Theoretical Frame

Descriving the principles behind Arfificial Intelligence, Machine Learning and Artificial Neural Networks.

Francesc Pifarré Esquerda

Index

1. Introduction

I once heard that for the last decade, there has been at least a headline related to Artificial Intelligence on any of the most important newspapers of the US every single week. Though I don't think it to be true, from a merely statistical point, it is surely true that AI has been one of the fastest evolving field of computer science.

But many people still believe Artificial Intelligence (also commonly referred to as AI) as a kind of "superhuman intelligence" that will eventually take out the world and the human race. This thought is not casual; in the last decades there have been an increasing number of films, books and so on describing a future where humans are mere subjects of an intelligent machine.

Though some people still argue that we may come to that situations, the present is far from all that. AI is present in a lot of places in our daily life, usually making small things that we frequently do easier and more reliable. It is there when you use your phone; not only when you talk to it via a virtual assistant but also when you take pictures and you see a small square surrounding people's faces to make sure they are on focus. It's there every time you fly anywhere with a plane, most of the time you are in the air the pilots are there just supervising the work of an AI driving the plane. When you search for something online, the results are shown based on what you have previously searched. When you see an online add, it has not been put there randomly, but it is based on a lot of information about you.

Banks operate with their millions of stocks and shares using mostly algorithms that do the raw work for them, trading huge amounts of money without human interference, and it has been so since the 70s, being these algorithms partly responsible for the Black Monday crash of 1987. By then algorithms were accountable for only 10% of market trading. This number is nowadays closer to 50% and that's only taking into account the most modern algorithms.

So, ¿what makes machines capable of all these things? ¿What makes a computer program be able to locate a face, translate your speech or buy and sell assets? That's the question I asked myself when I started digging into the field of AI.

The answer is that in the brief story of artificial intelligence there have been many techniques that have empowered computers to "think". But instead of doing an intensive research on all of them, I decided to focus more on the present ones.

And in the present day (2018) AI consists mostly of Machine Learning, also abbreviated as ML. Machine Learning's goal is to generalize behaviours based on data. The amount of data and its quality determine how good the machine can be at the task given to it. That has created a whole industry around data collection and sharing.

You may have heard the terms Big Data before. Big Data is a way to name huge amounts of data collected, stored and shared by big companies. If you take the top 10 biggest technology companies by revenue, you see that every one of them has invested huge resources into artificial intelligence and collect data to improve them.

Machine Learning has been the receiver of most of the resources invested in the field of AI. The importance of ML resides in its attempts to imitate how the human brain and mind works. A child learns by experiences that are explained to him (data). With enough experience, the child starts to find patterns on what he perceives and understands it, so it doesn't depend on explanations anymore. All of this is done by the neurons, cells in the brain that are responsible for human thoughts, memory and basically all of the human minds.

Machine Learning tries to replicate the same structure using Artificial Neural Networks, which are responsible for the "magic" of deep learning. Those Artificial Neural Networks are able to recognize faces without knowing who the person is, understanding natural human language without actually learning the language, translating sentences between languages without knowing any of the languages at all. They can imitate and produce things that we thought were limited to humans; poetry, theatre, articles, paintings, images…

During this work we will go deeper into the world of Artificial Intelligence and Machine Learning, understanding their structure and how they work. At the beginning of every part, a brief summary of the concepts exposed in that part can be found.

2. Historical Context

2.1 Ancient concept of AI:

To start with, we go through the history of Artificial Intelligence, closely related to the history of logic until the modern age. We will see Ramon's Lull work on a thinking machine.

To fully understand anything, it is important to know where it came from; its history.
One may think that the story of AI is short because of its obvious connection to computer sciences, with are themselves quite recent, but it is not the case.
The concept of a machine being able to think has been in the minds of philosophers for a long time. This concept was born quite close to logic and evolved with them until more recently in our history. Let's start then on Aristotle's work on syllogisms.
Aristotle theorized about how the human mind thought and established the grounds of logic. Syllogisms were the central part of his thesis. Syllogisms are a method of deductive reasoning. They consist of two statements and a conclusion that is itself a deductive inference of the statements. The structure of the four more basic syllogisms is the next:

Statements:	Every A is B B is C	No A is B B is C	Some A are B B is C	Some A is not B B is C
Conclusion:	C is A	C can't be A	C may be A	C may be A

Figure 1. Structure of basic syllogisms according to Aristotle.

Aristotle's theory of syllogisms and logic consisted of more than just this table, but this was the part he theorized could be used to create an intelligence with was not human. Given a machine the first two premises, it should be able to get to the conclusion by itself. This is a simplified version of Aristotle's work on logic, as he is considered to be the founding father of the field, establishing them as a formal science.
Logic lost importance during most of the low medieval era, until the appearance of the biggest writer on medieval logics who also set the precedent for artificial machines being able to think: Ramon Llull.
Llull was a proliferous author who wrote mostly about philosophy and theology. The book we're interested in is Ars Magna, a book in which Llull described a mechanical logical machine which would be able to separate the true from the false. He named his machine Ars Magna, which means "greatest art".

In the Ars Magna, Llull describes a logical machine which should be able to validate and invalidate arguments. He called this process “automatic reasoning”, and theorized about a mechanical machine being able to carry on this process.

He named this mechanical machine the Ars Generalis Ultima. It worked with a series of levers and gears, being considered as one of the first logical automatons to be ever designed, but it was never built because of Llull’s death.

For the machine to work, Llull defined what he called “roots”. The roots were simple ideas which could correspond to a principle, question, entity, virtue or vice. With the combination of this categories, one could form a complete sentence, having a subject and a predicate.

With the movement of the levers and gears, the three geometrical forms that were considered perfect (square, triangle and circle) would be combined in a way that would either make the sentence asked to the machine true or false.

Because of the few descriptions of the machine as a mechanical entity, this could never be built. But despite this fact, Llull was the first writer to propose a machine that could think for itself.

After Llull’s definition of a machine that could validate logical statements, it began to become obvious that a machine would eventually be able to imitate the human processes of thinking. As formal logic matured, certain laws that could be able to describe accurately the appropriate process of reasoning appeared.

With this laws, a bigger problem appeared. Logical laws have some resemblance to some basic mathematical principles, and those could easily be imitated by mechanical machines if build appropriately. The complexity of those theoretical machines, though, was huge, and even though many were designed and theoretically described, none was successfully built. The gap from mathematics to logics seem too big.

But if mathematical reasoning could be built into a machine, ¿why couldn’t logical processes be able to be imitated by a machine? This problem remained unsolved until the appearance of the first digital computers, which opened a whole new field to explore how to imitate the human brain in a machine.

2.2 Alan Turing's "Computer Machinery and Intelligence":

Alan Turing's paper about Computer Intelligence is considered to be the foundational paper of Artificial Intelligence. We will see the description of the imitation game he proposed, the definition and parts of a computer he described, the arguments against him he refused and the conclusion of his work.

Alan Turing was one of the most important mathematicians of the 20th century, is his most important contributions to the fields of cryptography and computer science. He is actually considered to be the father of computer sciences and he was the builder of the first computer, built to break Nazi cryptography during the Second World War.

Though his early death at 41 years old, he published quite a lot, especially taking into account that a lot of his work was classified during most of the century because of his contributions to the war effort. The paper we're here about is perhaps the most famous he ever wrote: "Computer machinery and intelligence".

The starting sentence of the work was the prologue to a whole new science that would change the world in ways Turing wouldn't have been able to imagine. *I propose to consider the question, "Can machines think?".* That's how Alan started his description of what he originally called the "imitation game", nowadays known as the Turing test. The imitation game is a key piece of his article, so let's describe it before going on the article content by itself.

The imitation game consists of three people: A (a man or a woman), B (a person of the opposite gender of A) and C, also named the interrogator. The three persons are in different rooms from each other and can only communicate with telegraphic messages. The goal of the game is for the interrogator to discover which one of either A and B is the man and which one is the woman. It can do so by asking questions which responder by both of the subjects of the game must be. Let's put as an example that the interrogator must guess who the woman is. One of the questions he could ask is about the length of their hairs or the kind of clothing they are wearing. The woman (either A or B) must try to be identified correctly, while the man must try the opposite by either lying or answering the questions ambiguously.

Turing then proposed the following question: Could a computer take the part of A or B and successfully win the game? The interrogator would then have to decide which one of the subjects is a computer and which one is a person. To succeed at this game, the computer would have to be able to realistically imitate human behaviour using the same answers a human would give to those questions, and even lying to try to lean the interrogator towards an incorrect guess.

Turing also added that to make the game a bit less difficult for the computer, it would be programmed to make some mistakes on certain mathematical calculations, an area where a computer would answer too precisely for it to be taken for a human being. [1]

With the imitation, Turing avoided getting into the subject of "thinking" as a philosophical activity and avoided needing a definition and a complete knowledge of what "thinking" is. Those items would have been needed if the original question had been to be resolved as it had been asked: "Can machines think?"

During this work, we will also set aside the philosophical concepts of thinking, if intelligence can be artificial and other similar philosophical issues related to AI, as those are complex and would deserve an entire work on their own.

Turing did, although avoiding discussing intelligence, define the machines that would be allowed to play the game. He stated that not every human-made machine was allowed to even attempt to make the game, and only one kind of them could do so. Those machines were digital computers. It may seem obvious nowadays, but it wasn't back in Alan's days. Digital computers were very uncommon, expensive and completely unknown to the general public. So, Alan proceeded to describe the three main components of a digital computer. Though computers are much more extended nowadays, we will go through those parts to give a general knowledge of how they work.

- Storage unit: is the part of the computer that stores information, usually in binary code. It consists of cells of memory that consist of a fixed number of bits. As so, it is measured in bits and its larger groupings (bytes, kilobytes, megabytes…).

[1] Turing argued that a machine which was always accurate in mathematical operations would be too easily identified as one, because of virtually any human being would be able to archive such mathematical accuracy.

- Executive unit: is the part of the computer responsible for changes in the storage, modifying it by doing calculations using the numbers stored. It is the part of the computer that is programmed, being "to program" to define the instructions of the processes the executive unit will carry out. It is measured by its computational power, the number of operations it can do in a second, or Hz (Herz).
- Control unit: is the part of the computer ensuring the executive unit does its job, and that makes sure that the calculations are done in the right and then saved accordingly in the storage unit. It is pre-programmed and cannot be modified.[2]

After having explained what a digital computer is and having defined the problem with the imitation game, Turing progressed to dismantle the arguments that were used to oppose the possibility of a computer being able to think or win the imitation game.

The arguments he opposed were the theological, the "heads in the sand objection", the mathematical one, the argument for consciousness, the one for various disabilities, Lady Lovelace's objection, the argument for the nervous system continuity, the one for the informality of behaviour and even one for extra sensorial perception. Those were all arguments he thought could be used to oppose his idea of a "thinking" machine.

During the process of refuting these arguments, he mostly limited himself to pointing out fallacies in them and arguing back why a computer would indeed be able to think in similar term to the ones used to oppose this notion.

Finally, and to conclude, he described his vision on how machines could be trained to succeed in the imitation game, referring to the human process of learning. He thought that this was formed by three parts, the genetic conditions, the education and other experiences not related to education.

The genetic conditions could be referred to as, in the case of a computer, the amount of storage available and of computational power. The education would be what is programmed inside the computer and so would be the other experiences, though both would be stored in different parts of the storage.[3]

[2] The control unit is not usually measured in any way. If it were to be measured, though, it would also be measured in Hertz and by its computational power, as the executive unit is.

[3] This last part of Turing's article, about how Thinking Machines could be trained, is the one that set the grounds of Machine Learning.

He concluded his article with the statement that computer intelligence was not only possible but achievable in the near future, as storage capacity and computational power kept improving. Artificial Intelligence, thinking machines, were no longer a theoretical concept, unavailable due to methodological or conceptual problems. The only think holding them back was computing and storage capabilities.

The ideas of Turing were formalized in the Dartmouth Conference, a congress in the United States was the concepts of Artificial Intelligence were made formalized, and where the investment of resources in AI as a whole new field of computer sciences started.

2.3 AI Definition, processes and types:

We go through the definition of Artificial Intelligence and the human processes it tries to imitate. We will also see the distinction between Narrow and General AI, one of the most used classifications in the field.

AI is nowadays defined as the part of computer science that tries to simulate human intelligence through processes like learning, reasoning, problem-solving, perception and language.

Let's briefly describe how some of the processes mentioned above work, in order to better understand this definition of Artificial Intelligence.

There are different possible ways to make Artificial Intelligence learn. The simplest one is by trial and error. It was used on the earliest states of AI science. The machine would guess randomly until it guessed correctly and would then save the correct guess to repeat it when the same situation was given to the AI. The problem of trial and error was that it only responded in the specific situations guessed before and it was not able to generalize. This has now been solved with Machine Learning, which we will go into in the next part.

To make an AI reason, the only solution nowadays is to implement logical processes like induction and deduction. AIs have proved quite successful at doing these kinds of logical operations. Their success is mostly due to the mathematical aspects of logic, easily implemented in computer sciences through programming.

Problem-solving AIs have mostly been implemented in board games such as chess. The process is done with the application of the MinMax algorithm, one designed to maximize the gain and minimize the loss in future moves, requiring perfect information about the game. So is done assuming the opponent will always effectuate the move that will cause the most loss to the AI player. Then only a goal is needed (such as to kill the king in chess) to make the computer go through all possible options to achieve it, requiring great computational power and therefore being quite inefficient. Nowadays other more complex algorithms have become more efficient at solving problems. [4]

Finally, perception and language. Perception is mostly done with sensors with giving data on real-time about the real world. They could be from relatively simple sensors like humidity sensors to more sophisticated ones such as cameras. The data is then processed by the artificial intelligence using one or more of the processes mentioned above.

[4] In the field of board games, AIs have proven to be very successful, achieving better results than humans and beating the world champions on different board games several times.

Even though the main goal of AI is and has always been to completely imitate human intelligence, we distinguish two different types of AIs. The most used classification of AIs is distinguishing between Narrow AI and General AI.[5]

General AI, also called Strong AI or Human-Level AI would be the one able to both passes the Turing test and be able to perform as a human in every task a human is able to perform. General AI does not exist nowadays, and it is not foreseeable in the nearest of futures, though remains as the theoretical objective for AI scientists.

The main problem of General AI is what will happen when it is reached. If the definition of this kind of intelligence makes clear that it has to be able to perform as a human in every situation, it would be able to create an AI like itself, maybe even improved, which could repeat the process afterwards. This theoretical moment of AI explosion is called the singularity.

On the other hand, we have Narrow AI, also called Weak AI. This is the kind of machine intelligence present in our days. Narrow AI is named so because it is only able to perform a small range of functions, very specific. One of the most obvious examples of Narrow AI limitations is Self-driving cars. The existence of AIs able to drive cars is a fact, but if those same AIs that drive cars almost perfectly were given the task to drive a motorcycle, they wouldn't be able to do so. Another Narrow AI would have to be trained to perform that task, completely independent from the first one used to drive cars.

Narrow AI has been getting much better at the tasks it can perform, improving in accuracy and reducing the requirements needed for it to work, but it is still very specific. Nowadays, it is the fastest evolving of all AIs, being the focus of most research while General AI remains as a theoretical concept.

[5] Though it being the most common classification of AI, it is not the only one.

3. Machine Learning

3.1 What is Machine Learning and how it works:

We go through the definition of Machine Learning and how it works. So, will be done using an example to see the steps of the Machine Learning process. The importance of data in the field and its relation to Artificial Intelligence will also be mentioned and explained.

Machine Learning has been for the last decade the most successful method to make Narrow AIs possible, and its possible applications for General AIs are being studied, though not possible yet. It consists of the development of algorithms able to analyze data and made predictions about it using a structure called Artificial Neural Network.

This image represents the basic structure of Machine Learning (also referred to as ML):

To make the understanding of the concepts easier, we will assume that we're facing an already trained Machine Learning algorithm as we go through the steps shown in the figure. Training will be explained afterwards.

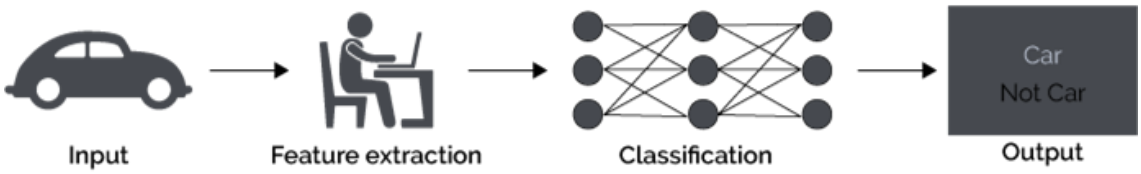


Figure 1. Diagram of a Machine Learning Algorithm. George Steif. ©

The first requirement of every ML algorithm is to have an input. The input is the data you want the algorithm to analyze and classify. In the case of the figure, the input consists of the image of a car.

The second step is Feature Extraction. It is usually done by a human individual through some of the most complex ML algorithms are able to do it by themselves. Feature Extraction consists of translating the input to terms the algorithm can understand, which basically means numbers or mathematically similar terms. This inputs can be called features. In the case shown on the figure, it would most likely be a grey scale of every pixel on the image from 0 (completely white pixel) to 1 (completely black pixel). The numbers would be put together into a matrix or a vector, which would be the direct input of the ML algorithm.

Then is when the classification phase comes in. The algorithm classifies the features and processes them making it go through an Artificial Neural Network. The classification process consists of weighted sums of the direct input and some functions applied to them, all repeated several times. This process will be further described when Artificial Neural Networks (also called ANNs) are studied in the following part of the work.

Finally, a number reaches the end of the ANN, in this case, probably a value between 0 and 1 or a percentage, which determines the possibility of the input being what the Network has been trained to identify. In the case of the figure, it would be whereas the image corresponds to a car or not. A value of 1 would be a 100% possibility of the input corresponding to a car (100% percentages are very rare in ML algorithms) and 0 would mean that the machine can't find any correspondence between the input and a car (0% percentages are also very rare). The value outputted by the ANN can be called a label.[6]

It must be taken into account that in the case of a binary classification such as the one shown in the figure (only possible outputs being "car" or "not car") even the most useless algorithm would achieve a 50% accuracy, because of the 50% chance of guessing correctly that is always present on binary classifications.[7] This percentage of guaranteed success goes down as the algorithm must classify more items, getting to 33% when the items are three, 20% when they are four and so on.[8]

So, now let's see how the algorithms get to the point of being able to classify inputs or features into "car" or "not car", the labels. This process is named training and is where the importance of data comes into place.

The training phase of the training of an ML algorithm consists of reproducing the same steps described above, with one last step where the output of the algorithm is compared with what it should be. Then, with a process named Backpropagation, which we will go deep into in a following part of the work, the algorithm adjusts some of its parameters to try to make its output as near as possible to what it is. It is done by comparing the label of the examples with the one outputted by the network.

The process is repeated for thousands of thousands of times with different sets of data. In the example shown in the figure, the ML would have already seen and classified millions of cars images, so it would have learnt what makes an image be of a car.

[6] This describes a classification problem. Together with regression problems, this are the most common problems machine learning algorithms are given.

[7] This would be if the network had been trained with as many images of "cars" as images of "not car".

[8] This is taking into account the same amount of every category in the data given to the algorithm. If the amounts of examples of each category to classify is not equal, these percentages could vary.

The fundamental problem of training? The amounts of data available. If the network is trained on a single image of a car, the results wouldn't be accurate at all. It needs lots of examples, and those examples must be classified by a human been to have something to compare the output of the algorithm with to make changes on the algorithm to try to make the output as similar as possible to what a human outputted.

Machine Learning is divided into three main fields:[9]

The most common one and the one we will work with is supervised learning. This is the method that requires the training data to have been previously labeled. This means that the algorithm has a real label to compare the label it gets to, and therefore can be trained to classify data on its own.

Unsupervised learning is the one which is just left alone with the raw data, with nothing to compare its outputs to. Unsupervised learning is mostly used to try to find patterns in the data we are feeding them, and those patterns must always be lately interpreted by a human being.[10]

The last field is reinforcement learning. In reinforcement learning, the network's success is based on a value which is always available to the network. This value could be seen as a score; the higher it gets, the better. The goal of the Machine Learning algorithm is to adjust itself to try to make this score as high as possible.[11]

[9] This is the main classification for Machine Learning algorithms, but not the only one.
[10] Unsupervised Machine Learning algorithms cannot classify data on their own, but can find pattern which could help the human who interprets the results make a more accurate classification.
[11] Reinforcement Machine Learning algorithms aren't commonly used for classification. They are used in field in which supervised or unsupervised ML algorithms would not be very effective, such as games.

3.2 Artificial Neural Networks and how they work:

We describe Artificial Neural Networks and their structure, establishing a relationship with the human nervous system. Keywords of the structure of Artificial Neural Networks are described and will be repeated through the following parts of the work. The distinction between Feed-Forward Neural Networks and Deep Neural Networks will also be explained.

Artificial Neural Networks, also abbreviated as ANNs, are the key element of machine learning. But, what do they consist of? To describe Artificial Neural Networks, the functioning of neurons in the human nervous system must be explained, so it can be compared to the functioning of ANNs. A complete and exhaustive knowledge of neurons is not needed, but a basis is needed in order to understand how Artificial Neural Networks try to imitate with their structure and functioning.

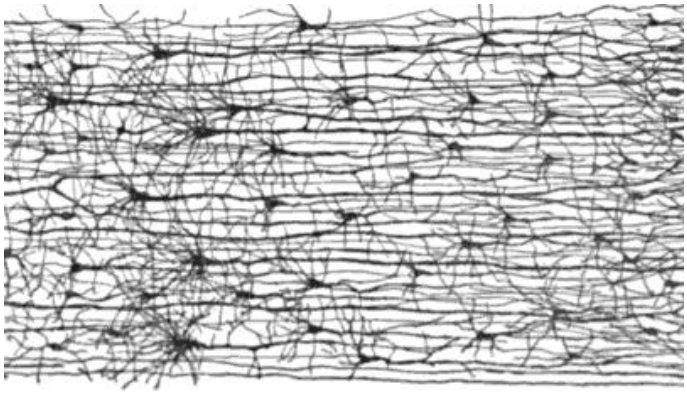


Figure 1. Drawing of interconnected neurons. Ramón y Cajal. ©

This image of Figure 2 is one of the first that was ever made of the nervous system, to be more concrete, in the brain. Ramón y Cajal was a Spanish neuroscientist who discovered the neurons. He spotted some weird looking cells in the brain. Those cells consisted of a body, which he called the nucleolus, and some prolongations, which could be either dendrites or axons depending on which role they had in transferring electrical pulses.
The neurons conform to a huge web of intercommunicated cells. A neuron receives an input, which comes from the axon of another neuron to the neuron's dendrite and is then sent to the nucleolus.

Depending on the strength of the electric pulse, it is transferred to another neuron through the original neuron's axons, and the process is then repeated in the following neuron of the web.[12]

Each neuron doesn't have a connection to only one other neuron, but many called synapsis. It is estimated that the human brain contains more than 86 x 10^9 neurons interconnected in a network that gives humans the ability of thinking.

This is the same structure the one that is tried to be replicated in Artificial Neural Networks. The most basic element of ANNs is the neuron, as it is in the human brain, which has the structure shown in Figure 2.

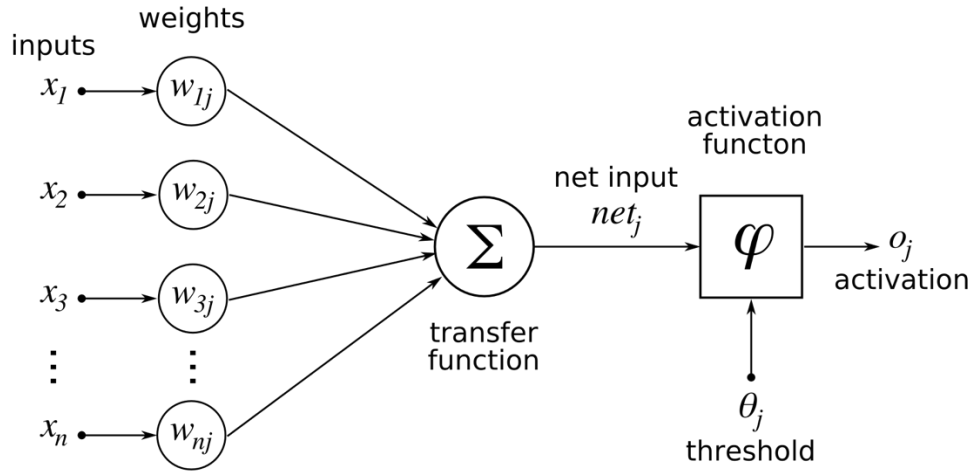


Figure 2. Diagram of the structure of an Artificial Neuron. Alex Castrounis. ©

An artificial neuron consists basically of a weighted sum of the inputs it is given. These inputs consist of several numbers, usually, the more the more accurate the algorithm is, which are multiplied by a weight. The inputs represent real-world elements in form of numbers and vary in each itineration of the machine learning algorithm. They can either be numerical values or categorical values.[13] They are referred to with the variable "x".

The weight is another number that multiplies the inputs. The number is invariable for each input. The value of the weight usually resides between 0 and 1 and is determined through the training phase of a machine learning algorithm, using various Backpropagation systems that will be described in following parts of the work. The weights are referred to with the constant "w".

[12] The actual structure and functioning of neurons is more complex than how its described. The intent of this description is only to show similarities to Artificial Neurons, not give an accurate description of Human Neurons and their functioning.

[13] Categorical and numerical values can be used as inputs, but they are usually treated differently. While numerical data is normalized or regularized, categorical is not, and can be expressed in the form of numbers using different systems.

All the values obtained from multiplying the inputs with the weights are summed together and with another value called threshold or more commonly bias to form the net input. The bias is there to make sure that the neuron activates even if the weights are 0.[14]

This net input is then run through an activation function, that consist of a mathematical operation which transforms the input into another value, usually between either -1 and 1[15] or 0 and 1[16]. The most common activation functions will be described in the next part of the work.

The net input after going through the activation function is the output of the neuron. This output then goes to another neuron as one of its inputs, and the same process is repeated.

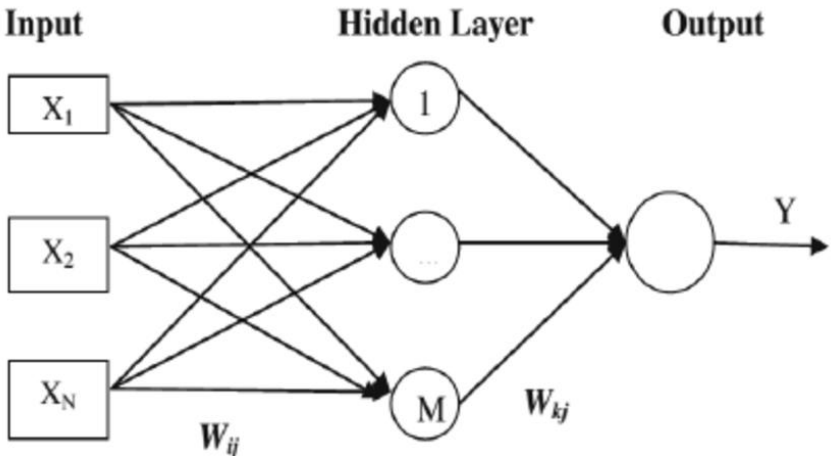


Figure 3. Structure of a Feed-Forward Neural Network. Zhongqi Wang. ©

The neurons are connected with each other forming a network of interconnected neurons. This network of neurons is the Artificial Neural Network. The neurons, when connected, are classified into three groups or three layers.

Input Neurons or the Input Layer, in which the original inputs are manually entered. The neurons of the input layer don't perform any operation. Hidden Neurons or Hidden Layer, in which the inputs are computed with a number of weighted sums and activation functions. Finally, the Output Neurons or the Output Layer, which displays the final output of the ANN, which classifies the information inputted into the Network.

[14] If the output of a neuron is always 0, regardless of the bias, the neuron is referred to as a dead neuron. This can specially happen when using some specific activation functions such as ReLU.
[15] The Hyperbolic Tangent Activation Function, described later, has a range of outputs from -1 to 1.
[16] The Sigmoid Function Activation Function, described later, has a range of outputs from 0 to 1, and so does the Threshold Activation Function.

Feed-Forward Neural Networks are the ones that only have one Hidden Layer or none, like the one shown in Figure 3. This makes this ANNs the simplest type on Neural Networks because the inputs are only computed once in a single layer.[17, 18]

Deep Artificial Neural Networks, abbreviated as DNNs, instead, consist of more than one Hidden Layer. The bigger the number of Hidden Layers, the more operations the network will do and the more accurate will the result be.[19] The bigger the number of hidden layers, moreover, increases the computational power needed to run the network and, consequently, the efficiency of the network may be reduced. The efficiency of an Artificial Neural Network in the relationship between either the computational power required to run and train the network or the time it takes and the accuracy of the Network.

[17] ANNs which use linear activation functions are usually Feed-Forward, as can be ANNs which perform regression or simple binary classification tasks.

[18] The number of neurons in the Hidden Layer of a network does not impact its type according to this classification. Because of this, a Feed-forward neural network could actually have more neurons than a Deep neural network, although it is not common.

[19] Even with a Feed-forward neural network having more neurons than a Deep one, the Deep one would probably archive better results because of the greater number of connection between neurons. Because of every neuron on each layer being connected to every neuron in the previous and next layer, the more Hidden Layer a DNN has, the more connection between neurons it develops and the more accurate it gets, even with a small number of neurons.

3.3 Activation functions and their types:

Now activation functions will be explained in depth. The different types of activation functions will be shown together with their mathematical expression. The difference between them will be mentioned and also what each type of function is used to.

As shown in Figure 2 and explained in the previous part of the work, activation functions are the functions that are applied over the net input. The type of activation function used in an Artificial Neuron is so important that the neuron is often referred to as the name of the function it uses.

But, what are the activation functions? Activation functions are essentially mathematical functions that were traditionally used in statistics. They are needed to limit the range of the possible output value of a neuron and to calculate its output.

$$Y = \sum(input * weight) + bias$$

Figure 4. The equation of an Artificial Neuron without an activation function.

Figure 4 represents the output of a neuron, referred to as Y, can be any value between infinity and minus infinity. This represents a problem for the Artificial Neural Networks, that needs the values to be as similar to each other as possible in other to work properly. Activations functions are the way to solve this problem. They compact the possible values that an Artificial Neuron could output into a range of smaller numbers that are more easily manageable by the ANN.

The most basic of activation functions is the Threshold Function. The Threshold Functions assign a value of either 0 or 1 to the output of the neuron. Its mathematical expression is the next.

$$Y = \begin{cases} 0 \; if \; \sum(input * weight) + bias < k \\ 1 \; if \; \sum(input * weight) + bias > k \end{cases}$$

Figure 5. Threshold Function.

The equation of Figure 5 is the one used by the Threshold Function, where Y is the output of the neuron and k is a constant. The main problem of the Threshold function is that the actual value of the net input doesn't carry into the next neuron. The only thing that does is whether the value is bigger than a previously defined constant or nor. Another problem happens when the net input is equal to the constant. This would be a very strange case because the constant is usually an integer while the net input is usually a number with many decimals because of the weights and biases usually having many decimals, but it can still happen. This is easily fixed adding an equal to any of the two operators of the function. Even though the Threshold Function can be useful in some cases, it is not used in most ANNs.

Another activation function is the linear function, which has the following form.

$$Y = [\sum(input * weigh) + bias] * m + n$$

Figure 6. Linear Function.

The equation of Figure 6 defines the Linear Function. It consists of the multiplication of the net input by m, a previously defined constant that is usually a small value used to make the net input smaller. Then n is added to the calculation, as another previously defined constant. The Linear Function resembles the geometrical form of a straight line and therefore is used in many regression tasks.

The main problem of the Linear Function is that it does not solve the problem of the values having an infinite range. Its particularity is that if it is used on more than a single Hidden Layer of an ANN it makes every one of the layers that use is unnecessary. The value outputted by a Linear Function goes into the next layer as input and the second layer calculates the net input of many inputs that come from Linear Functions and it, in turn, returns another value based on the same Linear Function

No matter how many layers we have, if all are linear the final activation function of the last layer is nothing but just a linear function of the input of the first layer, eliminating the need for all the previous layers and therefore making all extra layers useless.

Let's move over the most popular and widely used function in ANNs: The Sigmoid Function, a nonlinear function.

$$Y = \frac{1}{1 + e^{-[\sum(input * weight) + bias]}}$$

Figure 7. Sigmoid Function.

The equation of Figure 7 is the Sigmoid Function. This nonlinear function outputs a value between 0 and 1, which can be very easily turned into a percentage. This is why this function is the one usually used in the last layer of an ANN, especially in binary classification, the main field of speciality of ANNs. It also has the advantage of minimizing the impact of either very big net inputs or very small net inputs. The Sigmoid Function has a variation which is also used: Hyperbolic Tangent Function.

$$Y = \tanh\left(\sum(input * weight) + bias\right) = \frac{2}{1 + e^{-2[\sum(input*weight)+bias]}} - 1$$

$$= 2 * sigmoid[2(\sum(input * weight) + bias)] - 1$$

Figure 8. Hyperbolic Tangent Function.[20]

The equation of Figure 8 is the one used for the Hyperbolic Tangent Function. It is represented in different ways, which output the same value. It is basically a scaled version of the Sigmoid Function, providing a range of values between -1 and 1. Both the Sigmoid Function and Hyperbolic Tangent Function allow Hidden Layer stacking in ANNs, because of their nonlinear nature.

While the Sigmoid Function is most popular in the last layer of a Network, the Hyperbolic Tangent Function is in all of the other Hidden Layers, working very efficiently together. Their main problem is that they require much more computational power than the other functions, making the Network slower, sometimes less efficient and the many decimal points these functions can produce may provoke errors depending on the programming language used and type of variables used to store the numbers.

One of the activation functions that has become increasingly popular during the last years is the ReLU function.

$$Y = \max(0, [\sum(input * weight) + bias])$$

Figure 9. ReLU Function.[21]

[20] Hyperbolic Functions are mathematical functions based on the exponential function with the following expressions: $e^x = \cosh(x) + \sinh(x)$ and $e^{-x} = \cosh(x) - \sinh(x)$. They are the Hyperbolic Sinus: $\sinh(x) = \frac{e^x - e^{-x}}{2}$, Hyperbolic Cosinus: $\cosh(x) = \frac{e^x + e^{-x}}{2}$. The Hyperbolic Tangent can be expressed as: $\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$.

[21] Another advantage of the ReLU function is that it is easy to compute and to derivate, while other Activation Functions such as the Sigmoid Function or the Hyperbolic Tangents are harder to compute; both themselves and their derivatives.

The equation of Figure 9 is the ReLU Function, the abbreviation of Rectified Linear Units Function. The ReLU function outputs 0 if the net input is negative or the raw net input if it is positive. It doesn't fix the range problem completely, but it has proven to be very effective in training very big ANNs.

Its problem is that it is possible that the output of the ReLU neuron is always zero because the weights are set in a way to make the net input always negative. The weights are set through a process called backpropagation that will be explained later, but if they are set to negative in the learning phase, the ReLU Neuron would always output a 0. When this happens, the neuron is referred to as a Dead Neuron.

A way to fix it is with a mixture of the Threshold Function and the ReLU Function.

$$Y = \begin{cases} \max(0, \left[\sum(input * weight) + bias\right]) \; if \; \sum(input * weight) + bias \geq 0 \\ \max(0, k[\sum(input * weight) + bias] \; if \; \sum(input * weight) + bias < 0 \end{cases}$$

Figure 10. Leaky ReLU Function.[22]

This is the Leaky ReLU Function, shown in Figure 10. If the net input is positives, its output is the same as if it was a ReLU Neuron, while if the net input is negative, the output is the net input scaled to make it smaller multiplying it by a k constant. The constant in the Leaky ReLU Function is a negative power of 10. Leaky ReLU solves the problem of Dead Neurons, while it also emphasizes the positive net inputs and reduces the impact of negative ones.

The ReLU Functions also allow layer stacking, as Sigmoid and Hyperbolic Tangent do, because of their nonlinear nature.

$$Y = \frac{e^{[(input*weight)+bias]_j}}{e^{\sum(input*weight)+bias}} \; from \; j = 1 \; to \; j = n$$

Figure 11. Softmax Function.

Finally, we have the Softmax Function, shown in Figure 11. The Softmax Function has the peculiarity of outputting more than one value, and that all of the values it outputs summed together equal to one. This is why it is used in multiple classification problems, similarly to the way the Sigmoid Function is used in binary classifications. Its outputs can be transposed to a categorical probability distribution, telling the probability of any of the cases it classifies being true. The variable n is to the number of outputs the function will output which has to equal its number of inputs.

[22] Leaky ReLU has proven to outperform ReLU, especially in large datasets, while it can over fit to the training data in smaller datasets, reporting lower accuracy than ReLU.

There are more activation functions besides the ones described here, but this is the most commonly used in Artificial Neurons and will be the ones used in the construction of a Machine Learning Algorithm at the end of the work.

3.4 Backpropagation and how it works:

The process of backpropagation and its steps will be described. Loss functions and optimization algorithms will be defined and exemplified using the gradient descent optimization algorithm.

When describing the functioning of a Machine Learning algorithm we have supposed that the algorithm has already been trained. The training of the algorithm is where the most important part of Machine Learning comes in: backpropagation. Backpropagation is the process the algorithm uses to learn to distinguish the patterns in the data and classify it.

The objective of an ML algorithm is usually to classify or regress. In classification tasks, the algorithm has to classify the input into premade groups, outputting a number with categorical value.[23] In regression tasks, the algorithm must output a number with an actual numerical value.[24]

This is done feeding data into an Artificial Neural Network, which runs through the different layers of the ANN through the different neurons, which apply different activation functions until a number reaches the Output Layer, either in the form of a single number if the neuron of the Output Layer uses a Sigmoid Function, or of a probability distribution if it uses the Softmax Function.

As described above, every neuron has different values that it uses to process the data it is given: the weights and the biases. This is the values that are "learnt" using backpropagation. The weights and biases affect the calculations of the ANN immensely and even the slightest variation can have a very important impact on the output of the network and, consequently, the classification of the input.

This is why this values must be carefully adjusted to make the classification as precise as possible. Backpropagation makes it possible.

To start, the weights and biases of each neuron are initiated using random values between 0 and 1. The range of the random values is important because it makes the training process shorter, as the weights are usually values between 0 and 1.

[23] A very common example of a classification task is classifying points in a two or three dimensional space which are in different groups based on their position in that two or three dimensional space. The value outputted by the ML algorithm would be an integer which refers to one of the groups. (With 0 being a group, 1 being another one, and so on…) This example will be explored later in the work with an actual ML algorithm and a Neural Network.

[24] A very common example of a regression task is having to guess the price of a house using the number of rooms, size, place in a city… The value outputted by the ML algorithm would be the estimated prize of the house based on the value of the features given as an input.

Then, a dataset with previously classified data is run through the network. When the calculations are done, the outputs are compared to the ones in the dataset, generated by humans and are considered an accurate and reliable classification.

The loss of the ANN is calculated using a Loss Function, which can also be called an Error Function. There are many Loss Functions available, closely related to statistics, but because of their relatively low impact on the final calculations, they will not be enumerated and mathematically described as the activation functions were.[25]

Now, we have the output of the network or actual output, the previously classified output or ideal output and the error of the network. With this values, we can proceed to update the weights and the biases using an optimization algorithm.

Optimization algorithms are complex mathematical expressions which calculate for how much of the error is every weight and bias responsible and by how much it should be adjusted in order to minimize this error or loss. Many optimization algorithms have appeared during the last years because of the increasing popularity of machine learning, but the most common one and the simplest to explain is the gradient descent.

Gradient descent works on the principle of the loss function being convex. If so, there is a point on the function's space where the error will be 0 for every weight and bias involved.[26] This seems easy to picture in a two-dimensional graph or even a three dimensional one, but in the reality, every weight and bias gets its own mathematical dimension, so the graph is impossible to imagine for us.

As you can see in Figure 12, each weight is represented in one direction and the loss, referred to as cost in the figure, also gets its own dimension for representation. The point where we want the network to get is the one where the loss of every weight and bias is 0, or as near to 0 as possible.

Figure 12 represents a convex loss function. Depending on the loss function used, the graphical representation of it would not actually be convex, meaning that it would have local minimums where the loss would be near zero but also an absolute minimum where the loss would be actually at its minimum (perhaps even 0).

[25] This doesn't mean that Loss Functions are not important for the final calculations, but rather that their impact on those is smaller than the one of Activation Functions. There are also many more Loss Function that there are Activation Functions, and their use is sometimes very specific to certain ANNs, so they will not be enumerated.

[26] The point of an error of absolute 0 is actually almost never reached.

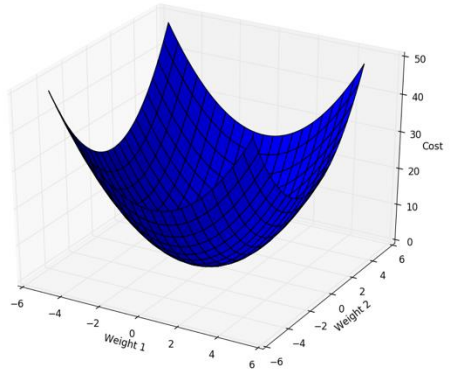


Figure 12. Representation of a three-dimensional Mean Square Error function. Rohan Kapur. ©

Gradient descent is the mathematical method used to calculate at which point of the loss function the network actually is. Because of the weights and biases being started randomly, the network could be at any point of the graph. Remember that when we refer to the graph it has as many dimensions as weights and biases the network has.[27]

Using the derivate of the position of loss of the network respect the derivate of the weights and biases that we want to update. When the derivate equals 0, we will have found the point in the loss function where the loss is also 0.

The operation with derivate actually tells us the slope of the point where the network is. Knowing the slope, the network is adjusted to move the weights and biases towards the point where this slope is smaller. Because of the loss function is convex, this point will always be closer to the 0 loss point than the previous one.[28]

$$W = W - \alpha \frac{\partial L}{\partial W}, B = B - \alpha \frac{\partial L}{\partial B}$$

Figure 13. The mathematical expression of gradient descent.

[27] In the graph on Figure 12 there are only a two weights and their respective loss. This would represent a very simple network which would likely not be very effective, and has been used only for visualization purposes, due for us being impossible to visualize the graph of a complex DNN with thousands of dimensions or more.

[28] In case of non-convex loss functions, other optimizations would be needed. One of the most popular in that field would be stochastic gradient descent, an altered version of gradient descent designed for non-convex function. Stochastic gradient descent could also be successfully applied to convex loss functions.

Figure 13 shows the mathematical expression of gradient descent. W and B refer to the weighs and biases respectively. The Greek letter α refers to a hyperparameter called the learning rate. Hyperparameters are the setting for the neural networks previously defined by the programmer and which the network cannot change by itself. Finally, L refers to the loss calculated using a loss function.

A learning rate is a number which multiplies the value for which the weights and biases will be modified, making learning easier and faster. It is the most important of all the hyperparameters in a Neural Network because of its impact on learning speed and effectiveness. If it is too big, the network would never reach the 0 loss point[29], while if it is too small it would take ages for the network to reach that point.[30]

When the weights and biases are updated using gradient descent, the process of learning is repeated. The network calculates its output, the loss is computed and gradient descent takes place again. This process repeated during usually thousands of times makes the network minimize the loss and, thereby, become actually effective. All the training data, though, is not fed at the same time, but rather in small pieces called batches. The number of times the optimization algorithm is run is called epochs, or training epochs.[31]

To summarize, backpropagation is the process by which the weights and biases of a neural network are updated in order to make the loss of the network as small as possible. This is done with an optimization algorithm, the most common of which is gradient descent. Then, it is multiplied by the learning rate to affect the speed of the networks weights and biases adjustment.

[29] When a gradient descent optimizer doesn't reach a minimum because of the learning rate it leads to exploding gradient. Exploding gradient happens when the loss grows bigger instead of smaller.

[30] While a learning rate too big can cause an exploding gradient, a learning rate too small is not the cause of the problem referred to as vanishing gradient. This can often lead to confusion.

[31] Epochs is one of the most common names, but the number of times the training of a ML algorithm is done can also be referred to as iterations or training steps.

There are many other optimization algorithms.[32] The main problem of gradient descent is that it is very slow and computationally intensive. Another issue is that loss functions are rarely convex[33], usually having local maximums and minimums. This issues can be solved with other optimization algorithms, but because of its mathematical complexity, they will not be described in this work. [34]

[32] Other popular optimization algorithms could be the Adam optimizer, the normal equation, polynomial regression, linear regression…

[33] The main convex loss function is the Mean Squared Error (MSE), which has the following form: $MSE = \frac{1}{n}\sum(Y_j - \hat{Y}_j)^2$ where n is the number of predictions, Y is the correct label, $\hat{Y}$ is the label predicted by the algorithm and j is the current prediction being calculated, going from 1 to n.

[34] Optimization algorithms use complex mathematical operations such as derivatives, partial derivatives or integrals, and therefore won't be described in more detail.

4. Artificial Neural Networks Types

4.1 Convolutional Neural Networks

One of the subtypes of an Artificial Neural Network is Convolutional Neural Networks (CNNs). This is mostly used to process images or videos. Their structure will be described and functioning will be described.

Artificial Neural Networks are, as stated before, the key element that makes Machine Learning and therefore AIs possible. Because of this, their structure is not as simple and regular as one might think. There are many types of ANNs, all of them variations of the previously described structure: a number of neurons organized in three types of layers which are the Input Layer, Hidden Layer and Output Layer.

Convolutional Neural Networks, abbreviated as CNNs, are a subfield on ANNs which are used to process images or videos instead of working with the raw numbers directly. They are capable of doing the feature extraction from a picture or video and create a series of numbers associated with them in order to be able to run them through the network; multiplying them by weights, adding biases, applying the activation functions…

Convolutional Neural Networks are a type of Deep Neural Networks. Remember the distinction between Feed-Forward ANNs and Deep ANNs[35]; DNNs have more than one hidden layer, while Feed-Forward ANNs have only one. CNNs, therefore, have more than one Hidden Layer.

Convolutional Neural Networks come from the need of teaching an AI to recognize what appears on an image. This has been a long-standing problem in the field of computer sciences. A child can, from a very early age, identify and classify what it sees. A computer, though, cannot carry on such task so easily. In fact, recognizing objects in pictures was virtually impossible until the appearance of CNNs.

The main objective of CNNs is to be eventually able to describe a picture as a person would do. This is done through a classification of every single object in the picture. If the network is able to identify and classify an object in an image, in the form of an apparently random succession of pixels, the CNN is considered successful.

[35] Remember that Deep ANNs are usually abbreviated in the form of DNN. This abbreviation may be used during the work.

CNNs, like every other type of ANN, require an input. In this case, the input consists of a raw image[36], either in black-and-white or in colour. The network translates the input image into a grid of pixels, with the mathematical form of a matrix, giving to each of them a value which represents the colour the pixel is displaying. In case of a black-and-white image, this value consists of a grayscale[37] and in the case of a coloured picture, each picture might get more than on value depending on the colour space used (RGB, sRGB, CMYK…).[38] Colour spaces are different ways of displaying the colours, associating different numbers to them according to their features.

The inputs are then fed to the CNN and, if the process were as simple as the one of a Feed-Forward ANN, the training process would proceed as normal, but it does not. Why is that? Because if the process I have just described is followed the CNN would only be able to identify objects in the centre of the image, and not if they are off-centre as the matrix representation of an image is sensitive to position.

This problem has several solutions. The simplest one consists of making an area, smaller than the whole picture, and then move it around the picture until it finds an object perfectly centred in it. The area has to change its size and check for the presence of the object in all areas of the picture. This method requires a lot of calculations because it has to run the CNN several times around the same picture, being very inefficient and relying too much on brute power.

Another simple method is to take the dataset in which the training of the ML algorithm will take place and edit it so the objects to be recognized are no longer in the centre. This can be easy to do with simple objects that are not part of a real-life or complex picture but is nearly impossible to do complex images.

The truly effective solution is a process called convolution, which gives its name to the Convolutional Neural Networks. Convolution makes small changes to the process described previously of feeding the network the value of the pixels in the image and running them straight into the ANN.

[36] CNNs can also work in videos but they use a different structure in that case, mixing elements of a CNN and a Recurrent Neural Network.

[37] The necessity to use a grey scale is because it allows for a pixel's darkness to be used as a feature in the form of a number, which a ANN always requires. Other ways to measure the darkness of a pixel could be used.

[38] The necessity to use a colour space is because it allows for colours to be used as features in the form of numbers, which a ANN always requires. Other ways to measure the colour could be used.

The first step consists of breaking the input into smaller parts. These smaller parts are not only independent groups of pixels, but rather overlapping parts of the image. The fact that the parts of the image overlap are important to make sure that no data is lost in following parts of the CNN work.

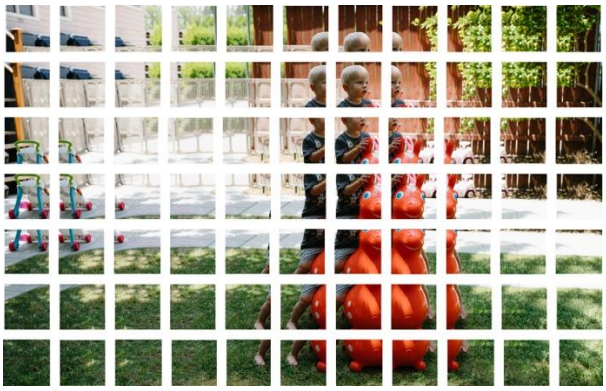


Figure 1. Example of a how a picture could be divided by a CNN. Adam Geitgey. ©

These small parts of the picture are run through a standard ANN, being the input values the grey scale of the pixels or the colour values. The goal of this small network is not to classify the part of the picture is given, but rather process it with the same values in every small square of the original picture. This will generate a matrix or array with numbers that represent different values in the image. This process is called embedding.[39] We, as humans, cannot know what these values represent, but we know that they somehow define the image for the ML algorithm, and that the CNN will be able to find objects in the images with this random-looking numbers.

After dividing the image into segments and processing them separately, they have to be simplified because, as you may remember, the fragments of the image overlapped on each other, giving the network more information than it needs. The simplifying is done through a process called max pooling, that keeps only the highest values of the matrix or array. This is done to try to make the network focus on what is important on the picture, rather than losing time processing information the network itself marked as less relevant during the previous phase of convolution.[40]

Finally, the arrays or matrixes are put together into a big deep ANN, which leads to a classification of the image.

[39] Embedding is not only used in CNNs and is also frequent in other types of ANNs, especially in Natural Language processing. The formal definition of embedding is the process to transcribe objects into vectors of a given dimension, with the individual dimensions of the vector having no inherent meaning.

[40] There are other ways of making the CNN focus in specific part of the picture, but max pooling has proven to be one of the most useful while easily computable.

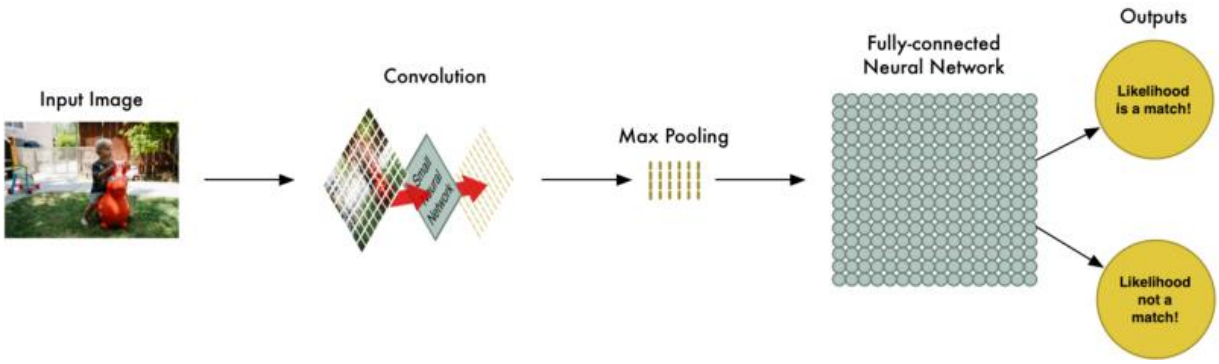


Figure 2. Representation of the process of a CNN to classify a picture. Adam Geitgey. ©

In conclusion, a CNN can be defined as the sum of small ANNs that divide an image through convolution, a max pooling algorithm that simplifies its results, and a deep ANN that classifies the image.[41] A Convolutional Neural Network is the sum of all these programming structures, and not only the last deep ANN.

Because of the complexity of object classification in images, the dataset required to train the ML algorithm with the CNN has to be much bigger than the ones used for other ML tasks such as predictions based on data or natural language processing. The fact that the images must be previously classified also adds complexity to the task of creating a database to train CNNs.

This is why Convolutional Neural Networks have only recently started to appear and visualization and perception are one of the most recent fields AIs have successfully mastered.

[41] This is one the most common structure of a CNN. Other more complex structures exist which can achieve higher accuracy.

4.2 Recurrent Neural Networks:

Another of the subtypes of an Artificial Neural Network is Recurrent Neural Networks (RNNs). This is used in tasks that require the network to take the last prediction it made as its input. RNNs are mostly used to find complex patterns in data. They can be understood as ANNs with short-term memory.

Recurrent Neural Networks, abbreviated as RNNs, are other of the subtypes of Artificial Neural Networks. Their particularity is the ability to be able to have as input the last prediction (output or label) they made, and use it as an input the next time they have to make a prediction. This gives what resembles short-time memory in the human brain. RNNs are born, in fact, to try to give Machine Learning algorithms this short-time memory. Usual ANNs without this memory are called stateless algorithms or stateless ANNs.

A subtype of Recurrent Neural Networks, Long-Short Term Memory Reclusive Neural Networks (LSTMs), give the network not only a short-term memory but a long-term one. LSTMs will be explained after the basic structure of RNNs is exposed.

The main goal of Recurrent Neural Networks is to find complex patterns in the data they are presented with, usually sequential data. Their most usual application is in Natural Language Processing, abbreviated as NLP. Natural Language Processing tries to give the machine the ability to understand human language and answer accordingly to what they understand. RNNs are incredibly useful in text generation, one of the main fields of NLP, where the last output, which is essentially the last word or character, has a great impact in the moment of choosing which the next word or character will be.

The structure of a Recurrent Neural Network differs a bit from the one of a stateless Artificial Neural Network.

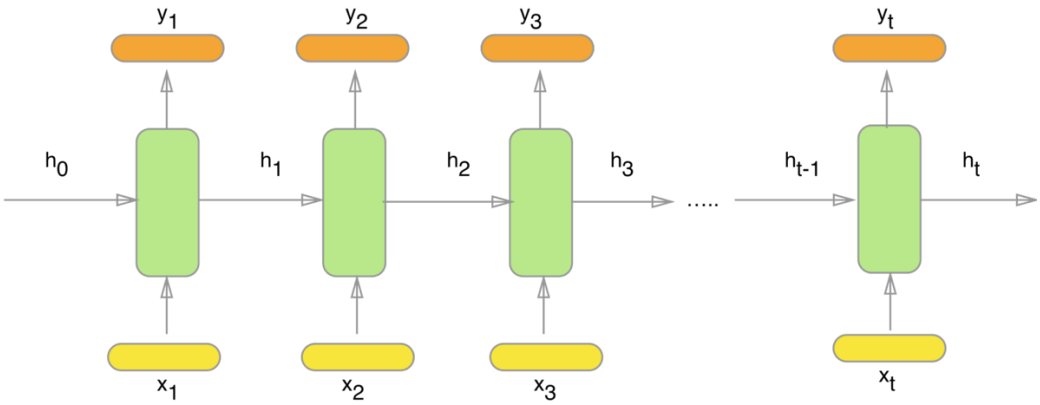


Figure 3. Diagram of a Recurrent Neural Network. Simeon Kostadinov. ©

In Figure 3 an RNN is represented using a diagram that looks different from the ones used to represent stateless ANNs. This is because in the representation of RNNs you add one important variable; time, or iteration, represented in the diagram with an “h”.

Usually, when representing an RNN, it is done as representing each layer instead of each neuron. Therefore, the yellow squares in the diagram do not represent a single input neuron but the Input Layer as a whole. The same happens with the green squares representing the Hidden Layers and the orange ones the Output Layers.

As in stateless ANNs, the input or features enter the network (x), are run through the hidden layers and then an output or label is outputted (y). In the first iteration of the RNN, the process is exactly the same. In the second one, though, when the Neural Network is given an input to classify, the output of the previous iteration is also inputted into the Hidden Layer of the network.

This means that when labelling this second output, the network will have taken into account its last prediction, giving the network short-term memory. The previous output has its associated weights and biases and is run through the Hidden Layer as if it were a normal feature.

The process is repeated in every iteration, which take into account the calculation of the previous one until the network is stopped. This means that, no matter how many iterations your network has, if input the same values, it will not always return the same output.

To understand this better, you can imagine a Recurrent Neural Network in the field of Natural Language Processing. When human talk, we always take into account the last word we pronounced, and that last word influences our next work according to the laws of grammar and syntax.

RNNs have been successful at trying to imitate human speech. In fact, when given a big enough dataset to be trained on, RNNs can output some human-like text which only experts could tell apart from the real human text.

```
                                KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
    Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
  So drop upon your lordship's head, and your opinion
                Shall be against your honour.
```

Figure 4. Text generated by an RNN trained on Shakespeare’s work. Andrej Karpathy. ©

The text in Figure 4, for instance, has been generated by an RNN which was trained using the whole work of Shakespeare using three Hidden Layers of 512 neurons each. Then, the network is executed with no direct input and predicts the most likely character to respond to that blank input with. Taking that last character into account, it outputs another one, and one more after that, creating words and sentences which obey grammar rules. Remember that in any moment the Network has been taught how grammar works, it has just found a pattern in the data and applied it to generate text of its own.

In conclusion, Recurrent Neural Networks are a very powerful tool, empowering Neural Networks to find very complex patterns in very complex sequential data while giving them the ability to have what resembles short-time memory in the human brain.

RNNs, though their effectiveness, have a problem when being trained related to backpropagation; vanishing gradient. Vanishing gradient happens because when trying to calculate the impact each weight has had in the final label, the input that comes from the previous iteration must also be taken into account. But in order to calculate the impact of this input of the previous iteration, the one from an even earlier iteration must be taken into account too, as it contributed to the result.

This essentially means that when trying to optimize the RNN through backpropagation, the optimization algorithm would have to go up to the initial iteration to successfully optimize the RNN. This, due to the mathematical usage of derivatives in the process, makes the gradient diminish tending to 0 in the initial iteration, even when it shouldn't be 0. This happens because of the time-distributed structure of RNNs, and is the major problem they face.

Vanishing gradient don't make RNNs useless, but makes them less effective than they could hypothetically be. It is also the reason why RNNs are almost never used with their raw structure, the one described before, but rather using different neuron structures. The most common of this variations of a RNN is Long-Short Term Memory networks.

4.3 Long-Short Term Memory Recurrent Neural Networks:

Recurrent Neural Networks are, as explained before, a variant of ANNs with short time memory. Long-Short Term Memory Recurrent Neural Networks abbreviated as LSTMs are a type of RNN which features a structure that gives them a long-term memory as well as a short-term one.

Long-Short Term Recurrent Neural Networks (abbreviated as LSTMs) share the same network structure as Recurrent Neural Networks, which is responsible for their short-term memory but have a particular structure in a neuron level that also gives them a long-term memory. LSTMs were created to make RNNs even more capable, and most of the RNNs success actually comes from LSTMs.

They were also created in order to fix the vanishing gradient problem, a mathematical issue related to the optimization algorithms which made impossible to correctly modify the parameters of a Recurrent Neural Network, weights and biases, because their gradient tended to zero due to the time-distributed structure of RNNs.

LSTMs have a particular structure in a neuron level, meaning that each individual neuron does not have the same structure as in feed-forward ANNs, deep ANNs or CNNs.

Remember how a neuron or perceptron[42] worked: they take one or more inputs (x), multiply them by a weight associated with each input (w) and then add a bias to the sum (b). Finally, they apply an activation function to the result and output it to the next neuron. LSTMs add complexity to this structure, and they are best explained using a diagram:

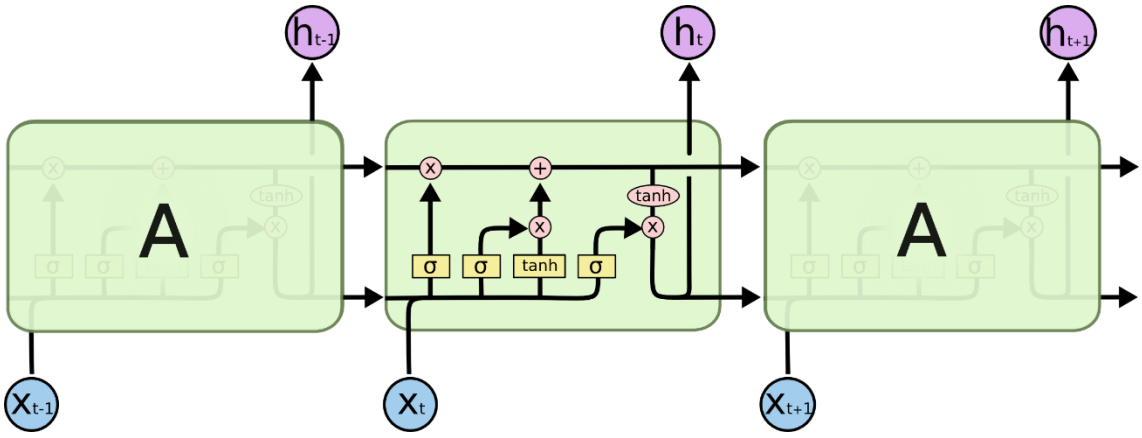


Figure 5. Diagram of a Long-Short Term Recurrent Neural Network. Christopher Olah. ©

[42] Neuron and perceptron can be used as synonyms when talking about not LSTM networks. In the context of LSTMs or any other type of ANN which changes the basic neuron structure, only the term network can be used.

This is the diagram of one of the possible structures of an LSTM neuron. Be aware that there are more possible structures that an LSTM could have, using different activation functions and mechanisms, and this is only one possible structure, although one of the most common ones.

Ignoring what happens inside of a neuron, the layer structure of LSTMs is the same as in RNNs, because of the first being a subtype of the latter. They have an input layer, identified in the diagram with an X, a hidden layer, identified with an A, and an output layer, identified with an h. Then, there is a channel connecting the output of the previous iteration of the network with the current one, identified with an arrow between the hidden layers.

Now, let's pay attention to what happens inside the neurons. In the diagram, the arrows identify numbers moving through the neuron, which can be either integers, floats, arrays, matrixes, vectors… The pink circles represent mathematical operations; such as sums or multiplications.

Finally, the yellow squares represent activation functions; such as the hyperbolic tangent or the sigmoid function, represented with the Greek character "σ", usually used to represent the sigmoid function in diagrams.

The neuron has two memory lanes, inherited from the Hidden Layer of the previous iteration. One is the long-term memory lane, also called the cell state, the upper one in the diagram, while the other one is the short term one.

As you can see in the diagram, the cell state only gets modified if the new input is important enough (its importance being measured with a sigmoid function and a hyperbolic tangent). The modification of the cell state is the second thing done by the neuron using the input, as it can be seen in the diagram; the first three activation functions and operations are in order to do so. Before that, though, the input and the short-term memory, equivalent to the previous iteration's output are summed together.

Next, after the short-term memory and input being added and the cell state calculated, the output of the neuron is processed using two more activation functions, one for the input and short-term memory (which had been previously combined), and one for the cell state (which has been previously modified). In the case there are more iterations of the RNN, the output is carried onto the Hidden Layer of the next iteration and so is the cell state, providing the next iteration with a new cell state and a new short-term memory, that will be combined with its new input.

5. Tensorflow

5.1 What is Tensorflow:

Tensorflow is an open-source library based on Python used to build, train and deploy Machine Learning algorithms. Tensorflow runs in three different levels: High Level APIs, Estimators and Low Level APIs.

Because of the complexity of Artificial Neural Networks and Machine Learning algorithms they are hard to implement when programming them. This has led to the appearance of several libraries to implement Machine Learning. These libraries allow to program ML algorithms easily, with the libraries taking care of most of the mathematical operations, activation functions, loss functions and optimization algorithms. If all of those were to be manually defined by the programmers of the ANNs every time one is to be built, it would take a lot of time, effort and resources.[43]

Most of this libraries are open-source and based on the programming language Python. Most of the big companies in the tech world have committed themselves to develop a specific library for Machine Learning.[44] Tensorflow has become one of the most popular. Developed by Google, Tensorflow was first built as an internal tool for Google's ML tasks, was later made open for everybody to access it.

Tensorflow was first released on November 9th 2015 for Python. Nowadays, besides Python, it supports other programming languages such as C++ or Java, but Python remains the main language. ML algorithms built with Tensorflow can be run in all main desktop platforms (Linux, Windows and macOS) supporting GPU[45] acceleration, and mobile platforms (Android and iOS).

[43] The implementation of complex mathematical operations in code is complex and often requires other libraries, such as Numpy, the most popular mathematical open-source library. The mathematical operations on most ML libraries are built on Numpy.

[44] Other popular ML libraries include Torch and PyTorch, by Facebook, OpenAI, by Tesla, Core ML and Create ML, by Apple, Microsoft Cognitive Toolkit (CNTK), Scikit Learn, by Scipy…

[45] GPUs, which stands for Graphics Processing Units, can run ANNs faster and more efficiently than CPUs (Central Processing Units). Tensorflow supports the execution of ANNs in Nvidia GPUs with the CUDA architecture.

Besides GPUs and CPUs, ML algorithms built in Tensorflow can be trained and executed in Tensor Processing Units or TPUs. TPUs are a circuit designed by Google specifically for Machine Learning. TPUs offer a huge computational[46] power and are offered by Google in its cloud computing services.

Tensorflow packs three levels of APIs. The most basic one is the Low-Level API. A more abstract and simple one is the Tensorflow estimator API, which packs premade estimators which automatically build on the structures of the Low-Level API. Finally, the High-Level APIs are the most abstract and easy-to-use APIs in Tensorflow. They allow to build and train complex ANNs with less code than the Low-level API and estimators. The most popular High-Level API for Tensorflow is Keras. Keras was not originally built by Google but its popularity has made them implement it natively in Tensorflow.[47]

Besides its ML APIs, Tensorflow includes a data visualization library called Tensorboard. Tensorboard allows displaying the different parameters of a Tensorflow ANN. It includes a visual representation of the network, the evolution of different parameters during the training phase and even the visualization of images if the network in question is a CNN.

All of this Tensorflow features; its APIs will be deeply explained in following parts of the work.

[46] The last generation of TPUs (third generation) offers up to 100 PFLOPS. FLOPS are Floating Point Operations Per Second, and a PFLOPS represent 10^{15} FLOPS.

[47] Besides Tensorflow, Keras can run on top of other ML libraries such as CNTK or Theano.

5.2 Low Level API:

The Low Level API of Tensorflow deals with data in the form of Tensors. The data is processed in Graphs built of Operations which run in Sessions. Operations, Graphs and Sessions are the basic structures of Tensorflow's Low Level API and they will be described.

Tensorflow's Low Level API allows to create complex ANNs with relative ease. The data computed by Tensorflow must always be in the form of Tensors. Tensors are defined inside of Tensorflow as n-dimensional Numpy arrays. This means that they can be arrays or matrixes of any given dimension. They are also of a given type, which can be floats[48], integers[49] or strings[50]. Each element of each Tensor must have the same type.

Tensors are computed in Graphs. Graphs are a network of Operations which define the direction in which the data flows through these Operations and the connections between the Operations. The data that flows between Operations in a Graph is in the form of Tensors.

An Operation is every node that conforms a Graph. Operations can be of four main types: constants, variables, placeholders or mathematical operations.[51]

Constants are a Tensor whose values cannot be changed by the network. The Tensor can contain either integers or floats.

Variables are also a Tensor which can contain either integers of floats, but whose values can be changed by the network. Because of that, they are usually used to represent weights and biases, which need to be changed during the training phase.

Placeholders are Operations without any data. For a Graph to be run, a Feed Dictionary must be included, which relates every placeholder in the Graph with some data to fill it. They are usually used as the input nodes of the network, or the Input Layer. They do not contain any value because in must be fed from outside the network. Variables must always be initialized within a Session, before they can be called.

[48] Floats represent floating points, or numerical values with a decimal plane.
[49] Integers represent whole numbers, every numerical value that can be represented without a fractional unit.
[50] Strings represent a chain of characters such as words or sentences. They cannot be used for mathematical operations without being embedded.
[51] Not all Operations (nodes in a graph) represent mathematical operations. This can often lead to confusion.

Finally, Operations that represent mathematical operations express the relationship between Operations of the other types and output the result of the mathematical operation between them. For instance, a Matrix Sum Operation could relate a constant and a variable and would output its matrix sum.[52]

Graphs, then, are the relationship between Operations, but they are just that. A Graph cannot be run by itself, so a Session is required. Sessions don't actually run Graphs, but rather operations. When an operation is run within a Session, Tensorflow automatically refers to the Graphs in which this operation is in and runs it until the Operation specified is met. If the Operation specified is the final one in a Graph, what would be an Output Layer, then the whole Graph is run in order to reach this one Operation.

All of this basic Tensorflow structures can be easily related to the concepts in an Artificial Neural Network. The Input Layer of the Network would be built of placeholders, which wait to receive an input. The Hidden Layers would be formed of variables, containing the values of the weights and biases, and mathematical operations, which would multiply the inputs with the weights, sum the biases and apply an activation function. The Output Layer would be the output of the last mathematical operation of the last Hidden Layer, outputting the label predicted by the network.

Besides providing this structures to build ANNs, Tensorflow's Low Level API also provides structures to quickly create Hidden Layers with a given number of neurons and a given activation function, with erases the need to create each neuron manually as a set of Operations.

It also provides with a lot of built-in Loss Functions and Optimization Algorithm which allow for the network build using a Graph of Operations to be trained with relative ease within a Session.

In conclusion, Tensorflow's Low Level API contains the structures to build an Artificial Neural Network from scratch and train it without requiring to manually define the neurons, mathematical operations, activation functions, loss functions or optimization algorithms. All of those are provided by Tensorflow.

The Estimator API and the High Level APIs are built on top of this Low Level API, and therefore use the same structures. This means that the only difference between those APIs and the Low Level one is that the higher ones provide a bigger abstraction and are simpler to use.

[52] This would be taking into account that the dimensions of the constant and the variable allow for them to be summed.

5.3 Estimator API:

The Estimator API of Tensorflow is built on top of its Low Level API, and uses the same structures in the background. To make networks easier to build, though, it offers faster ways to create and train them, without requiring to define each Operation in a Graph or create Sessions. It uses feature columns as a data structure, input functions to feed data into the network and estimators as the network itself.

Tensorflow's Estimator API is built on top of the Low Level API. Practically, this makes the Estimator API a High Level API, but they are separated inside of Tensorflow's documentation, so will be explained separately in the work.

Being a High Level API, essentially means that though never specified when building a network with the Estimator API, it is actually being built of Operations in Graphs and being run in Sessions. It also means that it deals with ANNs in a more abstract way, without directly referring to all of their elements and making them easier to build.

The Estimator API, though, gives more options to more easily create network and specially to manage the data before it is fed into the network. In the Estimator API Artificial Neural Networks are referred to as estimators, which can be of different types, such as a linear or dense[53] regression and a linear or dense classified.

The basic data structure inside an estimator is a feature column. Feature columns in an estimator would be the equivalent of the Input Layer of an ANN. Feature columns must be defined individually and they must have a type. The most common type is a feature column with numerical data, but the Estimator API also provides ways to create feature columns with categorical data.

Feature columns automatically transform that categorical data into numerical values. They can do so in different ways, such as assigning each category an integer or using embedding to define the categories. As mention when describing CNNS, embedding is a process that transforms an object into a vectors of a given dimension. Each dimension does not mean anything but as a whole they represent the embedded object. The more dimensions used, the more abstract and complex the embedding will be.

[53] Dense is the term used in the Estimator API to refer to Deep Neural Networks. Dense Neural Networks can also be used in other contexts to refer to Deep Neural Networks, and both are abbreviated as DNN.

The ability to deal with categorical data is one of its biggest advantages against the Low Level API, which requires the data to be already preprocessed. The Estimator API also allows data to be imported from pandas[54], while the Low Level API only supports Numpy arrays or matrixes.

When the feature columns have been defined, they must be filled with the actual data. This is done through an input function, which is the one that allows the data to be inputted from either pandas or Numpy.

In the input function the features must be specified, and they must correspond to the feature columns; there must be the same number of features in the data and feature columns in the estimator and their type of data (categorical or numerical) must correspond to the one specified in the feature column.

The input function also requires to specify the labels of the inputs, which must be in the same datatype of the features.[55] It also requires the size of the batches in which the data will be given to the estimator. Remember that the data is never fed entirely at the same type to an ANN, but is rather given in smaller pieces called batches. The size of the batch is the number of elements that will be fed at the same type. It must also be specified whether to shuffle the data or leave it in the same order as it is given.

Finally, the proper network, the estimator, must be created. It will always require a list of all the feature columns. If the estimator used is linear, it doesn't require anything else. If it is dense, though, it requires a size (the number of Hidden Layers and of neurons in each one of them). You can also specify the Activation Function and the Optimization Algorithm. If you don't, it will set them to the default provided by Tensorflow.[56]

Then, the estimator is automatically created. You can now train it, run it in new data when it is trained or test it. All of this can be done with built-in functions which automatically train the estimator, output labels if given new data and evaluate it with different metrics if tested.

To summarize, the Estimator API provides an easier way to build ANNs, called estimators, train them, use them and evaluate them. All while based on Tensorflow's Low Level API.

[54] Pandas is an open-source library for data manipulation. It can be used to import form a file into an ML algorithm using its DataFrame object. The Estimator API allows Tensorflow to use the DataFrames data as feature columns, both numerical and categorical data.

[55] If the features are imported from pandas, so must the labels. If they are imported form Numpy, so must the labels.

[56] The default Activation Function is ReLU and the default Optimization Algorithm is Adagrad Optimizer, based on Gradient Descent.

6. Example of an Artificial Neural Network

6.1 Low level API

In this part of the work an Artificial Neural Network will be created to exemplify all the concepts explained previously using Tensorflow's Low Level API. The goal of the network will be to classify points in a 3D space into its correspondent blobs, based only in their position. To do so, a deep ANN will be created, trained, and tested.

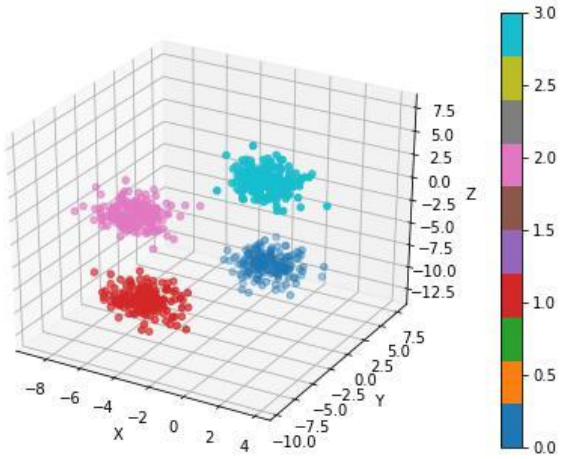
To exemplify and better understand all the concepts explained previously about Machine Learning, perceptrons and Artificial Neural Networks, we will continue to see a real ANN used to classify blobs. The blobs, created randomly, will have three features or three coordinates (X, Y and Z) and a label. Each blob will be formed by a number of points which will be close to each other, and therefore will have close coordinates, and the same label.
The objective of the ANN will be to be able to tell to which blob each point belongs.

```
In [1]: import tensorflow as tf

        import numpy as np

        import matplotlib.pyplot as plt

        from mpl_toolkits.mplot3d import Axes3D

        %matplotlib inline

        from sklearn.datasets import make_blobs

        from sklearn.model_selection import train_test_split as split
```

To start, all libraries and dependencies used to must be imported. We will be using the following dependencies:

- **Tensorflow**: an open-source Machine Learning library.
- **Numpy**: an open-source library used to perform mathematical operations and to manage data
- **Matplotlib**: an open-source library used for data visualization purposes. (Only some of its tools are imported and used).
- **Scikit-learn**: an open-source Machine Learning library. (Only some of its tools are imported and used).

```
In [2]: data = make_blobs(1000, 3, 4, random_state = 1)

In [3]: features = data[0]

        labels = data[1]

In [4]: features_train, features_test, labels_train, labels_test = split(features, labels,
                                                                         test_size=0.3)

In [5]: x_features = features_train[:,0]

        y_features = features_train[:,1]

        z_features = features_train[:,2]

In [6]: x_test = features_test[:,0]

        y_test = features_test[:,1]

        z_test = features_test[:,2]

In [7]: fig = plt.figure()

        ax = Axes3D(fig)

        graphic = ax.scatter(x_features, y_features, z_features, c = labels_train, cmap =
                                                                    "tab10")
        ax.set_xlabel('X')

        ax.set_ylabel('Y')

        ax.set_zlabel('Z')

        plt.colorbar(graphic)

Out[7]: <matplotlib.colorbar.Colorbar at 0x2a662fb14a8>
```



Here the dataset used for the classification has been created. As you can see in the figure, four clear blobs have been randomly generated. The position of the points on the graph is the features of the point (its three coordinates) and the colour of the point is its label. As a human, you can easily see that all points that share a label have something in common; they are close to each other, forming a blob. A computer, though, cannot reach this conclusion by itself because all it sees is random looking numbers.

In order for a computer to be able to tell the blobs apart, it must learn how to do so. This is where Machine Learning comes in and ANNs come in. Let's remember the steps used to build a Machine Learning algorithm shown in Figure 1. First we must have the input and its features. In this case, the input of our Machine Learning algorithm is a point, and its features will be the three coordinates of a point: a X coordinate, a Y coordinate and a Z coordinate. This are randomized by the "make_blobs" function from the Scikit-learn library and are given in the form of floats, or floating points.

Then we must classify all the points. We will do so with an Artificial Neural Network. Finally, we will test our classification using some other points the Machine Learning algorithm hasn't been trained on. This is the reason that we have split our data in two sets: the training set (with the variables "features_train" and "labels_train") and the testing set (with the variables "features_test" and "lables_test") with the 70% of the data being used for training and the remaining 30% for testing.

So there is only one thing remaining to have out blobs classification algorithm built: creating the ANN.

```
In [8]: n_hidden1_neurons = 16

        n_hidden2_neurons = 8

        n_hidden3_neurons = 4

        n_outputs = 1

        lr = 0.01

        epochs = 1000

        n_batches = 70

        batch_size = 10
```

Now the ANN has been created, but it has not been trained yet. Let's go through the creation of the network before training it. First, the hyperparamethers of the network have been defined. As you can see, it has three Hidden Layers; the first one containing 16 neurons or perceptrons, the second one 8 and the third one 4. The number of neurons in each hidden layer is defined by the variables: "n_hidden1_neurons", "n_hidden2_neurons" and "n_hidden3_neurons". Finally, in the output layer, we have a single output, meaning that the output will be a single value, as defined by the "n_outputs" variable.

Then we have other hyperparamethers such as the learning rate and the number of epochs the training will perform. This values can be easily changed and the ones used are probably not the optimal ones. If one were to carefully tune this values, different results and probably better accuracy could be achieved. But because of the nature of the classification task the ANN has to perform and its relative simplicity, we can leave these values at relatively standard values. The learning rate is defined with the variable "lr" and the number of epochs of the training by the variable "epochs".

```
In [9]: x = tf.placeholder(dtype = tf.float32, shape = [None, 3])
        y = tf.placeholder(dtype = tf.float32, shape =[None, 1])
```

Now the Tensorflow graph of the network is created. A placeholder "x", which will be the input of the features in the network, is created, as well as a placeholder "y", which will be the label of each input, or the blob to which a point with the "x" labels belongs to. The dimensions or shape of the placeholders is also specified as a matrix, containing an unspecified number of items ("None") and, in the case of the "x" placeholders, 3 features, and in the case of the "y" placeholder, a single label.

```
In [10]: h1 = tf.layers.dense(x, n_hidden1_neurons, name = "hidden1", activation = tf.nn.relu)
         h2 = tf.layers.dense(h1, n_hidden2_neurons, name = "hidden2",
                              activation = tf.nn.relu)
         h3 = tf.layers.dense(h2, n_hidden3_neurons, name = "hidden3",
                              activation = tf.nn.relu)
         y_predicted = tf.layers.dense(h3, n_outputs, name = "output")
```

Then the three hidden layers are created using a Tensorflow function to make the process easier. The class "layers.dense()" in Tensorflow automatically creates a Layer and its neurons, which all its weights and biases, requiring the following arguments:

- **Data input:** the inputs of the layer, either a previous layer or the features themselves.
- **Number of neurons:** the number of neurons in the layer.
- **Name:** the name of the layer. The name is optional but useful for data visualization using Tensorboard as will be done later.
- **Activation function:** the activation function of the neurons in the layer. If none is specified, none will be used.

In our particular ANN, we have three Hidden Layers ("h1", "h2" and "h3") and an output layer ("y_predicted"). The value in the output layer will correspond to the prediction of the net-work, its classification of the point given at the beginning. If the network were to be 100% accurate, the output would always be equal to the label, therefore "y_preddicted" would be equal to "y". We have chosen ReLU as the activation function used by all the networks in all the hidden layers. If a different activation function was used, different results could be achieved. This could include either a bigger or smaller loss at the end, but because of the relative simplicity of the classification problem, ReLU is enough to classify the points into blobs.

```
In [11]: loss = tf.losses.mean_squared_error(y, y_predicted)
```

This calculation to determine the accuracy of the network is done with the loss. The higher the loss, the less accurate the network is. We define the "loss" variable using the Mean Squared Error formula between the label ("y") and the predicted label or output ("y_predicted"). The Mean Squared Error is just one of the possible methods that could be used to calculate loss, but also one of the most common ones. Many more loss calculation formulas are included with Tensorflow.

```
In [12]: optimizer = tf.train.GradientDescentOptimizer(learning_rate = lr)
```

```
In [13]: train = optimizer.minimize(loss)
```

Finally, we create the optimizer of the Neural Network. Remember that this is the most important item as it is the responsible for the ML algorithm's learning. Luckily, Tensorflow also provides a great number of optimization algorithms. If we had to implement them manually, it would require very high mathematical knowledge that would be very difficult to implement in Python. Tensorflow's built in optimization algorithms have already been defined and perform all these operations in the background. For our particular ANN, the Gradient Descent optimizer has been chosen and it has been given the learning rate of the network as an argument. Then, we specify what the optimization algorithm must do; minimize the rate, tuning the Network's weights and biases to do so.

```
In [14]: init = tf.global_variables_initializer()
```

Finally, we have the variable initializer. Tensorflow always requires the variable to be initialized, so this is what "init" does, initialize all the variables using the default Global Variables Initializer ("global_variables_initializer()"). Now the ANN has been built, but is not operational yet. To make it run, we have to create a session. It is only within a session that a Tensorflow network can run and be trained.

First of all, the session in which the network will run is created. Because of this code being run in a Jupyter Notebook, an Interactive Session is used. Interactive Sessions, though, are only useful in programs such as Jupyter Notebook, while usually all the code that follows the Interactive Session would be in the following form:

```
In [15]: with tf.Session() as sess:
             print("All the code following would be executed if written inside this loop.")
```

```
All the code following would be executed if written inside this loop.
```

If you were to run that with the same code that follows the Interactive Session you would get the same results, but for Jupyter Notebook based project Interactive Notebooks are more useful because they allow the session to be used through different cells, while regular Session with the previously seen structure can only be used inside one cell, because they automatically close at the end of the loop.

```
In [16]: sess = tf.InteractiveSession()
```

```
In [17]: sess.run(init)
```

First of all, as said before, the variables must be initialized, so “init” is run first, which corresponds, as you may remember, to the Global Variables Initializer. Then is when the training takes part. As hard as it may seem to understand, we will go through it line by line to make it easier.

```
In [18]: for epoch in range(epochs + 1):
             for batch in range(n_batches):
                 batch_x = np.column_stack((x_features
                                            [batch*batch_size:batch_size*(batch+
                                            1)],
                                            y_features
                                            [batch*batch_size:batch_size*(batch+
                                            1)],
                                            z_features
                                            [batch*batch_size:batch_size*(batch+
                                            1)]))
                 batch_y =
                 np.column_stack((labels_train[batch*batch_size:batch_size*(batch+1)])).
                 T
                 sess.run(train, feed_dict = {x: batch_x, y:batch_y})
             if epoch%100 == 0:
                 print("Epoch:" + str(epoch))
                 print("Loss:" + str(sess.run(loss, feed_dict = {x: batch_x, y:batch_y})))
```

Epoch:0 Loss:0.08059037

Epoch:100 Loss:0.00018433068

Epoch:200 Loss:6.233634e-05

Epoch:300 Loss:3.980687e-05

Epoch:400 Loss:3.625945e-05

Epoch:500 Loss:3.4087432e-05

Epoch:600 Loss:3.1305615e-05

Epoch:700 Loss:2.9387598e-05

Epoch:800 Loss:2.773869e-05

Epoch:900 Loss:2.5857948e-05

Epoch:1000 Loss:2.0983156e-05

First the main loop is created to make the code run for as many times as we specify. Each iteration of the code is an epoch, and we defined the number of epochs in the training previously with the "epochs" hyperparameter. So this loop makes the network run the training code several times, as many as epochs were declared. Then another loop is created. This loop is responsible of feeding the different batches into the network.

To get the different batches, we must get different parts of the dataset, each one containing a number of points equal to "batch_size". The number of batches we want to get correspond to the variable "n_batches". To get the points and coordinates of each batch, we have to get their "x_features", "y_features" and "z_features" and stack them in a matrix. The labels of this points must be also separated from the dataset, and made transformed into a different matrix. Using numpy's function "column_stack()", a matrix is created using the data specified as an argument of the function.

The specific points to get for each batch is in the indexes of the arrays containing the features as the following:

Operation to get the points in each batch: [batch*batch_size:batch_size*(batch+1)]

In Python, the first item in any data structure is always the item "0", while the second one being the item "1". With this operation, we get all the points from the point which corresponds to the index of current batch multiplied by the number of points in each batch to the point which corresponds to the index of current batch plus one multiplied by the number of points in each batch. For instance, in the first iteration of the batch loop the operation would be the following:

Points of the first batch: [0*10:(0+1)*10] = [0:10]

This would get all the points from point 0 to point 10. In the second iteration, it would take all points from the point 10 to the 20:

Points of the second batch: [1*10:(1+1)*10] = [10:20]

When the batches have been created, we proceed to the actual training of the network. We do so by running the "train" object, whose objective was to run the optimizer, a Gradient Descent Optimizer, and try to minimize the loss. With the "feed_dict" we specify which data to feed into the graph in the placeholders "x" and "y" that had been previously defined. We feed the two batches: "x_batch", containing the features of the points, its coordinates, and "y_batch", containing its labels, or the blob to which they belong.

Finally, the last two lines of the training loop are just used to print the loss on the current epoch of training if that epoch is divisible by 100. This is used in order to avoid getting the loss in every epoch of training, which would not be very useful because it would not vary much between each individual epoch. Every 100 epochs, though, we can see a clear tendency of the loss value diminishing, which means that the network's training is being successful.

```
In [19]: test_x = np.column_stack((x_test, y_test, z_test))

test_y = np.column_stack((labels_test)).T
```

Now we create the matrixes containing more points that were not used during training to test the networks accuracy. We do so in the same way as when we were creating the batches, stacking the columns of the three features and transforming the array containing the labels of those points into a matrix.

```
In [20]: print("Test Loss:" + str(sess.run(loss, feed_dict = {x: test_x, y: test_y})))

Test Loss:0.0007578826

In [21]: sess.close()
```

Finally, we print the Final Loss of our ANN. That final loss is actually the loss calculated using the testing points. The smaller the final loss is; the more accurately can the network classify the points. As you can see, the loss is indeed very low, meaning that the network has achieved great accuracy and would be able to correctly classify any given point into its correspondent blob.

So we have successfully created a Deep Artificial Neural Network and we have trained it to classify points in a three-dimensional space into different blobs, using all the different structures and mechanisms explained before.

6.2 Keras API

In this part of the work an Artificial Neural Network will be created to exemplify all the concepts explained previously using Tensorflow's the Keras High Level API, which will be simpler than using the Low Level API. The goal of the network will be again to classify points in a 3D space into its correspondent blobs, based only in their position. To do so, a deep ANN will be created, trained, and tested.

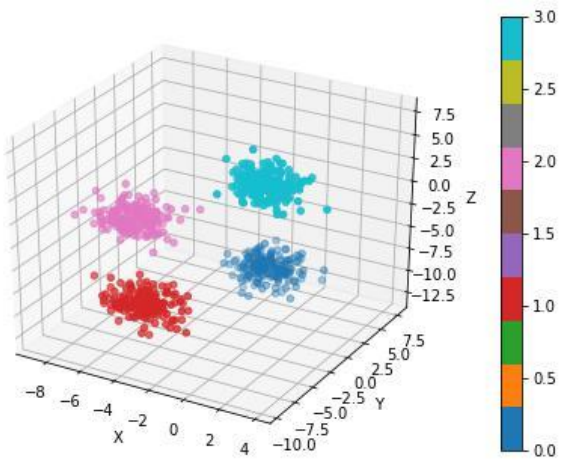
As you have seen, creating an Artificial Neural Network using Tensorflow's Low Level API can be quite complex, even when dealing with very simple tasks like classification of points into blobs. To make it simpler, High Level APIs can be used, such as the Estimators API or Keras. To show how much simpler it can be and why Keras will be used later in the work, an ANN with the same task as before will be built using Keras. This will be again to classify points in a three dimensional space into blobs. The points used will be the same, so that the same results can be achieved.

```
In [1]: import tensorflow as tf
        import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        %matplotlib inline
        from sklearn.datasets import make_blobs
        from sklearn.model_selection import train_test_split as split
```

As before, all dependencies will be imported. The dependencies will be the same as when using the Low Level API since Keras is included in Tensorflow. Beware that if using an old version of Tensorflow, Keras might not be included with it.

```
In [2]: data = make_blobs(1000, 3, 4, random_state = 1)
In [3]: features = data[0]
        labels = data[1]
In [4]: features_train, features_test, labels_train, labels_test = split(features,
                                                                         labels, test_size=0.3)
```

```
In [5]: x_features = features_train[:,0]

        y_features = features_train[:,1]

        z_features = features_train[:,2]

In [6]: x_test = features_test[:,0]

        y_test = features_test[:,1]

        z_test = features_test[:,2]

In [7]: fig = plt.figure()

        ax = Axes3D(fig)

        graphic = ax.scatter(x_features, y_features, z_features,
                             c = labels_train, cmap = "tab10")
        ax.set_xlabel('X')

        ax.set_ylabel('Y')

        ax.set_zlabel('Z')

        plt.colorbar(graphic)

Out[7]: <matplotlib.colorbar.Colorbar at 0x253d581e160>
```



We get the exact same data as before, using the random state when generating the points to be sure it is the case. Using the same random state in both ANNs ensures that the data randomly generated will be the same for both networks, making it essentially not random. If any random state was given, different datasets would be generated every time, but the ANN would be able to work with them anyway.

```
In [8]: n_hidden1_neurons = 16
```

```
n_hidden2_neurons = 8

n_hidden3_neurons = 4

n_outputs = 1

lr = 0.01

epochs = 1000

n_batches = 70

batch_size = 10
```

We, again, define the hyperparamethers of the network. They are exactly the same as we used in the Low Level API example. Actually, in Keras, less hyperparamethers would be required, so not all of them will be used.

```
In [9]: network = tf.keras.Sequential()
```

Now the model must be built. Keras has a Sequential model built in, which is the most com-mon ANN model. The model though, when created, is empty, meaning it has only an input layer. The Hidden Layers and Output Layers must be added manually, but in a simpler way than using the Low Level API.

```
In [10]: network.add(tf.keras.layers.Dense(n_hidden1_neurons, activation = 'relu'))

         network.add(tf.keras.layers.Dense(n_hidden2_neurons, activation = 'relu'))

         network.add(tf.keras.layers.Dense(n_hidden3_neurons, activation = 'relu'))

         network.add(tf.keras.layers.Dense(n_outputs, activation = 'relu'))
```

This adds all layers to the network. They are of the same dimensions and use the same activation functions as when the Low Level API was used, in order to achieve the same results.

```
In [11]: network.compile(optimizer = tf.train.GradientDescentOptimizer(learning_rate = lr),
                         loss = tf.losses.mean_squared_error)
```

Now we compile the network, meaning that we define the optimizer and loss functions it will use during training. Again, Gradient Descent and Mean Squared Error are used.

```
In [12]: network.fit(features_train, labels_train, batch_size = batch_size,
                    epochs = 1000, verbose = 0)

Out[12]: <tensorflow.python.keras.callbacks.History at 0x253dbf34b38>
```

In [13]: network.evaluate(features_test, labels_test)

300/300 [==============================] - 0s 196us/step

Out[13]: 0.0004519956519652624

Finally, we train and test the network. Training is the process where Keras' simplicity is most easily seen. Without needing to create a Session, or get the data from the batches, a built in feature can be used to fit the data into the network. All it need in order to train the network are the features and the labels. It doesn't even need to have the features separated, as the Low Level API required. The verbose setting set to two means that it will not output the results of each epoch during training. If verbose was not specified, it would output the results of every epoch during training, which would be the same as we saw when using the Low Level API.

To end with, we see that the final loss when evaluating the networks performance quite similar as it was before. This is because all the same parameters were used in both examples. This also hap-pens because, although we manually define it, Keras is actually building a Graph of Operations and running it within a Session. Using the Low Level API requires all of these to be manually built, while Keras builds them automatically, greatly simplifying that process. Any small difference in the loss is due to some points being shuffled during training and the random initiation of weights and biases.

7. Sources

Ahirwar, K. (2017). Everything you need to know about Neural Networks. *Medium: Hackernoon*.

Ars Magna. (2018). *Wikipedia*.

Artificial intelligence. (2018). *Merrian-Webster*.

Bell, L. (12/1/2016). Machine learning versus AI: what's the difference? *Wired*.

Borgen, H. (2016). Machine Learning in a Year. *Medium: Learning New Stuff*.

Castrounis, A. (2016). Artificial Intelligence, Deep Learning, and Neural Networks, Explained. *KDnuggets*.

Copeland, B. (2018). Artificial intelligence. *Encyclopaedia Britannica*.

Cuartero, F. (10/11/2012). Ramon Llull, el Ars Magna y la Informática. *El País*.

Curso intensivo de aprendizaje automático (2018).

Dertat, A. (2018). Applied Deep Learning. *Medium: Towards Data Science*.

Deutsch, D. (10/3/2012). Philosophy will be the key that unlocks artificial intelligence. *The Guardian*.

Fidora, A., Sierra, C., Barberà, S., Beuchot, M., Bonet, E., Bonner, A., . . . Wyllie, G. (2011). *Ramon Llull: From the Ars Magna to Artificial Intelligence.* Barcelona: Artificial Intelligence Research Institute.

Geitgey, A. (2014). Machine Learning is Fun! *Medium*.

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn & Tensorflow* (Fifth Rele ed.). (N. Tache, Ed.) Sebastapol: O'Reilly Media Inc.

Hartikka, L. (2017). A step-by-step guide to building a simple chess AI. *Medium: freeCodeCamp*.

History of Logic. (2018). *Wikipedia*.

Jaulent, E. (2008). El Ars Generalis ultima de Ramon Llull: Presupuestos metafísicos y éticos. *Intituto Brasileiro de Filosofia e Ciência*.

Kapur, R. (2018). Neural Networks & The Backpropagation Algorithm, Explained. *Medium: A Year Of Artificial Intelligence*.

Karpathy, A. (2015). The Unreasonable Effectiveness of Recurrent Neural Networks.

Kashnitskiy, Y. (2018). Open Machine Learning Course. *Medium: Open Machine Learning Course*.

Kingma, D., & Lei Ba, J. (2015). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION.

Kostadinov, S. (2017). How Recurrent Neural Networks work. *Medium: Towards Data Science*.

Kwiatkowski, S. (2018). Machine Learning From Scratch. *Medium: Towards Data Science*.

Machine Learning. (2018). *Wikipedia*.

Machine Learning and AI Foundations: Value Estimators (2017).

Mazur, M. (2018). A Step by Step Backpropagation Example.

McDonald, C. (2017). Machine Learning Fundamentals. *Medium: Towards Data Science*.

Nag, D. (2017). What separates us from AI. *Medium*.

Nielsen, M. (2017). How the backpropagation algorithm works.

Nielsen, M. (2017). *Neural Networks and Deep Learning.*

Philosophy of Artificial Intelligence. (2018). *Wikipedia*.

Ramon Llull. (2018). *Wikipedia*.

Rodriguez, T. (4/5/2017). Machine Learning y Deep Learning: cómo entender las claves del presente y futuro de la inteligencia artificial. *Xataka*.

Rouse, M. (December of 2016). What is AI?

Ruder, S. (2017). An overview of gradient descent optimization algorithms.

Seif, G. (2018). I'll tell you why Deep Learning is so popular and in demand. *Medium: The Startup*.

Sequence2Sequence. (2018). *Amazon*.

Sharma, A. (2017). Understanding Activation Functions in Neural Networks. *Medium: The Theory Of Everything*.

Sharma, S. (2017). Activation Functions: Neural Networks. *Medium: Towards Data Scienc*.

Simonite, T. (7/12/2017). Artificial Intelligence Seeks An Ethical Conscience. *Wired*.

Soni, A. (2017). Machine Learning with JavaScript. *Medium: Hackernoon*.

Tensorflow. (2018). *Google*.

Turing, A. (1 de 10 de 1950). Computing Machinery and Intelligence. *Mind, LIX*(236), 433-460.

Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent. (2017). *Medium: Towards Data Science*.

Willems, K. (2016). Jupyter Notebook Tutorial: The Definitive Guide. *DataCamp*.

Yang, J. (2017). ReLU and Softmax Activation Functions. *Github*.

Yufeng, G. (2017). Interactive Data Science with Jupyter Notebooks. *Medium: Towards Data Science*.

Yufeng, G. (2017). Plain and Simple Estimators. *Medium: Towards Data Science*.

Practical Frame

Building an Artificial Intelligence to
generate Classical music.

Francesc Pifarré Esquerda

Index

1. Introduction

Now that the way Machine Learning algorithms and Neural Networks work has been explained, we can proceed to building one. The main goal of this part of the work will be to build a Machine Learning algorithm to generate music. As explained before, Artificial Neural Networks are powerful tools to classify and predict data. Is this ability to predict new data that opens the door for what humans would call "creativity", something rather controversial when talking about an Artificial Intelligence.

The aim of this part of the work is only to check if an Artificial Intelligence is able to generate music, being aware of the complex philosophical debate about intelligence and creativity in the field of AI. The aim of the work is not to determine of the music produced by an AI is actually original and creative or whether the AI was intelligence when creating this music.

Using Artificial Neural Networks to imitate human art has been done for some time with success. Let's think about text, for instance. As seen when explaining LSTMs, those can be used to predict the next character in a sequence of text. For instance, a trained LSTM could receive the following sentence as an input and output the predicted next character for it:

$$[H, E, L, L] \rightarrow [O]$$

In this case, a single character has been generated from a previous sequence, but this is hardly creative. To create a more complex sentence, we would have to keep the LSTM running, giving it as an input the same sequence plus the character it has predicted:

$$[H, E, L, L, O] \rightarrow [\ ]$$

Now, it predicts a white space to separate this word from a hypothetical next one. But, what if we were to keep the algorithm running in its own predictions? Then it would keep generating characters, and then sentences, and eventually a full text. This text would then have been created by an Artificial Intelligence completely and would, therefore, be original. This task, as simple as it may seem, is actually quite complex. The Network would have to learn by itself how grammar works, how to write words and full sentences only by looking at sample data, which would be full texts in this case.

And the complexity of creating an ML algorithm to generate text doesn't only lie on the complexity of the task. Remember that ANNs can only deal with numerical data and text is, obviously, not numerical. This requires to develop systems to "translate" text into

numerical values so that the Network can deal with them. And only then, when dealing with numerical data, could the network reach an "understanding" of language to generate text by itself. This way to generate text is so abstract that it is hard to say that the Network "understands" language, or that it is "creative" to generate completely original pieces. But instead of diving into the task of making clear whether the ML algorithms are creative or can understand something as complex as language, we will focus on building those algorithms.

So, seeing that ANNs are able to generate text; could they generate music?

One may think that music and text are completely different from each other, but they are not that different. You could think of musical notes as characters in a text, and of musical pieces as sentences. In fact, both are sequential structures of different elements, either characters or notes. This means that an Artificial Neural Network should be able to predict musical notes as it can predict text.

In fact, due to the mathematical structure of music, it could be even easier for an ML algorithm to predict music than text. Music is highly mathematical, and ANNs can only deal with mathematical terms, which makes music quite suitable for them.

This means that an AI using ML should be perfectly able to compose music following the same principles of text prediction; giving it a starting sequence and giving the network the task of predicting the following elements, in this case, musical notes.

So let's try to build an Artificial Intelligence based on Machine Learning to compose music using Long-Short Term Memory Recurrent Neural Networks to answer this question; can an Artificial Intelligence compose music?

2. Methodology

Our goal will be to build a Machine Learning algorithm to generate music based on a dataset of 80 violin scores from Johann Sebastian Bach, Antonio Vivaldi, Ludwig van Beethoven, Wolfgang Amadeus Mozart and Niccolò Paganini, classical music composers. The scores used have been downloaded from the MuseScore open dataset of scores.
The 80 scores contain over 40.000 notes which will be the dataset used to train the algorithm.
The hardware in which the network will be trained will be an Intel® Core™ i7-8650U CPU with 16GB of DDR3 RAM and an Nvidia GeForce® GTX 1050 with 2GB of GDDR5 RAM dGPU running Windows 10 Pro 1803 as Operating System.
The ML algorithm will be coded in Python 3.6.6 on a Jupyter Notebook environment (Jupyter Lab version 0.35.2). To extract the music data from their files we will use the music21 library (version 5.3), developed by the MIT and "os" to access them in the memory. To preprocess this data and make it numerical we will use Pandas (version 0.23.4) and Numpy (1.14.5), libraries used to manage data and perform operations on it. Finally, we will also use the Tensorflow library (GPU 1.10 version) to build the Artificial Neural Network that will process the data and, eventually, generate new music.

```
In [1]: import music21

import os

import numpy as np

import pandas as pd

import tensorflow as tf
```

2.1 Extracting the data from MIDI files

To predict music, the first thing that must be done is extract the training data from its source files. In our case, the source files are MIDI files, gathered from the MuseScore public score database. All the files have been saved in the same folder for convenience.

```
In [2]: path = " /Data"
```

As stated before, there are 80 different files in that folder, so a list of them must be created to extract data from them individually.

```
In [3]: files = os.listdir(path)
```

Using “os”, we list add the name of every file in that folder to a list, and then, we add them to the original path so that we can read them.

```
In [4]: for file in range(len(files)):
            files[file] = path + files[file]
```

Now that we have a list of all files, we must extract the musical data from them. To do so, an empty list is created and two values are added to it. The first value will be the note pitch and the second value will be its duration.

```
In [5]: data = []
```

To add those values to the list we must first convert the MIDI file to a score object in music21. Once the score object has been created, we check for every note on it and add its pitch and length to the previously created data list. Checking if the current element is a note is done in order to avoid having other musical structures in the data file, such as chords, as this would greatly complicate the preprocessing of data and the eventual prediction of music.

```
In [6]: for file in files:
            score = music21.converter.parse(file)[0]
for note in score.getElementsByClass("Note"):
            indv_note = [note.pitch, note.duration.quarterLength]
            data.append(indv_note)
```

So now that we have a list of all notes in the dataset we can proceed to preprocessing them.

2.2 Data preprocessing

```
In [7]: data = pd.DataFrame(data, columns = ["Note", "Duration"])

In [8]: data.head()

Out[8]:
  Note  Duration
0 F#4   1
1 F#4   1
2 F#4   1
3 D4    4
4 E4    1

In [9]: data_length = len(data)

In [10]: data_length

Out[10]: 41811
```

Preprocessing data is of great importance and it can determine the ML algorithms success. As you can see above, we have over 40.000 notes with two features each; their pitch and duration. This are categorical features, meaning that a note can be categorized by its pitch or the musical note it represents and by its duration. This data has to be preprocessed so that it can be used as an input for an ANN.

```
In [11]: sequence_length = 25
```

It is also now that we define the length of the sequences that will be used later in the network. The sequence length can be described as the network's memory. A sequence length of 25 means that the network will take into account the last 25 notes it has seen before predicting the next one. A sequence length too short can lead to the network failing to correctly predict the next note, but a sequence length too long is also not ideal, as it can lead to the network memorizing the prediction from the dataset rather that finding patterns in the data, which would lead to bad results when generating its own data.

```
In [12]: %store data_length

%store sequence_length

Stored 'data_length' (int)
```

Stored 'sequence_length' (int)

2.2.1 Notes preprocessing

In [13]: notes = data["Note"]

In [14]: notes.head()

Out[14]: 0 F#4

1 F#4

2 F#4

3 D4

4E4

Name: Note, dtype: object

First, let's preprocess the notes. We will transform them into categorical numerical values, so that they can be used by the network. We will do so by using the one-hot encoding method. A very easy way to visualize how this encoding method works is looking at the following example:

[1,2,3] -> [[1,0,0], [0,1,0], [0,0,1]]

In this example, you can see that each value has been given an index in the one-hot encoded array, being as many indexes as categories in the original data. In our case, we have a much higher number of classes (notes), going as high as 54, as seen bellow with the variable "n_notes". This means that manually encoding them would require a lot of effort. Luckily, Keras packs a one-hot encoder which automatically does what we have manually done in the previous example.

The real purpose on one-hot encoding will become clearer once we build the Neural Network that will process the data. They are useful because they can be interpreted as a probability distribution. Returning to our example, a one in the second index of the array can be interpreted as a 100% probability of the original value encoded being part of the second category. This probability distribution form will be very useful when trying to predict notes.

2.2.1.1 Notes to categorical

The Keras function to one-hot encode requires the data to be in the form of a Numpy ndarray. Numpy ndarrays are n-dimensional arrays, meaning that they store data in a "n" number of dimensions. We transform the Pandas Series, the data structure used above, to a Numpy array with the data type being strings.

```
In [15]: notes = np.asarray(notes, dtype = "str")

In [16]: notes

Out[16]: array(['F#4', 'F#4', 'F#4', ..., 'C5', 'A4', 'C5'], dtype='<U64')

In [17]: all_notes = set(notes)
```

We now list every different note, without them being in the "all_notes" list more than once. We do so in order to be able to transform the notes, currently in the form of strings to integers, so that we can one-hot encode them. But before being able to translate them into integers, dictionaries must be created.

```
In [18]: notes_dictionary = {}

notes_dictionary_inv = {}

In [19]: counter = 0

In [20]: for note in all_notes:

        notes_dictionary[note] = counter

        counter += 1

In [21]: counter = 0

In [22]: for note in all_notes:

        notes_dictionary_inv[counter] = note

        counter += 1
```

Two dictionaries are created. The first one translates the string of the note to an integer, while the second one does the opposite; translate an integer to a note in the form of a string. For instance; the note with the identifier 1 corresponds to A6 in musical notation, and another one like B4 has a different integer assigned, 13 in this case. Using this correspondence and this dictionaries, we can proceed to transform the original notes list into integers, so we can later encode them.

```
In [23]: notes_int = []

In [24]: for note in notes:

             notes_int.append(notes_dictionary[note])
```

Now we have all the notes extracted from the original files in the form of integers, so we can proceed to one-hot encoding them.

```
In [25]: notes_preprocessed = tf.keras.utils.to_categorical(notes_int)

In [26]: notes_preprocessed.shape

Out[26]: (41811, 54)
```

Using a built-in Keras function, one-hot encoding them is very simple. As you can see we end up with two-dimensional array consisting of 41.811 items (the total number of notes in the original dataset) with 54 different features per item (the total number of different notes in the dataset). This 54 features per item are actually a one-hot encoded array representing the note, being formed of 53 zeroes and a one. The index of the array where the one is located is the one that gives the array its value.

```
In [27]: n_notes = notes_preprocessed.shape[1]

In [28]: n_notes

Out[28]: 54
```

And to finish the notes encoding, we store the total number of different notes, as it will be needed later when defining the Neural Network.

```
In [29]: %store n_notes
```

```
%store notes_dictionary_inv
```

```
Stored 'n_notes' (int)
Stored 'notes_dictionary_inv' (dict)
```

2.2.1.2 Notes to sequences

```
In [30]: inputs_notes = []
outputs_notes = []
In [31]: for i in range(data_length - sequence_length):
             sequence = notes_preprocessed[i:i + sequence_length]
             following_character = notes_preprocessed[i + sequence_length]
             inputs_notes.append(sequence)
             outputs_notes.append(following_character)
In [32]: features_notes = np.asarray(inputs_notes)
labels_notes = np.asarray(outputs_notes)
In [33]: features_notes.shape
Out[33]: (41786, 25, 54)
In [34]: labels_notes.shape
Out[34]: (41786, 54)
```

Finally, to finish the preprocessing of the notes, we must save them in the form of inputs and outputs. This requires storing the notes in groups of 25, the sequence length we defined previously, or the number of previous notes the Neural Network will store in its memory when it is built. So we take 25 notes and group them together into the "inputs_notes" array while we store the next note in the "outputs_notes" array, as it is the value the network will try to predict.

Finally, we transform these two arrays into ndarrays. The "features_notes" ndarray is a three-dimensional array; being the first one the number of samples (the length of the

dataset minus the sequence length), the second dimension the length of the sequence, and the third one the number of features in each item, or the total number of different notes. The “labels_notes” ndarray has only two dimensions, as it doesn’t have a sequence length because of it corresponding to the output of the network and the network predicting a single note at a time. Here, the first dimension also corresponds to the number of samples, the same as before, and to the total number of different notes, also the same as before.

In [35]: %**store** features_notes

%**store** labels_notes

Stored 'features_notes' (ndarray)

Stored 'labels_notes' (ndarray)

2.2.2 Durations preprocessing

In [36]: data.head()

Out[36]:

	Note	Duration
0	F#4	1
1	F#4	1
2	F#4	1
3	D4	4
4	E4	1

But remember that the note was not the only feature that we had. We also have another feature, the duration of the note. To preprocess it, we will do exactly the same as when encoding and preprocessing the notes.

2.2.2.1 Durations to categorical

In [37]: durations = data["Duration"]

In [38]: durations.head()

Out[38]:

0	1
1	1
2	1
3	4

```
41

Name: Duration, dtype: object

In [39]: durations = np.asarray(durations, dtype = "float32")

In [40]: all_durations = set(durations)

In [41]: durations_dictionary = {}

durations_dictionary_inv = {}

In [42]: counter = 0

In [43]: for duration in all_durations:

             durations_dictionary[duration] = counter

             counter += 1

In [44]: counter = 0

In [45]: for duration in all_durations:

             durations_dictionary_inv[counter] = duration

             counter += 1

In [46]: durations_int = []

In [47]: for duration in durations:

             notes_int.append(durations_dictionary[duration])

In [48]: durations_preprocessed = tf.keras.utils.to_categorical(durations)

In [49]: n_durations = durations_preprocessed.shape[1]

In [50]: n_durations

Out[50]: 10

In [51]: %store n_durations

%store durations_dictionary_inv

Stored 'n_durations' (int)

Stored 'durations_dictionary_inv' (dict)
```

2.2.2.2 Durations to sequences

```
In [52]: inputs_durations = []

outputs_durations = []

In [53]: for i in range(data_length - sequence_length):

             sequence = durations_preprocessed[i:i + sequence_length]

             following_character = durations_preprocessed[i + sequence_length]

             inputs_durations.append(sequence)

             outputs_durations.append(following_character)

In [54]: features_durations = np.asarray(inputs_durations)

labels_durations = np.asarray(outputs_durations)

In [55]: features_durations.shape

Out[55]: (41786, 25, 10)

In [56]: labels_durations.shape

Out[56]: (41786, 10)

In [57]: %store features_durations

%store labels_durations

Stored 'features_durations' (ndarray)

Stored 'labels_durations' (ndarray)
```

2.3 Creating the networks

Now it comes the time to define the Neural Network that we will use to generate music. For convenience, two different networks will be built; one to predict the musical note and another one to predict its duration.
But even if building and training two completely different networks, we will use the same optimizer for both of them. If more than one optimizer was used, one for each with different learning rates, better results might be achieved. For simplicity, though, we will use the same optimizer for both networks.
The optimizer in question is the Adam optimizer. This optimizer is built upon Gradient De-scent, but is much more complex and effective. The version of this optimizer in Keras is the one proposed by Diederik P. Kingma and Jimmy Lei Ba in their 2015 paper "Adam: a method for stochastic optimization". The details of this optimization algorithm will not be described as it involves many complex mathematical calculations.

```
In [58]: optimizer = tf.keras.optimizers.Adam(lr = 1e-3)
```

2.3.1 Notes network

```
In [59]: network_notes = tf.keras.Sequential()
```

First of all, we define the Keras model that will be used. This will be a sequential model, that is one corresponding to the usual ANN structure. This will be the model tasked with predicting notes, and another one will be later built to predict the durations of those notes. Both models will have been of the same type; Recurrent Neural Network with Long-Short Term Memory cells.
This will give the model both a short term memory, the last note predicted by the model, and a long term memory, the last 25 notes predicted by the model (corresponding to the sequence length, as stated before when defining this variable).

```
In [60]: network_notes.add(tf.keras.layers.LSTM(128, input_shape = (sequence_length,
n_notes),

network_notes.add(tf.keras.layers.Dropout(0.1))

network_notes.add(tf.keras.layers.LSTM(128))
```

```
network_notes.add(tf.keras.layers.Dropout(0.1))

network_notes.add(tf.keras.layers.Dense(n_notes, activation = "softmax"))

In [61]: network_notes.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 25, 128)           93696
_________________________________________________________________
dropout (Dropout)            (None, 25, 128)           0
_________________________________________________________________
lstm_1 (LSTM)                (None, 128)               131584
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense (Dense)                (None, 54)                6966
=================================================================
Total params: 232,246
Trainable params: 232,246
Non-trainable params: 0
_________________________________________________________________
```

After defining the model, we will add layers to it. In the first layer, we must specify the shape of its input. This shape would be the equivalent to the Input Layer of the model, as Keras doesn't have a specific Input Layer module. The dimensions of this Input Layer, of the shape of the first Hidden Layer will be the sequence length and the total number of notes to classify. The layer we add to the model is a LSTM with 128 neurons. We set the parameter "return_sequences" to true for the first layer as another LSTM layer will be

following. This second LSTM also has 128 neurons. This parameter is needed in order to give the short term memory ability to the second layer of the network.

Between the Hidden Layers of the model, we add what Keras calls a Dropout Layer. The dropout layer is the amount of neurons of the following layers that will not be trained during the current epoch. It represents a percentage; 0.1 actually refers to 10% of the neurons.

This means that at each training batch, 10% of the neurons will not be trained. This is done in order to avoid overfitting. Overfitting happens when the network's weights and biases are tuned to much to match the current training batch.

If this were to happen, the network would be to fitted to process the training examples, but would not be able to generalize what it has learned to new hypothetical data that it has not processed before.

The last layer of the network is a Dense Layer. This is the Output Layer, with the same number of neurons are different types of notes exist in the dataset, and the Softmax function is used as the activation function. For the other layer the default activation function (Hyperbolic Tangent) was used.

The main characteristic of the Softmax function is that it outputs a number of values that sum 1. This means that it essentially outputs a probability distribution. This means that it is ideal to process one-hot encoded sequences, as this sequences also sum 1 (with the index of the value one being the note and the other ones being zero).

The network, when used to generate new notes, will output a number of values in each index of the newly formed array. The index with the higher number will be the one the network "believes" would be the following one based on its training.

We end up with a RNN with a LSTM structure formed by two Hidden Layers with a Hyperbolic Tangent activation function and an Output Layer with the Softmax function. So now we can proceed to training the network on the dataset.

```
In [62]: network_notes.compile(loss = "categorical_crossentropy",
                               optimizer = optimizer)
```

Before training it the last step is compiling it. Doing so requires to choose a loss function and an optimizer. The loss function used here is Categorical Crossentropy, the one used when the Softmax layer is used in the Output Layer.

```
In [63]: network_notes.fit(features_notes, labels_notes, batch_size = 100,
```

```
                                    epochs = 50, verbose = 0)
```

Out[63]: <tensorflow.python.keras.callbacks.History at 0x13d3d7d3518>

Now we train the network with a batch size of 100 examples and for 50 epochs. This takes quite a long time, depending on the computing power of the computer used for training

```
In [64]: save_notes = "/Models/notes.h5py
```

```
In [65]: network_notes.save(save_notes)
```

```
In [66]: %store save_notes
```

Stored 'save_notes' (str)

Finally, we save the network to the "save_notes" path. Keras allows networks being saved in a custom file format called h5py. This format allows Keras to save networks and later restore them with the exact same structure, weights and biases it had before been saved. In case we were to train the network more, the optimizer for the network and its loss function would also be saves in the same exact state.

2.3.2 Durations network

For the network that will predict the duration of the notes, the exact same thing as with the network used to predict the notes.

```
In [67]: network_durations = tf.keras.Sequential()
In [68]: network_durations.add(tf.keras.layers.LSTM(128, input_shape =
(sequence_length, n_dur
```

```
network_durations.add(tf.keras.layers.Dropout(0.1))
```

```
network_durations.add(tf.keras.layers.LSTM(128))
```

```
network_durations.add(tf.keras.layers.Dropout(0.1))
```

```
network_durations.add(tf.keras.layers.Dense(n_durations, activation = "softmax"))
```

```
In [69]: network_durations.compile(loss = "categorical_crossentropy",
                                   optimizer = optimizer)
```

```
In [70]: network_durations.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 25, 128)           71168
_________________________________________________________________
dropout_2 (Dropout)          (None, 25, 128)           0
_________________________________________________________________
lstm_3 (LSTM)                (None, 128)               131584
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 204,042
Trainable params: 204,042
Non-trainable params: 0
_________________________________________________________________
```

```
In [71]: network_durations.fit(features_durations, labels_durations, batch_size = 100,
epochs = 50, verbose = 0)

Out[71]: <tensorflow.python.keras.callbacks.History at 0x13d13664fd0>

In [72]: save_durations = '' Models/durations.h5py

In [73]: network_durations.save(save_durations)

In [74]: %store save_durations

Stored 'save_durations' (str)
```

2.4 Generating new melodies

Finally, it's time to generate music using the two Neural Networks created and trained. To generate the music, we start importing the same dependencies as before and reading all the stores variables to be able to build some of the data stored during the data preprocessing and network definition and training.
From now on, another different Python file is used from the one used previously. This is done in order to clearly separate the definition of the network and the data preprocessing and the music generation.
The usage of a new file also means that all dependencies and variables created before have to be reloaded and imported again.

```
In [1]: import music21

import numpy as np

import tensorflow as tf

In [2]: %store -r data_length

%store -r sequence_length

%store -r n_notes

%store -r n_durations

%store -r features_notes

%store -r features_durations

%store -r notes_dictionary_inv

%store -r durations_dictionary_inv

%store -r save_notes

%store -r save_durations
```

We also load the two networks using the save created before by Keras using a Keras function and using the file path as its argument.

```
In [3]: network_notes = tf.keras.models.load_model(save_notes)

In [4]: network_durations = tf.keras.models.load_model(save_durations)
```

```
In [5]: notes_to_generate = 500

temperature_notes = 0.5

temperature_durations = 0.5
```

Now it's time to define the hyperparamethers of the music generations. Those are three: the number of notes to generate in the variable "notes_to_generate" and the temperatures of the two networks. The temperature is a mathematical parameter that changes the output of the Softmax function in the Output Layer. The lower the temperature is, the more confident the network will be. More about the temperature and its mathematical expression will be explained when defining the temperature function.

```
In [6]: random_start = np.random.randint(0, data_length)

In [7]: sequence_notes = features_notes[random_start]

In [8]: sequence_durations = features_durations[random_start]
```

But before starting to generate new music, an initial sequence must be loaded from the original dataset. Because of the way the ANN was built, it requires a sequence before being able to generate new music on its own. This sequence will be loaded from a random point in the original dataset for both the notes dataset and the durations dataset.

```
In [9]: notes = []

In [10]: for note, duration in zip(sequence_notes, sequence_durations):

             notes.append([notes_dictionary_inv[np.argmax(note)],
                           durations_dictionary_inv[np.argmax(duration)]]).
```

We also append this original sequence to the newly created "notes" array. This array will end up containing every note generated by the algorithm and the original notes serving as the original sequence. To append the sample sequence to the array in the form of a real note and duration, the process followed to encode them must be inverted. We do so using the inverse dictionaries created when encoding and preprocessing the data and picking the index of the maximum argument in the one-hot encoded array. What we are essentially doing is inverting the preprocessing of the data in order to be able to end having the same items as we extracted from the source; a note and its duration.

After the networks has generated new notes, we will use music21 again to encode the notes in a midi file.

```
In [11]: def apply_temperature(predictions, temperature = 0.5):
             predictions = predictions.astype("float")
             predictions = np.log(predictions) / temperature
             predictions = np.exp(predictions) / np.sum(np.exp(predictions))

         return np.random.multinomial(1, predictions, 1)
```

Now, about the temperature. The mathematical formula of the code above is the following:

$$temperature(predictions) = \frac{e^{\frac{\ln(prediction_j)}{temperature}}}{\sum e^{\frac{\ln(prediction)}{temperature}}} \; from\, j = 1 \; to \; j = n$$

The following code does the same as the formula above using numpy's functions for the exponentials and natural logarithm. It returns the predictions modified accordingly to the temperature.

These modifications are actually quite simple: if the temperature is smaller than 1, the differences between the different predicted values is made bigger, thus making the network more confident of its predictions and make it output more consistent results with the original dataset. If the temperature is higher than 1, the opposite happens: the differences between the values are made smaller, thus making the network less confident and make it output less consistent results with the original dataset.

```
In [12]: def predict_note():

             note_feature = np.reshape(sequence_notes, (1, sequence_length, n_notes))

             predicted_note = network_notes.predict(note_feature)[0]

             predicted_note_temperature = apply_temperature(predicted_note,
                                                            temperature =
                                                          temperature_notes)

             final_note = notes_dictionary_inv[np.argmax(predicted_note_temperature)]

             sequence_notes = np.concatenate((sequence_notes[1:len(sequence_notes)],
                                              predicted_note_temperature), axis = 0)

         return final_note, sequence_notes
```

```
In [13]: def predict_duration():

             duration_feature = np.reshape(sequence_durations, (1, sequence_length,
                                                                n_duration))

             predicted_duration = network_durations.predict(duration_feature)[0]

             predicted_duration_temperature = apply_temperature(predicted_duration,
                                                                temperaturature_durations)

             final_duration =
             durations_dictionary_inv[np.argmax(predicted_duration_temperature)

sequence_durations = np.concatenate((sequence_durations[1:len(sequence_durations)],
                                     predicted_note_temperature), axis = 0)

return final_duration, sequence_durations
```

Now we define the functions to actually use the networks and predict the notes and their durations. First we reshape the sequence to make it fit into the network, and then we use the network to predict values based on that sequence. Once we have the predictions, we apply the temperature function to them and we append the most likely next note and duration to the sequence that will be used in the next epoch, removing the first one at the same time to make sure this sequence always have a length of 25 (the sequence length used all the time).

```
In [14 ]: for i in range(notes_to_generate):

              predict_note()

              predict_duration()

              new_note = [final_note, final_duration]

              notes.append(new_note)
```

So all that is left is use this functions and append all the predicted notes to the notes array previously defined.

```
In [15]: midi = music21.stream.Stream()

midi.insert(music21.instrument.Violin())

In [16]: for note in notes:
```

```
        new_note_midi = music21.note.Note(str(note[0]), quarterLength =float(note[1]))
        midi.append(new_note_midi)
```

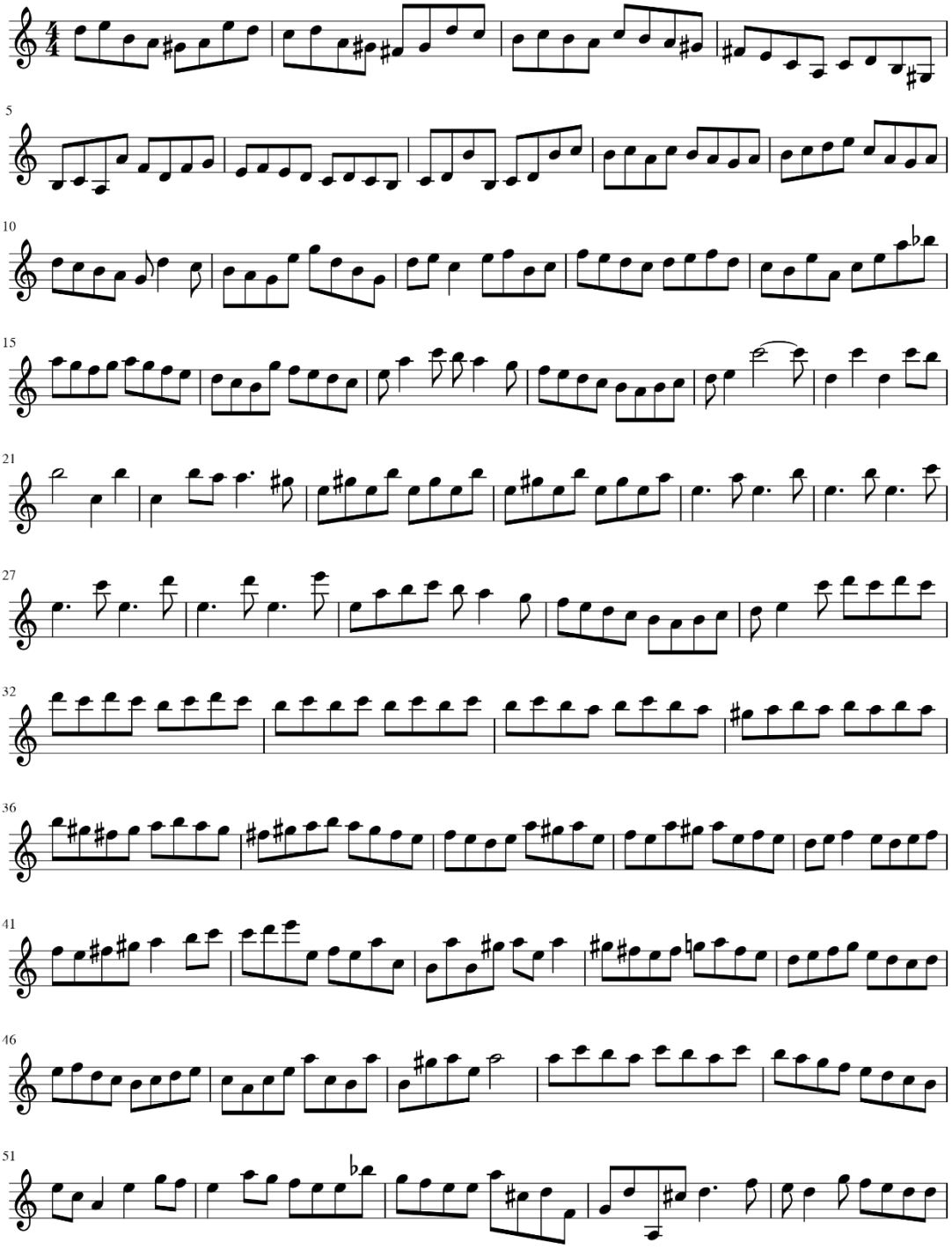
```
In [17]: midi.write("midi", fp = "song.mid")
```

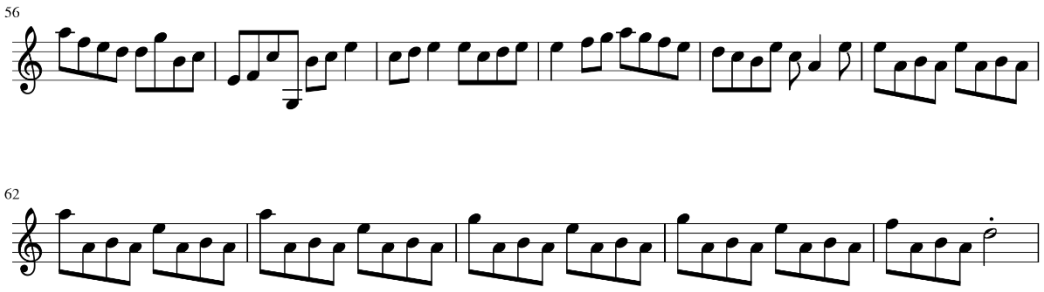
```
Out[17]: 'song.mid'
```

And, at last, we use music21 to transform those notes into a midi file, so we end with a music file with the same format and structure as the original files in the dataset. And with this new file comes the music generated by the ML algorithm. So let's have a look at what the network has generated.

3. Results

This is the score generated by the ML algorithm crafted before, adapted for presentation using MuseScore.




56
62

4. Discussion

Seeing the results, we can state confidently that an AI is able to compose or generate music. For an untrained listener, the music generated by the network would be hard to differentiate from music actually composed by a classical composer. This means that the network has succeeded in its goal.

There are, though, some elements that can be used to tell apart this score from a real one, considering a real score the one composed by an actual human being.

The first one is the great difference in styles. This can be clearly appreciated in the 19th compass, where the style of the music shifts. This couldn't happen on a real score, as there is a great time jump between the popularity of the different musical styles used by the network. This jump is likely due to the fact that in the original dataset there were different classical composer from different time periods and therefore with very different styles.

Another element which is inconsistent is the harmony. If some parts of the score were analysed individually, they could be considered harmonically similar. If the score as a whole is consider, there is no consistent harmony as well as tonality. Those change during the score without any apparent reason. The shift in tonality are not as noticeable as the ones with the style, but can be identified if the score is harmonically analysed.

The rhythms are also worth mentioning. A single ANN has been created only to predict the rhythms, but we can now see that these are quite simple and lack any complexity. This could be because of two different factors: either the rhythms in the original dataset were all quite simple or they should have been preprocessed in a different way. Most likely the latter is where the problem lies. To achieve more success with composing complex rhythms, maybe a different preprocessing of them should be used.

Another item where the network has not succeeded is in the beginning and end of the score. The beginning is feasible, as beginnings can be much more diverse and flexible than endings, but the ending lacks any of the characteristics, neither rhythmic nor harmonic, of a real ending. This is again due to the preprocessing. When extracting the data from the scores, there was no indicator of when a score ended and another one started, so the network has no way to deal with beginnings and endings. A different preprocessing and extraction of the data could again solve this problem.

But, overall, the results are quite good. The final score resembles a real score if it is not closely analysed. To get more complex and realistic results, a more complex network

could be used, with more Hidden Layers and/or more neurons per layer. Other activation functions could also be used and those would maybe report better results.

A better tuning of the hyperparamethers and of the optimization algorithm would probably also make the network more successful. To get the perfect combination of activation functions, hyperparamethers, size of the layer and optimization algorithm; would only be possible by trial and error, which would make the process very time consuming.

In fact, in the broad world of AI and Machine Learning, the network and algorithm used here are not very complex. The network just uses LSTMs in a Sequential Model and the preprocessing of the data relies on one-hot encoding, which is not the most complex and abstract way of preprocessing.

Some this preprocessing would probably be embedding the data instead of one-hot encoding it. It would be much more complex and computationally intensive but it would probably output better results.

Creating a single, probably deeper, ANN to process both the notes and their duration would probably give better results as well.

To sum up, let's go back to the original question: can an Artificial Intelligence compose music? The answer is yes, it can. It may not be perfectly harmonic music, with a consistent style, but it is music after all.

This leads to the issues of creativity and intelligence, which have been mentioned during the work but not developed. Is this ML algorithm creative? Is it intelligent? These complex questions make the main problem of AI evident; intelligence itself.

To understand Artificial Intelligence, intelligence must be understood. Because as realistic and maybe even beautiful the music composed by an AI can seem, in the end, it is just the product of thousands of matrix sums, multiplications, derivatives, functions…

After all, it is possible that human intelligence is not related to the ability to produce sound which are harmonically related to each other; to understand images or videos, drive vehicles, analyze data, trade assets, play board games, write text…

Because as Douglas R. Hofstadter, leading AI scientist at the MIT: "Each step towards AI, rather than producing something which everyone agrees is real intelligence, merely reveals what real intelligence is not."

5. Sources

Kingma, D., & Lei Ba, J. (2015). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION.

music21. (2018). *MIT*.

MuseScore. (2018).

NumPy. (2018).

Pandas. (2018).

Project Jupyter. (2018).

Tensorflow. (2018). *Google*.