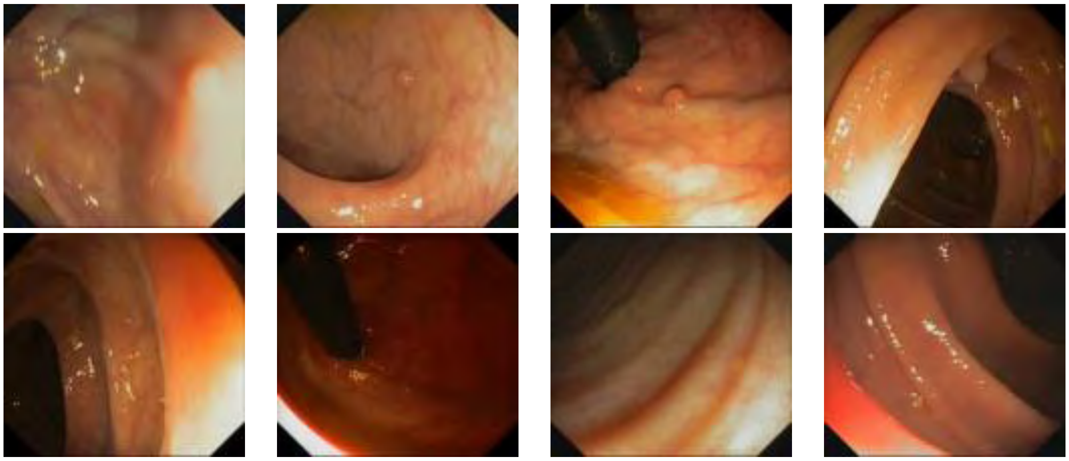


Figure 75 EndoVis 2017 – GIANA Polyp Detection challenge dataset, sample images. First row contain normal images and the second images with abnormalities.

The polyp detection sub-challenge dataset contains a training dataset of 6502 images with resolution 384×288 pixels (Fig. 75), obtained by 9 volunteer patients. The goal was to automatically classify the images as normal and abnormal, where the first stands for images that no polyp was detected and abnormal, the images that contain at least one polyp or part of it. The dataset was initially cropped in order to remove the black border that surrounded the images and then downscaled to the resolution of 120×112 pixels. No further transformation or automatic image augmentation was used. The cross-validation of the dataset was done using the technique of "leave one patient out" which means that for every trained model, the images of one patient were left out of the training dataset in order to be used as a test dataset. The maximum Area Under Curve obtained by the evaluation of the model on the entire dataset was 96.36% (Fig. 76) and lowest 90.82% with average classification accuracy of 89.45%.

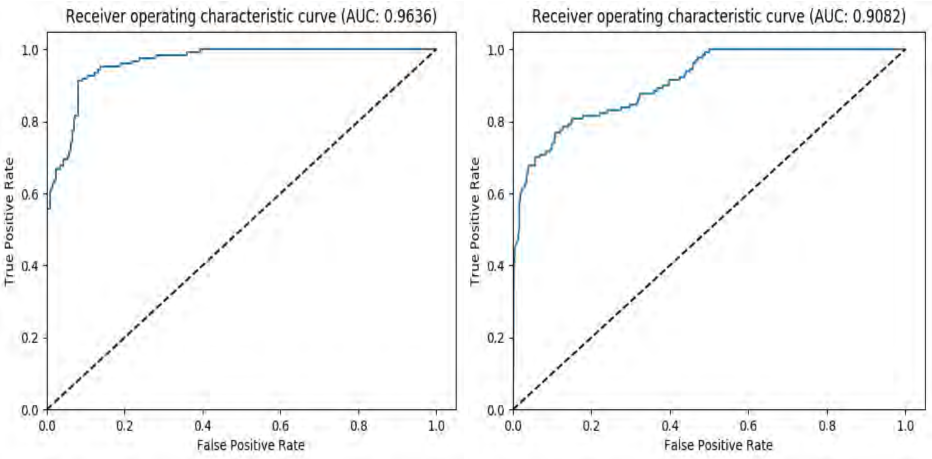


Figure 76 Receiver Operation Characteristic curve (ROC) with maximum (left) AUC 96.36% and lowest (right) AUC 90.86% obtained by training and testing the PFCNN on EndoVis 2017 – GIANA Polyp Detection challenge dataset.

8. Conclusions and Future Work

In this thesis the subject of artificial neural networks was presented. An introductory and historical overview was presented, including the basics of how the artificial neurons work and how, combining them together, layers can be formed in order to simulate from simple functions, like the logical OR gate to more complex ones, such us hand written number recognition. Furthermore, it has been shown that moving into more complex architectures and by increasing both the number of neurons on each layer and network depth, the prediction power of the model can be significantly increased. The thesis also examined the problems that arise on training such deeper networks and presented multiple methodologies that can be applied in the training process to overcome them. A special kind of deep neural network architecture was presented called CNNs, along with an architectural break down of each layer that forms them, in order to provide a deep understanding of how they function and why they perform better over the other traditional approaches that can be found in the field of computer vision. Furthermore a review of the most significant CNN architectures was presented along with the advantages that they brought into the field. An application of CNNs was presented to tackle the problem of facial makeup detection, showing the superiority of this approach over the traditional handcrafted feature extraction and classification methodologies.

More importantly a novel CNN named as "Peephole Fully Convolutional Neural Network" (PFCNN) was proposed. Extensive experiments were performed on datasets that contain images from the gastrointestinal tract of humans. The results showed the superiority of the proposed deep learning based model over the previous state-of-the-art approach.

Overall the following conclusions can be drawn:

- In recent years the attention has been drowned towards deep learning
- Deep neural networks, especially in computer vision problems, outperform the traditional hand crafted feature and classification approaches
- Over the course of the last years, network depth has vastly increased, as deeper networks tend to perform better, i.e AlexNet [33] which was developed 2012 had 5 convolution layers, while ResNet [53] of 2015 had 152 layers.

- Applying deep learning on facial makeup discover outperforms the state-of-the-art traditional hand crafted feature approach [116]
- Results from applying the presented PFCNN architecture on different dataset of images from the human gastrointestinal tract, shows that PFCNN outperforms the previous state-of-the-art methodology [132]

The usage of facial makeup affects the performance of facial recognition algorithms, as demonstrated in [121]. Therefore accurate prediction of the presence of makeup on human faces can be an important pre-processing stage for face authentication systems. A novel approach to facial makeup detection was presented based on a weekly supervised CNN. The proposed approach eliminates the need of handcrafted features as it is capable to extract the appropriate features automatically by providing only semantically annotated images. The experiments performed using benchmark datasets showed that it outperforms state-of-the-art methods that are based on handcrafted features. Future work includes further experimentation and optimization of the proposed methodology using larger and more diverse datasets.

The generality of the proposed PFCNN architecture is promising as it is expected to perform better when it is optimized for the specific dataset that is trained with. This generality will be examined in the future as the model will be used in a wider spectrum of problems, such as, but not limited, the ImageNet dataset.

The field of artificial neural networks, despite the fact that they exist for more than 50 years, is still under heavy research. More and more attention is drowning towards them, especially after the recent achievements that were obtained in the field of image classification, which shown that, even simple yet deep neural networks are powerful enough to achieve a remarkable performance. Unfortunately it is still unclear how deep neural networks function and why, and in some cases, the training of them is becoming challenging. This, along with the progress that is done in understanding the human brain, shows that the subject of neural networks is still widely open, waiting for exploration and thus, promising for even better performance in the future.

9. Bibliography

[1] F. Rosenblatt, “Principles of neurodynamics. perceptrons and the theory of brain mechanisms,” 1961.

[2] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[3] M. Minsky and S. Papert, “Perceptrons.” M.I.T. Press, 1969.

[4] M. Kubat, “Neural networks: a comprehensive foundation by Simon Haykin, Macmillan, 1994, ISBN 0-02-352781-7.,” *Knowl. Eng. Rev.*, vol. 13, no. 4, p. S0269888998214044, Feb. 1999.

[5] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Math. Control. Signals, Syst.*, vol. 2, no. 4, pp. 303–314, Dec. 1989.

[6] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1989.

[7] A. M. Turing, “On computable numbers, with an application to the Entscheidungs problem,” *Proc. London Math. Soc.*, vol. 2, no. 1, pp. 230–265, 1937.

[8] N. Murata, S. Yoshizawa, and S. Amari, “Network information criterion-determining the number of hidden units for an artificial neural network model,” *IEEE Trans. Neural Networks*, vol. 5, no. 6, pp. 865–872, 1994.

[9] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, “Greedy layer-wise training of deep networks,” in *Advances in neural information processing systems*, 2007, pp. 153–160.

[10] A. J. Jones, “Genetic algorithms and their applications to the design of neural networks,” *Neural Comput. Appl.*, vol. 1, no. 1, pp. 32–45, Mar. 1993.

[11] D. Montana, “Neural network weight selection using genetic algorithms,” *Intell. Hybrid Syst.*, 1995.

[12] H. Robbins and S. Monro, “A Stochastic Approximation Method,” *Ann. Math. Stat.*, vol. 22, no. 3, pp. 400–407, Sep. 1951.

[13] M. Moreira and E. Fiesler, “IDIAP Neural Networks with Adaptive Learning Rate and Momentum Terms,” 1995.

[14] C. Darken, J. Chang, and J. Moody, “Learning rate schedules for faster stochastic gradient search,” in *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, pp. 3–12.

[15] C. T. Kelley, *Iterative methods for optimization*. SIAM, 1999.

[16] E. W. Weisstein, “Hessian,” 2012. [Online]. Available: http://mathworld.wolfram.com/Hessian.html.

[17] R. S. Sutton, “Two problems with backpropagation and other steepest-descent learning procedures for networks,” in *Proc. 8th annual conf. cognitive science society,* 1986, pp. 823–831.

[18] S. Ruder, “An overview of gradient descent optimization algorithms,” Sep. 2016.

[19] N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural Networks,* vol. 12, no. 1, pp. 145–151, Jan. 1999.

[20] Y. Nesterov, “A method of solving a convex programming problem with convergence rate O (1/k2).”

[21] G. Hinton, N. Srivastava, and K. Swersky, “Neural Networks for Machine Learning Lecture 6a Overview of mini--- batch gradient descent.”

[22] J. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” *J. Mach. Learn. Res.,* vol. 12, no. Jul, pp. 2121–2159, 2011.

[23] J. Dean *et al.*, “Large Scale Distributed Deep Networks.” pp. 1223–1231, 2012.

[24] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation.,” in *EMNLP*, 2014, vol. 14, pp. 1532–1543.

[25] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” Dec. 2012.

[26] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Dec. 2014.

[27] T. Dozat, “Incorporating nesterov momentum into adam,” 2016.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature,* vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[29] M. A. Nielsen, “Neural Networks and Deep Learning.” Determination Press, 2015.

[30] F. Girosi, M. Jones, and T. Poggio, “Regularization Theory and Neural Networks Architectures,” *Neural Comput.,* vol. 7, no. 2, pp. 219–269, Mar. 1995.

[31] “. Early stopping the ANN training to avoid overfitting - Figure 2 of 2.” [Online]. Available: https://www.researchgate.net/figure/223790695_fig2_Figure-2-Early-stopping-the-ANN-training-to-avoid-overfitting. [Accessed: 09-Jul-2017].

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *J. Mach. Learn. Res.,* vol. 15, pp. 1929–1958, 2014.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks.” pp. 1097–1105, 2012.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[35] “affNIST,” 2013. [Online]. Available: http://www.cs.toronto.edu/~tijmen/affNIST/. [Accessed: 11-Jul-2017].

[36] K. Pearson, “LIII. On lines and planes of closest fit to systems of points in space,” *Philos. Mag. Ser. 6*, vol. 2, no. 11, pp. 559–572, Nov. 1901.

[37] J. Mohamad-Saleh and B. Hoyle, “Improved neural network performance using principal component analysis on Matlab,” *Comput. internet …*, 2008.

[38] “Unsupervised Feature Learning and Deep Learning Tutorial,” *Stanford University*, 2016. [Online]. Available: http://ufldl.stanford.edu/tutorial/unsupervised/PCAWhitening/. [Accessed: 13-Jul-2017].

[39] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” 2009.

[40] Chris McCormick, “Deep Learning Tutorial - PCA and Whitening · Chris McCormick,” 2014. [Online]. Available: http://mccormickml.com/2014/06/03/deep-learning-tutorial-pca-and-whitening/. [Accessed: 14-Jul-2017].

[41] J. Bergstra, Y. Bengio, J. Bergstra, and Y. Bengio, *Journal of machine learning research : JMLR.*, vol. 13, no. 1. MIT Press, 2001.

[42] J. Snoek, H. Larochelle, and R. P. Adams, “Practical Bayesian Optimization of Machine Learning Algorithms.” pp. 2951–2959, 2012.

[43] D. H. Hubel and T. N. Wiesel, “Receptive fields and functional architecture of monkey striate cortex.,” *J. Physiol.*, vol. 195, no. 1, pp. 215–43, Mar. 1968.

[44] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biol. Cybern.*, vol. 36, no. 4, pp. 193–202, Apr. 1980.

[45] S. Behnke, *Hierarchical Neural Networks for Image Interpretation*, vol. 2766. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[46] P. Y. Simard, D. Steinkraus, and J. Platt, “Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis.” 01-Aug-2003.

[47] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, p. 40, Mar. 2008.

[48] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[49] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The Pascal Visual Object Classes (VOC) Challenge,” *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, Jun. 2010.

[50] World4jason, “AlexNet · research-log,” 2012. [Online]. Available: https://world4jason.gitbooks.io/research-log/content/deepLearning/CNN/Model & ImgNet/alexnet/alexnet.html. [Accessed: 16-Jul-2017].

[51] Aarshay Jain, “Deep Learning for Computer Vision - Introduction to Convolution Neural Networks,” 2016. [Online]. Available: https://www.analyticsvidhya.com/blog/2016/04/deep-learning-computer-vision-introduction-convolution-neural-networks/. [Accessed: 17-Jul-2017].

[52] C. Szegedy *et al.*, “Going Deeper with Convolutions,” pp. 1–9, 2014.

[53] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” Dec. 2015.

[54] “CS231n Convolutional Neural Networks for Visual Recognition,” 2015. [Online]. Available: http://cs231n.github.io/convolutional-networks/#pool. [Accessed: 18-Jul-2017].

[55] Nouroz Rahman, “What is the benefit of using average pooling rather than max pooling? - Quora,” 2017. [Online]. Available: https://www.quora.com/What-is-the-benefit-of-using-average-pooling-rather-than-max-pooling. [Accessed: 18-Jul-2017].

[56] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, “Striving for Simplicity: The All Convolutional Net,” Dec. 2014.

[57] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes,” Dec. 2013.

[58] A. Rajaraman and J. D. Ullman, “Mining of Massive Datasets,” *Lect. Notes Stanford CS345A Web Min.*, vol. 67, p. 328, 2011.

[59] J. Weston and C. Watkins, “Multi-class support vector machines,” 1998.

[60] R. Rifkin and A. Klautau, “In Defense of One-Vs-All Classification,” *J. Mach. Learn. Res.*, vol. 5, no. Jan, pp. 101–141, 2004.

[61] Andrej Karpathy, “CS231n Convolutional Neural Networks for Visual Recognition.” [Online]. Available: http://cs231n.github.io/linear-classify/#softmax. [Accessed: 22-Jul-2017].

[62] D. Pathak, E. Shelhamer, J. Long, and T. Darrell, “Fully convolutional multi-class multiple instance learning,” *arXiv Prepr. arXiv1412.7144,* 2014.

[63] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440.

[64] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” Sep. 2014.

[65] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” Feb. 2015.

[66] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” Nov. 2013.

[67] Adit Deshpande, “The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3) – Adit Deshpande – CS Undergrad at UCLA (’19).” [Online]. Available: https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html. [Accessed: 24-Jul-2017].

[68] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” Nov. 2013.

[69] R. Girshick, “Fast R-CNN,” Apr. 2015.

[70] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” Jun. 2015.

[71] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders, “Selective Search for Object Recognition,” *Int. J. Comput. Vis.*, vol. 104, 2013.

[72] I. J. Goodfellow *et al.*, “Adversarial Networks,” Jun. 2014.

[73] C. Szegedy *et al.*, “Intriguing properties of neural networks,” Dec. 2013.

[74] R. Soumith, Emily, Arthur, “The Eyescream Project,” 2015. [Online]. Available: http://soumith.ch/eyescream/. [Accessed: 25-Jul-2017].

[75] A. Karpathy and L. Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions,” Dec. 2014.

[76] A. Karpathy, A. Joulin, and L. Fei-Fei, “Deep Fragment Embeddings for Bidirectional Image Sentence Mapping,” Jun. 2014.

[77] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, “Spatial Transformer Networks,” *Nips*, pp. 1–14, Jun. 2015.

[78] P. Gallinari, Y. Lecun, S. Thiria, and F. F. Soulie, “Memoires associatives distribuees: Une comparaison (Distributed associative memories: A comparison).” Cesta-Afcet, 1987.

[79] H. Bourlard and Y. Kamp, “Auto-association by multilayer perceptrons and singular value decomposition.,” *Biol. Cybern.*, vol. 59, no. 4–5, pp. 291–4, 1988.

[80] G. E. Hinton and R. S. Zemel, “Autoencoders, Minimum Description Length and Helmholtz Free Energy.” pp. 3–10, 1994.

[81] G. E. Hinton, G. E. Hinton, and J. L. Mcclelland, “Learning Representations by Recirculation,” *Proc. IEEE Conf. NEURAL Inf. Process. Syst.*, 1988.

[82] M. Ranzato, F. J. Huang, Y.-L. Boureau, and Y. LeCun, “Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition,” in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, 2007, pp. 1–8.

[83] M. Ranzato, Y. Boureau, and Y. L. Cun, “Sparse Feature Learning for Deep Belief Networks.” pp. 1185–1192, 2008.

[84] X. Glorot, A. Bordes, and Y. Bengio, “Deep Sparse Rectifier Neural Networks.” pp. 315–323, 14-Jun-2011.

[85] G. Alain and Y. Bengio, “What Regularized Auto-Encoders Learn from the Data Generating Distribution,” Nov. 2012.

[86] Y. Bengio, “Deep learning of representations: Looking forward,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013, vol. 7978 LNAI, pp. 1–37.

[87] A. Hyvärinen, “Estimation of non-normalized statistical models by score matching,” *J. Mach. Learn. Res.*, vol. 6, no. Apr, pp. 695–708, 2006.

[88] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[89] A. Vincent, J. Herman, R. Schulick, R. H. Hruban, and M. Goggins, “Pancreatic cancer,” *Lancet*, vol. 378, no. 9791, pp. 607–620, Aug. 2011.

[90] D. P. Kingma and Y. LeCun, “Regularized estimation of image statistics by Score Matching.” 2010.

[91] K. Swersky, D. Buchman, N. D. Freitas, B. M. Marlin, and others, “On autoencoders and score matching for energy based models,” in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 1201–1208.

[92] Y. Bengio and O. Delalleau, “Justifying and Generalizing Contrastive Divergence,” *Neural Comput.*, vol. 21, no. 6, pp. 1601–1621, Jun. 2009.

[93] H. Kamyshanska and R. Memisevic, “The Potential Energy of an Autoencoder,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 6, pp. 1261–1273, Jun. 2015.

[94] E. W. Weisstein, “Jacobian,” 2012. [Online]. Available: http://mathworld.wolfram.com/Jacobian.html.

[95] T. K. Kohonen, *Self-Organizing Maps*, vol. 30. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[96] “Kohonen Network - Background Information,” 2012. [Online]. Available: http://www.lohninger.com/helpcsuite/kohonen_network_-_background_information.htm. [Accessed: 02-Aug-2017].

[97] A. Jae-Wook and S. Sue Yeon, “SOM Tutorial.” [Online]. Available: http://www.pitt.edu/~is2470pb/Spring05/FinalProjects/Group1a/tutorial/som.html. [Accessed: 02-Aug-2017].

[98] H. Sak, A. Senior, and F. Beaufays, “Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition,” Feb. 2014.

[99] X. Li and X. Wu, “Constructing Long Short-Term Memory based Deep Recurrent Neural Networks for Large Vocabulary Speech Recognition,” Oct. 2014.

[100] G. Lewis, “Sentence Correction using Recurrent Neural Networks,” 2016.

[101] Jonamjar, “Recurrent Neural Networks from Scratch - Jonamjar,” 2016. [Online]. Available: http://jonamjar.com/2016/12/22/recurrent-neural-networks/. [Accessed: 05-Aug-2017].

[102] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[103] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber, “A Novel Connectionist System for Unconstrained Handwriting Recognition,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 5, pp. 855–868, May 2009.

[104] “ICDAR2009,” 2009. [Online]. Available: http://www.cvc.uab.es/icdar2009/. [Accessed: 06-Aug-2017].

[105] C. Olah, “Understanding LSTM Networks,” 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed: 06-Aug-2017].

[106] F. A. Gers and J. Schmidhuber, “Recurrent nets that time and count,” in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 2000, pp. 189–194 vol.3.

[107] K. Cho *et al.*, “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation,” Jun. 2014.

[108] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, 1997.

[109] M. Sundermeyer, T. Alkhouli, J. Wuebker, and H. Ney, “Translation Modeling with Bidirectional Recurrent Neural Networks,” *Proc. 2014 Conf. Empir. Methods Nat. Lang. Process.*, pp. 14–25, 2014.

[110] M. Liwicki, A. Graves, H. Bunke, and J. Schmidhuber, “A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks,” in *Proceedings - 9th Int. Conf. on Document Analysis and Recognition*, 2007, vol. 1, pp. 367–371.

[111] A. Graves, S. Fernández, and J. Schmidhuber, “Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition,” pp. 799–804, 2005.

[112] A. Graves, N. Jaitly, and A. R. Mohamed, “Hybrid speech recognition with Deep Bidirectional LSTM,” in *2013 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2013 - Proceedings*, 2013, pp. 273–278.

[113] P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri, “Exploiting the past and the

future in protein secondary structure prediction.," *Bioinformatics*, vol. 15, no. 11, pp. 937–946, Nov. 1999.

[114] G. Pollastri and A. McLysaght, "Porter: A new, accurate server for protein secondary structure prediction," *Bioinformatics*, vol. 21, no. 8, pp. 1719–1720, Apr. 2005.

[115] A. Łopaciuk and M. Łoboda, "Global Beauty Industry Trends in the 21st Century," *Knowl. Manag. Innov. Knowl. Learn.*, pp. 1079–1087, 2013.

[116] C. Chen, A. Dantcheva, and A. Ross, "Automatic facial makeup detection with application in face recognition," in *2013 International Conference on Biometrics (ICB)*, 2013, pp. 1–8.

[117] C. Siagian and L. Itti, "Rapid Biologically-Inspired Scene Classification Using Features Shared with Visual Attention," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 2, pp. 300–312, Feb. 2007.

[118] T. Ahonen, A. Hadid, and M. Pietikainen, "Face Description with Local Binary Patterns: Application to Face Recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 12, pp. 2037–2041, Dec. 2006.

[119] Y. Freund and R. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," *Comput. Learn. theory*, vol. 55, pp. 119–139, 1995.

[120] C.~Cortes, C.~Cortes, V.~Vapnik, and V.~Vapnik, "Support Vector Networks," *Mach. Learn.*, vol. 20, no. 3, p. 273~-~297, Sep. 1995.

[121] A. Dantcheva, C. Chen, and A. Ross, "Can facial cosmetics affect the matching accuracy of face recognition systems?," in *2012 IEEE 5th International Conference on Biometrics: Theory, Applications and Systems, BTAS 2012*, 2012, pp. 391–398.

[122] Guodong Guo, Lingyun Wen, and Shuicheng Yan, "Face Authentication With Makeup Changes," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 24, no. 5, pp. 814–825, May 2014.

[123] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[124] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," *Comput. Vis. Pattern Recognit.*, vol. 1, p. I--511--I--518, 2001.

[125] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems."

[126] F. Chollet, "Keras." GitHub, 2015.

[127] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *AMC Queue*, vol. 6, no. April, pp. 40–53, 2008.

[128] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Feb. 2015.

[129] "EndoVisSub-Abnormal - Home." [Online]. Available: https://endovissub-abnormal.grand-challenge.org/. [Accessed: 01-Oct-2017].

[130] Y. Cong, S. Wang, J. Liu, J. Cao, Y. Yang, and J. Luo, "Deep sparse feature selection for computer aided endoscopy diagnosis," *Pattern Recognit.*, vol. 48, no. 3, pp. 907–917, Mar. 2015.

[131] D. K. Iakovidis and A. Koulaouzidis, "KID: A Capsule Endoscopy Database for Medical Decision Support," *United Eur. Gastroenterol. J.*, vol. 2 (Supplem, 2015.

[132] M. Vasilakakis, D. K. Iakovidis, E. Spyrou, and A. Koulaouzidis, "Weakly-supervised lesion detection in video capsule endoscopy based on a bag-of-colour features model," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10170 LNCS, pp. 96–103.

[133] "EndoVisSub2017-GIANA - Home," 2017. [Online]. Available: https://endovissub2017-giana.grand-challenge.org/. [Accessed: 04-Oct-2017].

[134] "MICCAI 2017 Conference." [Online]. Available: http://www.miccai2017.org/. [Accessed: 04-Oct-2017].