NATIONAL AND KAPODISTRIAN UNIVERCITY OF ATHENS
PHYSICS DEPARTMENT
DEPARTMENT OF ELECTRONICS, COMPUTERS, TELECOMMUNICATIONS AND CONTROL

Postgraduate Specialization Degree on Control and Computing

Thesis

Artificial Intelligence with Reinforcement Learning on Video Games

Mavrothalassitis Kyriakos (RN: 2015513)

Athens
August 2019

Supervisor:

Dionysios Reisis, Associate Professor

Evaluation Committee:

Dionysios Reisis, Associate Professor

Ektoras Nistazakis, Associate Professor

Dr. Nikolaos Vlassopoulos, Research Associate

Contents

1 Introduction

If someone wanted to describe our modern world in a single word, he couldn't find a more appropriate word other than *data*. According to recent estimates, 2.5 quintillion (10^{18}) bytes are being produced daily and that pace is only accelerating with the growth of social media and the Internet of Things (IoT). Unfortunately, most of this huge amount of information cannot be used by humans, either because the data are not in a typical form needed by standard analytical methods, or it's too vast for a human mind even to comprehend.

Fortunately, modern computers consist of substantial amount of working memory and storage capacity and therefore are more than capable of processing, learning from and draw actionable insights out of this kind of big data. This process is called **Machine Learning** and from Google's massive server farms used for Google Services to the voice assistant of the very last smart-phone in the world, we rely on it to power our civilization, sometimes without even knowing it! Some other use cases of machine learning include spam email filtering, product recommendations for customers, detecting and diagnosing medical diseases, self-driving vehicles and the list goes on and on.

A more formal definition of Machine Learning can be phrased like this:

> ***Machine Learning*** *is the application of computer self-learning algorithms on big data in such a way that we can turn this data into knowledge, like finding data patterns or make predictions about future events.*

1.1 The Three Types of Machine Learning

Big data of the modern era comes in two flavours: *structured* and *unstructured* data. Instead of having humans process, manipulate and analyse manually that amount of data in order to derive rules and build models from it, Machine Learning consists a much more efficient alternative for implementing and improving the performance of predictive models, in order to achieve the best possible data-driven decisions[1].

Generally, there are three types of Machine Learning: **Supervised Learning**, **Unsupervised Learning** and **Reinforcement Learning**. There are fundamental differences between them so that each of them is being applied on different types of problems. A visual representation of the three types of Machine Learning along with their subcategories and some practical

[1] Conceptually, this section is based on chapter 1 of the book [1]

application examples can be seen in figure 1[2].

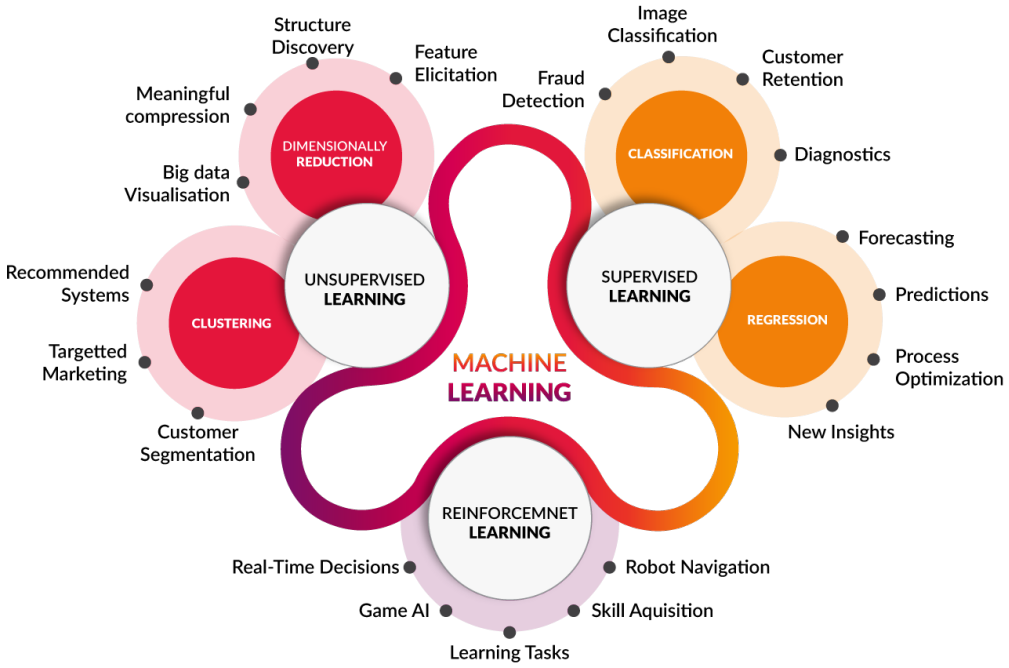

Figure 1: The Three Types of Machine Learning

Brief descriptions of the three types of Machine Learning and their subcategories follow in the next subsections.

1.1.1 Supervised Learning

In the case of Supervised Learning the objective is to make a model learn from training data, in order to be able to make predictions about other unseen data of the same type.

If the training data is a set of samples from a *categorized* (class-labelled) data set for which their categories (discrete class labels) are already known, the predictive model consist a *classifier*, hence this Machine Learning method is called **Classification**. The output signal of such a model is the class labels that the model predicts for each new unseen sample. In the case of a *binary* classification task, the number of categories is two, otherwise for more than two categories we have the case of a *multi-class* classification task. Typical examples of these two classification task types are spam email filters (spam, not-spam) and hand-written character recognition systems, respectively. The whole concept is visually described in figure 2[3].

[2]Image taken from: `https://towardsdatascience.com/coding-deep-learning-for-beginners-types-of-machine-learning-b9e651e1ed9d`

[3]Image taken from: `http://ogrisel.github.io/scikit-learn.org/sklearn-tutorial/tutorial/text_analytics/general_concepts.html`

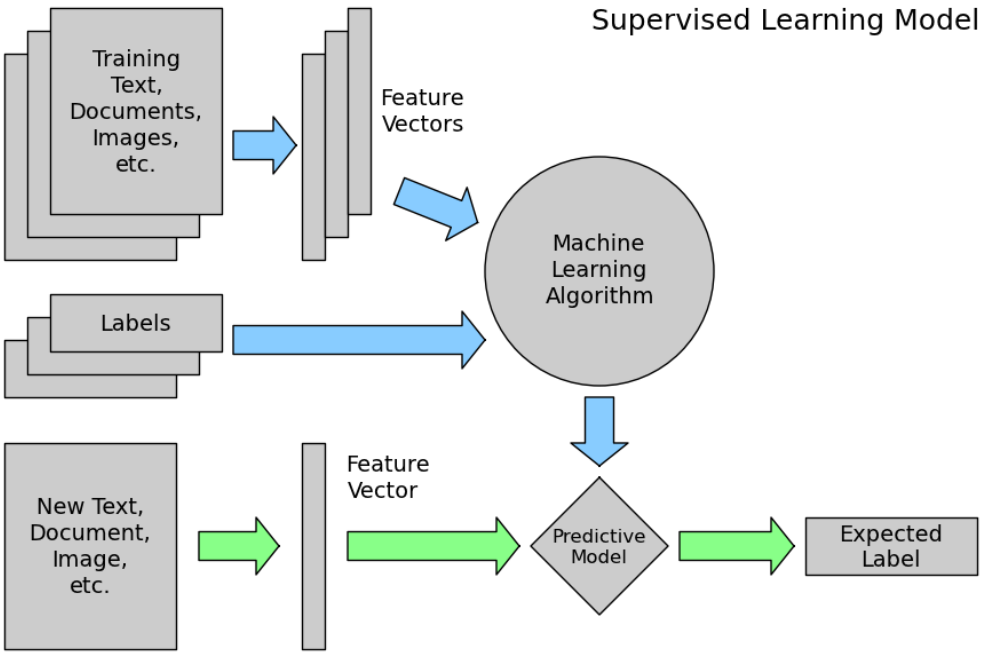

Figure 2: Supervised Learning - Classification

On the other hand, if the training data is a set of samples, each of which depends on a number of real-valued *predictor variables* and the output signal is a real-valued *response variable*, the machine learning method is called **Regression**. In this case, by training the model we try to find a relationship between those predictor variables, which allow us to predict the output variable. Typically, there are three types of Regression: **Linear Regression**, **Non-Linear Regression** and **Logistic Regression**.

In the first case we fit a straight line to the data that minimizes the distance (usually the average squared distance) between the sample points and the line. Then we use the intercept and slope learned from this data to predict the outcome variable of new data as shown in figure 3[4]. We do the same for the Non-Linear Regression, but in this case the fitted line is not a straight one. A common model of this case is the **Polynomial Regression**. Predictor variables can be continuous or binary. In Logistic Regression the response variable is binary in nature. In this case the line divides the data points into two big categories, hence this method is usually used for classification tasks.

[4]Image taken from: `https://medium.com/simple-ai/linear-regression-intro-to-machine-learning-6-6e320dbdaf06`

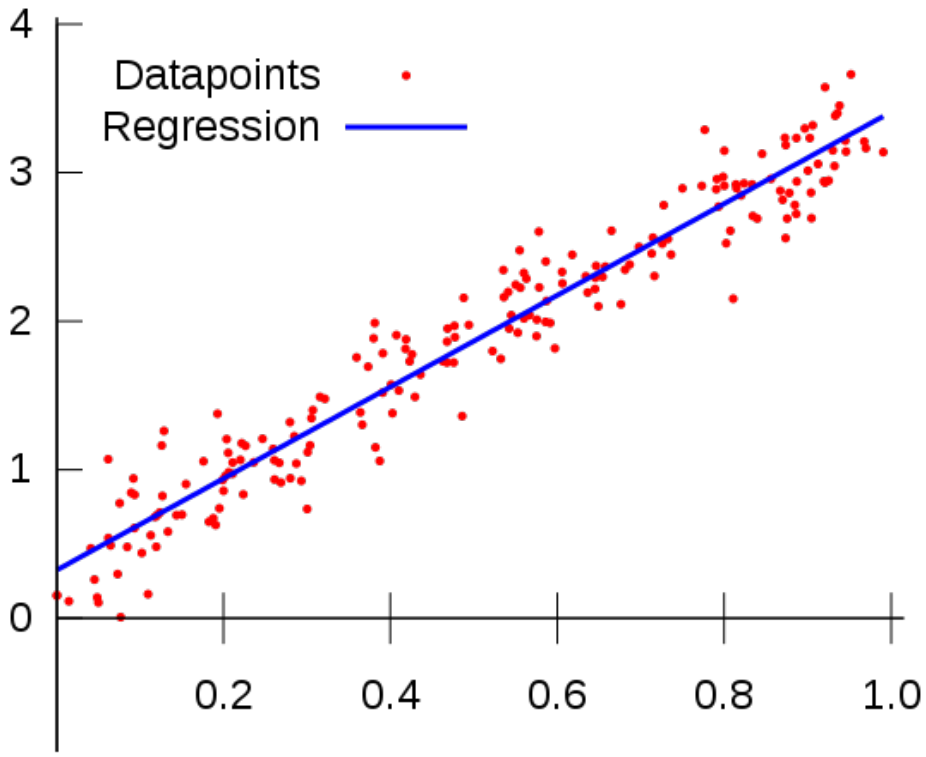

Figure 3: Supervised Learning - Linear Regression

1.1.2 Unsupervised Learning

In Unsupervised Learning, the data we are dealing with are *unstructured* and *unlabelled*. By using an Unsupervised Learning method, we can extract the hidden structure of the data and so retrieve meaningful information without the need of training the model with sample data. The most common applications of Unsupervised Learning are: **Clustering** and **Dimensionality Reduction**.

Clustering is a data analysis model that organizes an unstructured data set into meaningful subgroups (*clusters*), without having any prior information about their cluster memberships. Each derived cluster groups data points that share a degree of similarity with each other but look quite dissimilar compared to the data points that belong to another cluster. Clustering is a great technique for deriving information about the structure and the relationships between the members of the data set, as shown in figure 4[5].

In real life applications data sets have high dimensionality, in other words data sets come with a high number of predictor variables. For the modern big data sets, this fact rises challenges sometimes because the storage capacity and the computational capability of computers

[5]Image taken from: `https://chatbotsmagazine.com/lets-know-supervised-and-unsupervised-in-an-easy-way-9168363e06ab`

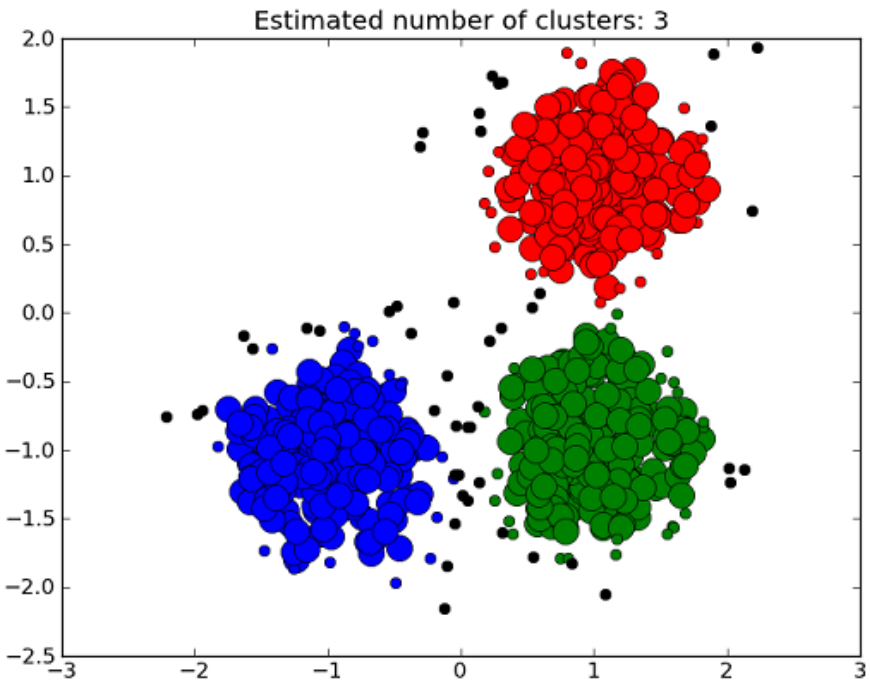

Figure 4: Unsupervised Learning - Clustering

(even the most advanced ones) are limited. **Dimensionality Reduction** is an Unsupervised Learning data preprocessing technique that reduces the number of predictor variables (*features*) of the data set in order to remove noise from data, but sometimes with the side-effect of reducing the predictive performance of the models. Generally, the goal is to compress the data onto a smaller dimensional subspace, retaining at the same time the majority of the relevant information.

1.1.3 Reinforcement Learning

In **Reinforcement Learning** there are two basic notions: The *agent* and the *environment*. The agent interacts with the environment using a Reinforcement Learning Algorithm and, via an exploratory trial-and-error approach, it tries to improve its performance. The performance of the agent is evaluated by a *reward* value given by the environment for each of the interactions with it. Typically, the agent being at a particular state in the environment makes an action and then receives feedback from the environment containing the reward for that action and the agent's new *state*, as shown in figure 5[6]. The model is trained using a large number of conceptual entities (in terms of time or space), the *epochs*. These entities are most commonly called *episodes*. The overall goal of the Reinforcement Learning Algorithm is to train the model in such a way that it finally learns a series of actions for which the cumulative reward after the

[6]Image taken from: `https://chinagdg.org/2018/12/deep-reinforcement-learning-on-gcp-using-hyperparameters-and-cloud-ml-engine-to-best-openai-gym-games/`

end of an episode is maximized.

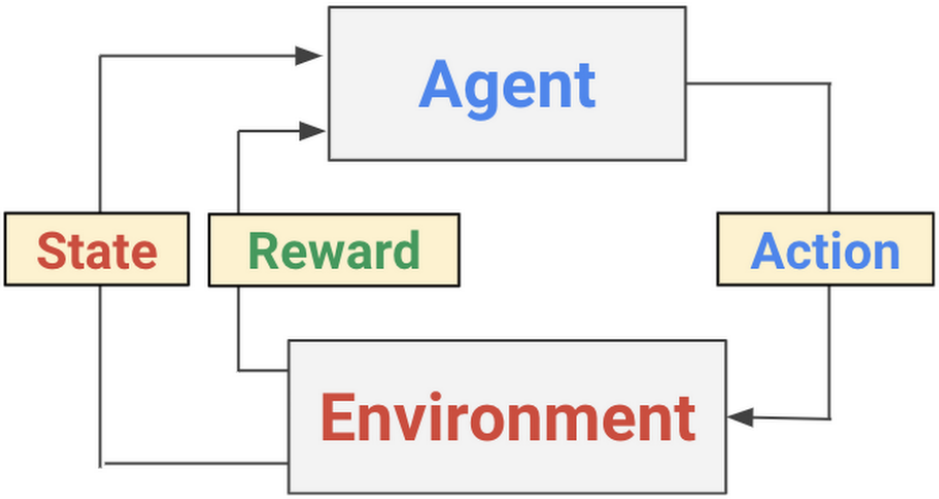

Figure 5: Reinforcement Learning - Flow Diagram

A popular application of Reinforcement Learning is a *game model*, in other words how to teach a Reinforcement Learning Model to play a particular game. In this example the agent decides the best next move considering its current state in the environment and the possible reward it will receive from it after the next move. The final goal is to win the game by maximizing the cumulative reward.

2 Neural Network Theory

2.1 The Neuron of the Human Brain

The human brain is the most complex organ in the human body. It produces our thoughts, actions, memories, feelings and experiences of the world and contains a staggering 100 billions nerve cells or **neurons**. Each neuron can make connections with thousands of others via tiny bonds, called **synapses**. The complexity of this network of synapses of the human brain is mind-blowing and the patterns these synapses make along with their strength are changing continuously[7].

The neuron is the basic cell of a nervous system and consists of the **cell body** or **soma**, the **dendrites**, and the **axon**. A dendrite is a long fibre and the receiver of electrical or chemical signals, the **impulses**. Each neuron has at least two of them and their job is to carry in the impulses to the cell body. The cell body contains a **nucleus** and is the part of the neuron that determines whether or not an impulse should be passed along to the axon. The axon usually consists of one fibre and is responsible to carry away the impulses from the cell body to other neurons, with which is attached to, via a number of synapses. The axon is capable of making synapses thanks to some nerve terminals it contains, the **axon terminals**. A visual representation of a neuron is shown in figure 6[8].

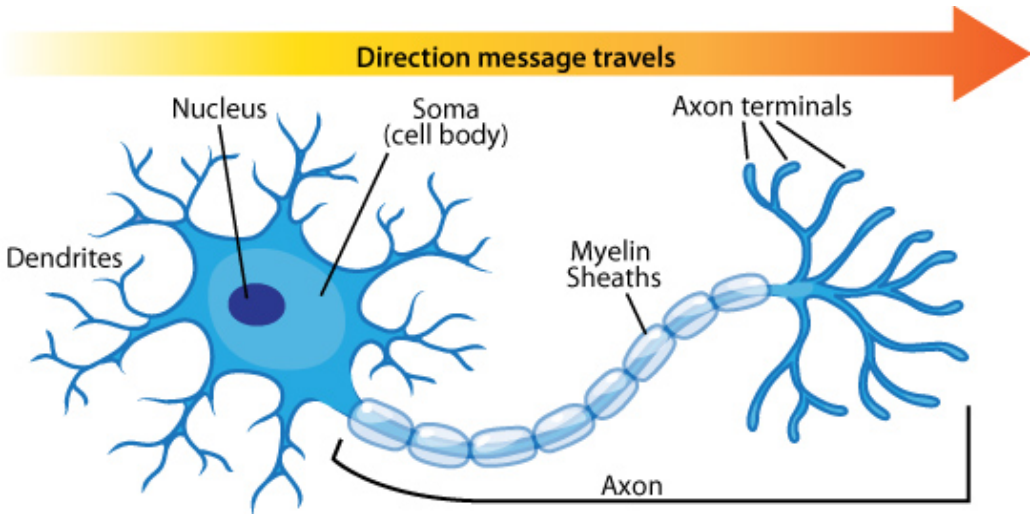

Figure 6: The Anatomy of a Neuron

[7]Conceptually, this section is based on chapter 2 of the book [1]

[8]Image taken from: `https://socratic.org/questions/why-do-peripheral-neurons-have-long-axons`

2.2 The Artificial Neuron

2.2.1 The Perceptron

In 1943, researchers Warren McCullock and Walter Pitts, in order to analyse and describe the function of the neuron, they came up with the notion of a simple *logic gate* with *binary outputs*. The dendrites receive multiple incoming signals from other neurons or external censors, that are aggregated into the cell body. If the aggregation of these signals exceeds a certain *threshold*, an output signal is generated and passed through the axon of the neuron to the other neurons that are attached to it via a number of synapses. The model is known as the **MCP neuron**.

A few years later, researcher Frank Rosenblatt, based on the MCP neuron model, published the concept of an artificial neuron, the **Perceptron**. The Perceptron is an *algorithmic model* of the MCP neuron combined with an *algorithmic learning process*. Suppose the number of the input signals is m. Then the input signals x_i are multiplied with weight coefficients or **weights** w_i and are transferred to the cell body where they are aggregated to the value of $\sum_i w_i x_i$ with $i \in (1, 2, 3, \cdots m)$, which is called the **net input** z. The cell body consists of the **aggregator**, an **activation function** $\phi(z)$ and a **threshold** θ. The activation function $\phi(z)$ used in the Perceptron model is the *step function* with respect to θ, which is:

$$\phi(z) = \begin{cases} 1, & z \geq \theta \\ 0, & z < \theta \end{cases} \tag{1}$$

If the value $\phi(\sum_i w_i x_i)$ of the activation function exceeds the given threshold θ, the Perceptron fires and since the output is binary, the output signal $\hat{y}$ of the activation function is 1, otherwise the output is 0.

More formally, we can pose this problem with the help of the linear algebra and define the input signals and the weights as the vectors $\boldsymbol{x}$ and $\boldsymbol{w}$, respectively. In addition, for simplicity reasons, we can bring the threshold θ to the left of the equation 1 and define $w_0 = -\theta$ and $x_0 = 1$, so we can now have a more compact form of the net input z and the activation function $\phi(z)$, as shown in equations 2.

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}, \ \boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix}, \ z = \sum_{i=0}^{m} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}, \ \phi(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases} \tag{2}$$

The Perceptron Algorithm can automatically learn the optimal weights $\boldsymbol{w}$ via the procedure of *training*, which in this case is called the **Perceptron Rule**. During training multiple sets of

training input signals $\boldsymbol{x}$ paired with their corresponding desired output signals $\boldsymbol{y}$ are transmitted into the Perceptron. The Perceptron rule uses the difference between the output signal $\hat{y}$ and the desired output y, which is called the **error** or **loss**, in order to update the weights $\boldsymbol{w}$ in the direction that minimizes that error. Generally, the Perceptron rule can be summarized by the following steps:

1. Initialize the weights $\boldsymbol{w}$ to 0 or a small random number
2. For each training sample $\boldsymbol{x}$:
   (a) Compute the output value $\hat{y}$
   (b) Update the weights $\boldsymbol{w}$

Conceptually, the whole process is described in figure 7[9].

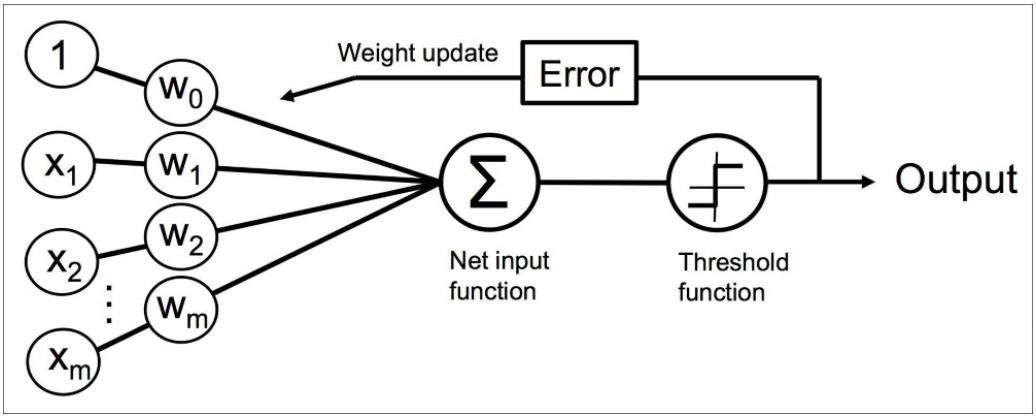

Figure 7: The Perceptron Rule

It is important to note that the convergence of the Perceptron rule is only guaranteed if the two classes (0, 1) are *linearly separable*, otherwise the Perceptron would keep updating the weights $\boldsymbol{w}$ forever.

2.2.2 The Adaline (ADAptive LInear NEuron)

The **Adaline** is an evolution of the Perceptron and published by Bernard Widrow and Tedd Hoff a few years after the Perceptron. The main difference between the Perceptron rule and the **Adaline rule** is that, in this case, the activation function $\phi(z)$ is the linear function $\phi(z) = z$, in other words the *identity function*. The benefit of this type of function is that the training process is now happening using the *continuous* value of the linear activation function $\phi(z)$, instead of the binary output in order to compute the error, which results to a much more efficient training. Consequently, we now need a **threshold function** or **quantizer** after the

[9]Image taken from: `https://ldapwiki.com/wiki/Perceptron`

linear activation function $\phi(z)$, in order to keep the the output of the Adaline binary, as shown in figure 8[10].

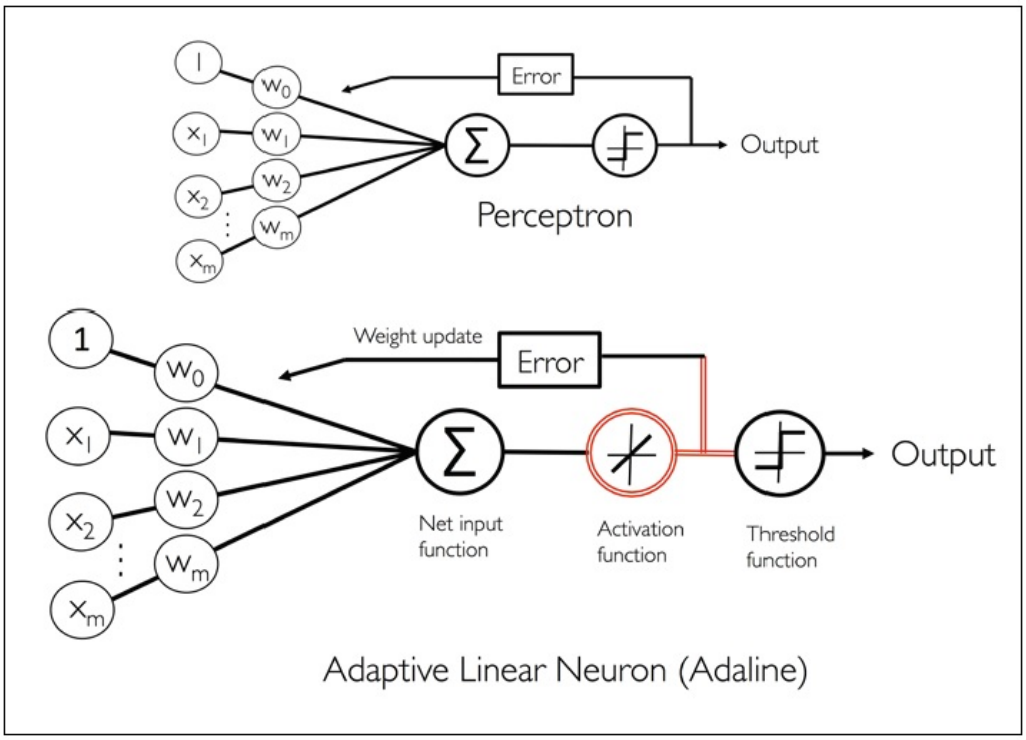

Figure 8: The Adaline Rule

Another critical difference between the Perceptron and the Adaline is the way we compute the error during the training process. In the case of Adaline, instead of the Perceptron's $(y - \hat{y})$ difference, we define an **error function** or **loss function** $J(\boldsymbol{w})$ and the objective is to minimize that function. The Adaline rule defines the error function as the *Sum of Squared Errors (SSE)* of all n samples of the training set, as shown in equation 3.

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{2}\sum_{i=1}^{n}[y_i - \phi(z_i)]^2 \tag{3}$$

The main advantages of the linearity of the loss function J are that it is *differentiable* and *convex*, which means that we can use the **gradient descent** optimization algorithm to calculate the optimal weights $\boldsymbol{w}$ that minimize the loss function J so as to classify the samples. Conceptually, the gradient descent algorithm can be described as a descenting down a valley movement until a local or global minimum is reached, as shown in figure 9[11].

The weight updates $\Delta\boldsymbol{w}$ are calculated based on all samples in the training set, unlike Perceptron's rule, which updates the weights incrementally after each sample. The training

[10]Image taken from: `https://www.simplilearn.com/how-to-train-artificial-neural-network-tutorial`
[11]Image taken from the book: `Sebastian Raschka, *Python Machine Learning*, Packt Publishing 2015`

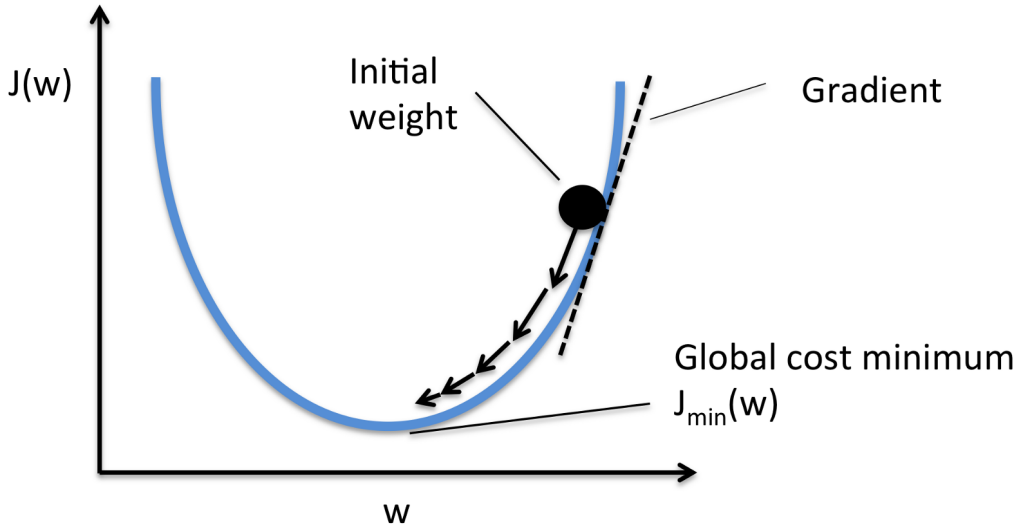

Figure 9: A Visualization of the Gradient Descent Algorithm

algorithm passes over the training set several times, which are called **epochs**, until it finally converges. In each iteration we make a small step away from the loss function's gradient on that certain point. The step size depends on the value of the **learning rate** η and the slope of the gradient. The learning rate is usually a very small number, in the range of $[10^{-5}, 10^{-3}]$. The weight updates $\Delta \boldsymbol{w}$, are proportional to the gradient $\nabla J(\boldsymbol{w})$ of the loss function J and the learning rate η and are calculated according to equation 4,

$$\Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}) \tag{4}$$

which means that each weight element w_j of the vector $\boldsymbol{w}$ is adjusted by the factor calculated in equation 5.

$$\Delta w_j = -\eta \frac{\partial J(\boldsymbol{w})}{\partial w_j} = \eta \sum_{i=1}^{n} [y_i - \phi(z_i)](x_i)_j \tag{5}$$

Finally, since all the weights w_j are updated simultaneously, the Adaline learning rule can be written in a more compact form, using vectors, as in equation 6.

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w} \tag{6}$$

2.2.3 The Artificial Neuron of Modern Neural Networks

Over the years, some other types of artificial neurons were invented based on the foundation principles of Perceptron and Adaline. The main differences between these neurons and the

Perceptron or Adaline are in the formulas of the activation functions that are used, the types of output signals and a slightly altered gradient descent algorithm, the **stochastic gradient descent**. Several of these neurons are used together, interconnected with each other, forming the modern **Artificial Neural Networks (ANNs)**.

In general, the activation functions of these new generation neurons can be non-linear and their usage depends on the type of the Neural Network we need, in order to solve a particular type of problem. A representative subset of these functions includes the **Sigmoid** $\sigma(x)$, the **Tanh** $tanh(x)$, the **ReLU**, the **Leaky ReLU**, the **MaxOut** and the **ELU**, as shown in figure 10[12].

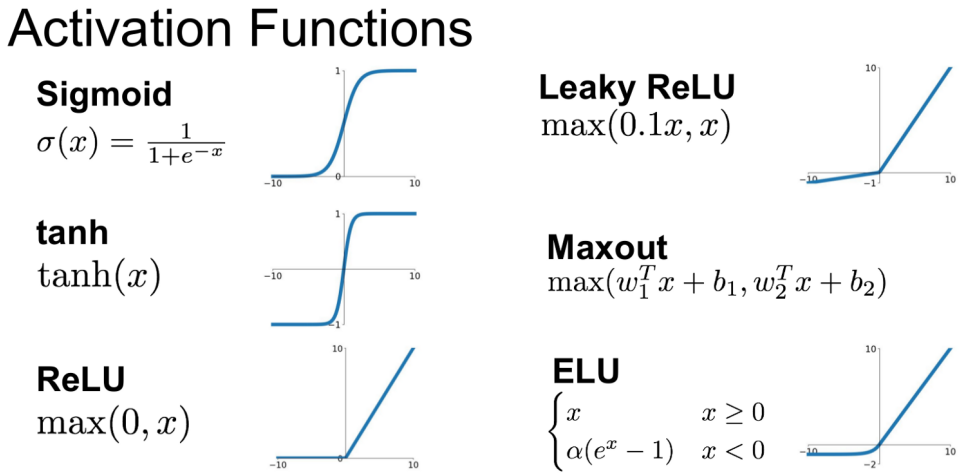

Figure 10: A Collection of Non-Linear Activation Functions

On the other hand, the output value $\hat{y}$ of these neurons is not necessarily limited to binary, but it can also be a *real continuous value* or a *class* of a multi-classed dataset. This theoretical extension differentiates the behaviour of the artificial neuron to that of a physical neuron of the human brain, but it is usually very convenient for achieving an efficient training process of the ANN which is used in.

Finally, in the case of the Adaline neuron, the weights w_j were calculated by the process of minimizing the SSE loss function (shown in equation 3), using the whole training set several times until the model converged. Nowadays, with the modern ANNs that are in use, the size of the training datasets is usually very large, which means that running through the whole dataset for each epoch would be computationally very costly. An answer to this problem is the **stochastic gradient descent** optimization algorithm, which consists of the following steps:

[12]Image taken from: `https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c`

1. *Shuffle* the dataset randomly.

2. *Apply* the gradient descent algorithm, using one sample at a time.

Since each gradient descent value is calculated based on a single training sample, the value of the error function $J(\boldsymbol{w})$ includes more noise than in gradient descent, which can be an advantage for escaping shallow local minima more easily. The deficit of this method is, of course, the reduced computational efficiency, because of the huge number of times the weights w_j are being updated. A good compromise between the gradient descent and the stochastic gradient descent is the **mini-batch stochastic gradient descent**. In this case, the whole dataset is divided into *mini-batches* (groups) of samples on which the stochastic gradient descent algorithm is then applied. The advantage of this method is the much faster convergence of the training process because of the more frequent weight updates compared to the Adaline's gradient descent algorithm. Many of the modern neural network learning processes use the mini-batch stochastic gradient descent algorithm.

2.3 The Artificial Neural Network (ANN)

As previously seen, the human brain is a collection of neurons connected together via synapses. Each neuron receives as inputs the outputs of other neurons that are connected to it, makes some calculations and if the result of the calculation exceeds some threshold θ, fires and propagates the output signal to the next connected neuron. An **Artificial Neural Network (ANN)** is an algorithmic construction that mimics a certain function of the human brain. The idea is based on taking a large dataset of training samples and developing a system which can learn from these samples, so the ANN can automatically infer rules that enables it to recognise previously unseen samples or make predictions about future events. The topology of an ANN can belong to one of three major categories: the **Feed-Forward ANNs**, the **Recurent ANNs** and the **Completely Linked ANNs**, all of which are discussed in more detail in the next subsections.

2.3.1 The Feed-Forward Artificial Neural Network (Feed-Forward ANN)

In general, a Feed-Forward ANN consists of:

1. a number of interconnected artificial neurons, which are divided in three categories:

   (a) The **input** neurons or **sensors**

   (b) The **hidden layer** neurons

   (c) The **output** neurons

2. a number of *directed weighted connections*, the **synapses**. The weight w of a synapse acts as the strength of this synapse.

The complexity of the human brain is enormous and if we want to replicate it, the only thing we can do is to approximate it. For some particular problems, a good approximation we can use is an idealized **Feed-Forward ANN**, which consists of discrete **layers** of neurons. Each layer of neurons is connected to the next with several synapses. Typically, the first layer is the **input layer** which receives the inputs $\boldsymbol{x}$ and feed-forward them to the next layer unchanged. The next part of the Feed-Forward ANN consists of one or more **hidden layers**, which take the outputs of the previous layer, make some calculations based on a **forward propagation function** f_{prop} and forward them to the next layer. The Feed-Forward ANNs that have more than one hidden layers of neurons are called **Deep Feed-Forward ANNs**. The last layer of the Feed-Forward ANN is called the **output layer** and is responsible for producing the final output $\hat{\boldsymbol{y}}$.

A common connection topology for a Feed-Forward ANN is the **Fully Connected** Feed-Forward ANN, in which every neuron of a particular layer is connected to all of the neurons of the next layer, as shown in figure 11[13]. The main characteristic of these connections is that they *never* form a circle, hence the term "feed-forward".

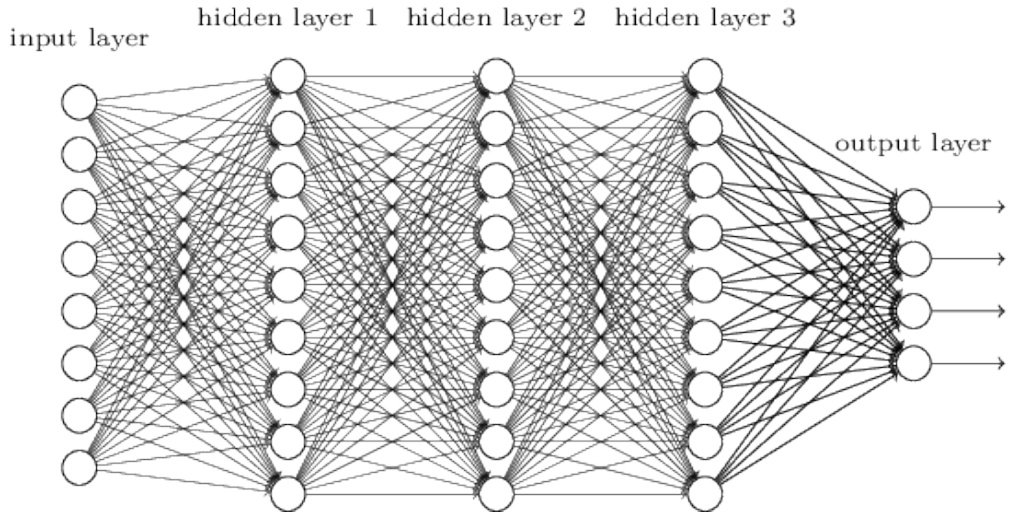

Figure 11: A Fully Connected Feed-Forward Artificial Neural Network

The most common forward-propagation function f_{prop} used in Feed-Forward ANNs is the previously defined as the net input of the Perceptron z, which is the weighted sum of the inputs x_i, as shown again in equation 7, where m is the number of neurons contained in the previous layer. This statement does not apply, of course, to the input layer.

[13]Image taken from: `www.http://neuralnetworksanddeeplearning.com/chap5.html`

$$f_{prop}(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=0}^{m} w_i x_i = \boldsymbol{w}^T \boldsymbol{x} \tag{7}$$

The most interesting aspect of a Feed-Forward ANN is its training process. The most popular training approach is the **back-propagation algorithm**, the inspiration of which is based on the stochastic gradient descent algorithm we looked at earlier. The exact steps of this algorithm in order to adjust the weights of a Feed-Forward ANN are:

1. We forward-propagate the input vector $\boldsymbol{x}$, layer by layer, using the forward propagation function f_{prop}, until the Feed-Forward ANN produces the output vector $\hat{\boldsymbol{y}}$.

2. This procedure results in an error for each of the neurons of the output layer, which is calculated according to an error function $J(\boldsymbol{w})$.

3. We compute the gradient of the error function $J(\boldsymbol{w})$ with respect to the weights w_j of the neurons of the output layer and then we update these weights in the direction of reducing the errors of the output neurons.

4. We back-propagate these output errors back to the hidden layers, one at a time, to infer errors to the hidden neurons, respectively.

5. We compute the gradients of these errors as well, using the same error function $J(\boldsymbol{w})$ and we update the weights of the hidden neurons, accordingly.

6. We repeat the previous steps for the entire dataset until the Feed-Forward ANN converges.

The Feed-Forward ANNs are the basis of many important types of Neural Networks that are currently being used, such as the **Convolutional Neural Networks (CNNs)** for computer vision applications [4], that will be discussed in more detail in a later section.

2.3.2 The Recurrent Neural Network (RNN)

The **Recurrent Neural Networks (RNNs)** are based on the notion of **recurrence**, which is defined as a process of an artificial neuron *influencing itself* by any connection (synapse) and along a *temporal* sequence. In general, all connections of a RNN form a directed graph. Some types of RNN allow for their artificial neurons to be connected to themselves. These types of connections are called **direct** or **self recurrences**. As a result, the artificial neurons that implement such connections strengthen themselves in order to reach their activation limits θ. If additional connections between artificial neurons and other artificial neurons of the same or different layer are allowed, then we have the case of **indirect recurrences**. A RNN paradigm

with direct connections and an additional feed-forward layer is shown in figure 12[14].

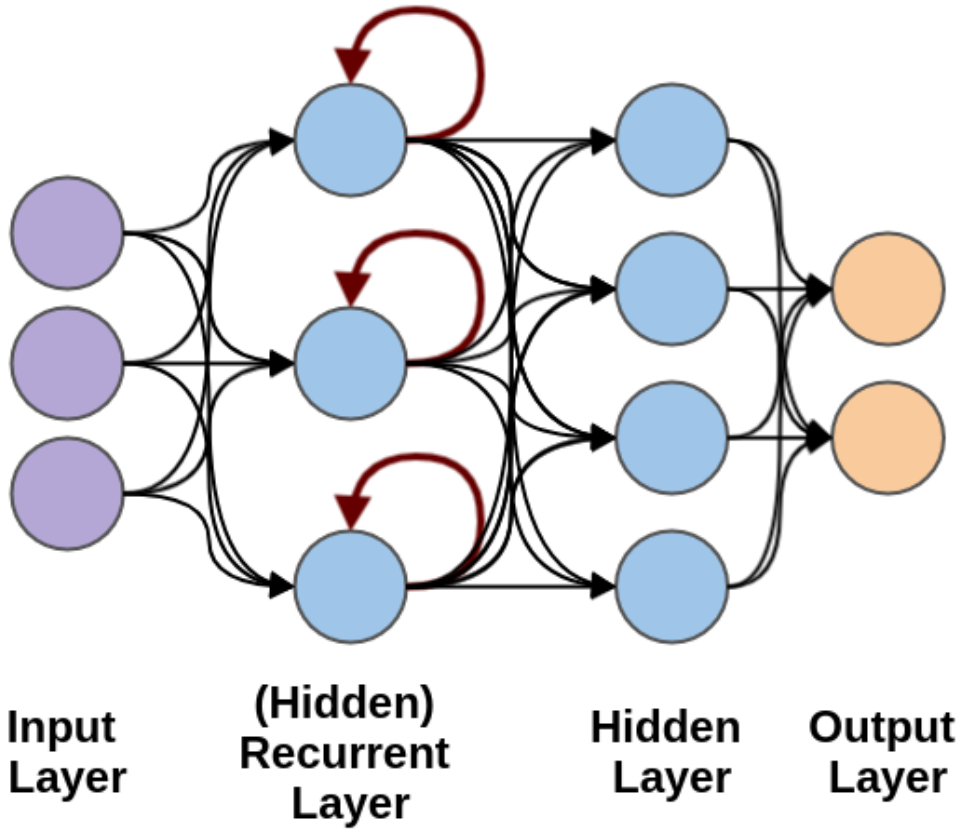

Figure 12: A Recurrent Neural Network Paradigm

Unlike Feed-Forward ANNs, RNNs can have **internal memory** so they can keep data from previous recurrences, in order to be used in a next recurrence. In the case of RNNs, the time is divided into **time steps**. RNNs work through loops and each loop belongs to a certain time step t. The computation algorithm of a RNN takes into consideration not only the current input $\boldsymbol{x_t}$ but also a number of results $\hat{\boldsymbol{h}}_{\boldsymbol{t}}$ it computed in previous time steps $(t-1, t-2, \cdots)$ and are stored in the internal memory of the RNN, as shown in figure 13[15]. Typically, RNNs have short-term memories and their recursive nature allows exhibiting a possible *temporal behaviour* of the input signal $\boldsymbol{x_t}$, therefore they are used in applications such as speech recognition or time series prediction.

In general, the RNNs are being trained through a process that is called **Back-Propagation Through Time (BPTT)**, which is an extension of the back-propagation algorithm used for

[14]Image taken from: `https://austingwalters.com/classify-sentences-via-a-recurrent-neural-network-lstm/`

[15]Image taken from: `https://medium.com/@camrongodbout/recurrent-neural-networks-for-beginners-7aca4e933b82`

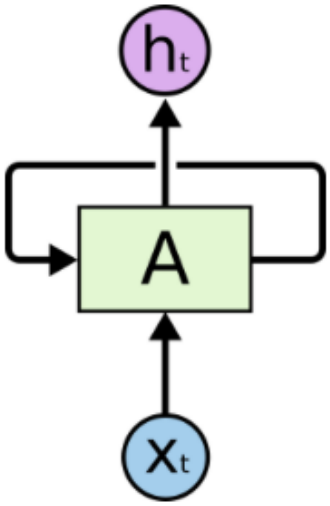

Figure 13: The Dataflow of a Recurrent Neural Network A

the Feed-Forward ANNs. A RNN can be thought of as multiple copies of the same network, each passing a message to its successor. Each copy belongs to a certain time step t. Conceptually, if we would like to unroll a RNN through time, we would sketch a figure as shown in figure 14[16].

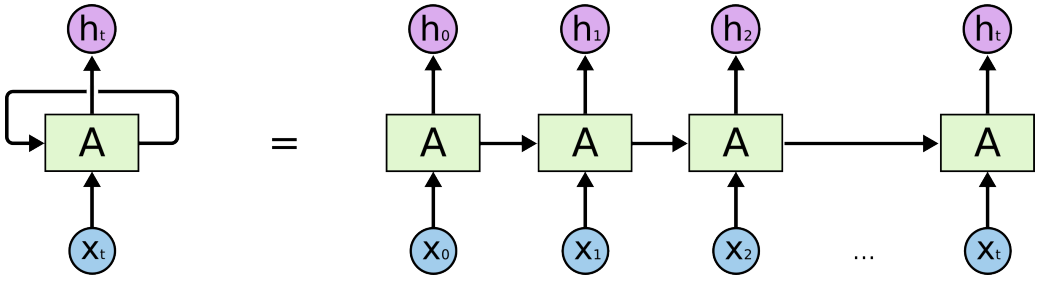

Figure 14: A Unrolled Recurrent Neural Network A

In the case of BPTT, we apply the back-propagation algorithm to the RNN, starting from the time step t, backwards to time step 0. This means that, for each new time step t, the back-propagation chain of the RNN is prolonged by 1, in other words the time element in the BPTT algorithm only extends the depth of the back-propagation algorithm step by step.

RNNs are very useful because of their ability to connect previous information to a current task and therefore extract *relationships* between them. This results to an enhanced capability of understanding better their current state in the context of older sequential data. This comes with a price though because of the **vanishing gradients problem** BPTT training algorithm faces. The vanishing gradients problem is the indirect result of the continuous extension of the

[16]Image taken from: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

depth of the RNN. Each time the BPTT algorithm steps backwards, it calculates the gradient of the error function J for each neuron of that time-layer with respect to its weights w_j and it multiplies the result with the learning rate η. Taking into account that the value of the learning rate η is usually much lower than 1 ($\eta \ll 1$), the more we travel backwards in time the more times that time-layer's gradients are multiplied with η, which means that after several steps back the values of the gradients are very small (practically 0). As a result, the RNNs stop learning after a few time steps backwards and consequently they are unable to connect old data of these time-layers to the current task.

A recently invented type of RNN, which has been used extensively in our assignment, is the **Long Short-Term Memory RNN (LSTM RNN)**[5]. LSTMs cope efficiently with the vanishing gradients problem and are extensively discussed in a later section.

2.3.3 The Completely Linked Neural Network (CLNN)

In the case of **Completely Linked Neural Networks (CLNNs)**, every neuron can be connected to all other neurons except for itself (the direct recurrences are prohibited). The only limitation is that all connections must be **symmetric**, which means that if there is a connection from neuron i to the neuron j ($i \longrightarrow j$), there must also be the reverse connection from neuron j to the neuron i ($j \longrightarrow i$), as shown in figure 15[17].

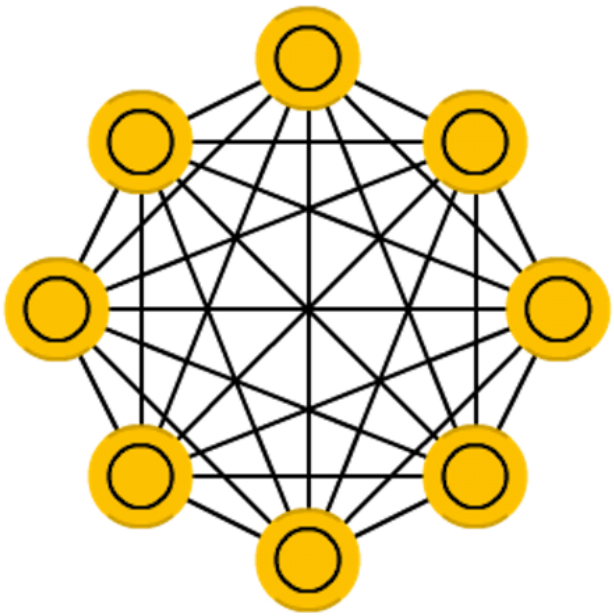

Figure 15: A Completely Linked Neural Network

This unique topology of the CLNNs results in two very interesting properties these networks

[17]Image taken from: `http://www.asimovinstitute.org/neural-network-zoo/`

have, which are:

1. Every neuron of an CLNN can become an input neuron.

2. Defined layers of neurons no longer exist.

A great application of the CLNNs are the **Self-Organizing Maps (SOMs)** or **Kohonen Neural Networks (KNNs)**[6], which are ANNs that produce a lower dimensional representation (usually a 2D representation) of the input space of the training samples $\boldsymbol{x}$ and therefore are used in unsupervised learning for dimensionality reduction applications. The basic aspect that differentiates SOMs from Feed-Forward ANNs and RNNs is the learning process that is used for their training. Their learning method is called **competitive learning**, in which all nodes compete for the right to respond to a subset of the input data $\boldsymbol{x}$, using a **neighbourhood function** $N(\boldsymbol{x})$ to preserve the topological properties of the input space. This results in increasing the specialization of each node of the network, which means that SOMs are well suited to finding *clusters* within data, as shown in figure 16[18].

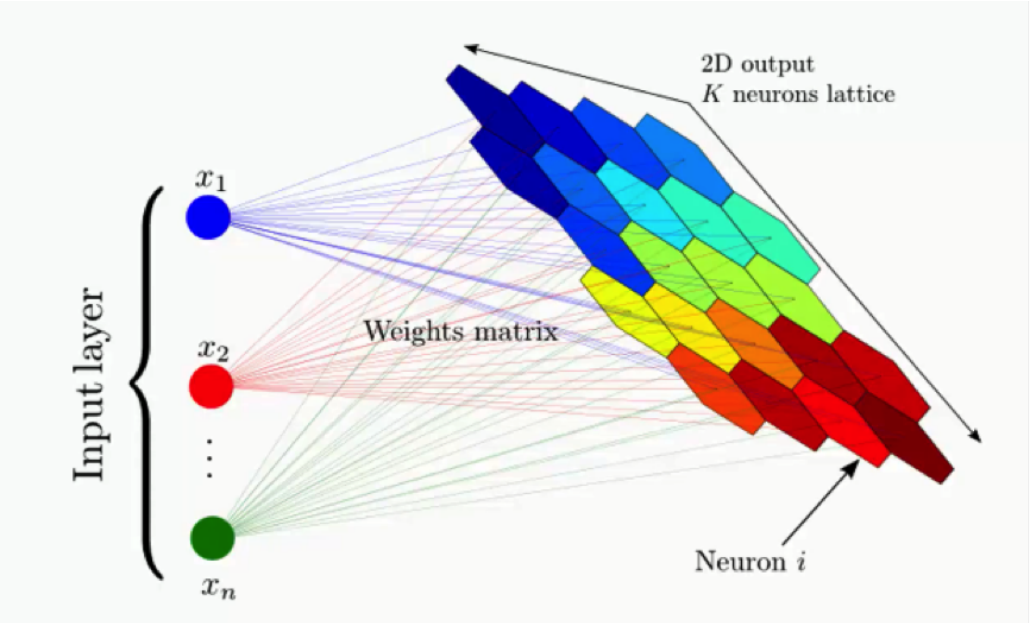

Figure 16: A 2D Diagram of a Self Organizing Map

2.4 The Convolutional Neural Network (CNN)

Human vision is a very complicated function. The way the human brain inputs and processes an image had been puzzling scientists until 1962, when a fascinating experiment by neuroscientists Hubel and Wiesel took place. In order to understand the functionality of the **visual**

[18]Image taken from: `https://www.superdatascience.com/blogs/the-ultimate-guide-to-self-organizing-maps-soms`

cortex, the part of the human brain that is responsible for the human vision, Hubel and Wiesel showed that some individual neurons in the brain fired only in the presence of edges of a certain orientation. For example, some neurons reacted when exposed to vertical edges, others to horizontal edges and some others to diagonal ones. But the most important discovery they made was that all of these neurons were organized in a columnar architecture, in a way that all these groups of neurons together were able to produce **visual perception**. In other words, what they found was that the visual cortex consists of *specialized components* (groups of neurons) that have specific tasks, which means that when the human brain processes an image, it tries to find certain characteristics or **features** of that image in order to comprehend and classify it.

Convolutional Neural Networks (CNNs)[4] are conceptually inspired from the virtual cortex of the human brain. They mimic the way the human brain processes an image in order to classify it into a predefined class by leveraging the facts that:

- Nearby pixels of an image are more strongly related than distant ones.
- Objects are built up out of smaller parts or **features**.

When a computer takes an image as input, in reality it receives a 3D array of pixel values with size $W \times H \times D$. Depending on the resolution of the image, W refers to the width in pixels, H refers to the height in pixels and D refers to the color depth of a pixel. In the case of a colored RGB image we have $D = 3$ because we need an integer value in the range of $(0 - 255)$ for each of the *red*, *green*, and *blue* colors of the RGB image. Each of these values represents the intensity of the corresponding color. Of course, in the case of a gray-scaled image we have $D = 1$ since the color used in the image is only one (white). The numbers of the image array are the only inputs available to the computer during the image classification process and the task for the computer is to output numbers that describe the probability of the image belonging to a certain class.

Technically, a typical CNN consists of eight types of layered concepts, which are:

1. The **Input** Layer
2. The **Convolutional** Layer
3. The **Non-Linear** Layer
4. The **Pooling** Layer
5. The **Flattening** Layer
6. The **Fully Connected** Layer
7. The **Normalization** Layer

8. The **Output** Layer

All eight types of layers will be discussed in more detail in the following subsections.

2.4.1 The Input Layer

In CNNs we take advantage of the fact that all the inputs they have to process are images and therefore we optimize their architecture in a more suitable way. In particular, unlike the regular ANNs in which all inputs are flattened out first to an 1D vector, the **input layer** of a CNN is a 3D construction of neurons with size $W \times H \times D$, the same size as the input image. Most of the times and if no additional computational process is required at this stage, the input layer just overlaps the convolutional layer.

2.4.2 The Convolutional Layer

The second layer in a CNN is the **Convolutional Layer**. The intuition behind this layer is similar to the convolution operation $(f * g)$ on two functions $f(t)$ and $g(t)$, from mathematics, which is shown in equation 8.

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{8}$$

In our case, the $f(t)$ function corresponds conceptually to the $W \times H \times D$ array of pixels of the input image, and the $g(t)$ function to an other, much smaller $F \times F \times D$ array which is called **filter** or **kernel**. Typical values for the F dimension belong to the range of $[2, 5]$. The previous dimensions of the image and the filter[19] suggest that:

- The color depth of the image and the filter must be the same (D).
- The two of the three dimensions of the filter, other than the color depth D, must be the same, forming a rectangle with size F.

The values of the filter are called **weights** or **parameters**, and all these values combined are shaping a **pixel pattern**, which is also called a **feature**. In general, during the convolution operation we aim to find if some well-known features are included somewhere in the input image, in order to be able to classify that image later on. In practice, the convolution operation consists of the following steps:

[19]From now on the terms **image** and **image array** or **filter** and **filter array** respectively are being used interchangeably.

1. We *overlap* the filter $\boldsymbol{f}$ with the top-left corner of the image $\boldsymbol{x}_f$, in a pixel by pixel fashion. In this case we have the top-left $F \times F \times D$ part of the image overlapped by the $F \times F \times D$ filter.

2. We compute the *dot product* $\boldsymbol{f} \cdot \boldsymbol{x}_f$, by making element-wise multiplications of the values $f_{i,j}$ of the filter with the corresponding values $x_{i,j}$ of the image and then we aggregate the results in a single value $\boldsymbol{f} \cdot \boldsymbol{x}_f$, as shown in equation 9.

$$\boldsymbol{f} \cdot \boldsymbol{x}_f = \sum_{i=1}^{W \cdot H} \sum_{j=1}^{D} f_{i,j} \cdot x_{i,j} \tag{9}$$

3. We slide or **convolve** the filter to the right by a number of pixels, which is called the **stride** $\boldsymbol{S}$ of the convolution layer, and we repeat the process of the previous step again, from left to right and from top to bottom, until all parts of the image are visited at least once. This whole process consists the **convolution** of the filter through the image. Typical values for the stride belong to the range of $[1, 5]$.

4. All convolution values $\boldsymbol{f} \cdot \boldsymbol{x}_f$ computed during the convolution process define a new array which is called **activation map** or **feature map**. The size of the feature map $W_{fm} \times H_{fm}$ with respect to the size of the image (W, H), the size of the filter F and the stride S is derived from equation 10.

$$W_{fm} \times H_{fm} = \left(\frac{W - F}{S} + 1 \right) \times \left(\frac{H - F}{S} + 1 \right) \tag{10}$$

   The whole process is visually described in figure 17[20].

5. We repeat the convolution process for various filters, scanning the image for many different features. In that way we construct a bunch of feature maps in order to obtain the **final convolutional layer** of the CNN, as shown in figure 18[21]. The convolutional layer is also a 3D construction of neurons with size $W_{fm} \times H_{fm} \times K_{fm}$, where K_{fm} if the number of filters that was used.

From the previous analysis it is now obvious that the whole convolution process has two main disadvantages:

[20]Image taken from: `https://www.vaetas.cz/posts/intro-convolutional-neural-networks/`

[21]Image taken from: `https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-1-convolution-operation`

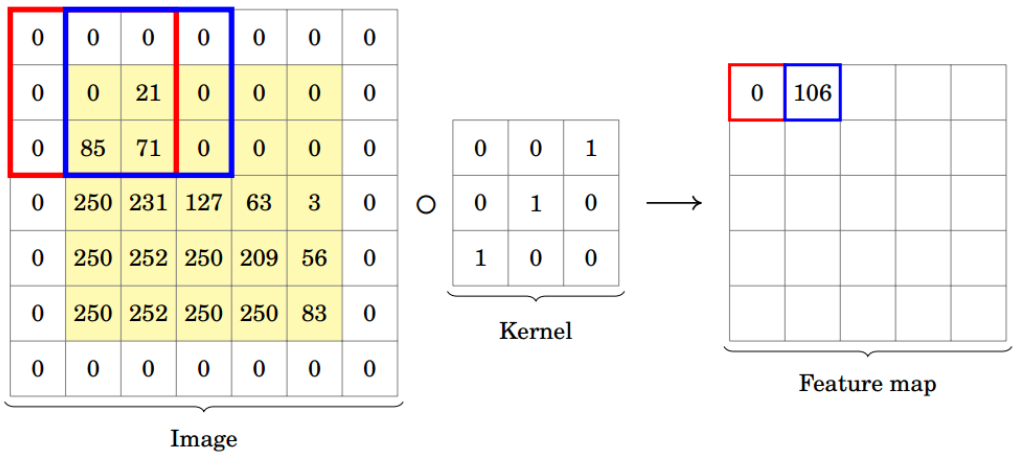

Figure 17: The Convolution Process ($L = W = 7, F = 3, S = 1$)

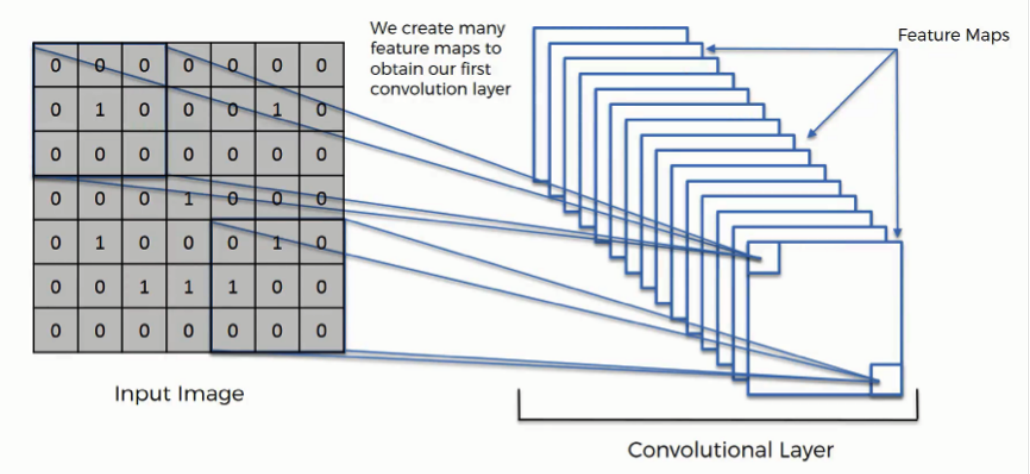

Figure 18: The Final Convolutional Layer of the CNN

1. After a complete convolution process the image that is produced (feature map) can be *reduced* in *size*. In modern CNNs where there are several convolutional layers, this could lead to smaller and smaller images and this is something that usually adds difficulties to the training process of a CNN.

2. During the convolution process, pixels that are positioned at the *corners* or at the *edges* of the image are visited fewer times compared to the pixels that are positioned in more centric areas, which means that there is a possibility of information loss for possible features that are positioned near these spatial-limiting areas.

In order to overcome these issues we can use **padding**. With padding we add symmetrically a number $\boldsymbol{P}$ of black pixels ($x_i = 0$) at the borders of the image. This procedure results in a larger image by $2P$ pixels in each dimension. In this case the size of the feature map $W_{fm} \times H_{fm}$ with respect to the size of the image (W, H), the size of the filter F, the stride S and the padding P is derived from equation 11.

$$W_{fm} \times H_{fm} = \left(\frac{W - F + 2P}{S} + 1\right) \times \left(\frac{H - F + 2P}{S} + 1\right) \tag{11}$$

The value of P depends on the padding style we want to apply and this style depends on the style of the data we focus on. In general there are two major categories of padding:

1. **Full Padding**: The value of P ensures that all pixels are visited the same amount of times by the filter. This method has the disadvantage of increasing the size of the input.
2. **Same Padding**: The value of P ensures that the output has the same size as the input.

Visually, the two padding styles are described in figure 19[22].

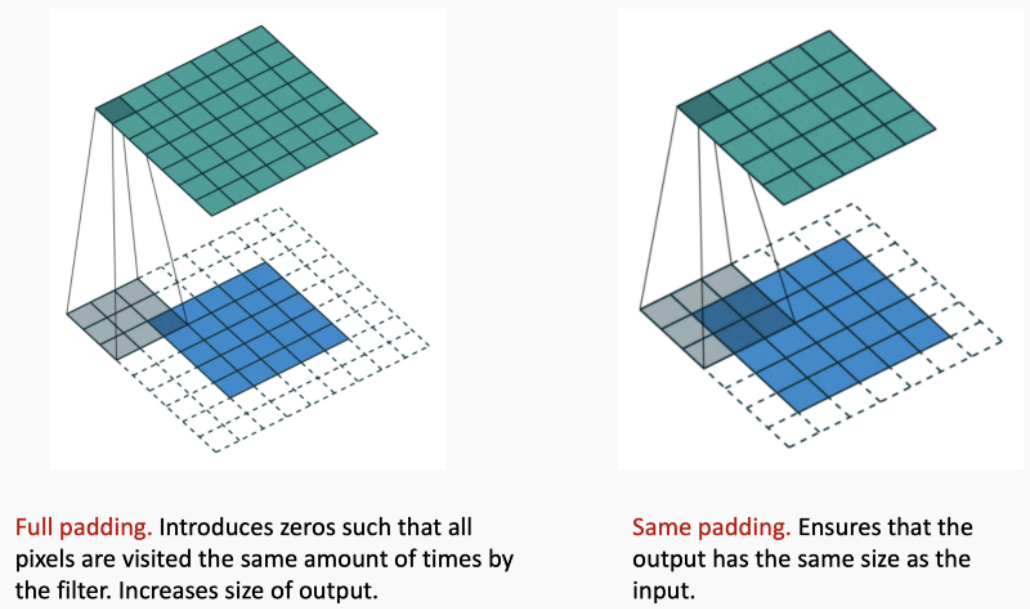

Figure 19: The Two Padding Styles

2.4.3 The Non-Linear Layer

Up until now, all operations that have been computed by the CNN were linear, just element-wise multiplications and summations. The **Non-Linear Layer** introduces non-linearity to the system, just after the convolutional layer. The main reason we need non-linearity into CNNs is that it helps us to alleviate from the **vanishing gradients** problem that mentioned before in the RNN section. All multi-layered ANNs that are trained with the use of the back-propagation algorithm face this problem because the gradients decrease exponentially through the layers,

[22]Image taken from: `https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac`

as they propagate from the back to the front layers.

In the case of CNNs, several members of the family of the non-linear activation functions can be used, but the most common one is the **Rectified Linear Unit (ReLU)** function. As we have seen in previous sections, the ReLU function is defined as shown in equation 12 and is graphically described in figure 20[23].

$$R(x) = max(0, x) \tag{12}$$

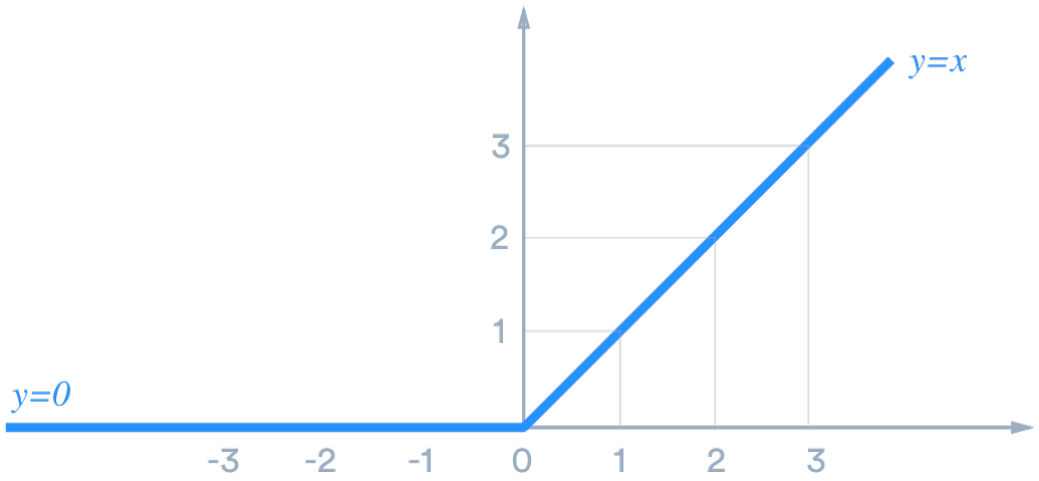

Figure 20: The ReLU Activation Function

Practically, the ReLU function turns all the negative weights of a feature map to zeros, which is a computationally efficient procedure, leading to a faster CNN training without making a significant difference to the accuracy of the results.

2.4.4 The Pooling Layer

The next layer in a CNN is the **Downsampling** or **Pooling Layer**. The way this layer operates is similar to the convolution layer. In the case of the pooling layer, a $F \times F$ filter convolutes through a $W \times H$ feature map with a stride of S, but the operations applied to each step are totally different, and belong to one of the following three main methods:

1. **Max Pooling**: The filter selects the *maximum* value of the $F \times F$ part of the feature map it currently convolves. This is by far the most popular method and this is the method that was used in this assignment.

2. **L2-Norm Pooling**: The filter computes the *L2-Norm* value of the $F \times F$ part of the feature map it currently convolves.

[23]Image taken from: `https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7`

3. **Average Pooling**: The filter computes and selects the *average* value of the $F \times F$ part of the feature map it currently convolves.

A visual representation of the max pooling operation of a 2×2 filter through a 4×4 feature map and a stride of 2 is shown in figure 21[24].

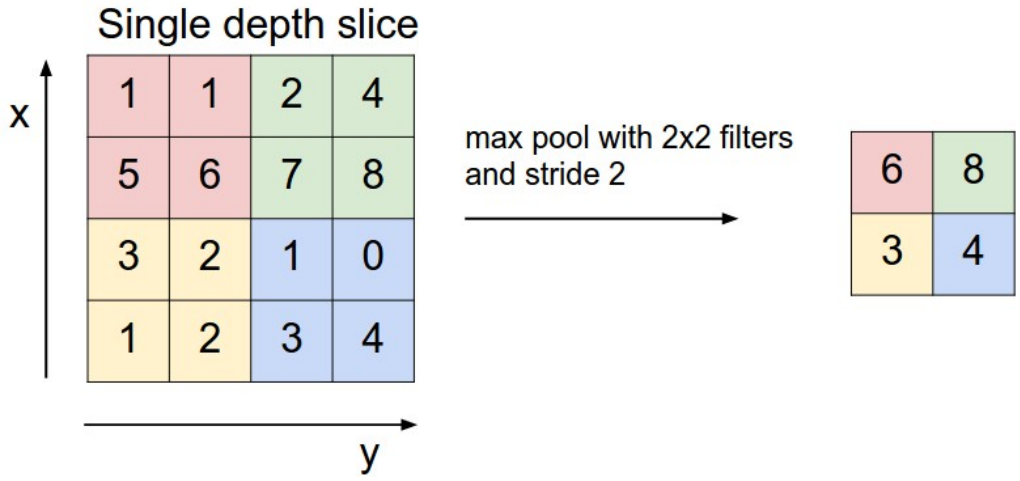

Figure 21: The Max Pooling Operation ($L = W = 4, F = 2, S = 2$)

Intuitively, the pooling layer acts as a **feature collection mechanism**. Considering the example in figure 21, a high activation value of a pixel of the activation map means that a specific feature was spotted nearby that area of the input image. The max pooling operation collects that information and stores it in the corresponding pixel of a new array, which consists the **max pooling map**.

The most important reasons for adding a pooling layer into the architecture of a CNN are:

1. It reduces the size of the feature maps considerably, without loosing at the same time the critical part of the information we need.

2. It can identify correctly the features of the input image, even though they are twisted or placed in a slightly different position.

3. Although it removes information from the input image, the important features of the input image are successfully preserved.

4. The information removal from the input image helps the training process to reduce **overfitting** significantly. The term overfitting refers to a model that is so accurately trained for the train data set that it cannot generalize well, and fails when processing the test data set.

[24]Image taken from: `http://cs231n.github.io/convolutional-networks/`

2.4.5 The Flattening Layer

The **flattening layer** receives the $W \times H \times D$ image array and reshapes (flattens) it into a column vector $\boldsymbol{x}$ of size $[W \cdot H \cdot D \times 1]$, as shown in equation 13. This step is essential in order to use the 1D vector as input to the fully connected layer that comes next.

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}, \; m = W \cdot H \cdot D \tag{13}$$

2.4.6 The Fully Connected Layer

The **fully connected layer** is a typical Deep ANN, like the ones discussed in more detail in previous sections, with its input layer fully connected to the flattening layer. Basically, it is responsible for the *classification* task of the CNN. Alternatively, depending on the nature of the problem, other types of neural networks can be used like RNNs. A special case of a RNN, the LSTM, is what was used in this assignment.

2.4.7 The Normalization Layer

The output of the fully connected layer are values that are analogous to the probability of an image belonging to a certain class, but in general, they are not normalized, which means that these probabilities don't add up to 1. The **normalization layer** normalizes the probabilities, using a normalization method like **SVM** or **Softmax**. In this assignment we used the Softmax approach which usually gives better results and practically consists of the application of the **Softmax function** to the output of the fully connected layer. Conceptually, we can say that the Softmax function is the generalization of the Binary Logistic Regression Classifier to multiple classes. The Softmax function is shown in equation 14, where $f(z_j)$ is the probability of the output value z_j.

$$f(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{14}$$

2.4.8 The Output Layer

The **output layer** consists of N neurons, where N is the number of the different classes of the training data set. Each of these neurons contains the value of the probability a certain image belonging to a certain class. These values are directly propagated from the normalization layer.

2.4.9 The Back-Propagation Operation in a CNN

One of the most significant operations of a CNN, if not the most significant, is its **back-propagation operation**[9] because through this operation we are able to train it. Depending on the layer of the CNN we focus on each time, we have different approaches for the loss function we have to use. In our case we divide the back-propagation procedure in three categories, which are:

1. **Back-Propagation in the Fully Connected Layer**: In this layer, during the forward propagation we used the Softmax function to normalize the probabilities of the output, which means that in the back-propagation we have to compute the gradients of that function in reverse. The resulting function is called the **cross-entropy loss function** and is shown in equation 15, where the notation f_j means the j-th element of the vector of class-scores f.

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{15}$$

   In practice, in order to calculate the cross-entropy loss function more efficiently we use the *information theory* version of the function, which is shown in equation 16.

$$H(p,q) = -\sum_x p(x)\log q(x) \tag{16}$$

   Equation 16 describes the cross-entropy between a **real distribution $\boldsymbol{p}$** and an **estimated distribution $\boldsymbol{q}$**. As seen above, the estimated class probabilities q are calculated according to the formula of equation 17.

$$q = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \tag{17}$$

   The values of the real distribution $\boldsymbol{p}$ is a **one-hot vector**, with the shape of that of equation 18, where the whole probability mass ($p_j = 1$) belongs to the correct class.

$$\boldsymbol{p} = [0, \cdots, 1, \cdots, 0] \tag{18}$$

2. **Back-Propagation in the Pooling Layer**: In the case of the max-pooling operation that was implemented in our assignment, and for the backward pass for the *max operation*, we only have to route the gradient to the input that had the highest value in the forward pass. A common practice to achieve this routing is to keep track of the index of the max activation during the forward pass.

3. **Back-Propagation in the Convolutional Layer**: The back-propagation pass for a convolution operation, for both the data and the weights, is also a convolution. The only difference is that during the backward pass we must use the corresponding **spatially-inverted filters**. Back-propagation in the convolutional layer is one of the most critical operations in a CNN because this is the process its filters are trained in, meaning that this is the operation the CNN learns which are the significant features of the input images and which are not, in order to be able to classify these images correctly. An visual example of an array of trained filters is shown in figure 22[25].

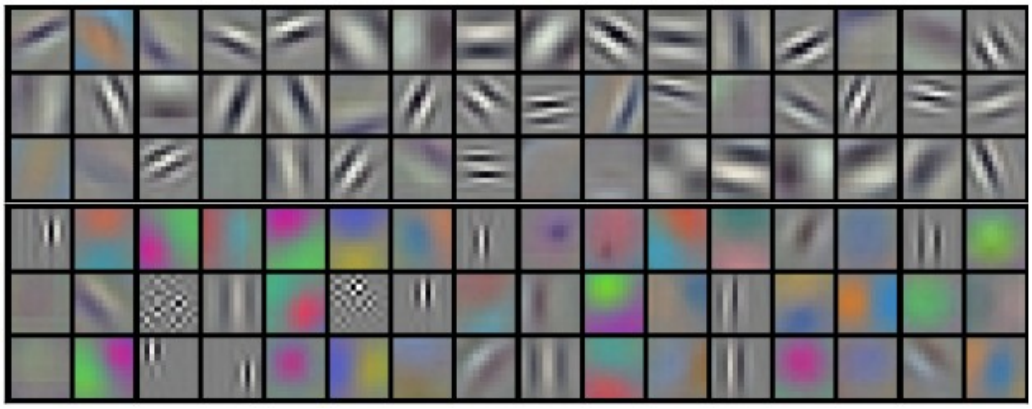

Figure 22: An Array of Trained Filters of a CNN

2.4.10 The Modern Architecture of a CNN

As CNNs have been evolving through time, new architectures have also been invented. The main areas the researches have been focusing on are:

1. **The number of the convolutional layers**: Modern CNNs can have more than one convolutional layer followed usually by a max pooling layer. These multiple-layer combinations play a big role to the increased accuracy of the CNNs.

2. **The type of the fully connected layer**: Several types of ANNs are being used nowadays with the DNNs and the RNNs the most common ones.

A visual representation of a modern CNN is shown in figure 23[26].

[25]Image taken from: `http://cs231n.github.io/convolutional-networks/#conv`

[26]Image taken from: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

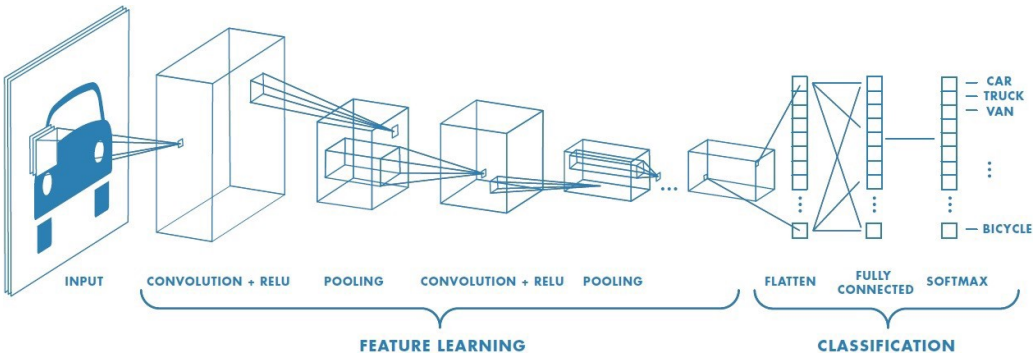

Figure 23: The Architecture of a Modern CNN

2.5 The Long Short-Term Memory Neural Network (LSTM)

As we discussed in a previous chapter, the **Long Short-Term Memory Neural Network (LSTM)**[5] is a member of the RNN family. They were introduced by Hochreiter & Schmidhuber in 1997, and its biggest attainment is its capability of coping efficiently with the *vanishing gradients problem*, meaning that they are capable of learning long-term dependences. The LSTM, like the RNN, can be thought of as multiple copies of the same network, each passing a message to its successor. In the case of LSTM though the repeating network A or **cell**, contains *four network layers*, instead of one. If we unroll a LSTM through time we get the structure shown in figure 24[27].

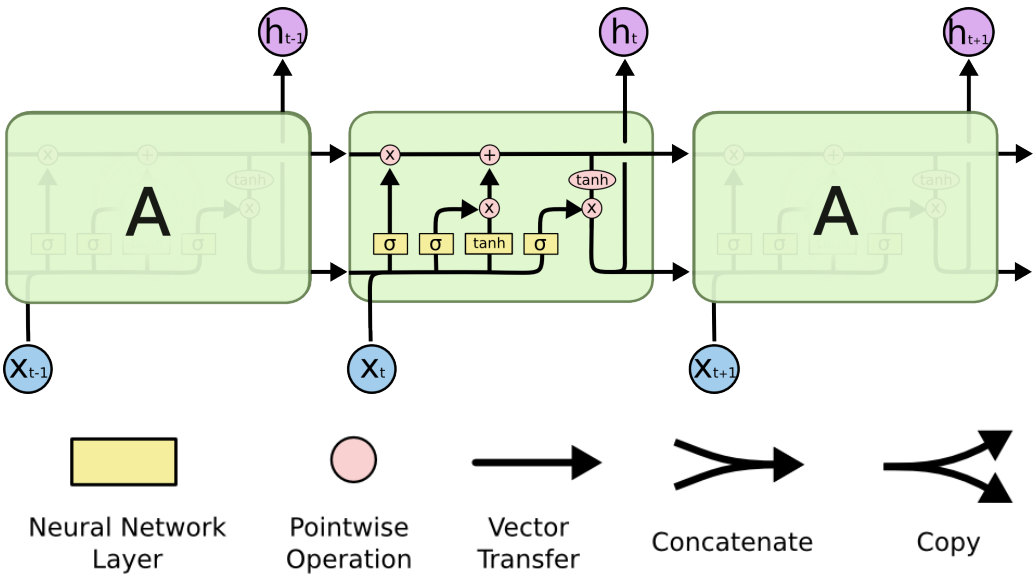

Figure 24: The Unrolled LSTM

[27]Image taken from: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

2.5.1 The Fundamental Concept behind LSTM

The core concept behind LSTM is the **cell state**, and the various **gates**[10]. The cell state is a vector that corresponds to the state of the previous cell, is modified in the current cell and is propagated to the next cell through a propagation line, annotated with the top horizontal line of figure 25[27].

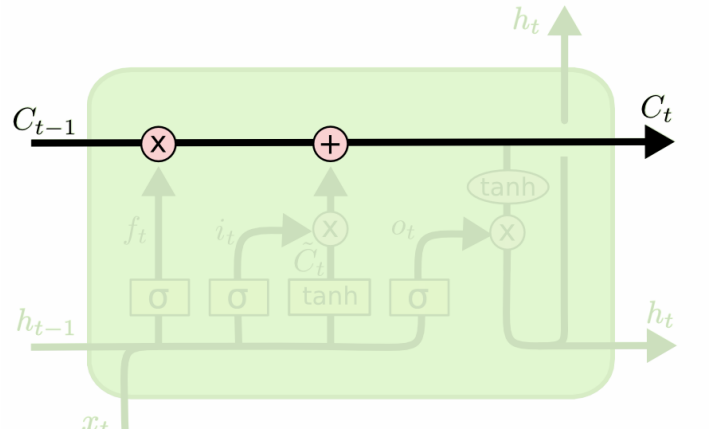

Figure 25: The Cell State Propagation Line

On its way, it interacts linearly with the four network layers of the current cell through some neural network components, which are called **gates**. Practically, these gates act as **filters** on the layers of the cell they are applied on, which means that they have the ability to optionally remove or add information to these layers. This is done through their sigmoid σ activation functions and a pointwise multiplication operation, as shown in figure 26[27].

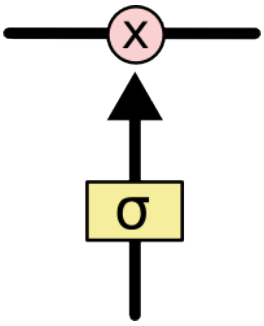

Figure 26: A LSTM Gate

As shown in figure 10 in the introduction section, the sigmoid layer outputs numbers between 0 and 1. These numbers correspond to the percentage of the content that should be let through. A value of 0 means the gate is blocking completely the content, while a value of 1 lets everything through. An LSTM has three of these gates, in order to control the cell and the hidden states. These are the **forget gate**, the **input gate** and the **output gate**, which are

discussed in more detail in the following sub-section.

2.5.2 Step-by-step Description of the Operation of the LSTM

The main element that distinguishes the RNNs from the other types of neural networks is their **internal memory**. In the case of LSTMs, as members of the RNN family, their internal memory corresponds to the **cell state** C_t of time-step t, while their output corresponds to their **hidden state** h_t. In general, a LSTM cell has three inputs and two outputs. The inputs are the cell state C_{t-1} and the hidden state h_{t-1} of the previous time step $t-1$ and the **input signal** x_t of the current time step t, while the outputs are the cell state C_t and the hidden state h_t of the current time step t.

The LSTM operation can be divided into three main steps, which are:

1. **The Forget Step**: In this step the LSTM cell has to decide what information is going to be thrown away or *forgotten* from the cell state C_{t-1}. This decision is taken by the **forget gate** f, which takes as inputs the concatenation of the hidden state h_{t-1} and the input signal x_t vectors and outputs a vector f_t with component values in the range of $[0, 1]$ for each component of the C_{t-1} vector. Values close to 0 mean "forget this component completely" and values close to 1 mean "keep this component completely". Taking into account the **weights** W_f and the **bias** b_f of the forget gate layer, we can compute the output f_t of the forget gate layer using equation 19.

$$f_t = \sigma\left[W_f\left(h_{t-1} + x_t\right) + b_f\right] \tag{19}$$

   A visual representation of the forget step is shown in figure 27[27].

2. **The Input Step**: In this step the LSTM cell has to decide what information is going to be stored into the cell state C_{t-1}. This decision is taken by the **input gate** i and an additional *tanh* **layer** $\hat{C}_t$. Both of them take as inputs the concatenation of the hidden state h_{t-1} and the input signal x_t vectors . The output of the input gate i_t decides which components of the C_{t-1} vector will be updated, while the *tanh* layer creates a vector of the new candidates $\hat{C}_t$ for the cell state C_t, as shown in figure 28[27].

   As shown in figure 10, the tanh function squashes its inputs in the range of $[-1, +1]$. That is a very important step because, depending on the problem we want to solve, the

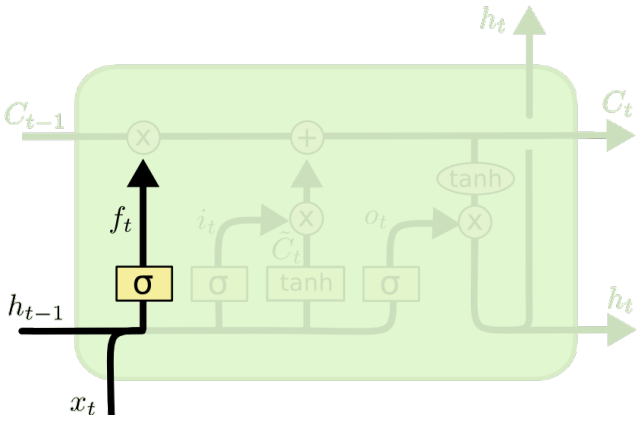

Figure 27: The LSTM Forget Gate Layer

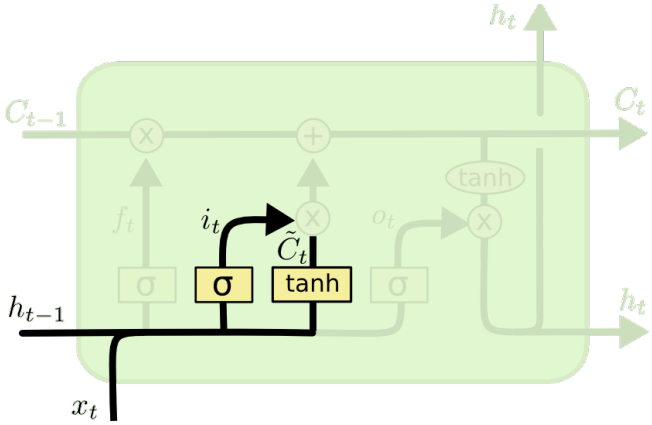

Figure 28: The LSTM Input Gate & Tanh Layers

values of the members of the input vector $\boldsymbol{h}_{t-1} + \boldsymbol{x}_t$ can become very big or very small, resulting in devastating consequences on the *stability* of the network.

The outputs of these layers with respect to the weights W_i, W_C and the biases b_i, b_C are computed using equations 20.

$$\begin{aligned} i_t &= \sigma\left[W_i\left(h_{t-1} + x_t\right) + b_i\right] \\ \hat{C}_t &= \tanh\left[W_C\left(h_{t-1} + x_t\right) + b_C\right] \end{aligned} \tag{20}$$

The outputs of the forget step and the input step are then combined using pointwise multiplication operations, in order to update the old cell state C_{t-1} and produce the new cell state C_t. First the output f_t of the forget gate is pointwise multiplied with the old cell state C_{t-1}. Then the outputs i_t and $\hat{C}_t$ of the input gate and the tanh layer are pointwise multiplied and then added to the old cell state C_{t-1}. All these operations create the new cell state C_t, as shown in equation 21.

$$C_t = f_t \times C_{t-1} + i_t \times \hat{C}_t \tag{21}$$

There are two important things to notice in the previous equation 21:

(a) As can be easily observed, equation 21 combines the previous cell state C_{t-1} with the candidate new cell state C_t, which means that this is the part of the LSTM that helps it learn the *relationship* between inputs separated by time.

(b) The output of the forget gate layer $f_t \times C_{t-1}$ is *added* to the output of the input layer $i_t \times \hat{C}_t$, instead of multiplied or mixed with it via weights and a sigmoid activation function as occurs in a standard RNN. This addition operation helps the LSTM to reduce substantially the *vanishing gradients* issue that standard RNNs face.

A visual representation of the computation of the new cell state C_t is shown in figure 29[27].

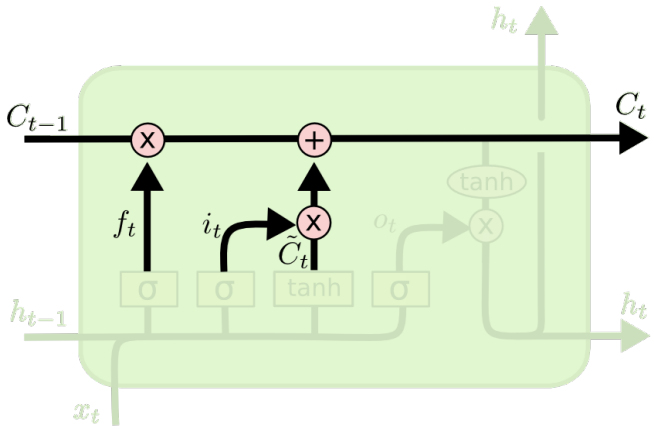

Figure 29: The Cell State C_t Computation

3. **The Output Step**: In this step the LSTM cell has to decide what information is going to output. This information will be the new hidden state h_t of the cell. Practically, the hidden state h_t will be a filtered version of the cell state C_t. The filter that is used for this purpose is the **output gate** o_t. First a copy of the cell state C_t is squashed in the range of $[-1, +1]$, using a tanh function, for the same reasons as in the input step. The result is then pointwise multiplied with the output of the output gate o_t in order to filter out the members of the vector of the cell state C_t, accordingly. All these operations create the hidden state h_t which consists the output of the LSTM cell of time step t. Defining the weights and the bias of the output gate o_t as W_o and b_o respectably, the hidden state h_t can be computed as shown in equations 22.

$$
\begin{aligned}
o_t &= \sigma\left[W_o\left(h_{t-1} + x_t\right) + b_o\right] \\
h_t &= o_t \times \tanh\left(C_t\right)
\end{aligned}
\tag{22}
$$

A visual representation of the computation of the new hidden state h_t is shown in figure 30[27].

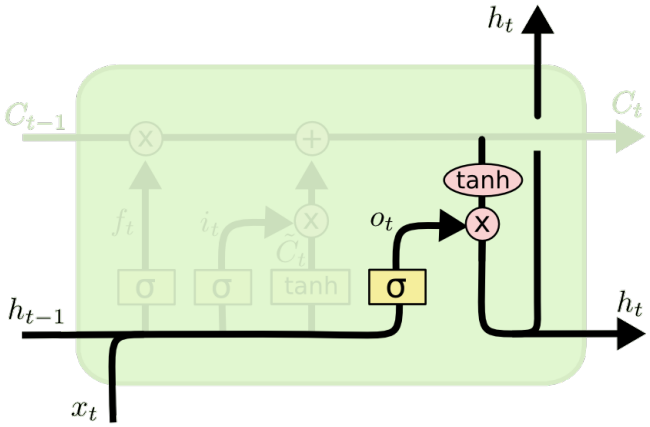

Figure 30: The Hidden State h_t Computation

Over the years, many variants of LSTM architectures have been defined. Some of them are:

1. The **Peephole-Connected LSTM**, where all the gate layers are allowed to modify the cell state.

2. The **Gate-Coupled LSTM**, where the forget and the input gate are coupled in order to decide what to forget and what new information should be added.

3. The **Gated Recurrent Unit (GRU)**, where the forget and input gates are combined into an **update gate**, and the cell and hidden states are merged into one "general" **hidden state** h_t, as shown in figure 31[27].

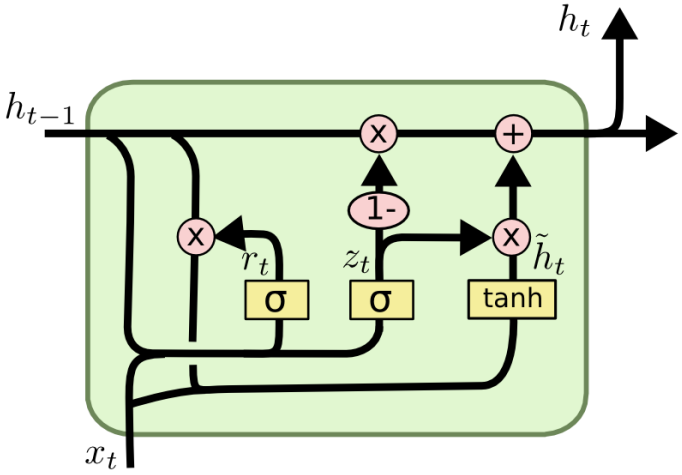

Figure 31: The Gated Recurrent Unit (GRU)

3 Reinforcement Learning Theory

As discussed briefly in the introductory section, **Reinforcement Learning (RL)** consists on its own one of the three major categories of Machine Learning, mostly because of the structure of its model being notably different from the other two. More specifically, the RL model along with the **Decision-Theoretic Planning (DTP)** deals with learning in *sequential decision making problems*, in which there is limited feedback. Both these models constitute the most important sub-categories of the **Markov Decision Processes (MDP)** for learning in stochastic domains[28].

3.1 Markov Decision Processes (MDP)

Conceptually, the **Markov Decision Processes (MDP)** are based on two main notions: The **environment** and the **agent**. Intuitively they can be defined as:

- **Environment**: The *world* in which an agent moves. It is an external system equipped with a scenario that an agent can interact with. Everything the agent cannot control is considered part of the environment. Typically, the environment consists the MDP system.
- **Agent**: A system that controls an environment by taking *actions* that can change the *state* of the environment. Typically, the agent is not a part of the MDP. It is the controlling system of the environment. In general, an agent could be a software entity, a mobile robot or an industrial controller.

In a MDP model an environment is modelled as a set of states. In order to control these states, an agent performs some actions. The goal is the agent to be able to control the environment in such a way that some performance quantity is maximized.

Formally, the definition of the MDP is:

---

*A **Markov Decision Process (MDP)** is a discrete time stochastic control process. It provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker.*

MDPs are equipped with a **discrete global clock** t, with $t = 1, 2, 3, ...$ .At each time step t, the MDP is in some state and the agent may choose any action that is available in that state.

[28] Conceptually, this section is based on chapter 1 of the book [2]

The MDP responds at the next time step by randomly moving into a new state, informing the agent of the new state and of the corresponding scalar **reward**, as shown in figure 32[29]. The probability that the MDP moves into its new state depends only on the current state and the agent's action, and is *conditionally independent* of all previous states and actions. In other words the state transitions of an MDP satisfies the **Markov Property**, the definition of which is:

---

*A stochastic process has the **Markov Property** if the conditional probability distribution of future states of the process (conditional on both past and present states) depends only upon the present state, and not on the sequence of events that preceded it. A process with this property is called a **Markov Process**[30].*

In Markovian environments the current state s gives enough information to the agent in order to decide for the optimal action. In other words, if the agent selects an action a with the environment being in a state s, the probability distribution over the next states remains the same as the last time the agent applied the same action a in the same state s.

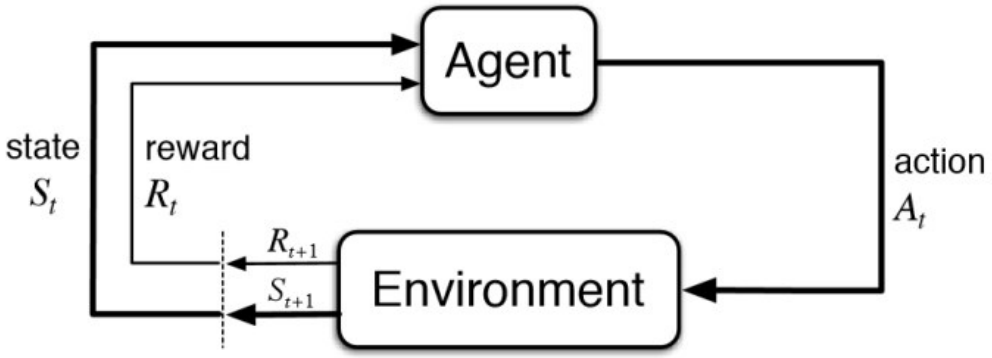

Figure 32: The Reinforcement Learning Process

Depending on the prior knowledge about a MDP environment, we can define two types of MDPs. The first is the **model-based MDP**, in which the full state transition dynamics and evaluative feedbacks, also known as rewards, are priorly known. Model-based MDPs use **dynamic programming (DP)** techniques for their learning. The second is the **model-free MDP**, in which no prior knowledge about the MDP is available. Therefore the algorithm interacts and experiments with the environment in order to gain knowledge on how to optimize

[29]Image taken from: `https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html`
[30]Definition taken from WikiPedia

its performance, based on the received rewards. Model-free MDPs use RL techniques for their learning, which have several advantages compared to other MDP techniques. The most important one is that they can cope well with *uncertainty* and *changing conditions*.

3.2 Formal MDP Framework

The components that consist a **Formal MDP Framework** are divided into three sub-categories: a set of MDPs, the policies and the optimality criteria, which are discussed in more detail in the following subsections.

3.2.1 Markov Decision Processes (MDPs)

A set of MDPs consists of states, actions, a transition function and a reward function. Their detailed definitions are:

1. **States**: A *state* is a concrete and unique characterization of all the important features of the problem that is modelled. Each state is represented by a distinct symbol s_t with respect to the time step t it is related to. The set of the states is defined as $S = \{s^1, s^2, ..., s^N\}$, where $|S| = N$.

2. **Actions**: An *action* can be used to control the state of the environment. Each action is also represented with a distinct symbol a_t with respect to the time step t it is related to. The set of actions is defined as $A = \{a^1, a^2, ..., a^K\}$, where $|A| = K$. In some environments there can be actions that cannot be applied to every state. The set of actions than can be applied to a particular state $s \in S$ is denoted as $A(s)$, where $A(s) \subseteq A$. In general, we assume that $A(s) = A$.

3. **The Transition Function**: When the agent takes an action $a \in A$ in a state $s \in S$ the environment makes a *transition* to a new state $s' \in S$ according to a probability distribution over the set of possible transitions. The *transition function* $T : S \times A \times S \longrightarrow [0, 1]$ defines a proper probability distribution over all the possible next states, which means that T must satisfy the property $\sum_{s' \in S} T(s, a, s') = 1$, where $T(s, a, s')$ is the probability of the environment ending up in state s' when an action a is applied to state s.

4. **Reward Function**: The *reward function* R specifies a scalar feedback signal for being in a state $R : S \longrightarrow \mathbb{R}$, for applying some action in a state $R : S \times A \longrightarrow \mathbb{R}$, or for a particular transition between states $R : S \times A \times S \longrightarrow \mathbb{R}$. The last one is usually used in model-free MDPs, like RL systems, where both the starting state s and the ending state s' are needed. In this case the reward function is denoted as $R(s, a, s')$. The reward

function gives to the agent the *direction* to which way the MDPs should be controlled.

The transition function T along with the reward function R define the **MDP model**. There are several types of systems that can be modelled by the MDP model. One typical case is the *episodic systems*, which are based on the notion of the **episode**. An episode is a sequence of states, actions and rewards, which starts at an initial state and ends with a terminal state. The goal for the agent is, starting at a random starting state, to manage to finish in the terminal state. For example, playing an entire game can be considered as an episode.

With all the elements of the MDP framework now available, we can redefine the MDP using a formal mathematical notation as follows:

---

*A **Markov Decision Process (MDP)** is a tuple $\langle S, A, T, R \rangle$ in which S is a finite set of states, A a finite set of actions, T a transition function defined as $T : S \times A \times S \longrightarrow [0, 1]$ and R a reward function defined as $R : S \times A \times S \longrightarrow \mathbb{R}$.*

3.2.2 Policies

Theoretically speaking, a **policy** is a mapping from the set of states S to an optimal action $a \in A$, based on decision theoretic measures of *optimality*, with respect to some goal to be optimized. In the case of MDPs, we can use the MDP framework elements to form a formal mathematical definition, as follows:

---

*Given an MDP $\langle S, A, T, R \rangle$, a **policy** is a computable function π that outputs an action $a \in A(s)$ for each state $s \in S$.*

A **deterministic policy** $\pi(s)$ is a function defined as $\pi : S \longrightarrow A$ and a **stochastic policy** $\pi(s, a)$ is a function defined as $\pi : S \times A \longrightarrow [0, 1]$, where $\pi(s, a) \geq 0$ and $\sum_{a \in A} \pi(s, a) = 1$ for each state $s \in S$. The policy is a part of the agent which controls an environment modelled as an MDP.

3.2.3 Optimality Criteria

The **optimality criteria** determine what is actually being optimized. In the case of MDPs the goal of learning is to gather as many rewards as possible. If we were concerned only for the **immediate reward** r_t, a simple optimality criterion would be the optimization of the **expected reward** $\mathbb{E}[r_t]$. However, in MDPs we also take into account the future rewards that are related to the current state s_t, in order to decide how to act now.

Considering episodic MDPs, there are three models of optimality, which are:

1. The **finite horizon model**, in which each episode consists of a finite number of steps h, or a finite horizon of length h. The agent optimizes its *expected reward* $\mathbb{E}\left[\sum_{t=0}^{h} r_t\right]$ over this horizon.

2. The **infinite horizon model**, in which each episode has an end but has also an arbitrary length. In this case the agent optimizes the *expected discounted reward* $\mathbb{E}\left[\sum_{t=0}^{+\infty} \gamma^t r_t\right]$, using a *discount factor* γ with $0 < \gamma < 1$. The discount factor ensures that the theoretically infinite sum converges to a finite value and also declares that the earlier rewards have more weight compared to the ones received far away in time. Most RL algorithms use this model.

3. The **average reward model**, in which the agent optimizes its *expected average reward* $\lim_{h\to\infty} \mathbb{E}\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right]$. The major disadvantage of this approach is that we cannot distinguish between two policies in which the one receives big amounts of rewards earlier than the other one, resulting in a more ambiguous route to the optimal goal.

3.3 Value Functions and Bellman Equations

In this section we define the **value functions**, which can link the optimality criteria to the policies. A value function describes how good a state is for an agent to be in, or how good a state is for an agent to make a certain action in and they are defined under some particular policies. Multiple policies can have the same value function, but for a given policy the corresponding value function is always unique. Some of the most commonly used value functions are described in the following subsections.

3.3.1 The State Value Function $V^\pi(s)$

Considering an infinite horizon model, the value of the **state value function** $V : S \longrightarrow \mathbb{R}$ of a state s under policy π is the expected return (reward) of the state s, as expressed by equation 23.

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left\{\sum_{k=0}^{+\infty} \gamma^{k} r_{t+k} \,|s_t = s\right\} \tag{23}$$

A useful property of all value functions is that they can be also written in a recursive form, which are called **Bellman Equations**. Taking this into account, equation 23 can be rewritten with the form of equation 24.

$$V^{\pi}(s) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi}(s')\right] \tag{24}$$

Equation 24 denotes that the expected value of the state value function $V^{\pi}(s)$ is defined in terms of the immediate reward and all the possible next states, weighted by their transition probabilities.

In order for an MDP to find the optimal policy π^*, it must find the policy that receives the most cumulative reward. It can be proven that the optimal solution of the state value function V^{π^*} is given by equation 25, and is known as the **Bellman optimality equation**.

$$V^{\pi^*}(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi^*}(s')\right] \tag{25}$$

Given the Bellman optimality equation, we can now derive the optimal policy, as shown in equation 26. The optimal policy is also known as the **greedy policy** because it greedily selects the best action using the optimal value of the state value function.

$$\pi^*(s) = \arg\max_{a} \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi^*}(s')\right] \tag{26}$$

3.3.2 The State-Action Value Function $Q^{\pi}(s, a)$

In a similar way as the state value function, we can define the **state-action value function** or **Q-function** $Q : S \times A \longrightarrow \mathbb{R}$ as the expected return (reward) of the state s if the action a is taken under policy π, which is expressed by equation 27.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left\{\sum_{k=0}^{+\infty} \gamma^{k} r_{t+k} \,|s_t = s,\ a_t = a\right\} \tag{27}$$

In this case, the Bellman equation $Q^{\pi}(s, a)$, the optimal state-action-value function Q^{π^*} and the optimal policy $\pi^*(s)$ are given by equations 28, 29 and 30 respectably.

$$Q^{\pi}(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma Q^{\pi}(s') \right] \tag{28}$$

$$Q^{\pi^*}(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q^{\pi^*}(s') \right] \tag{29}$$

$$\pi^*(s) = \arg\max_{a} Q^{\pi^*}(s,a) \tag{30}$$

The main advantage Q-functions have is that they are not depended on the weighted sum over the possible alternatives that use the transition function. In other words, in order to compute an optimal action in a state there is no need to look into future steps. That is the reason model-free MDPs use Q-functions instead of V-functions.

Equation 30 defines the greedy policy, based on the Q-function. This is the action that has the highest value based on all possible next states that are related to this action.

The relation between the optimal values of the Q-function and the V-function is shown in equations 31 and 32.

$$Q^{\pi^*}(s,a) = \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi^*}(s') \right] \tag{31}$$

$$V^*(s) = \max_{a} Q^*(s,a) \tag{32}$$

3.4 Markov Decision Process (MDP) Learning

Based on the definitions of the previous subsections, we can define the general procedure of how to solve a MDP problem. By solving a MDP problem we are actually trying to compute an optimal policy π^*, which heavily depends on the model type of the MDP: *model-based* or *model-free*.

In order to solve model-based MDPs we use DP. Assuming that the model is known, we can compute value functions and policies based on the Bellman equation, using iterative procedures. On the other hand, model-free MDPs are solved via RL methods, which rely on the interaction with the environment. Being completely agnostic about the structure of the model,

the agent has to explore the environment to collect information about it.

Both solutions are based on the generalized **policy iteration (GPI) principle**, which consists of two interaction processes, the **policy evaluation** and the **policy improvement** processes, as shown in figure 33[31].

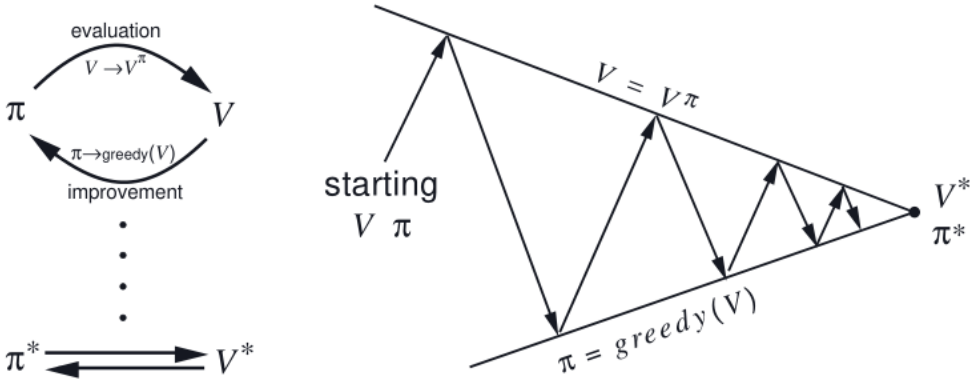

Figure 33: The Generalized Policy Iteration (GPI)

1. The **policy evaluation process** computes the value V^π of the current policy π and collects information about the policy.

2. The **policy improvement process**, evaluates the values of the actions in every state, in order to find possible improvements and computes an improved policy π', using the current policy π and the value of the current policy V^π.

In other words, we have a policy π that computes the value of a value function V^π using a learning procedure and in turn we have a value function that can be used to improve the policy in order to make good actions.

3.5 Model-Free MDP Solution Techniques

In this section we focus on model-free MDP learning. As discussed earlier, model-free MDPs use RL techniques for their learning, which are based on approximation and incomplete information for the MDP model. They estimate values for actions without estimating the MDP model. This type of RL is also called **direct RL**, in contrast to the **indirect RL**, which uses DP methods for its learning.

[31]Image taken from book [3]

An additional consideration about RL is how to deal with the **temporal credit assignment** problem, which is based on the fact that the value of the earlier actions are completely determined by the distant goal the agent is trying to achieve, resulting in *delayed rewards* for these actions. In RL we deal with this problem by adjusting the estimated value of a state based on the immediate reward and the estimated discounted value of the next state. This process is called **temporal difference (TD) learning** and the class of RL algorithms that interacts with the environment and update their estimates after each step is called **online**.

Because of the fact that the model-free MDPs use **RL algorithms** for their learning, these systems are very commonly called **RL systems** and this is the terminology we will be using in the rest of this text.

3.5.1 Exploration vs Exploitation

A RL algorithm needs to *explore* the environment, try various actions and evaluate the results, because the MDP model is unknown. At the same time, it needs to *exploit* the gained knowledge of the environment in order to perform well and gain a lot of rewards. The balance between these two actions is known as the **exploration vs exploitation problem**. In general, there are two main strategies for achieving this balance, which are:

1. **ϵ-greedy Exploration Policy**: This is the most basic strategy. The agent takes the currently best action with probability $(1 - \epsilon)$ or an other available random action with probability ϵ.

2. **Boltzmann** or **Softmax Exploration Policy**: The agent takes the action randomly based on the probabilities derived from the weighted Q-values, according to equation 33, in which $P(a_n)$ is the probability of selecting the action a_n and T the temperature parameter. High values of T result in a more random selection strategy, as opposed to lower values of T that make the strategy more greedy.

$$P(a_N) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_i e^{\frac{Q(s,a_i)}{T}}} \tag{33}$$

3.5.2 Temporal Difference (TD) Learning

As discussed earlier, TD learning algorithms learn estimates of values based on other estimates, a procedure that is also called **bootstrapping**. A new action in an environment produces a *learning example*, which can be used to re-evaluate the value of a state or state-action function in relation to the immediate reward and the estimate value of the next state or the next state-action pair.

The family of the TD learning algorithms have many members, the most common of which are:

1. **TD(0)**: This algorithm estimates the value of a state function $V^{\pi}(s)$, using the update rule of equation 34.

$$V_{t+1}^{\pi}(s) := V_t^{\pi}(s) + \alpha \left[r + \gamma V_t^{\pi}(s') - V_t^{\pi}(s) \right] \tag{34}$$

   where α is the **learning rate**, with $\alpha \in (0, 1)$. The agent makes an action a and receives a reward r, while the system transitions from state s to the state s'. In this algorithm only the value of the immediate successor $V_t^{\pi}(s')$ is used, instead of the weighted average of all possible next states.

2. **Q-Learning**: This algorithm estimates incrementally the Q-values for the corresponding actions, based on the Q-value function of the agent, using the update rule of equation 35.

$$Q_{t+1}^{\pi}(s_t, a_t) := Q_t^{\pi}(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a_{t+1}} Q_t^{\pi}(s_{t+1}, a_{t+1}) - Q_t^{\pi}(s_t, a_t) \right] \tag{35}$$

   where α is the **learning rate**, with $\alpha \in (0, 1)$. The agent makes an action a_t and receives a reward r_t, while the environment transitions from state s_t to the state s_{t+1}. The update is applied to the value $Q_t^{\pi}(s_t, a_t)$. Q-Learning is an **off-policy** learning algorithm, which means it will finally converge to the optimal policy π^* regardless of the initial exploration policy that is being followed. The necessary conditions for this convergence is that each state-action pair is visited infinite (theoretically) number of times and the learning parameter α is decreased appropriately through the process.

3. **SARSA**: This algorithm is the **on-policy** version of the Q-learning algorithm, which means that the algorithm learns the Q-value function of the policy the agent is currently executing. The update rule for the SARSA algorithm is shown in equation 36.

$$Q^{\pi}_{t+1}(s_t, a_t) := Q^{\pi}_t(s_t, a_t) + \alpha \left[r_t + \gamma Q^{\pi}_t(s_{t+1}, a_{t+1}) - Q^{\pi}_t(s_t, a_t) \right] \tag{36}$$

where the agent makes the action a_{t+1} according to the current policy in the state s_{t+1}. The α parameter is the **learning rate**, with $\alpha \in (0, 1)$. In the limit, the SARSA algorithm converges to the optimal policy π^* if each state-action pair is visited infinite number of times.

4. **Actor-Critic Learning**: This is another on-policy learning algorithm, but this time the policy is kept *separated* from the value function. The policy is called the **actor** and the value function the **critic**. After each interaction of the agent with the environment, the critic evaluates the action made by the actor, using the TD-error function that is shown in equation 37.

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{37}$$

The sign of the TD-error δ_t defines if the weight of the action selected by the actor in the state s_t must be increased or decreased. For this purpose, the **preference** $p(s_t, a_t)$ for an action a_t in some state s_t can be modified using the update rule that is shown in equation 38.

$$p(s_t, a_t) := p(s_t, a_t) + \beta \delta_t \tag{38}$$

A visual representation of the actor-critic learning algorithm is shown in figure 34[32].

Having separate policy and value function representations has many advantages in the learning procedure. The most important of them are:

(a) Actor-critic algorithm does not take into account all the Q-values of each action in order to select one of them. This is a big advantage in environments where the size $|A|$ of the set of the available actions is big, or the action space is continuous.

(b) Actor-critic algorithm can learn stochastic policies effectively.

[32]Image taken from: `https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html`

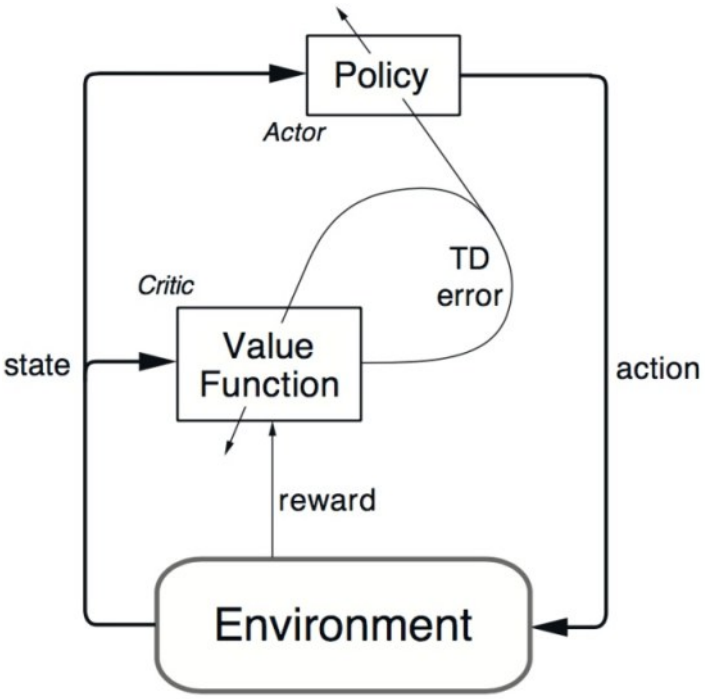

Figure 34: The Actor-Critic Learning Algorithm

3.6 Asynchronous Advantage Actor-Critic Algorithm (A3C)

In 2016, Google's DeepMind Research Team published the **Asynchronous Advantage Actor-Critic (A3C)**[7] RL algorithm, which is the RL algorithm that has been used in this assignment. The A3C algorithm is presented in more detail in the following subsections.

3.6.1 The Advantage Function $A(s, a)$

In value-based RL methods, the state-action value function (or Q-function) $Q^{\pi}(s, a; \theta)$ is approximated, usually through a NN, where θ are the parameters being optimized. In the previous subsections we discussed a variety of RL algorithms that can be used to update these parameters, such as the Q-learning algorithm. This algorithm aims to directly approximate the optimal Q-function $Q^{\pi^*}(s, a)$, assuming that $Q^{\pi^*}(s, a) \approx Q(s, a; \theta)$. This algorithm is also called **1-step Q-learning** because it updates the Q-function $Q^{\pi}(s, a; \theta)$ using the 1-step return $r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta)$. This approach can make the learning process slow because it takes many iterations for a reward to propagate back to the relevant preceding state-action pairs.

We can deal with this problem by using an **n-step Q-learning** algorithm, in which the Q-function $Q^{\pi}(s, a; \theta)$ is updated using the *n-step return*:

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \max_{a_{t+n}} \gamma^n Q(s_{t+n}, a_{t+n})$$

which directly affects all the Q-values of the n preceding state-action pairs. This makes the

propagation procedure of the rewards to the relevant state-action pairs much more efficient.

On the other hand, during a RL process a policy $\pi(s|a;\theta)$ can be improved by using a learning value function V^{π}, as discussed previously. Typically, this is done by approximating the gradient ascent on the expected return $\mathbb{E}[R_t]$. It can be proven that an unbiased estimate of $\nabla_\theta \mathbb{E}[R_t]$ with respect to the parameters θ is the gradient $\nabla_\theta \log \pi(a_t|s_t;\theta) R_t$. We can reduce the variance of this estimate by subtracting a learned estimate b_t of the value function of a state $V^{\pi}(s_t)$, known as baseline, which results in the gradient $\nabla_\theta \log \pi(a_t|s_t;\theta)\,(R_t - b_t)$. The quantity $R_t - b_t$ we use to scale the policy gradient can be seen as an estimate of the **advantage** of the action a_t in state s_t, leading to the definition of the **Advantage Function** $A^{\pi}(a_t, s_t)$ under policy π that is given by equation 39, because R_t is an estimate of the Q-function $Q^{\pi}(a_t, s_t)$ and b_t is an estimate of the value function $V^{\pi}(s_t)$.

$$A^{\pi}(a_t, s_t) = Q^{\pi}(a_t, s_t) - V^{\pi}(s_t) \tag{39}$$

This approach can be seen as an **actor-critic architecture**, with the policy π being the actor and the estimate b_t of the value function $V^{\pi}(s_t)$ being the critic.

3.6.2 Asynchronous Reinforcement Learning Framework

In Asynchronous RL we use many asynchronous learning processes, also known as **actor-learners**. Actor-learners run on a separate thread, forked by the OS of a single machine that is equipped with several multi-core CPUs. Each actor-learner consists of a copy of the RL algorithm and a copy of the environment it interacts with. There is also a separate **shared model** of the environment, with which all actor-learners are synchronized and an **optimizer**, which is a process responsible for the updating procedure of the parameters θ of the shared model. The overall learning loop goes as follows:

1. At the beginning of each episode, each actor-learner copies the parameters θ of the shared model into its own environment's parameters θ'.

2. Each actor-learner executes the RL algorithm for a number of t_{max} time steps or less if the end of the episode has reached.

3. Each actor-learner informs the optimizer for its newly learned parameters θ' and continues the execution of the RL algorithm.

4. For each update of any of the actor-learners, the optimizer updates the parameters θ of the shared model accordingly.

A visual representation of the whole process is shown in figure 35.

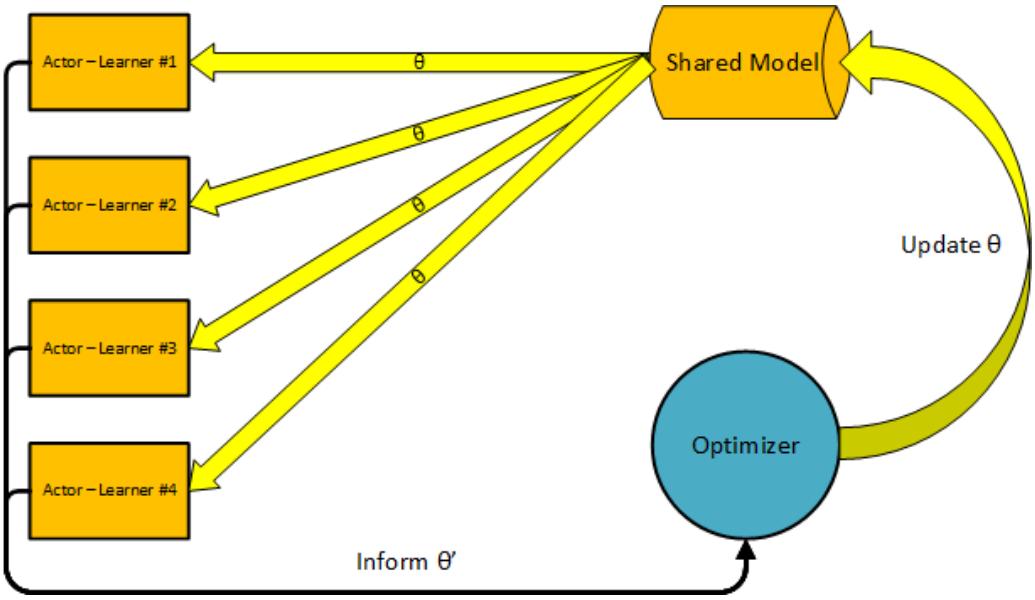

Figure 35: The Asynchronous RL Framework

Using an asynchronous RL procedure has many advantages, the most important of which are:

1. We obtain a substantial reduction in training time, which most of the times is linear in the number N of the actor-learners, given that each actor-learner occupies a single CPU core.

2. Multiple actor-learners running asynchronously are likely to be exploring different parts of the environment, which means that an actor-learner may be exploring a certain part that had been previously explored by an other actor-learner, resulting to a much more efficient learning.

3. The overall online updates that are being made to the parameters of the shared model by the asynchronous actor-learners are likely to be *less correlated* compared to the online updates applied by a single actor-learner.

3.6.3 Asynchronous Advantage Actor-Critic Algorithm (A3C)

Combining all the information of the previous related subsections together, we can define the **Asynchronous Advantage Actor-Critic Algorithm (A3C)**[7]. The A3C algorithm learns a policy $\pi(a_t|s_t;\theta)$, which is the *actor* and an estimate of the state value function $V^{\pi}(s_t;\theta_v)$, which is the *critic*, where θ and θ_v are the parameters of the actor and the critic respectively. It also uses an *n-step return* learning approach, in order to update the policy and the value function, which are updated every t_{max} time steps or less if a terminal state is reached. The

update of the policy is given by the expression:

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta') A^{\pi}(a_t, s_t; \theta, \theta_v)$$

where $A^{\pi}(a_t, s_t; \theta, \theta_v)$ is an estimate of the advantage function, which is given by the expression:

$$\sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V^{\pi}(s_{t+k}; \theta_v) - V^{\pi}(s_t; \theta_v)$$

and $0 < k < t_{max}$. The pseudo-code of the A3C algorithm is presented in figure 36[33].

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 36: The A3C Pseudo-code

Although the parameters θ of the policy $\pi(a_t|s_t;\theta)$ and θ_v of the state value function $V^{\pi}(s_t;\theta_v)$ are in general being separate, practically we always share some or all of them. For example, we can have a softmax output for the policy $\pi(a_t|s_t;\theta)$ and one linear output for the state value function $V^{\pi}(s_t;\theta_v)$, with all hidden layers of the NN being shared, as shown in figure 37.

Finally, Google researchers experimentally found that adding the **entropy** H of the policy π to its gradient function improved the exploration because it helped the algorithm not to

[33]Image taken from paper [7]

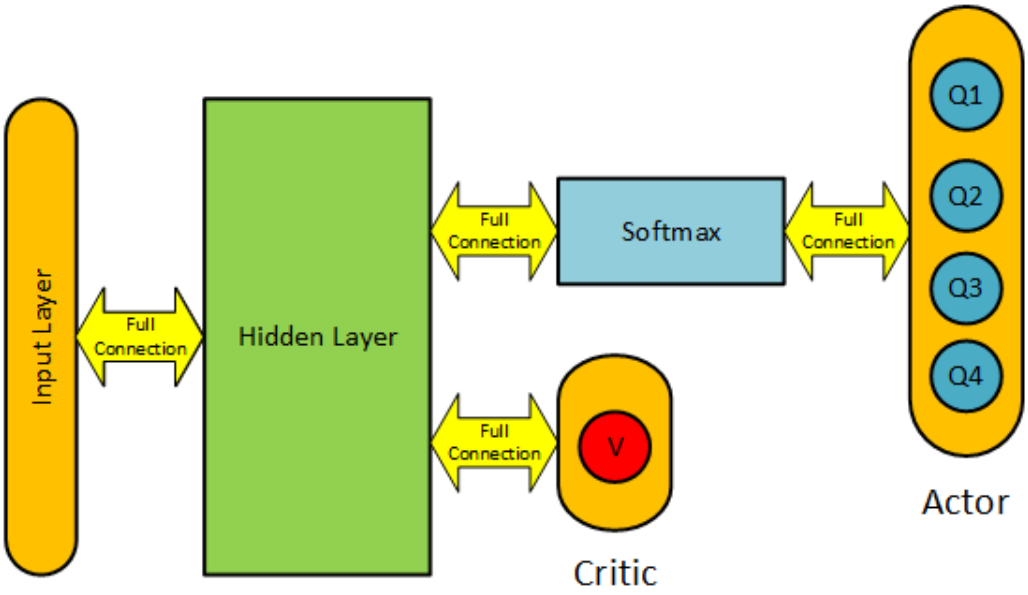

Figure 37: The Actor-Critic Architecture

converge in suboptimal policies. The gradient of the final objective function which includes the entropy regularization is given by expression 40, where β is a hyper-parameter that controls the strength of the entropy regularization term.

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta')A^{\pi}(a_t, s_t;\theta,\theta_v) + \beta\nabla_{\theta'}(H(\pi(s_t;\theta'))) \tag{40}$$

4 Assignment: Atari Breakout© Game

4.1 Introduction

In this assignment we studied the case of a computer, learning how to play the video game **Atari Breakout©**. In this game the player controls a pad at the bottom of the screen in order to force a tiny moving ball smash the bricks of a brick wall, which is located on the north part of the screen, as shown in figure 38. For every smashed brick the player collects points and the ultimate goal is to earn as much points as possible. If the player manages to clear out all the bricks, he wins the level and moves on to a next, more challenging level. If the player misses the ball, it passes through to the bottom of the screen and a "life" is lost. There are five lives available in the game. The game finishes as soon as all lives are lost.

Figure 38: Atari Breakout© Game

4.2 Assignment Implementation

An arcade game is a text-book example of a Reinforcement Learning application. We trained a model which was a combination of two types of Neural Networks: A CNN along with a LSTM, and named it as the **breakout model** for practical reasons. The breakout model is visually described in figure 39. The Reinforcement Learning algorithm was Google DeepMind's **Asynchronous Advantage Actor – Critic (A3C)**[7], which means that we used *multiple agents* during the training procedure. In addition, we needed a *shared model* in order to give the

agents the ability to share their knowledge. Each agent along with the shared model consisted of a breakout model.

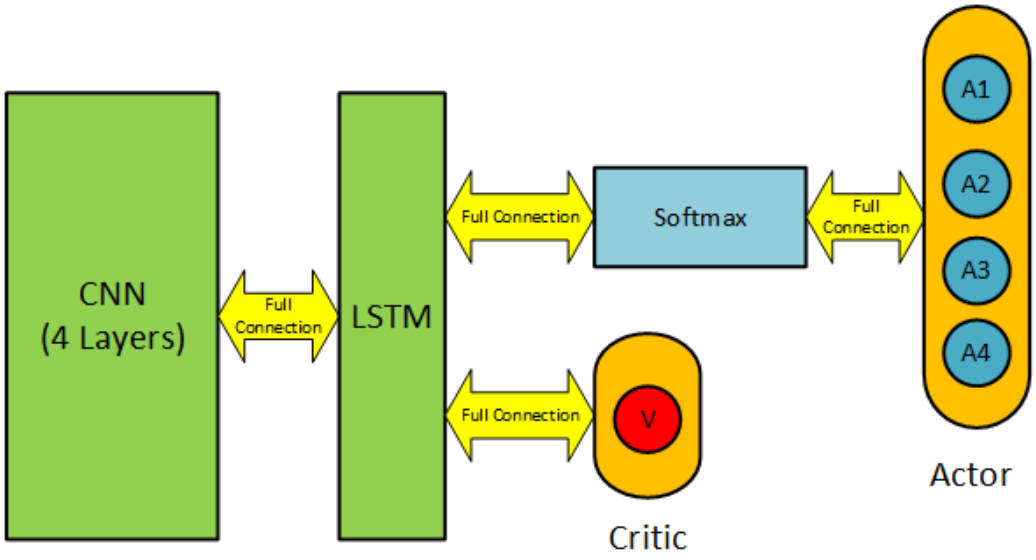

Figure 39: The Breakout Model

The size of the **action space** that is available in this game is four (4), which means that there are four possible actions an agent (actor) can choose from. These actions are coded by the OpenAI Gym platform with the integers of $\{0, 1, 2, 3\}$. The actions along with their corresponding integer codes and interpretations are:

- **NOOP - 0**: Do not move.
- **FIRE - 1**: Press fire after a "life" is lost in order to continue the game.
- **LEFT - 2**: Move the pad left.
- **RIGHT - 3**: Move the pad right.

The screen resolution of the platform of the game, after the removal of the black vertical side stripes, was (160×80) with color depth of 3 (RGB) and 32 colors. In order to accelerate the training process, each frame was pre-processed in the following way:

1. The black vertical side straps of the frame were cropped out. In this way we removed the parts of the screen that did not contain any information, reducing at the same time the total amount of data that were processed for each frame.
2. The resolution of the frame was reduced from (160×80) to (80×80).
3. The color depth of the frame was reduced from 3 to 1, by calculating the mean value of the three RGB values, which resulted in a gray-scale colored frame.

These three pre-processing steps resulted in a significant speed-up of the training process.

4.2.1 CNN layer

The CNN is an image processing network and therefore gives the agent the ability to see the environment. During the training process the CNN learns to identify the important objects in each frame and passes this information to the LSTM layer.

The CNN we used in our assignment consisted of four identical **combo-layers**. Each combo-layer was a convolutional / non-linear / pooling combination of layers. From left to right, these combo-layers were capable of identifying objects of a smaller and smaller size. That was a critical architectural decision because, as we can see in figure 38 the size of the objects varies. For example, it was the last (far right) combo-level that was able to identify and learn the shape of the game's tiny ball.

4.2.2 LSTM layer

Regarding the fully-connected layer of the CNN, we used a LSTM. Experiments suggest that using LSTMs in game-learning problems result in reduced learning times and in much more efficient learning. The reason behind these ascertainments is that game-learning, in general, depends heavily on time-related series of actions, in which LSTMs excel. For example, two consecutive game frames include two consecutive positions of the ball, which in turn suggest a particular route for the ball, resulting in a much more effective learning.

4.2.3 Optimizer

An **Optimizer** is an algorithm that updates the weights on stochastic gradient-based NNs, in order to make them learn. In this implementation we used a modified version of the **ADAM optimizer**[8]. The applied modification made ADAM capable of working in multi-threading environments, which is critical for the parallel nature of the A3C algorithm.

4.2.4 Technical Details

Python 3 programming language was used throughout the whole concept because nowadays Python and its Machine Learning open source libraries are the de-facto standard to any Machine Learning application out there. As environment for the model, we used the **OpenAI Gym**[11] platform which consists a great abstraction of the game Breakout, because of its simplistic approach on interacting with it via the Gym API for the Python 3 programming language. The software we used in a little more more detail is shown in table 1.

Operating System	`Ubuntu 18.04 LTS`
Environment	`OpenAI Gym 0.10.9`
Programming Language	`Python 3.6.7`

Python Libraries	`NumPy 1.15.4`
	`SciPy 1.1.0`
	`matplotlib 2.1.1`
	`pandas 0.22.0`
	`OpenCV 3.2.0`
	`PyTorch 0.4.1`

Table 1: Software used in the Assignment

The computer we used for the model training procedure was a PC with a 6-core Intel© i5-8400 @ 2.80Ghz CPU and 8GB DDR4 of RAM.

5 Experiments and Results

5.1 Basic Setup

The first factor that we had to set, in relation to our hardware resources, was the **number of the asynchronous agents**. In order not to overload our machine we used 16 **training agents**, which is a good compromise between the demand for resources and training efficiency. Additionally, we used a **testing agent**, which was continuously testing the breakout model being trained, in order to collect and store the scores it achieved in each episode. In this way we were able to study the quality and speed of the model, using different settings for its main factors.

The two main factors related to the quality and speed of the learning procedure of the breakout model are the **learning rate** η and the **number of the steps** n of the n-step Advantage Function. For this reason, our experiments were based on these two factors, selecting the values $\{10^{-3}, 10^{-4}, 10^{-5}\}$ for the learning rate η and the values $\{10, 20, 50\}$ for the number of steps n.

5.2 Experiments with the Learning Rate, η

The term **quality** of the training procedure is related to the highest score the algorithm can achieve in the game. We calculated two separate score values for each episode, the **raw score** and the **100-period moving average of the scores, MA(100)** and we collected the maximum of these two factors of all the episodes. With this computational approach we achieved a more generalized understanding of the intuitive meaning of the highest scores.

In order to study the learning rate η more thoroughly, we first ran our tests for its three different values, keeping at the same time the number of steps n constant ($n = 20$). The maximum scores we achieved for each of the three learning rates η are shown in table 2.

	$\eta = 10^{-3}$	$\eta = 10^{-4}$	$\eta = 10^{-5}$
max raw score	11.00	864.00	815.00
max MA(100) score	6.73	450.36	400.20

Table 2: Breakout scores with respect to the learning rate η

A visual representation of the three training procedures is shown in the graphs of figures 40, 41, 42.

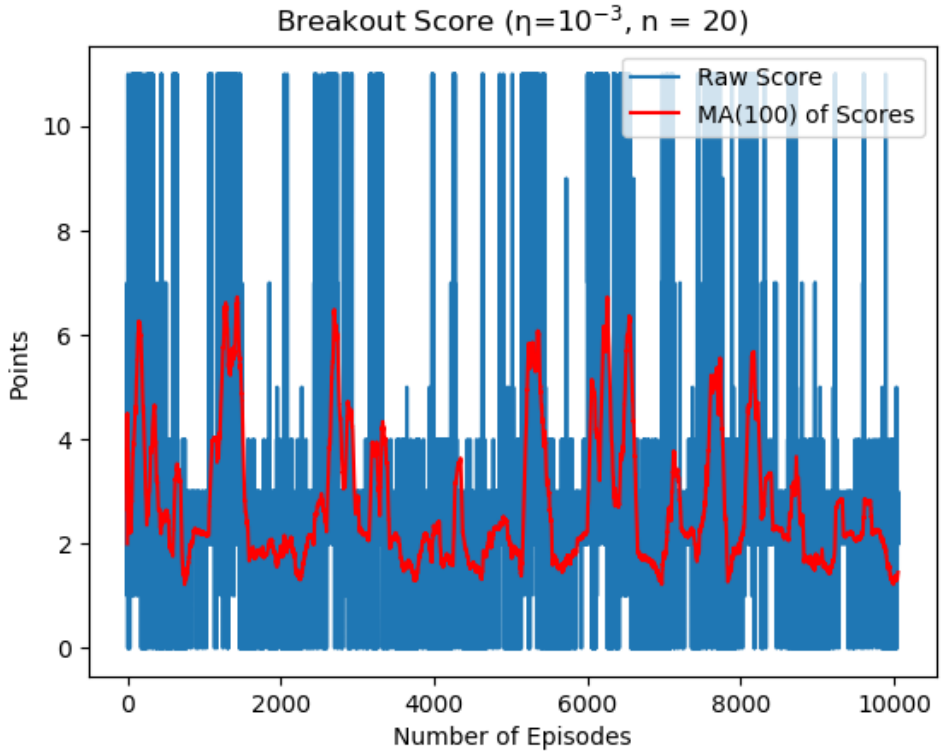

Figure 40: Breakout Scores for $\eta = 10^{-3}$ and $n = 20$

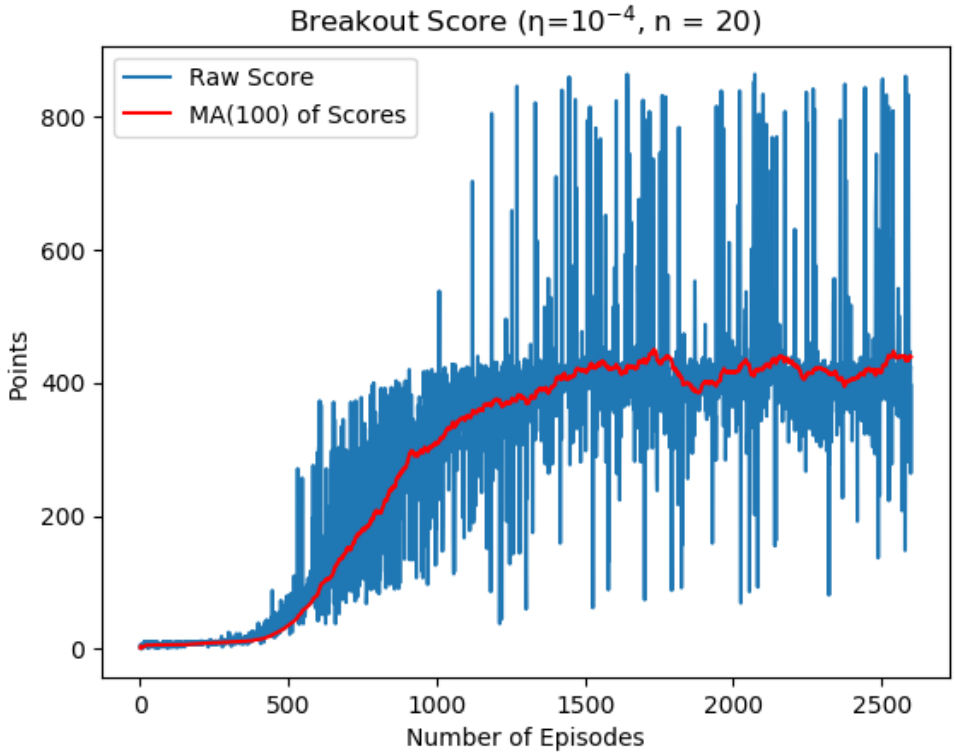

Figure 41: Breakout Scores for $\eta = 10^{-4}$ and $n = 20$

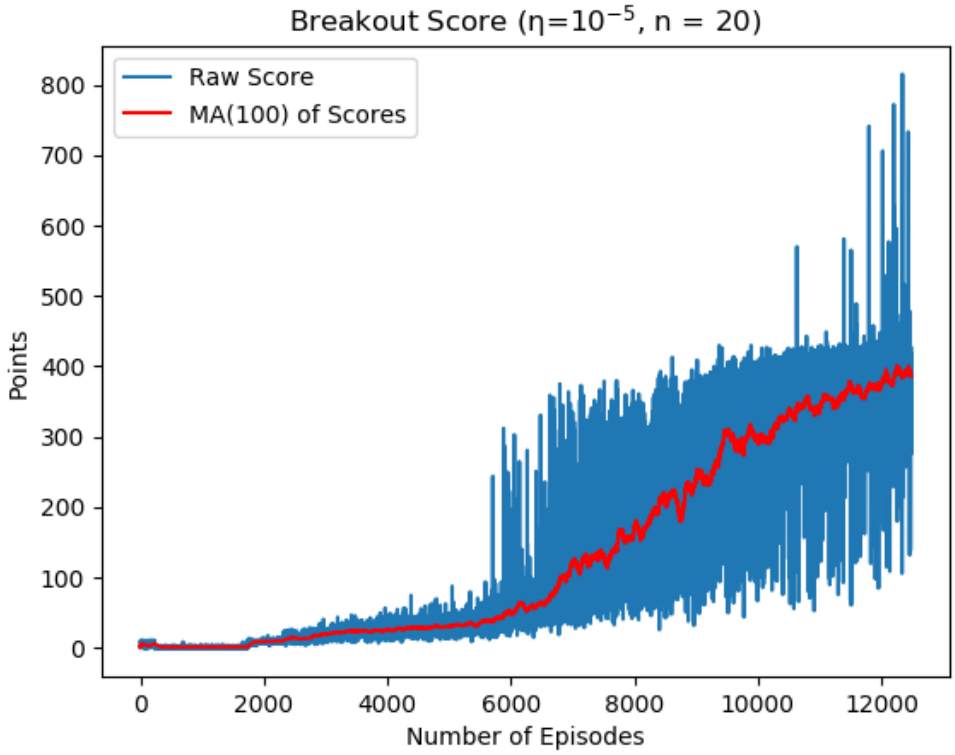

Figure 42: Breakout Scores for $\eta = 10^{-5}$ and $n = 20$

After studying the previous results and graphs regarding the learning rate values, we ended up in the following conclusions:

1. $\boldsymbol{\eta = 10^{-3}}$: It is more than obvious that in this case the breakout model did not learn anything at all. The graph of figure 40 is just noise, made from random movements of the agent. This result was somewhat expected and is due to the **global minimum overshooting phenomenon** as can be seen in figure 43[34]. Choosing a large learning rate η results in large corrections δw of the error function $J(w)$, which in turn results in overshooting the global minimum and finally in increasing the value of the error function instead of decreasing it. The final outcome is a back and forth adjustment process of the value of the error function, and therefore noise.

2. $\boldsymbol{\eta = 10^{-4}}$: In this case the training procedure was fast and efficient. After ≈ 500 episodes the scores started to grow almost linearly and after ≈ 1750 episodes the MA(100) reached the 450 level. We kept the training procedure going for a total of ≈ 2500 episodes, just for confirmation reasons.

3. $\boldsymbol{\eta = 10^{-5}}$: In this case the training procedure was successful but very slow, as expected. After ≈ 2000 episodes the scores started to grow almost linearly but at a very slow pace, and only after $\approx$ 6000 episodes the scores grew a little more rapidly. After $\approx$ 12000

[34]Image taken from the book [1]

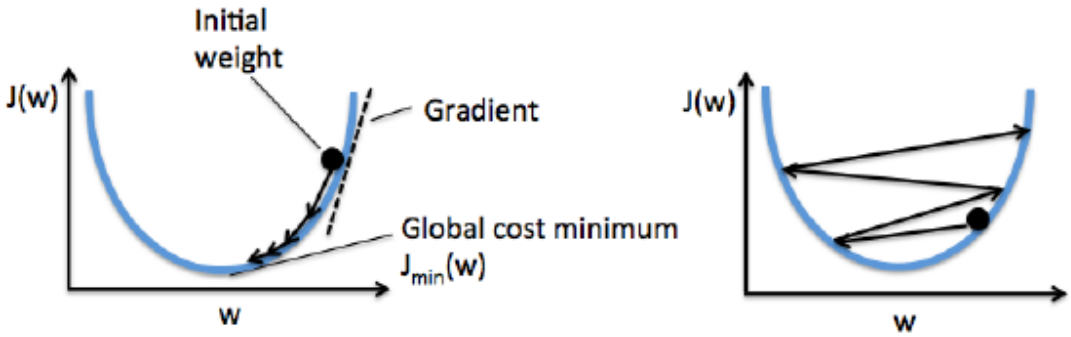

Figure 43: Successful Gradient Descent (left) & Global Minimum Overshooting (right)

episodes the MA(100) reached the 400 level, maybe having the potential to move even higher, but we stopped the training as the whole procedure would require weeks of training to finally converge. Choosing a small learning rate η results in tiny corrections δw of the error function $J(w)$, which in turn results in a slow converging procedure. Additionally, with a small learning rate η we are risking the case of the agent getting trapped in a possible **local minimum** of the error function $J(w)$. That would reduce the quality of training, which means lower scores. Fortunately, in the case of the A3C RL algorithm the possibility of being trapped in a local minimum is significantly reduced due to the asynchronous nature of this algorithm. In case an agent gets trapped in a local minimum, the sharing knowledge of all the other asynchronous agents would most probably help it overcome the issue, as the probability of all agents being trapped at the same time in a local minimum is extremely small.

In conclusion, the value $\eta = 10^{-4}$ of the learning rate produces the best balanced results and therefore consists an optimal generalization of the solution of the problem.

5.3 Experiments with the Number of Steps, n

As discussed in the previous subsection, the term **speed** of the learning procedure is related to the number of episodes the algorithm had to be trained with in order to achieve the highest score. In addition to the learning rate η, the speed of the learning procedure also depends on the number of steps n an agents makes in the environment before that cumulative experience is transferred to the optimizer.

In order to study the impact the number of steps n have to the learning process, we ran our tests for its three different values $\{10, 20, 50\}$ keeping at the same time the value of the learning rate constant ($\eta = 10^{-4}$). The maximum scores we achieved for each of the three values of the number of steps n are shown in table 3. The results of $\eta = 10^{-4}$ and $n = 20$ are the ones

computed in the previous subsection.

	$n = 10$	$n = 20$	$n = 50$
max raw score	857.00	864.00	501.00
max MA(100) score	418.71	450.36	361.19

Table 3: Breakout scores with respect to the number of steps n

A visual representation of the three training procedures is shown in the graphs of figures 44, 41, 45.

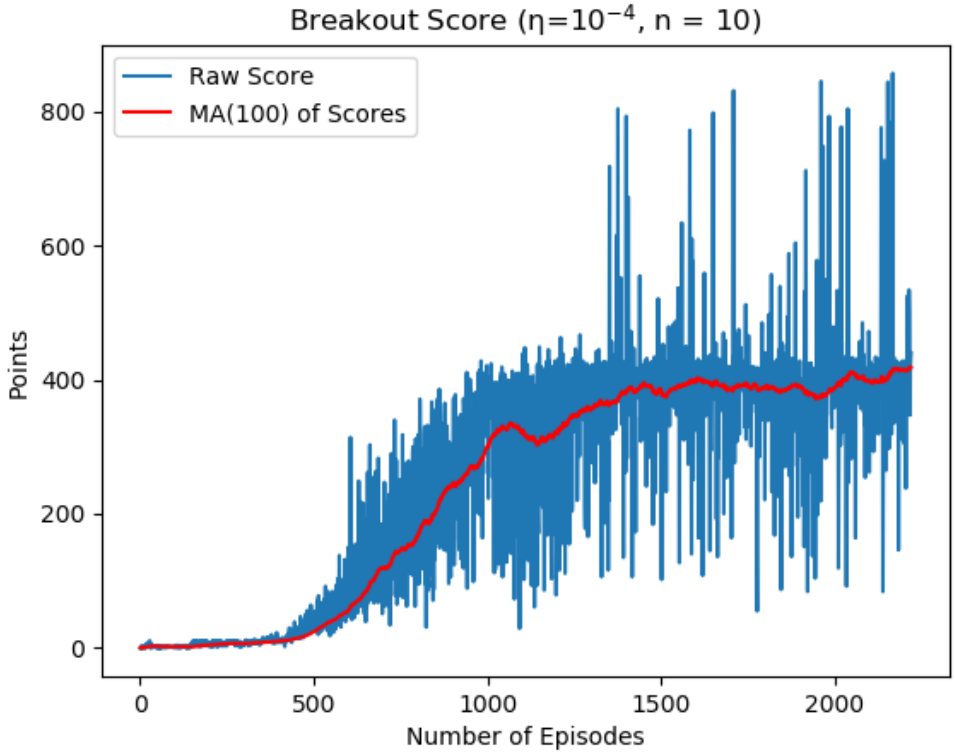

Figure 44: Breakout Scores for $\eta = 10^{-4}$ and $n = 10$

Having studied the previous results and graphs regarding the number of steps values, we ended up in the following conclusions:

1. $\boldsymbol{n = 10}$ & $\boldsymbol{n = 20}$: In the first two cases the training processes had more or less the same behaviour and the same results, with the $n = 20$ setting being a slightly better option. After ≈ 500 episodes the scores started to grow almost linearly and after ≈ 1500 episodes the MA(100) reached the $420 \backsim 450$ level. These results are considered as fast

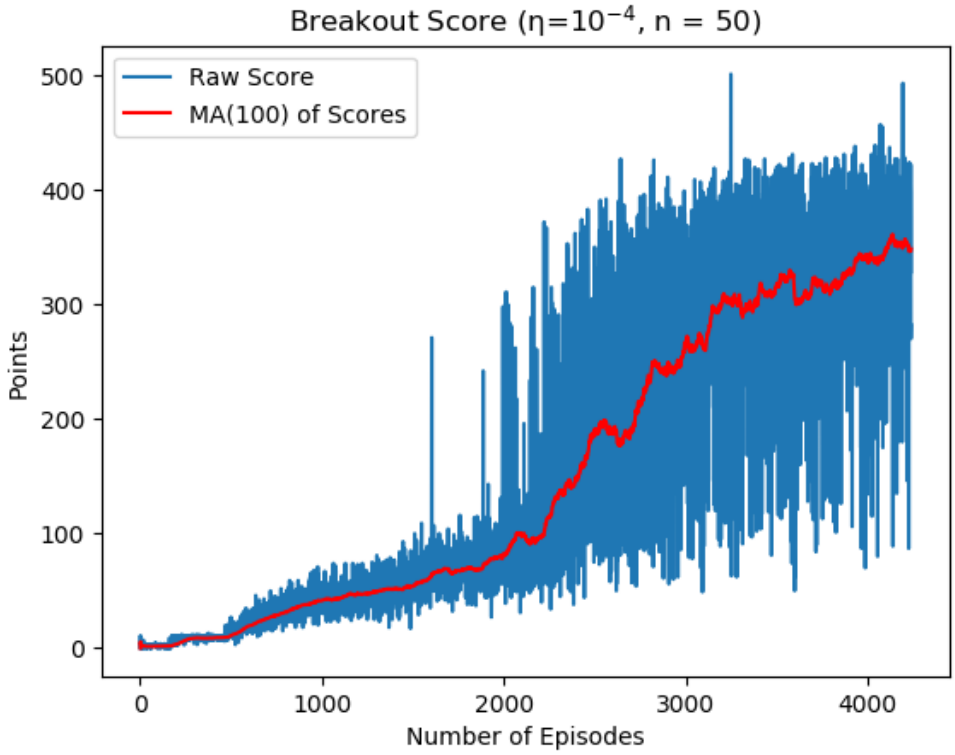

Figure 45: Breakout Scores for $\eta = 10^{-4}$ and $n = 50$

and efficient. We kept the training procedure going for a total of ≈ 2500 episodes, just for confirmation reasons.

2. $\boldsymbol{n = 50}$: In this case the training process was relatively slower and noisier compared to the two previous ones. After ≈ 500 episodes the scores started to grow almost linearly but at a slower pace, and only after ≈ 2000 episodes the scores grew a little more rapidly. After ≈ 4500 episodes the MA(100) reached the 360 level, maybe having the potential to move a little bit higher. These less efficient and more noisy results were more or less expected. A high number of steps in the environment means many continuous steps into the "future", resulting to a much more unpredictable journey of the agents. That is the reason that during a batch of episodes the scores were ranging from ≈ 50 points to ≈ 500 points, hence the more noise.

5.4 Final Conclusion

In conclusion, the optimal settings of the learning rate η and the number of steps n of the A3C RL Algorithm we used to solve the Breakout problem, were: $\eta = 10^{-4}$ and $n = 20$. These were the settings that resulted in the most balanced results in terms of speed and efficiency.

Appendices

A Code Listings

In this Appendix are included the eight (8) Python Code Listings we used in this assignment.

A.1 main.py

```
import os
os.environ['OMP_NUM_THREADS'] = '1'
import torch
import torch.multiprocessing as mp
from environment import atari_env
from model import A3C_LSTM
from train import train
from test import test
from optimizer import SharedAdam
import time

# game configuration constants
env_config = {'id': 'BreakoutDeterministic-v0', 'crop1': 34, 'crop2': 34, '
    dimension2': 80}

class Params():

    def __init__(self):

        # environment parameters
        self.env_conf = env_config

        # training parameters
        self.lr = 1e-4
        self.gamma = 0.99
        self.tau = 1.00
        self.num_steps = 20
        self.max_eps_len = 10000

        # thread parameters
        self.seed = 1
        self.num_agents = 16

if __name__ == '__main__':

    # initialize global parameters
```

```
    params = Params()

    # initialize environment
    env = atari_env(params)
    print('environment ID = ',  params.env_conf['id'])

    # initialize shared model
    shared_model = A3C_LSTM(env.observation_space.shape[0], env.action_space
)
    shared_model.share_memory()

    # load trained model if available
    print('Loading trained model...')
    try:
        saved_state = torch.load('a3c_lstm_model.dat')
        shared_model.load_state_dict(saved_state)
        print('Trained model loaded successfully...')
    except:
        print('Trained model load procedure failed!')

    # initialize optimizer
    optimizer = SharedAdam(shared_model.parameters(), lr=params.lr)
    optimizer.share_memory()

    # start agents (fork threads)
    processes = []

    # start training agents
    for rank in range(params.num_agents):
        p = mp.Process(target=train, args=(rank, params, shared_model,
optimizer))
        p.start()
        processes.append(p)
        time.sleep(0.1)

    # start testing agent
    p = mp.Process(target=test, args=(params, shared_model))
    p.start()
    processes.append(p)
    time.sleep(0.1)

    # stop agents (join threads)
    for p in processes:
        time.sleep(0.1)
        p.join()
```

A.2 environment.py

```
import gym
import numpy as np
from gym.spaces.box import Box
from cv2 import resize


# Parts of the following code were taken from the OpenAI Gym / Universe and
    modified accordingly.
# https://github.com/openai/universe-starter-agent/blob/master/envs.py

def atari_env(params):
    env = gym.make(params.env_conf['id'])
    env._max_episode_steps = params.max_eps_len
    env = EpisodicLifeEnv(env)
    env = FireResetEnv(env)
    env = AtariRescale(env, params.env_conf)
    env = NormalizedEnv(env)
    return env


def process_frame(frame, conf):
    frame = frame[conf['crop1']:conf['crop2'] + 160, :160]
    frame = frame.mean(2)
    frame = frame.astype(np.float32)
    frame *= (1.0 / 255.0)
    frame = resize(frame, (80, conf['dimension2']))
    frame = resize(frame, (80, 80))
    frame = np.reshape(frame, [1, 80, 80])
    return frame


class AtariRescale(gym.ObservationWrapper):

    def __init__(self, env, env_conf):
        gym.ObservationWrapper.__init__(self, env)
        self.observation_space = Box(0.0, 1.0, [1, 80, 80], dtype=np.uint8)
        self.conf = env_conf

    def observation(self, observation):
        return process_frame(observation, self.conf)


class NormalizedEnv(gym.ObservationWrapper):

```

```
    def __init__(self, env = None):
        gym.ObservationWrapper.__init__(self, env)
        self.state_mean = 0
        self.state_std = 0
        self.alpha = 0.9999
        self.num_steps = 0

    def observation(self, observation):
        self.num_steps += 1
        self.state_mean = self.state_mean * self.alpha + observation.mean()
    * (1 - self.alpha)
        self.state_std = self.state_std * self.alpha + observation.std() *
    (1 - self.alpha)

        unbiased_mean = self.state_mean / (1 - pow(self.alpha, self.
    num_steps))
        unbiased_std = self.state_std / (1 - pow(self.alpha, self.num_steps)
    )

        return (observation - unbiased_mean) / (unbiased_std + 1e-8)


class EpisodicLifeEnv(gym.Wrapper):

    def __init__(self, env):
        gym.Wrapper.__init__(self, env)
        self.lives = 0
        self.was_real_done = True

    def step(self, action):
        obs, reward, done, info = self.env.step(action)
        self.was_real_done = done
        lives = self.env.unwrapped.ale.lives()
        if lives < self.lives and lives > 0:
            done = True
        self.lives = lives
        return obs, reward, done, self.was_real_done

    def reset(self, **kwargs):
        if self.was_real_done:
            obs = self.env.reset(**kwargs)
        else:
            obs, _, _, _ = self.env.step(0)
        self.lives = self.env.unwrapped.ale.lives()
        return obs


```

```
87 class FireResetEnv(gym.Wrapper):
88
89     def __init__(self, env):
90         gym.Wrapper.__init__(self, env)
91         assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
92         assert len(env.unwrapped.get_action_meanings()) >= 3
93
94     def reset(self, **kwargs):
95         self.env.reset(**kwargs)
96         obs, _, done, _ = self.env.step(1)
97         if done:
98             self.env.reset(**kwargs)
99         obs, _, done, _ = self.env.step(2)
100        if done:
101            self.env.reset(**kwargs)
102        return obs
103
104    def step(self, ac):
105        return self.env.step(ac)
```

A.3 agent.py

```
import torch
import torch.nn.functional as F
from torch.autograd import Variable
import environment
import model


class Agent(object):

    def __init__(self, params):

        # global parameters
        self.params = params

        # agent environment
        self.env = environment.atari_env(params)

        # agent model
        self.model = model.A3C_LSTM(self.env.observation_space.shape[0],
    self.env.action_space)
        self.eps_len = 0

        # values returned from agent environment (openai gym)
        self.state = None
        self.reward = 0
        self.done = True
        self.info = None

        # agent LSTM values
        self.hx = None
        self.cx = None

        # agent buffers
        self.values = []
        self.log_probs = []
        self.rewards = []
        self.entropies = []


    def action_train(self):

        # retrieve values for current state
        value, logits, (self.hx, self.cx) = self.model((Variable(self.state.
    unsqueeze(0)), (self.hx, self.cx)))
```

```
        prob = F.softmax(logits, dim=1)
        log_prob = F.log_softmax(logits, dim=1)
        entropy = -(log_prob * prob).sum(1)

        # choose action with respect to probs
        action = prob.multinomial(num_samples=1).data
        log_prob = log_prob.gather(1, Variable(action))

        # make next step
        self.state, self.reward, self.done, self.info = self.env.step(action
    .numpy())
        self.state = torch.from_numpy(self.state).float()
        self.reward = max(min(self.reward, 1), -1)

        # store values into agent buffers
        self.values.append(value)
        self.log_probs.append(log_prob)
        self.rewards.append(self.reward)
        self.entropies.append(entropy)

        self.eps_len += 1

        return self

    def action_test(self):

        # retrieve values for current state
        with torch.no_grad():
            value, logit, (self.hx, self.cx) = self.model((Variable(self.
    state.unsqueeze(0)), (self.hx, self.cx)))
        prob = F.softmax(logit, dim=1)

        # choose action with respect to max prob
        action = prob.max(1)[1].data.numpy()

        # make next step
        self.state, self.reward, self.done, self.info = self.env.step(action
    [0])
        self.state = torch.from_numpy(self.state).float()
        self.eps_len += 1

        return self

    def clear_agent_buffers(self):
        self.values = []
        self.log_probs = []
        self.rewards = []
```

```
87          self.entropies = []
88
89          return self
```

A.4 optimizer.py

```
import math
import torch
import torch.optim as optim
from collections import defaultdict


# Parts of the following code were taken from the PyTorch library (v0.4.1) and modified accordingly.

class SharedAdam(optim.Optimizer):

    def __init__(self, model_params, lr=1e-3, betas=(0.9, 0.999), eps=1e-3, weight_decay=0, amsgrad=True):
        defaults = defaultdict(lr=lr, betas=betas, eps=eps, weight_decay=weight_decay, amsgrad=amsgrad)
        super(SharedAdam, self).__init__(model_params, defaults)

        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['step'] = torch.zeros(1)
                state['exp_avg'] = p.data.new().resize_as_(p.data).zero_()
                state['exp_avg_sq'] = p.data.new().resize_as_(p.data).zero_()
                state['max_exp_avg_sq'] = p.data.new().resize_as_(p.data).zero_()

    def share_memory(self):
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['step'].share_memory_()
                state['exp_avg'].share_memory_()
                state['exp_avg_sq'].share_memory_()
                state['max_exp_avg_sq'].share_memory_()

    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                amsgrad = group['amsgrad']

```

```
                state = self.state[p]
                exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']

                if amsgrad:
                    max_exp_avg_sq = state['max_exp_avg_sq']
                beta1, beta2 = group['betas']

                state['step'] += 1

                if group['weight_decay'] != 0:
                    grad = grad.add(group['weight_decay'], p.data)

                exp_avg.mul_(beta1).add_(1 - beta1, grad)
                exp_avg_sq.mul_(beta2).addcmul_(1 - beta2, grad, grad)

                if amsgrad:
                    torch.max(max_exp_avg_sq, exp_avg_sq, out=max_exp_avg_sq
    )
                    denom = max_exp_avg_sq.sqrt().add_(group['eps'])
                else:
                    denom = exp_avg_sq.sqrt().add_(group['eps'])

                bias_correction1 = 1 - beta1**state['step'].item()
                bias_correction2 = 1 - beta2**state['step'].item()
                step_size = group['lr'] * math.sqrt(bias_correction2) /
    bias_correction1

                p.data.addcdiv_(-step_size, exp_avg, denom)
```

A.5 model.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np


class A3C_LSTM(nn.Module):

    def __init__(self, num_inputs, action_space):
        super(A3C_LSTM, self).__init__()

        # Variable Initialization
        num_outputs = action_space.n
        relu_gain = nn.init.calculate_gain('relu')  # additional coefficient
                                                    # for ReLU
        # Convolution Layers (4)
        self.conv1 = nn.Conv2d(num_inputs, 32, kernel_size=5, stride=1,
    padding=2)
        self.maxp1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 32, kernel_size=5, stride=1, padding=1)
        self.maxp2 = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=4, stride=1, padding=1)
        self.maxp3 = nn.MaxPool2d(2, 2)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
        self.maxp4 = nn.MaxPool2d(2, 2)

        # LSTM Layer
        self.lstm = nn.LSTMCell(1024, 512)

        # Actor Output
        self.actor_linear = nn.Linear(512, num_outputs)

        # Critic Output
        self.critic_linear = nn.Linear(512, 1)

        # Normalized Weights Initialization (Xavier)
        self.apply(weights_xavier_init)

        self.conv1.weight.data.mul_(relu_gain)
        self.conv2.weight.data.mul_(relu_gain)
        self.conv3.weight.data.mul_(relu_gain)
        self.conv4.weight.data.mul_(relu_gain)

```

```
44          self.actor_linear.weight.data = norm_tensor_init(self.actor_linear.
    weight.data, 0.01)
45          self.actor_linear.bias.data.fill_(0)
46
47          self.critic_linear.weight.data = norm_tensor_init(self.critic_linear
    .weight.data, 1.00)
48          self.critic_linear.bias.data.fill_(0)
49
50          self.lstm.bias_ih.data.fill_(0)
51          self.lstm.bias_hh.data.fill_(0)
52
53      def forward(self, inputs):
54          inputs, (hx, cx) = inputs
55          x = F.relu(self.maxp1(self.conv1(inputs)))
56          x = F.relu(self.maxp2(self.conv2(x)))
57          x = F.relu(self.maxp3(self.conv3(x)))
58          x = F.relu(self.maxp4(self.conv4(x)))
59
60          x = x.view(x.size(0), -1)   # flatten tensor (LSTM input)
61
62          hx, cx = self.lstm(x, (hx, cx))
63          x = hx
64
65          return self.critic_linear(x), self.actor_linear(x), (hx, cx)
66
67  # Computation of policy loss (Actor) & value loss (Critic)
68  def loss_function(agent):
69
70      # variable initialization
71      R = torch.zeros(1, 1)    # Reward
72      gae = torch.zeros(1, 1) # Generalized Advantage Estimation
73      policy_loss = 0.00       # Actor policy loss
74      value_loss = 0.00        # Critic value loss
75      gamma = agent.params.gamma
76      tau = agent.params.tau
77
78      # retrieve value from agent model
79      if not agent.done:
80          value, _, _ = agent.model((Variable(agent.state.unsqueeze(0)), (
    agent.hx, agent.cx)))
81          R = value.data   # last state reward = Critic value
82
83      R = Variable(R)
84      agent.values.append(R)
85
86      for i in reversed(range(len(agent.rewards))):
87
```

```
        # compute Reward & Advantage
        R = gamma * R + agent.rewards[i]
        advantage = R - agent.values[i]

        # compute Value loss
        value_loss = value_loss + 0.5 * advantage.pow(2)

        # compute GAE (Generalized Advantage Estimation)
        delta_t = agent.rewards[i] + gamma * agent.values[i + 1].data -
    agent.values[i].data

        gae = gae * gamma * tau + delta_t

        # compute Policy loss
        policy_loss = policy_loss - agent.log_probs[i] * Variable(gae) -
    0.01 * agent.entropies[i]

    return policy_loss, value_loss

# Xavier Weights Initialization
def weights_xavier_init(m):
    if isinstance(m, nn.Conv2d):
        weight_shape = list(m.weight.data.size())
        fan_in = np.prod(weight_shape[1:4])
        fan_out = np.prod(weight_shape[2:4]) * weight_shape[0]
        w_bound = np.sqrt(6.00 / (fan_in + fan_out))
        m.weight.data.uniform_(-w_bound, w_bound)
        m.bias.data.fill_(0)
    elif isinstance(m, nn.Linear):
        weight_shape = list(m.weight.data.size())
        fan_in = weight_shape[1]
        fan_out = weight_shape[0]
        w_bound = np.sqrt(6.00 / (fan_in + fan_out))
        m.weight.data.uniform_(-w_bound, w_bound)
        m.bias.data.fill_(0)

# Normalized Weights Initializer (with respect of std value)
def norm_tensor_init(weights, std = 1.00):
    x = torch.randn(weights.size())
    x *= std / torch.sqrt((x**2).sum(1, keepdim=True)) # normalization (var(
    out) = std ** 2)
    return x
```

A.6 train.py

```
import torch
from agent import Agent
from torch.autograd import Variable
import model


def train(rank, params, shared_model, optimizer):

    print('Train Agent: {} -> started'.format(rank))

    # initialize seed
    torch.manual_seed(params.seed + rank)

    # initialize agent
    agent = Agent(params)
    agent.env.seed(params.seed + rank)
    agent.state = agent.env.reset()
    agent.state = torch.from_numpy(agent.state).float()
    agent.model.train()

    while True:
        agent.model.load_state_dict(shared_model.state_dict())

        if agent.done:
            agent.cx = Variable(torch.zeros(1, 512))
            agent.hx = Variable(torch.zeros(1, 512))
        else:
            agent.cx = Variable(agent.cx.data)
            agent.hx = Variable(agent.hx.data)

        # make num_steps actions in a row
        for step in range(params.num_steps):
            agent.action_train()
            if agent.done:
                break

        if agent.done:
            agent.state = agent.env.reset()
            agent.state = torch.from_numpy(agent.state).float()

        # compute policy loss & value loss
        policy_loss, value_loss = model.loss_function(agent)

        # compute variable gradients
```

```
45          agent.model.zero_grad()
46          optimizer.zero_grad()
47          (policy_loss + 0.5 * value_loss).backward()
48          ensure_shared_grads(agent.model, shared_model)
49
50          # update model weights
51          optimizer.step()
52
53          # reset agent
54          agent.clear_agent_buffers()
55
56  def ensure_shared_grads(model, shared_model):
57      for param, shared_param in zip(model.parameters(), shared_model.
        parameters()):
58          if shared_param.grad is not None:
59              return
60          else:
61              shared_param._grad = param.grad
```

A.7 test.py

```
import torch
from agent import Agent
from torch.autograd import Variable
import time
import csv


def test(params, shared_model):

    print('Test  Agent:   -> started')

    # initialize seed
    torch.manual_seed(params.seed)

    # initialize agent
    agent = Agent(params)
    agent.env.seed(params.seed)
    agent.state = agent.env.reset()
    agent.eps_len += 2
    agent.state = torch.from_numpy(agent.state).float()
    agent.model.eval()

    # initialize scores
    eps_num = 0

    try:
        print('Checking for old score file...')
        csv_file = open('score_log.csv', 'r', newline='')
        with csv_file:
            for row in reversed(list(csv.reader(csv_file))):
                eps_num = int(row[0])
                break
        print('Old score file found. Last episode number = {}'.format(
    eps_num))
    except:
        print('Old score file not found!')

    reward_sum = 0
    reward_max = 0
    score_buffer = []

    start_time = time.time()
    eps_ended = True
    while True:
```

```
        if eps_ended:
            agent.model.load_state_dict(shared_model.state_dict())
            eps_ended = False

        with torch.no_grad():
            if agent.done:
                agent.cx = Variable(torch.zeros(1, 512))
                agent.hx = Variable(torch.zeros(1, 512))
            else:
                agent.cx = Variable(agent.cx.data)
                agent.hx = Variable(agent.hx.data)

        # make next action
        agent.action_test()
        reward_sum += agent.reward

        if agent.done and not agent.info:
            state = agent.env.reset()
            agent.eps_len += 2
            agent.state = torch.from_numpy(state).float()
        elif agent.info:
            eps_ended = True
            if agent.eps_len < agent.params.max_eps_len - 50:
                eps_num += 1
                score_buffer.append((eps_num, reward_sum))
            if reward_sum >= reward_max:
                reward_max = reward_sum
            print('Episode #{}: total time {}, reward = {}, total frames =
{}, max reward = {}'.format(
                  eps_num, time.strftime("%Hh %Mm %Ss", time.gmtime(time.
time() - start_time)), reward_sum, agent.eps_len, reward_max))
            if eps_num % 10 == 0:
                print('Saving trained model...')
                try:
                    state_to_save = agent.model.state_dict()
                    torch.save(state_to_save, 'a3c_lstm_model.dat')
                    print('Trained model saved successfully...')
                except:
                    print('Trained model saving procedure failed!')
                print('Saving scores...')
                try:
                    csv_file = open('score_log.csv', 'a', newline='')
                    with csv_file:
                        csv_writer = csv.writer(csv_file)
                        csv_writer.writerows(score_buffer)
                        print('Scores saved successfully...')
                    score_buffer.clear()
```

```
                except:
                    print('Score saving procedure failed!')
            start_time = time.time()
            reward_sum = 0
            agent.eps_len = 0
            agent.state = agent.env.reset()
            agent.eps_len += 2
            agent.state = torch.from_numpy(agent.state).float()
```

A.8 play.py

```
import os
os.environ['OMP_NUM_THREADS'] = '1'
import torch
from agent import Agent
from torch.autograd import Variable
import time

# game configuration constants
env_config = {'id': 'BreakoutDeterministic-v0', 'crop1': 34, 'crop2': 34, '
    dimension2': 80}

class Params():

    def __init__(self):

        # environment parameters
        self.env_conf = env_config

        # training parameters
        self.lr = 1e-5
        self.gamma = 0.99
        self.tau = 1.00
        self.num_steps = 20
        self.max_eps_len = 10000

        # thread parameters
        self.seed = 1
        self.num_agents = 16


def play(params):

    print('Play  Agent:   -> started')

    # initialize seed
    torch.manual_seed(params.seed)

    # initialize agent
    agent = Agent(params)
    agent.env.seed(params.seed)
    agent.state = agent.env.reset()
    agent.eps_len += 2
    agent.state = torch.from_numpy(agent.state).float()
    agent.model.eval()
```

```
44
45      # load saved model
46      print('Loading trained model...')
47      try:
48          saved_state = torch.load('a3c_lstm_model.dat')
49          agent.model.load_state_dict(saved_state)
50          print('Trained model loaded successfully...')
51      except:
52          print('Trained model load procedure failed!')
53
54      # initialize scores
55      reward_sum = 0
56      eps_num = 0
57
58      start_time = time.time()
59      while True:
60          with torch.no_grad():
61              if agent.done:
62                  agent.cx = Variable(torch.zeros(1, 512))
63                  agent.hx = Variable(torch.zeros(1, 512))
64              else:
65                  agent.cx = Variable(agent.cx.data)
66                  agent.hx = Variable(agent.hx.data)
67
68          # render screen
69          agent.env.render()
70
71          # make next action
72          agent.action_test()
73          reward_sum += agent.reward
74
75          if agent.done and not agent.info:
76              state = agent.env.reset()
77              agent.eps_len += 2
78              agent.state = torch.from_numpy(state).float()
79          elif agent.info:
80              eps_num += 1
81              print('Episode #{}: total time {}, reward = {}, total frames =
    {}'.format(
82                    eps_num, time.strftime("%Hh %Mm %Ss", time.gmtime(time.
    time() - start_time)), reward_sum, agent.eps_len))
83              start_time = time.time()
84              reward_sum = 0
85              agent.eps_len = 0
86              agent.state = agent.env.reset()
87              agent.eps_len += 2
88              agent.state = torch.from_numpy(agent.state).float()
```

```
89
90 if __name__ == '__main__':
91
92     # initialize global parameters
93     params = Params()
94
95     # initialize environment
96     print('environment ID = ',  params.env_conf['id'])
97
98     # play the game
99     play(params)
```

B Computer Setup

In this Appendix are included all the commands that are required for setting up an Ubuntu 18.04 LTS system, in order to be able to run the training and testing codes that were listed in Appendix A.

B.1 setup.sh

```
// Python 3 dependencies
sudo apt-get install build-essential
sudo apt-get install python3-dev
sudo apt-get install python3-pip

// OpenAI Gym dependencies
pip3 install 'gym[atari]'

// Pytorch 0.4.1 installation (CPU-only build)
pip3 install torch==0.4.1 -f https://download.pytorch.org/whl/cpu/stable
```

References

[1] Sebastian Raschka. *Python Machine Learning*. Packt Publishing 2015.

[2] Marco Wiering, Martijn van Otterlo. *Reinforcement Learning, State of the Art*. Springer 2012.

[3] Richard Sutton, Andrew Barto. *Reinforcement Learning, an Introduction*. MIT Press, 1998.

[4] Yann LeCun, Leon Bottou, Yoshua Bengio, Patrick Haffner. *Gradient-Based Learning Applied to Document Recognition*. Proc of the IEEE, 1998.

[5] Sepp Hochreiter , Jurgen Schmidbuber. *Long Short-Term Memory*. Neural Computation 9(8): 1735-1780, 1997.

[6] Teuvo Kohonen. *Self-Organized Formation of Topologically Correct Feature Maps*. Biological Cybernetics. 43(1): 59–69, 1982.

[7] Volodymyr Mnih, Adrià Puigdomènech Badia1, Mehdi Mirza, Alex Graves, Tim Harley, Timothy Lillicrap, David Silver, Koray Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning*. 1602.01783v2 [cs.LG] 16 Jun 2016.

[8] Diederik P. Kingma, Jimmy Lei Ba. *ADAM: A Method for Stochastic Optimization*. 1412.6980v9 [cs.LG] 30 Jan 2017.

[9] Andrej Karpathy. *CS231n Convolutional Neural Networks for Visual Recognition*.
`http://cs231n.github.io/convolutional-networks/`
Stanford University, 2019.

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

[11] Open AI Gym. *https://gym.openai.com/*