Prediction of Epileptic Seizures with Graph-based Deep Learning

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana*

Master of Science in Informatics

Major in Artificial Intelligence

presented by

Alessia Ruggeri

under the supervision of

Prof. Cesare Alippi

co-supervised by

PhD. Daniele Grattarola

June 2019

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Alessia Ruggeri
Lugano, Yesterday June 2019

Contents

Chapter 1

Introduction

Epilepsy Epilepsy is a neurological disorder characterized by epileptic seizures [1][2], which are episodes of vigorous shaking. Each shaking episode can last from brief to long periods and it can result in physical injuries. In epilepsy, seizures tend to recur without warning or any immediate underlying cause; for this reason, people with epilepsy experience varying degrees of discomfort in their social life due to their condition [3].

The cause of most cases of epilepsy is unknown. Some cases can be due to brain injury, infections or tumours; other manifestations of the disorder are directly linked to genetic mutations; but the majority of cases don't present any underlying cause [3]. Epileptic seizures are the result of excessive and abnormal neuronal activity in the cortex of the brain [2]; they are diagnosed by excluding other conditions that might cause similar symptoms. Epilepsy can often be confirmed with an electroencephalogram (EEG), but a normal test does not rule out the condition [4]. Not all cases of epilepsy are lifelong, and many people improve to the point that treatment is no longer needed [3].

Seizures are controllable with medication in about 70% of cases [5]. For people that don't respond to medication, surgery may be considered, but it is only an option when the area of the brain that causes the seizures can be clearly identified and is not responsible for critical functions. The condition of people that don't respond to medication is called *drug-resistant epilepsy* (DRE) and people suffering of this situation are forced to deal with it in everyday life [6].

Drug-resistant epilepsy is defined as failure of adequate trials of two tolerated and appropriately chosen and used antiepileptic drugs (AED schedules) to achieve sustained seizure freedom [6]. The probability that, after two failed AEDs, the next medication will achieve seizure freedom is around 4% [7]. It is important that people with DRE

undergo other treatments to control seizures, like neurostimulation or diet.

Approximately 50 million people currently live with epilepsy worldwide, which corresponds to about 0.65% of the world population. Globally, an estimated 2.4 million people are diagnosed with epilepsy each year. This chronic disorder of the brain affects people of all ages and it is one of the most common neurological diseases globally. Approximately 30% of people with epilepsy have a drug-resistant form, which corresponds to about 15 million of people [3].

People with epilepsy and especially those suffering from DRE, as a result of the frequent epileptic seizures, are subject to social discrimination and discomfort, which is often more difficult to overcome than the seizures themselves. The prejudice due to the disorder can even discourage people from seeking treatment for symptoms, so as to avoid becoming identified with the disorder [3].

Machine Learning Nowadays, there exist machine learning techniques that could be applied to improve the everyday-life quality of people suffering from drug-resistant epilepsy. Research on machine learning application to epilepsy could also lead to a possible contribute for a study in the medical field in order to understand better the causes of epileptic seizures.

Machine learning is a sub-field of Artificial Intelligence: while artificial intelligence studies are focused on the creations of machines that can mimic a human mind, the main aim of machine learning is to create machines that are able to perform a specific task. the difference from classic programs is that, to learn performing the task, a machine learning model automatically and progressively improve itself and its performance by analyzing sample data (called *training data*), finding patterns and deriving a mathematical model from that. This allows the model to make predictions or decisions without being explicitly programmed to perform that task [8].

Machine learning techniques can solve a variety of different tasks, like regression, prediction, classification, clustering, dimension reduction, density estimation, and many others [8]. These tasks fit with as many applications, like personal assistance, natural language processing, data security, healthcare, financial trading, marketing personalisation, image recognition, anomaly detection, robotics and so on. Machine learning is also strongly related to mathematical fields like data mining, optimization, and statistics.

ML and medicine In the last few years, machine learning techniques have been largely applied also to the medical field. Many of the machine learning start-ups are intensively

working on healthcare solutions that could potentially help doctors to diagnose illness, make patients' life much easier or offer information able to save lives. The reason why machine learning models in general have been heavily used just in recent years and are continuing to grow can be identified in the technological development: only starting from the new millennium we dispose of machines that have the computational power that is needed for these type of techniques.

In healthcare, there already exists several different applications of machine learning [9]. One of the most popular applications is the diagnosis in medical imaging, which makes great use of computer vision and pattern recognition techniques. A machine learning model can be trained by feeding to it a dataset of images labelled with the corresponding disease, so that it can process the data and "learn" disease-specific patterns. In this way, when a new image is fed to the model, it is able to recognize if the disease is present or not, hopefully with a good accuracy. Another utilization of machine learning algorithm in the medical field is the suggestion of treatment ideas or options, based on the previous experience with the disease and the analysis of what worked before. These information can help a doctor to make more informed decisions. Machine learning algorithms are used also for drug discovery in the pharmacy field, through the analysis and creation of new chemical compositions, and for robotic surgery, allowing surgeons to manipulate robotics devices having great precision and reaching tight spaces.

ML and epilepsy In the case of epileptic seizures, machine learning can be very useful if used for seizure prediction. Indeed, the ability to predict epileptic seizures could be essential mainly for two types of applications: the notification of the incoming seizure to the patient, or the anticipation of the seizure using neurostimulation in order to avoid it. In the first case, we could think about a case scenario in which a person suffering from epilepsy receives a notification on his phone warning him of an incoming seizure in 10 minutes. By being alerted by a notification, the person have the time to put himself in a safe state before the start of the shaking, in order to avoid any subsequent injury. This is an example in which machine learning doesn't help avoiding the actual problem, but it is very helpful in order to contain the subsequent damages. In the second case, the application of seizure prediction is even more interesting, because it is able to avoid the actual seizure occurrence. Indeed, some medical researches [10] have proved the effectiveness of neurostimulaton in the treatment of epilepsy: there exist devices that can provide stimulation to the entire brain or to specific areas of the brain (the ones responsible for the seizures) in order to reduce the number of seizures over the years or to actually avoid the seizure. If applied to the seizure focus in advance,

the neurostimulation is able to "block" a single occurrence of shaking; therefore the ability to predict a seizure some seconds before its beginning could allow the device to intervene in time and to avoid the episode [11].

Previous approaches Epileptic seizure is a topic that has been deeply analysed since 1970s. Tests involving recordings of EEG are used in order to look for the causes of epilepsy and to observe the brain activity during seizures to find patterns that make it predictable ([12] [13]). The seizure prediction task has been initially approached using classic statistical tools, like probability, thresholds, correlation, Monte Carlo methods and even classic machine learning methods, like SVM or k-clustering ([14] [15] [16] [17] [18] [19] [20]). However, given the complexity of the problems, during the last few years several deep learning approaches have been proposed, involving Convolutional Neural Networks, LSTM Neural Networks, Generative Adversarial Networks and even Graph Deep Learning ([21] [22] [23] [24] [25] [26]). The deep learning models were able to obtain acceptable results, but the capacity of the classifiers to predict future sample classes between seizure and non-seizure remains uncertain.

Our approach This master thesis project represents a review of methods for the epileptic seizure prediction task, but also an attempt to improve the performances in this field by applying new technologies. The study is performed using intracranial electroencephalography data (iEEG) and it includes the implementation of a variety of both classic and graph-based machine learning and deep learning models, in order to realize a comparison between the two approaches for the seizure prediction problem. The aim of the project is to find out if the graph-based models can actually outperform the classical ones for this type of task.

The data used is a collection of 24 hours of iEEG monitoring on a patient, which contains three seizures. The signals of the 90 electrodes is used both for the seizure detection and prediction tasks. For this purpose, several methods have been tested: classic machine learning techniques such as Random Forests, Gradient Boosting and Support Vector Machine (SVM), plus a Dense Neural Network have been used only for the detection task; while a Convolutional Neural Network (CNN), an LSTM Network and some Graph Neural Networks (GNN) have been tested both on the detection and the prediction tasks. All the models results have been compared in order to find out which one works better and how far it is able to predict a seizure.

The thesis is organized as follows: Chapter 2 gives a high-level overview of all the machine learning techniques used and the theory behind them and it also presents the state-of-the-art for the problem concerned; Chapter 3 describes the work done for this project and how the machine learning methods have been used; Chapter 4 explains the implementation of the project and all the related technical details; finally Chapter 5 draws the conclusions, making some final comments about the project and suggesting future works.

Chapter 2

Background

This chapter explains the theory behind the methods and techniques used in this project: it begins by describing the EEG data and presenting a formal definition of the problem; later, it presents logistic regression and the related evaluation metrics, followed by the description of activation functions, optimizers and regularizers used in the project; subsequently, it describes the classic machine learning models and deep learning models used and, finally, it presents the the functional connectivity, the graph representation and the graph-based deep learning models.

2.1 Epileptic seizures and EEG

An epileptic seizure is a disruption of the electrical signals in our brain [27]. The brain controls the way we function millions of neurons, which pass messages to each other by electrical signals. If these signals are disrupted, our body and feelings are hugely influenced by this change.

Since epileptic seizures arise inside the brain, the most common tests that specialist use in order to diagnose epilepsy are the electroencephalogram (EEG) and MRI (brain scans). Neither of these tests is able to confirm for certain if the patient has epilepsy or not, but they can help the specialist to decide whether to consider epilepsy as possible diagnosis. We are going to focus on the EEG, since the project is based on a dataset of iEEGs.

EEG gives an overview of the activity of the brain cells, that is the traffic of nerve impulses that neurons send each other to communicate, also called *brain waves*. The messages between neurons consist of changes in the electrical charge of the cells: when a neuron sends a message, it "gives off" electricity. In the EEG test, the brain waves

are picked up by electrodes, which are small sensors placed on different areas of the patient's head. When the electrodes are placed directly on the surface of the brain, we talk about *intracranial electroencephalogram* (iEEG). The electrodes are able to record the electrical activity from small areas of the brain, therefore each electrode outputs an electrical signal that varies over time. The EEG is the combination of all the electrodes signals in one single "chart", where we can see a signal for each electrode (y-axis) vary over time (x-axis). Figure 2.1 shows an example of an EEG, which in this case presents some epileptic spikes.

Figure 2.1. Example of epileptic spikes monitored with EEG (from *Wikipedia, the free encyclopedia*)

The number of electrodes used can range from a few dozen to hundreds. They are positioned on specific sections of the patient's head, therefore different electrodes monitor the activity of different areas of the brain. In this way, by looking at the EEG and knowing the position of each electrode, the specialist can understand which areas of the patient's brain are involved in the seizure. The electrodes are identified based on whether they are placed on the left or right side of the head and also based on the lobe they are recording from (frontal lobes, temporal lobes, parietal lobes or occipital lobes). The EEG monitoring captures several types of brain waves, which differ in the frequency (waves per second). Specific ranges of frequencies are determined by

particular moments of the day, situations and states of mind.

The EEG is a very helpful tool for epilepsy diagnosis, but it can't show for certain the presence of epileptic seizures. In fact, despite the existence of typical signal patterns associated with epilepsy, usually epileptic seizures look very different on the EEG depending on the patient: some seizures are recognizable thanks to the presence of typical spikes in the electrical signals; other seizures happen without showing any visible change in the EEG. Therefore, we have reason to think that for some patients the seizure is "hidden" in the relations between signals behaviour, more than in the behaviour of each signal itself. Even when the change in the EEG pattern is clear, it could involve only a subset of electrodes (partial seizures) or all the electrodes (generalized seizures), and it is difficult to determine if the disruption of the signal is actually caused by epilepsy or if it simply represents an 'abnormal' EEG, which could happen without the presence of seizures.

2.2 Problem definition

The focus of this thesis is on the seizure prediction problem, but we are going to analyze also the seizure detection task in order to have a clearer and more complete idea of the machine learning models potentialities on epileptic seizures data.

The seizure prediction/detection problem can be treated as a classification problem, whose aim is to predict a class associated with each sample. In the case of an EEG, if we consider discrete instants of time, we can represent the EEG information as a matrix $N \times F$, where the rows represent the N electrodes and the columns represent the features associated with the F time instants. So, for each of the N electrodes, we have a sequence of F values representing the amplitude over time. The task can be treated as a classification problem by associating each time instant with a label or target, which represents the class of membership between *seizure* and *non-seizure*. This means that each time instant is classified based on the presence or absence of an ongoing epileptic seizure in that instant. If we indicate with x_t the set of signals features related to a single time instant, we can say that each time instant t with features x_t has an associated target y_t, which is equal to 1 if there is an ongoing seizure and to 0 otherwise. Thus considered, the seizure prediction/detection problem becomes the task of predicting the class target y_t related to the time instant features x_t.

Let's consider a time portion of $T+1$ discrete instants in an EEG: the portion will be characterized by a sequence of features $[x_t, x_{t+1}, x_{t+2}, ..., x_{t+T}]$, one for each time instant t, representing the signals amplitude (measured in volts) in that instant. Let's

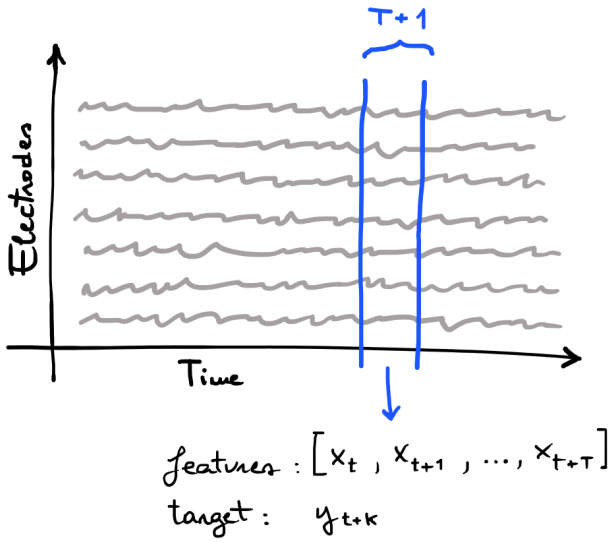


Figure 2.2. Problem definition

imagine that the time portion is used to predict a single target y_{t+k}, which indicates whether the instant $t+k$ is part of an ongoing epileptic seizure or not. The described situation is illustrated in Figure 2.2. Considering this scenario, we can individuate three cases of study of the problem, depending on the value of the variables T and k:

- $T = 0, \quad k = 0$: in this case, the length of the time portion is equal to 1 and the target corresponds to label of the same time instant of the sample. So at each sample x_t is associated the target y_t (as in the case described above).
  EXAMPLE: $T = 0, k = 0 \quad \Rightarrow \quad [x_3] \rightarrow y_3$

- $T > 0, \quad k = T$: in this case, the length of the time portion is greater that 1, so it consists of a sequence of samples, and the target is the label associated to the last sample of the sequence. So at each sequence $[x_t, x_{t+1}, ..., x_{t+T}]$ is associated the target y_{t+T}.
  EXAMPLE: $T = 5, k = 5 \quad \Rightarrow \quad [x_0, x_1, ..., x_5] \rightarrow y_5$

- $k > T$: in this case, we have a sequence of samples whose target is associated to a sample that doesn't belong the sequence, but is forwards in time ($t > T + 1$). This means that at each sequence $[x_t, x_{t+1}, ..., x_{t+T}]$ is associated the target y_{t+k}, where $(t+k) > (t+T)$.
  EXAMPLE: $T = 5, k = 8 \quad \Rightarrow \quad [x_0, x_1, ..., x_5] \rightarrow y_8$

The first two cases are examples of a detection problem, since the target y_{t+k} to predict

is associated to an instant that is still inside the time window we look at to predict it ($k \leq T$). The third case, on the other hand, is a prediction problem, since the target y_{t+k} to predict is associated to an instant that is outside the time window we look at to predict it and it is forwards in time with respect to the time window ($k > T$). The difficulty of the prediction task is that we cannot look at the features x_t in order to predict the target y_t, but we have to rely on information from previous timestamps (for this reason, we consider $T > 0$, since it would be almost impossible to predict the target based on a single sample backwards in time).

2.3 Logistic Regression

In this project, the binary classification problem related to epileptic seizure detection/prediction is approached using logistic regression. The logistic regression algorithm is used to estimate the probability that an instance belongs to a particular class [28]. If the estimated probability is greater than a fixed threshold, which is commonly set to 0.5, then the model assigns the instance to the positive class (in our case, *seizure* class), otherwise it assigns the instance to the negative class (in our case, *non-seizure*). In this way, logistic regression can be used as a binary classifier.

Let's look at a very simple logistic regression model as an example: it computes the weighted sum of the input features $\mathbf{x}$ multiplied to the model parameter θ and it outputs the logistic of the result:

$$\hat{p} = h_\theta(\mathbf{X}) = \sigma\left(\theta^T \mathbf{x}\right) \tag{2.1}$$

The logistic σ is a sigmoid function that generates a number between 0 and 1, representing the probability $\hat{p}$ that an instance $\mathbf{x}$ belongs to the positive class. The sigmoid function is illustrated in Figure 2.3 and it is mathematically defined as:

$$\sigma(t) = \frac{1}{1 + \exp(-t)} \tag{2.2}$$

The predictions $\hat{y}$ for the binary classifier can be easily obtained by applying the following equation to the probability $\hat{p}$:

$$\hat{y} = \begin{cases} 0 \text{ if } \hat{p} < 0.5 \\ 1 \text{ if } \hat{p} \geq 0.5 \end{cases} \tag{2.3}$$

The model parameter θ is trained in order to estimate high probabilities for positive

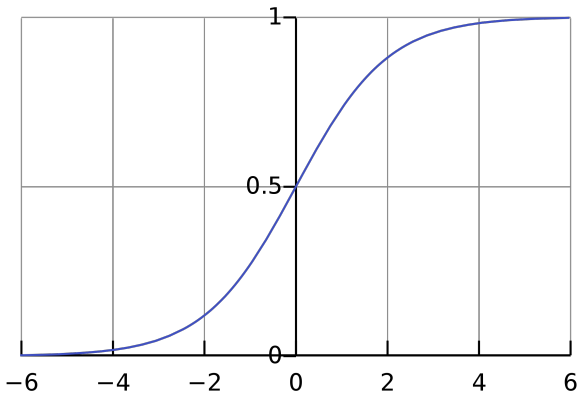


Figure 2.3. Sigmoid function (from *Wikipedia, the free encyclopedia*)

instances and low probabilities for negative ones. In order to train it, we need a cost function that directs the model on the right track. For logistic regression algorithms, the most suitable cost function is the *log loss*, since it captures exactly the aim of the classifier:

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1-\hat{p}) & \text{if } y = 0 \end{cases} \tag{2.4}$$

$-\log(\hat{p})$ becomes higher and higher and tends toward infinity the more $\hat{p}$ get close to 0, while it gets near to 0 when $\hat{p}$ approaches 1. $-\log(1-\hat{p})$ behaves in the opposite way, so it makes sense to use the first one for the positive class and the second one for the negative class. The cost function is applied to multiple training instances by computing the average cost over all the n instances:

$$error = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)] \tag{2.5}$$

The log loss cost function is then used by an optimizer, for instance Gradient Descent or Root Mean Square Propagation algorithms, in order to update the model parameters accordingly. Usually, like in this project, the optimizer works on mini-batches, so the model parameters are updated after each mini-batch.

The project models make use of the same process, but instead of using such a simple model consisting of only one parameter, we used a set of more complex machine learning models and applied logistic regression to each of them to build several binary classifiers.

2.4 Metrics

In order to evaluate the models, four metrics have been chosen, which are commonly used for binary classifiers: loss, accuracy, ROC-AUC and recall. The first two metrics are the most standard performance measures for all the machine learning models.

Loss The loss is nothing but the error computed by the cost function in use, which in our case is the log loss.

Add description of brier_score_loss (used for classic ml models)

Accuracy The accuracy is simply the ratio of correct prediction, which in a binary classifier is represented by the number of true prediction (true positive and true negative) over the total number of predictions:

$$acc = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{2.6}$$

where TP = true positive, TN = true negative, FP = false positive, FN = false negative.

Recall The recall, also called sensitivity or true positive rate, is the number of positive instances correctly classified over the total number of positive instances:

$$recall = \frac{(TP)}{(TP + FN)} \tag{2.7}$$

ROC-AUC The ROC curve (receiver operating characteristic curve) is a plot of the true positive rate (recall) against the false positive rate (shown in Figure 2.4):

$$TPR = \frac{(TP)}{(TP + FN)} \qquad FPR = \frac{(FP)}{(FP + TN)} \tag{2.8}$$

The false positive rate is the number of negative instances incorrectly classified as positive over the total number of negative instances.

The ROC curve represents a plot of TPR and FPR at different classification thresholds: if we lower the threshold, the classifier predicts more items as positive, so both TPR and FPR increase. In order to find the best classification threshold, we need to find a tradeoff between TPR and FPR, trying to obtain a higher TPR as possible and a lower FPR as possible (looking at the curve, we need to find the point that is closest to the upper left corner). For this purpose we can rely on the *area under the ROC curve*

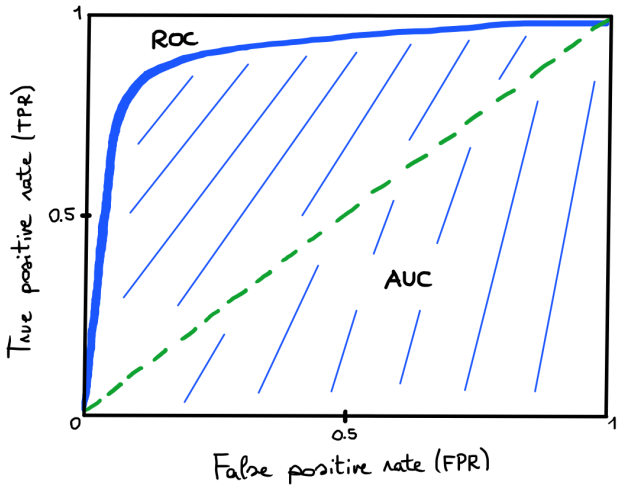


Figure 2.4. ROC curve and respective AUC

(AUC), which provides a performance measure for different classification thresholds. The ROC-AUC ranges from 0 to 1 and its value represents the quality of the classifier predictions. A perfect classifier has AUC $= 1$, while a completely random classifier has AUC $= 0.5$.

2.5 Activation, optimization and regularization

This section describes topics that are related only to the deep learning models.

2.5.1 Activation functions

An activation function is a non-linear mapping between the input and the output. Activation functions are essential in neural networks layers, since they introduce the non-linearity properties to the model: without activation functions, the neural network would be a simple linear transformation and it wouldn't be able to learn from complex data.

Some activation functions are more suited to hidden layers, while others are perfect to compute the final output of the model. In this project, we used the sigmoid function as activation function for the last layer of our models in order to classify data, as previously explained in Section 2.3. For the hidden layers, we used both ReLU and Tanh activation functions.

ReLU ReLU (Rectified Linear Unit) is a very popular and extremely fast and effective activation function for hidden layers. Its principle is very simple: it keeps only positive values, while setting to 0 all the negative ones.

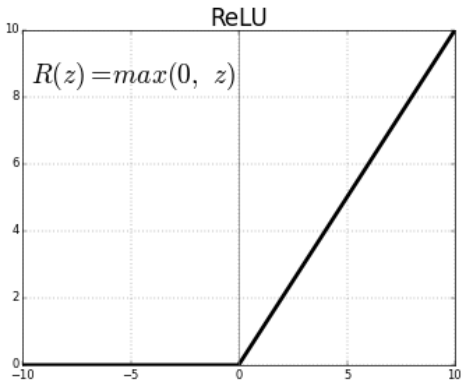
$$\mathrm{R}(x) = max(0, x) \tag{2.9}$$

The ReLU function is illustrated in Figure 2.5a.

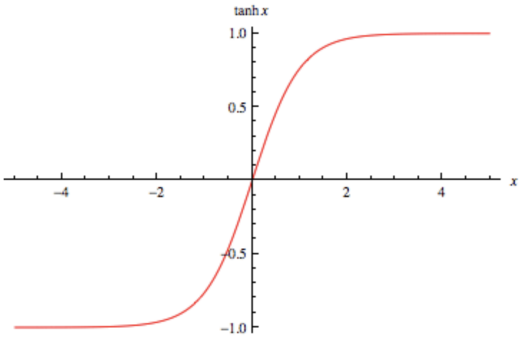
Tanh Tanh (Hyperbolic Tangent) is similar to sigmoid, but it ranges from -1 to 1. It maps negative input as strongly negative, positive ones as strongly positive and zero inputs near zero. Its formula is:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.10}$$

The Tanh function is illustrated in Figure 2.5b.



(a) ReLU activation function (from *Medium*)



(b) Tanh activation function (from *Wolfram MathWorld*)

Figure 2.5. Activation functions

2.5.2 Optimizers

An optimization algorithm determines the way model's learnable parameters (*weights*) are updated during the training process. The aim of an optimizer is to minimize (or maximize) an objective function that is dependent on the model's weights, which is the cost function. Since the cost function is computed using the model's predictions and the latter are determined by the weights and input features, the weights have a crucial role in minimizing the cost function. The update of the model's learnable parameters

takes place through *backpropagation*: after the generation of the predictions, the gradient of the loss function with respect to the weights is computed; then the weights are updated in the opposite direction of the gradient (Gradient Descent). The "magnitude" of the update is given by the *learning rate*, which can be decisive in the search for the loss function's global minimum. The role of an optimizer is to optimize the Gradient Descent process, in order to make it more efficient and effective.

In this project, the optimizer that has been used for all the models is Adam optimizer.

Adam optimizer Adam (Adaptive Moment Estimation) is one of the most popular and used optimizers in deep learning, that usually has good performances. It is based on two intuitions: adaptive learning rate (inspired by RMSProp optimizer) and momentum optimization [29] [30]. An adaptive learning rate method computes individual learning rates for different model's learnable parameters; while a momentum method helps accelerating the Gradient Descent process by adding a fraction of the update vector of the past time step to the current update vector. This optimizer stores two moments: the first moment m_t (the mean), which is estimated as the exponentially decaying average of past gradients, and the second moment v_t (the uncentered variance), which is estimated as the exponentially decaying average of past squared gradients. The two moments are computed respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t \tag{2.11}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2 \tag{2.12}$$

where g_t is the gradient on current mini-batch and β_1, β_2 are optimizer's decay rates, usually set close to 1. Since m_t and v_t are initialized as zero-vectors, they are biased towards zero, therefore they need to be corrected:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.13}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.14}$$

The first moment m_t is used as momentum optimization parameter, while the second moment v_t is used to adapt the learning rate to each different weight. The resulting Adam update rule for model's weights is the formula:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.15}$$

where θ is the model weight and η is the learning rate (ϵ is just a smoothing term usually initialized to a tiny number).

2.5.3 Regularization

Regularization techniques are very useful in deep learning in order to avoid overfitting. Deep neural networks generally have a huge amount of parameters, which represent their power to learn from complex datasets; this flexibility, however, sometimes leads to the overfitting of the training set, preventing the model from making good prediction on any other dataset [28]. To solve this problem, there exist several regularization methods, which in some way limit the amount of freedom of the model by adding some penalty as the model complexity increases.

The regularization techniques used in this project are the ℓ_2 regularization and dropout.

ℓ_2 regularization The ℓ_2 regularization, also called Ridge Regression or Tikhonov regularization, constrains the model's flexibility by adding a regularization term to the cost function. In this way, while trying to fit the data, the model is forced to keep the weights as small as possible. The regularization term added to the loss function is the 2-norm (Euclidean distance) squared, controlled by the parameter λ, which handles the amount of regularization of the model. This is expressed by the equation:

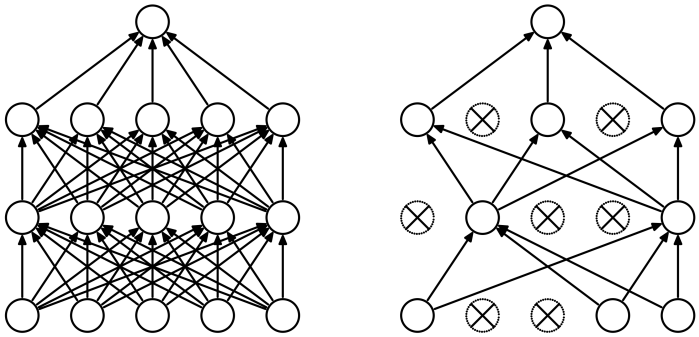
$$\textit{regularization term } = \lambda \|\theta\|_2^2 = \lambda \sum_{i=1}^{m} \theta_i^2 \tag{2.16}$$

$$\textit{error } = (\textit{loss function}) + \lambda \sum_{i=1}^{m} \theta_i^2 \tag{2.17}$$

where $\|\cdot\|_2$ represents the 2-norm and m is the length of the weights vector θ of the model. If $\lambda = 0$, then there is no regularization, since the regularization term added to the loss function is zero. If λ is too large, then the weights take a value very close to zero and the model underfit the data, not being able to learn from it.

Dropout The dropout is probably the most popular regularization technique in deep learning. The principle is very simple: during the training process, some layer's neurons are "dropped out", which means they are set to zero and completely ignored. The neurons to shut down are selected from the input layer and the hidden layers, excluding the output neurons, and they are chosen based on a probability p. This means that, at

each training step, each neuron has a probability p of being shut down for that training step. The probability p represents the dropout rate, that is the fraction of nodes to ignore.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.6. Illustrated example of dropout (from *Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014*)

Figure 2.6 shows and example of dropout applied to a simple dense neural network. The idea behind dropout is that the network cannot rely on the features of each individual neuron, because if an essential neuron is shut down, then the network is not able to perform well. By ignoring some percentage of different neurons at each training step, the model is forced to learn more robust features, that work well together with the features of other groups of neurons. In this way, it is not dependant on the features of few individual neurons, leading to a better generalization.

2.6 Classic machine learning models

Machine learning models are able to perform a specific task efficiently, without the need of explicit instructions, but relying uniquely on the patterns and features they learn from data. Giving the nature of this project's problem and the type of data we are working on, all the models that have been used have been trained in a supervised manner. This means that the training data fed to the algorithm includes the labels, that are the solutions to the problem. In the case of a classification task, the training data consist of both the data features and the respective classes of membership, so that the model can learn from examples and discover patterns that allow it to make good predictions on new data.

In this section we are going to describe the classic machine learning models that have been used, leaving the classic deep learning models (neural networks) for the

next section. The models in this section are much weaker than the neural networks, but they are more efficient when the problem is simple enough to be handled by a simpler model.

2.6.1 Random forest

A random forest is an ensemble of decision trees, which are machine learning algorithms able to perform both classification and regression [28]. To understand how a random forest works, we first need to say a few words about decision trees.

A decision tree is a model represented as a tree in graph theory, with a root node, internal vertices, leaf nodes and edges between them. Each node can be seen as a decision unit: it contains a boolean expression based on some input features and it makes decisions based on this condition. The leaf nodes make and exception since, for classification tasks, each of them correspond to a different class. The classification of a single input instance, then, works as follows: starting from the root node, each node evaluate its boolean expression on one (or multiple) instance's feature and, depending on whether the condition is *True* or *False*, the path to follow continues on the left or right child of that node. The process goes on until a leaf node is reaches and the instance is finally assigned to the class corresponding to that leaf node. Figure 2.7 shows and example of a decision tree applied to the famous Iris dataset for classification. In that case, the features used by nodes to create a condition are at first the petal length and then the petal width. The three leaf nodes correspond to the tree classes: setosa, versicolor and virginica. The *gini* attribute assigned to each node measures its impurity, which depends on the class distribution of the instances to which the node is applied. If $gini = 0$, then the node is pure, meaning that all the instances to which it is applied belong to the same class. If all the leaf nodes' *gini* attribute is equal to zero, then the decision tree has perfectly classified all the instances.

When the final prediction is computed using an ensemble of decision trees, we talk about random forest. In a random forest, k different decision trees are trained in parallel on different random subsets of the training set. The sampling of the data can be performed with replacement (*bootstrapping*), like in this project, or without replacement. Both methods allow training instances to be selected multiple times for different decision trees, but, when using bootstrapping, the same instance could be selected multiple times also for the same decision tree. Once all the decision trees have been trained, the random forest prediction is computed by aggregating the predictions of all its decision trees. Typically the statistical mode is used as aggregation function, but the average can be used as well. A random forest can be regularized by setting the

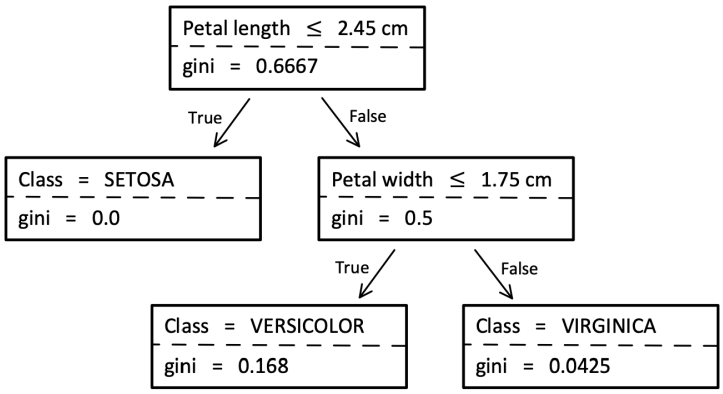


Figure 2.7. Example of decision tree applied to the famous Iris dataset for classification

number of decision trees to use and their maximum depth.

In general, random forest algorithms work better than decision trees and are less prone to overfitting. The reason is the greater tree diversity: in addiction to the data sampling, when splitting a node during the construction of the tree, the split is chosen no longer as the best among all features, but as the best among a random subset of the features.

2.6.2 Gradient boosting

Gradient boosting (tree-based) is very similar to random forest, as it is itself an ensemble of decision trees. Differently from random forest, in gradient boosting the decision trees are trained in a sequential way, one at a time, and each one tries to correct the errors made by its predecessor. In particular, each new decision tree is fitted on the residual errors made by the previous one. Therefore, while the first decision tree of the sequence will fit the instances $\mathbf{X}$ and the targets $\mathbf{y}$, the second decision tree will fit $\mathbf{X}$ and the residual errors $y_1 = \mathbf{y} - \hat{y}_1$, where $\hat{y}_1$ are the predictions made by the first decision tree on $\mathbf{X}$. Likewise, the third decision tree will fit $\mathbf{X}$ and the residual errors $y_2 = y_1 - \hat{y}_2$, where $\hat{y}_2$ are the predictions made by the second decision tree on $\mathbf{X}$, and so on. So we can consider gradient boosting as a gradient descent algorithm.

Usually gradient boosting have better performance with respect to random forest, but, since decision trees are not trained independently, it is more prone to overfitting.

2.6.3 Support vector machine

SVM (Support Vector Machine) is a very powerful machine learning algorithm able to perform both linear and non-linear classification and regression [28]. It is a binary classifier, but it can be used also as a multiple-class classifier through some trick. In order to classify data, the SVM represents the instances as points on a decision surface and divides them through a decision boundary that is placed in the middle of the largest possible gap between the two classes' instances (positive and negative class). In other words, if we imagine the data projected in a 2-dimensional space, the SVM tries to find a line able to separate the positive and negative instances staying as far away as possible from the closest instances. This situation is illustrated in Figure 2.8, where the two group of instances, belonging to the blue and green classes, are divided by the red line. The boundary generated by the SVM is placed as far as possible from the instances, trying to maximize the distance w from the nearest instances. In this way, in case we made a small error in the location of the boundary, we have a margin of error that prevents a misclassification. The instances that lie on the edges of the decision boundary's margins are called *support vectors* (in Figure 2.8 they are represented as filled circles). The position of the support vectors completely determine the max-margin decision boundary and the distance between support vectors of different classes determine the width of the optimal boundary's margins.

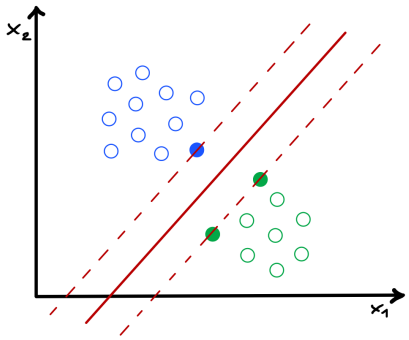


Figure 2.8. Example of SVM decision boundary for data classification

In SVMs, the data are projected to a d-dimensional space, where d is the number of data features, and the decision boundary that classifies the data is a hyperplane. A hyperplane is a subspace of an ambient space that is one dimension less than the ambient space: for example, if a space is 2-dimensional, then its hyperplanes will be 1-dimensional (lines). In general, if the ambient space is n-dimensional, its hyperplanes will be (n-1)-dimensional. In SVMs context, an hyperplane is the subspace of the deci-

sion space that the SVM constructs in such a way that the margin of separation between the two classes' instances is maximized.

If the data points are linearly separable, the SVM is able to construct two parallel hyperplanes that separate the data points in the two respective classes. The hyperplanes can be respectively described by the two equations:

$$\vec{w} \cdot \vec{x}^{+} - b = 1 \tag{2.18}$$

$$\vec{w} \cdot \vec{x}^{-} - b = -1 \tag{2.19}$$

The distance between the two hyperplane is $\frac{2}{\|\vec{w}\|}$, so, in order to maximize the distance between them, we want to minimize $\|\vec{w}\|$, while avoiding margin violation (*hard margin*) or limiting them (*soft margin*).

Hard margin can be used for linear problems, where the data points are linearly separable. In this case, the max-margin hyperplane is completely determined by those data points which are nearest to it and we want to prevent data points from falling into the margin. To reach this result, we can use the following objective function, subjected to the following constraints:

$$\text{objective function } = \min \frac{1}{2}\|\vec{w}\|^2 = \min \frac{1}{2} w^T w \tag{2.20}$$

$$\text{constraints } = \begin{cases} \text{if } y = 1: & w^T x + b \geq 1 \\ \text{if } y = 0: & w^T x + b < -1 \end{cases} \tag{2.21}$$

These constraints guarantee that each point lies on the correct side of the margin.

In some cases, the problem can be non-linear and therefore not solvable with hard-margin; in other cases, the problem can be linear, but we would like a general solution that takes into account the presence of some errors in the data. In these situations, soft-margin can be used in place of hard-margin. Soft-margin SVMs allow some exceptional data points to fall into the margin or to lie on the wrong side of the margin. Soft-margin SVM introduces two new parameters: ε_i represents the distance of x_i from his real class margin hyperplane; C is a parameter that allows to control better the softness of the margin, defining the tradeoff between the objective and the constraints. If C is very big, the SVM becomes very rigid and behaves similarly to hard-margin; on the other hand, if C is very small, the SVM allows a lot of errors. Soft-margin is defined by the

following objective function, subjected to the following constraints:

$$\text{objective function } = \min \frac{1}{2} w^T w + C \sum_{i=1}^{n} \varepsilon_i \tag{2.22}$$

$$\text{constraints } = \begin{cases} \text{if } y = 1: & w^T x + b \geq (1 - \varepsilon_i) \\ \text{if } y = 0: & w^T x + b < (-1 + \varepsilon_i) \\ \text{for all } i: & \varepsilon_i \geq 0 \end{cases} \tag{2.23}$$

Figure 2.9 shows a geometric representation of the parameters and formulas just presented.

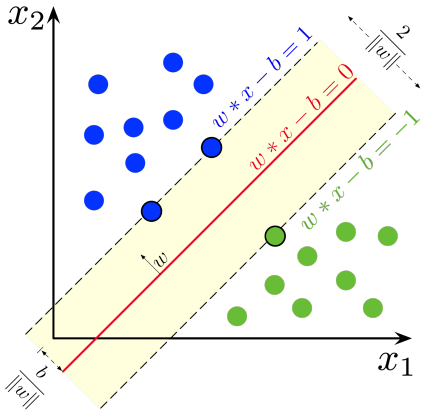


Figure 2.9. Geometric representation of SVM's equations

When data are not linearly separable, we can still use SVM in two ways: we can simply use soft-margin and permit some tolerance in presence of errors, or we can rely on a kernel function $K(a, b)$. A kernel function allows the SVM to lead the data points to a situation where they are linearly separable again. Indeed, the data can be mapped to a higher-dimensional feature space, where they are linearly separable by an optimal hyperplane. A feature map is the function that maps the data points to an higher-dimensional feature space: the function $\phi(x_i)$ maps every x_i to the new feature space:

$$< \phi(a), \phi(b) > \tag{2.24}$$

The increase of space's dimensions can be computationally expensive, due to the computation of all the additional features. To avoid an high-cost computation, we can use the *kernel trick*: applying a kernel function allows to compute the dot product of

weights in the lower-dimensional space instead of in the higher-dimensional space:

$$< \phi(a), \phi(b) >= K(a, b) \tag{2.25}$$

The trick consists in expressing the problem just in function of the dot products, without the need to know the entire mapping to the new feature space. Actually, a kernel is a function that is able to compute the dot product $\phi(a)^T \phi(b)$ based only on the original vectors a and b, without the need to compute the transformation ϕ on the vectors. Figure 2.10 shows an example of a non-linear separable problem that could be solved by mapping data to a higher-dimensional space.

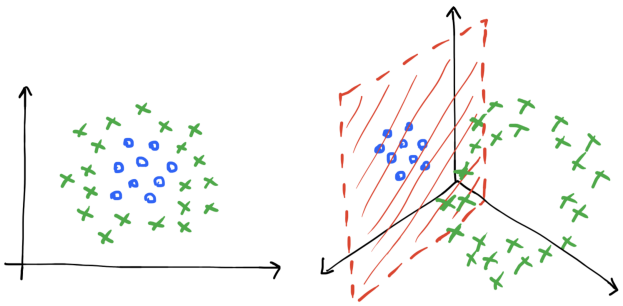


Figure 2.10. Example of a non-linear separable problem solved in a higher-dimensional space

A frequently used kernel is the Gaussian RBF kernel (Radial Basis Function kernel). This kernel generates new features by measuring the distance between the support vectors and all other instances: the function value decreases as the distance from the support vectors grows, so it can be considered a similarity measure. Through Gaussian RBF kernel, the input vector is mapped to an infinite vector and then normalized by dividing each component by the vector's length. The Gaussian RBF kernel is defined by the equation:

$$K(a, b) = \exp\left(-\frac{\|a - b\|^2}{2\sigma^2}\right) = \exp\left(-\gamma\|a - b\|^2\right) \tag{2.26}$$

The hyperparameter $\gamma = \frac{1}{2\sigma^2}$ can be used as regularization term, since it controls how strict the decision boundary is by determining a strong sharpness if γ is big (if σ is small) and a weak sharpness otherwise (if σ is big). So if the SVM is overfitting, γ should be reduced, and if it is underfitting, γ should be increased.

The kernel trick can be used together with the soft-margin in order to reach good performance and better generalization.

2.7 Classic deep learning models

Deep learning is a subfield of machine learning that differentiate itself for the specific type of models that it uses in order to learn a certain task. These models are *artificial neural networks*.

Artificial neural networks are very powerful and versatile models that are inspired by human neural networks: the structure of artificial neural networks is comparable to the one of our brain, which has neurons and connection between them to transmit electrical impulses. Similarly, artificial neural networks is composed by several neurons which are the computational cores of the model and which transmit the processed information to each other.

Artificial neural networks (or simply neural networks) organize the neurons in layers: the input layer is represented by the input data, the output layer is the last one that process data and returns the result of the entire model, while all the layers in between are called *hidden layers* and, in addition to transforming the data, they send the processed information to the next layer. Figure 2.11 shows an example of a typical network's structure. Each neuron processes the input data by computing a linear

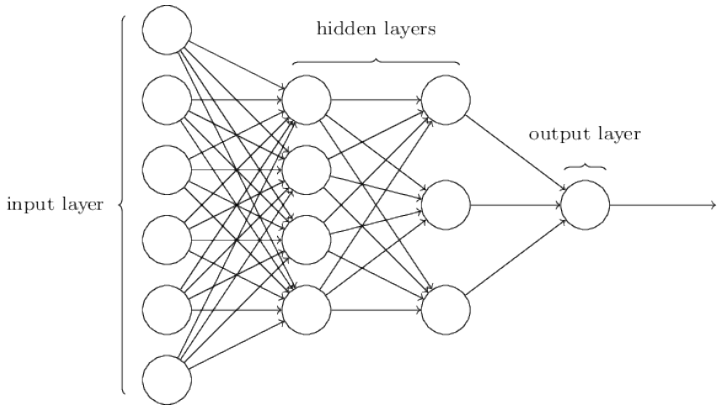


Figure 2.11. Neural network's structure and layers (from *Neural Networks and Deep Learning* free online book)

transformation, that is the weighted sum $\sum$ plus a bias, followed by a non-linear transformation, that is the activation function σ. The weighted sum is performed by multiplying each input feature x_i for its corespondent weight w_i and then taking the sum of all the resulting products. A bias term $b = 1$ is added to the weighted sum to help the network approximating the objective function. The result of this linear transformation represents the input of the activation function, which, as already mentioned, introduces the non-linearity properties by mapping the data to the desired output. Fig-

ure 2.12 illustrates the functioning of a single neuron. Each neuron in a layer performs

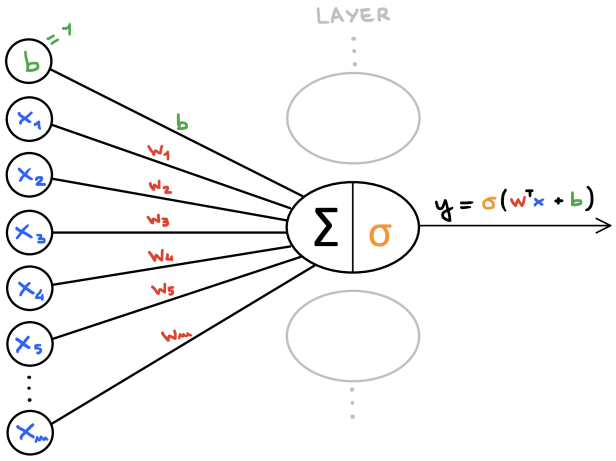


Figure 2.12. Illustrated description of the functioning of a single neuron

these operation and the outputs of all the neurons of that layer will be the inputs for the next layer. Formally, the inputs and outputs of layers of a neural network can be described by the equations:

$$h_0^{out} = \mathbf{X} \qquad \text{(input layer)} \tag{2.27}$$

$$h_i^{in} = h_{i-1}^{out} \times \mathbf{W}_i + \mathbf{B}_i \tag{2.28}$$

$$h_i^{out} = \phi_i(h_i^{in}) \tag{2.29}$$

where $\mathbf{X}$ is the features matrix of input data, $\mathbf{W}_i$ is the weights matrix of layer i, $\mathbf{B}_i$ is the bias vector of layer i, ϕ_i is the activation function used by neurons of layer i, and the operator $\times$ represents matrix multiplication.

2.7.1 Dense neural network

A dense neural network (or fully-connected neural network) is a network that consists of a sequence of fully-connected layers [31]. Each layer represents a function (linear and non-linear transformations) from $\mathbb{R}^F$ to $\mathbb{R}^{F_m}$. This means that, for each input instance having F features, the corresponding output will have dimensionality F_m, where F_m is the number of neurons in the network's output layer (the m-th layer). Even for the hidden layers, the output dimension depends on the number of neurons present in the layer.

The network is called *fully-connected* because the output of each neuron in one

layer is sent as input of each neuron in the next layer, so there is a direct connection between all the neurons belonging to subsequent layers. Figure 2.13 shows and example of dense neural network, making visible the full interconnection between subsequent layers.

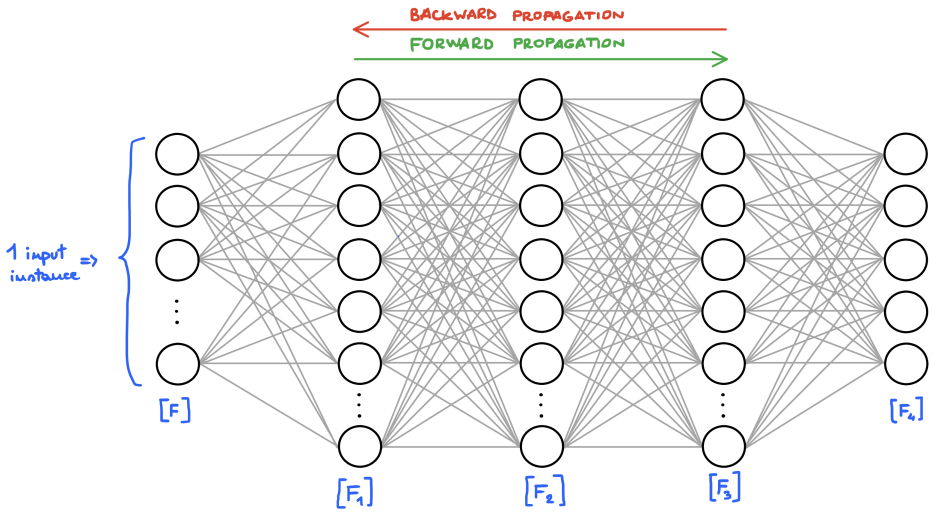


Figure 2.13. Example of dense neural network

For each training input instance, the model computes the output of every neuron in each consecutive layer until it reaches the final predictions from the output layer (*forward pass)*. After that, it compares its prediction with the actual targets and measures the output error. The error is then propagated through each layer in reverse in order to compute the error contribution (error gradient) from each neuron's connection, until it reaches the input layer (*backward pass*). The error is then used by the optimizer to update the connections weights and allow the model to learn. Dense neural networks are included in the category of feed-forward neural networks, since the activation flows only in one direction.

When used for classification tasks, the output layer of a dense neural network has a number of neurons that is equal to the number of classes of the data. In this way, each output of the network (computed through softmax or sigmoid activation function) corresponds to the estimated probability that the input instance belongs to the corresponding class. The instance is then assigned to the class which obtained higher probability.

2.7.2 Convolutional neural network

Convolutional neural networks (CNN) are feed-forward networks that work very well especially for image recognition and complex visual tasks in general [28]. A typical CNN architecture consists of different building blocks: several conv-pool blocks, each one containing a convolutional layer and a pooling layer, followed by a regular dense neural network at the end (see Figure 2.14). Each building block has a specific role: convolution layers identify low-level features in the image and learn to assemble them into higher-level features; pooling layers subsample the image in order to reduce its dimensionality; the dense network at the end of the architecture uses the extracted features to make a prediction. Let's look at the functioning of convolutional and pooling layers.

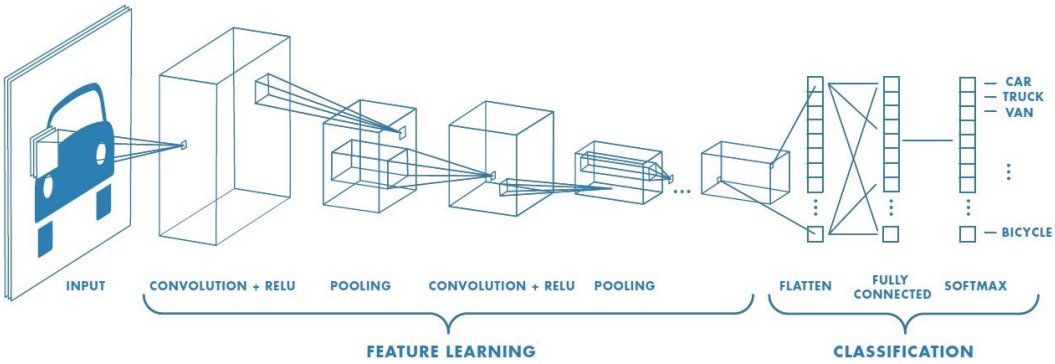


Figure 2.14. Typical architecture of a convolutional neural network (from *Medium - Towards Data Science*)

Convolutional layers are different from dense layers since they are not fully-connected: in convolutional layers, neurons are not connected to all previous layer's neurons, but only to neurons in their receptive fields. If we represent one layer's neurons mapped in 2D as a matrix, we can say that each neuron in the current matrix l looks at a limited region of the previous matrix $l-1$, so it is connected only to neurons located within that region, as shown in Figure 2.15. This architecture allows to focus on low-level features in the first layer, where neurons look at small regions of the input image, and to aggregate them in high-level features in the following layers, where neurons combine lower-level features form the previous layer.

In order for a layer to have the same dimensionality of the previous layer, zero padding can be applied by adding a zero-frame to the previous layer (Figure 2.16a). In order for a layer to have a much smaller dimensionality with respect to the previous layer, we can use a higher stride, which is the distance between two subsequent

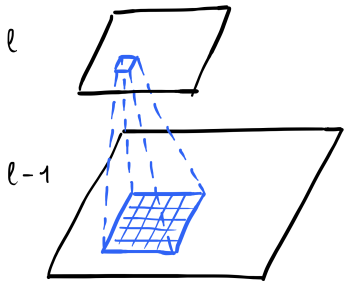
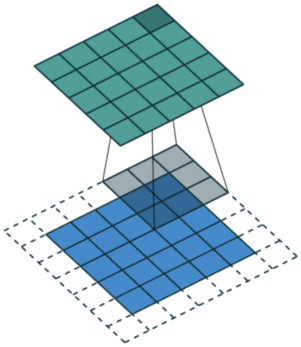
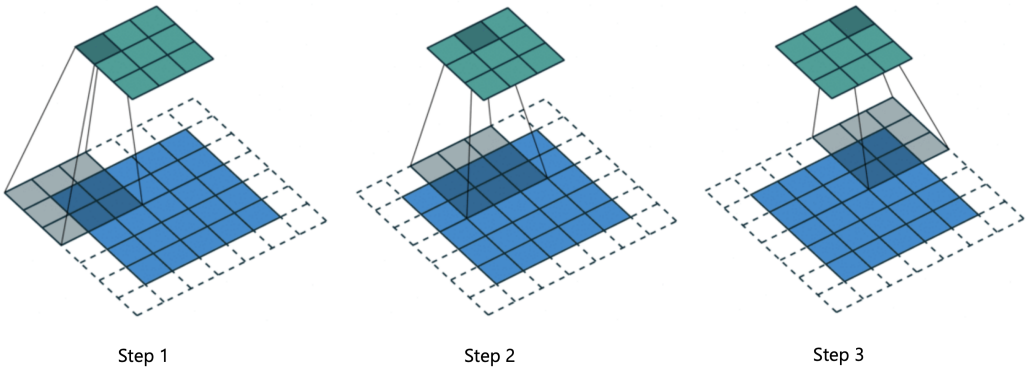


Figure 2.15. Relation between convolutional layer's neurons



(a) Zero padding



(b) Sequence of receptive fields with stride equal to 2

Figure 2.16. (from *Medium - Towards Data Science*)

receptive fields. By default, the stride is set to 1, so usually the difference between two receptive fields is is one row/column of neurons; by increasing the stride, a smaller number of receptive fields is taken into consideration, therefore the following layers is much smaller (Figure 2.16b).

In a convolutional layer, neurons weights are represented by *filters* (also called *kernels*). A filter is a matrix of the same size of the receptive field and it contains the weights that are applied to the corresponding receptive field. The application of the same filter to all the receptive fields of an image produces a feature map, which identify and highlight the details of the image that are most similar to the filter. A convolutional layer is composed by several feature maps of equal size, that in practice corresponds to having multiple matrices of neuron stacked together generating a third dimension, which is the number of feature maps. In a convolutional layer, all neurons in the same feature map share the same weights and bias, so they all use the same filter, while from layer to layer the filter used is different. In this way, the convolutional layer applies different filters to the input image and each feature map can specialize on identifying a specific image feature. Figure 2.17 shows a 3×3 filter applied over three feature maps (depth); for example, it could be the case of a color image composed of three color channels, which correspond to the three feature maps.

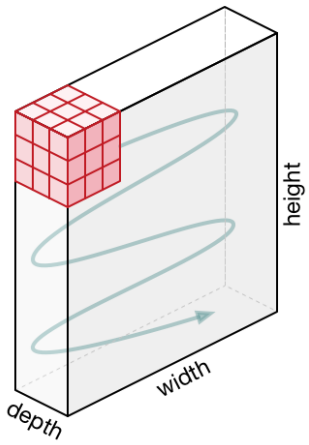


Figure 2.17. Movement of a 3×3 filter applied over three feature maps (from *Medium - Towards Data Science*)

Convolutional layers are usually followed by pooling layers. As already mentioned, the goal of a pooling layer is to subsample the input image in order to reduce its dimensionality; this action has the positive effects of cutting down the computational and memory loads and of reducing the number of parameters, thus diminishing the

risk of overfitting. A pooling layer is very similar to a convolutional layer, since it is focused on a receptive field too and it has configurable size, padding and stride. The crucial difference is that a pooling layer doesn't have weights, but instead it aggregate all the information in a receptive field by using an aggregation function (for example the mean or max functions). The pooling layer works on every channel (or feature map) independently, so the output has the same depth as the input.

Through convolutional and pooling layers, a CNN is able to learn high-level features, while reducing the data dimensionality, and to use this information to make predictions through the dense neural network at the end of the architecture.

2.7.3 LSTM neural network

LSTM neural networks are a special case of a more general category, which is the recurrent neural networks [28]; therefore we are going to first present the basics of recurrent neural networks and then explore LSTM neural networks.

Recurrent neural networks (RNN) are usually used to analyze sequences of data and time series in order to predict the future, thanks to their ability to "memorize" past information. This is due to the fact that they are not feed-forward network, since the output of a neuron is sent back to itself. In other words, at each time step t, a recurrent neuron receives the features of the input x_t as well as its own output y_{t-1} from the previous time step. In this way, each input of a recurrent neuron is determined by the current input instance alongside with some information from all the previous instances. Expanding this logic to an entire layer of recurrent neurons, at each time step t, each neuron receives both the input x_t as well as the entire layer's output y_{t-1} from the previous time step. Figure 2.18 illustrates the logic behind a recurrent neural network, showing the network unrolled through time.

Each recurrent neuron has two sets of weights, one for the inputs x_t and the other for the previous output y_{t-1}. Naming $\mathbf{W}_x$ and $\mathbf{W}_y$ respectively these two sets of weights, we can rewrite the equation for the output $\mathbf{Y}_t$ of a recurrent layer:

$$\mathbf{Y}_t = \phi(\mathbf{X}_t\mathbf{W}_x + \mathbf{Y}_{t-1}\mathbf{W}_y + \mathbf{b}) \tag{2.30}$$

where ϕ is the activation function of the layer and $\mathbf{b}$ is the bias. Concatenating the inputs $[\ \mathbf{X}_t \quad \mathbf{Y}_{t-1}\ ]$ and the weights $\mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$, we can rewrite the equation as:

$$\mathbf{Y}_t = \phi([\ \mathbf{X}_t \quad \mathbf{Y}_{t-1}\ ]\mathbf{W} + \mathbf{b}) \tag{2.31}$$

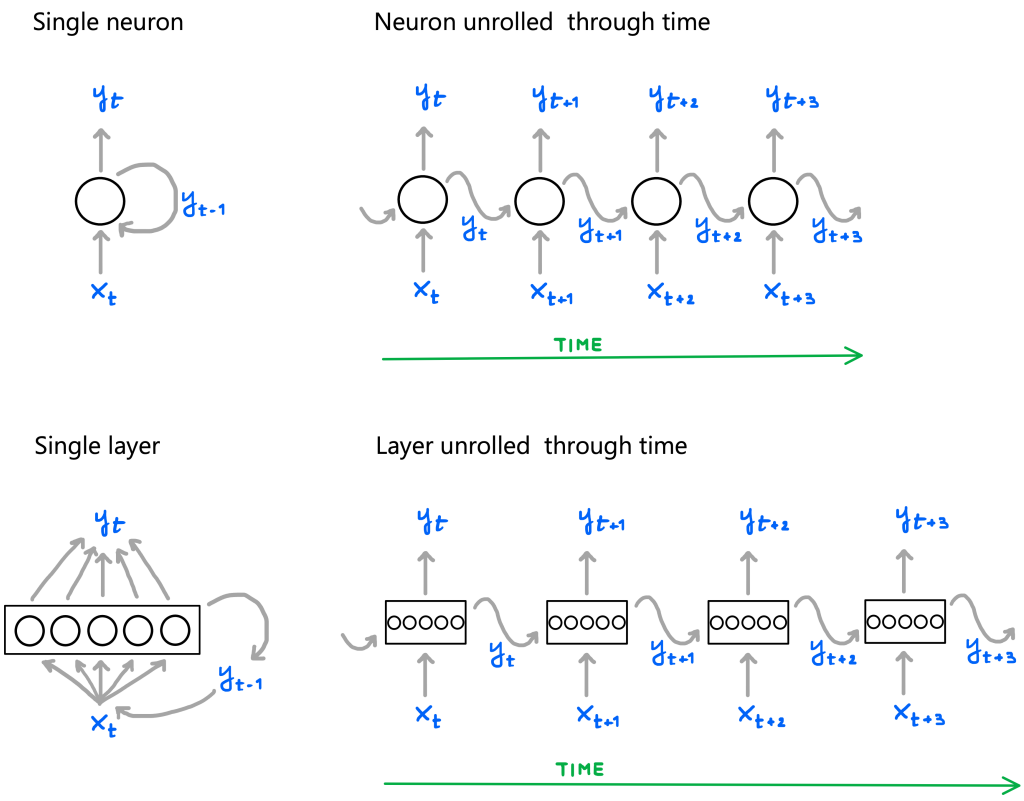


Figure 2.18. Logic behind a recurrent neural network

It should be clear by now that each output at time step t is a function of the inputs from all the previous time steps. For this reason, we can say that the network is able to store some sort of "memory" of the past inputs and to use this information for the outputs of the the following time steps. A neuron, or layer of neurons, that is able to preserve some state across time steps is called a *memory cell*. A cell's state at time t can be denoted with $\mathbf{h}_t$, which corresponds to a function $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$. In the case of basic memory cells like the ones in a recurrent neural network, the element $\mathbf{h}_{t-1}$ corresponds exactly to the notation $\mathbf{y}_{t-1}$ that we used until now to denote the output from the previous time step.

The classic recurrent neural network model presents some drawbacks. First of all, when the network is trained on long sequences, it may suffer form the vanishing/-exploding gradient problem, taking forever to train. Moreover, on the long run, the memory of information about the first time steps tends to fade away, since part of the information is lost at each step. We could consider the RNN to have a short-term memory, which is not optimal when we need to predict events based on a long sequence of

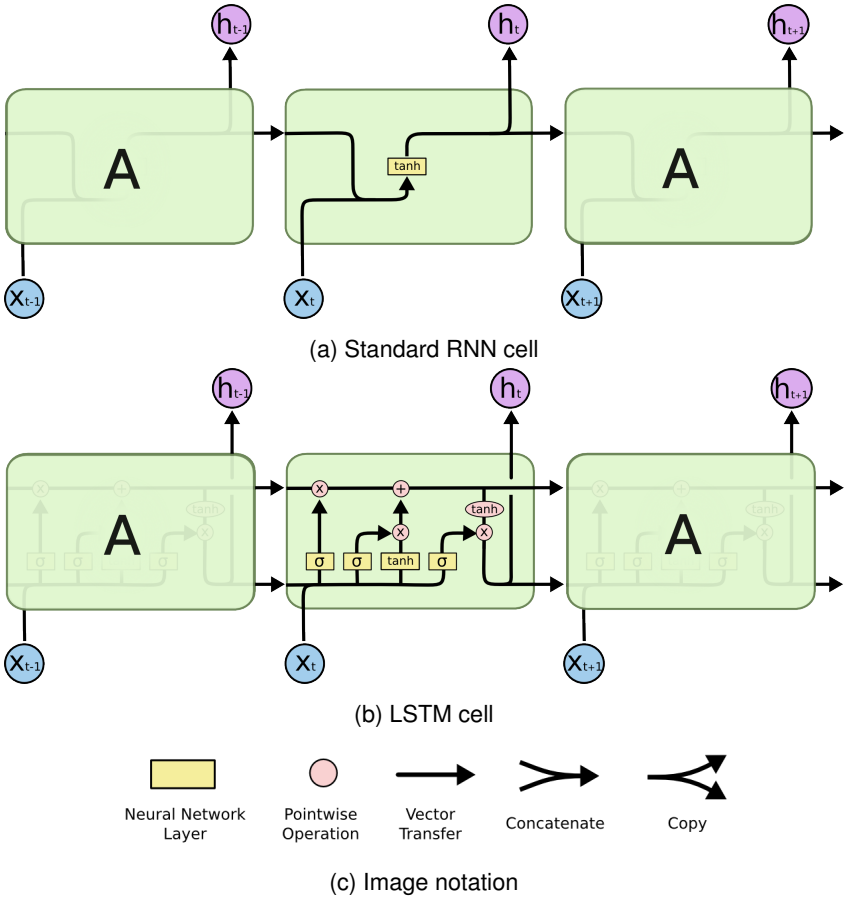


Figure 2.19. Difference between a standard RNN cell and a LSTM cell (from *colah's blog*)

time steps. That is where the LSTM neural network comes in.

An LSTM neural network, differently from a RNN, is made of *Long Short-Term Memory* (LSTM) cells [32]. This type of cells allows the model to perform much better and to converge faster, in addition to preserving long-term information in long sequences. As shown in Figure 2.19, while the standard RNN cell contains only a single layer (showed as a yellow box), the LSTM cell contains four interacting layers, each one having a well-defined purpose. An LSTM cell keeps "in memory" two states, one for the short-term memory and the other for the long-term memory, respectively $\mathbf{h}_t$ and $\mathbf{C}_t$ (also called *cell state*). The first one is the same that we saw for classic recurrent network's cells, while the cell state is what makes the LSTM model so special.

In addition to the three inputs $\mathbf{x}_t$, $\mathbf{h}_{t-1}$ and $\mathbf{C}_{t-1}$, the LSTM cell makes use of four different fully connected layer, three of them using a sigmoid activation function and the remaining one using a tanh activation function. Each one of the sigmoid activation

outputs goes through a *gate*, indicated in Figure 2.19 with the symbol ⊗, representing an element-wise multiplication. Gates use the output of the sigmoids (between 0 and 1) to determine how much information should be let through. Let's dive into the functioning of an LSTM cell.

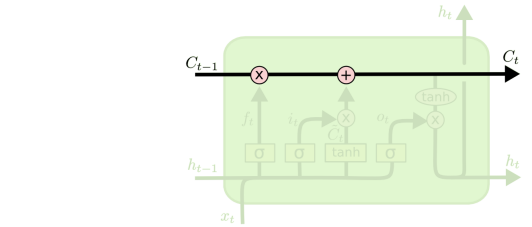
The cell state allows the network to decide what to store in the long-term memory and what to discard. Traversing the cell from left to right (Figure 2.20a), the cell states goes through two operations, the first being the *forget gate* and the second being and addition operation. The forget state determines which long-term information to drop from $\mathbf{C}_{t-1}$ and it is controlled by the first sigmoid layer's output f_t, calculated from $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ (Figure 2.20b). The addition operation, on the other hand, adds information to the cell state based on the output of the *input gate*, which determines which new information coming from $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ will be stored in the cell state (Figure 2.20d). The input gate is controlled by the second sigmoid layer's output i_t and by the tanh layer's output $\tilde{C}_t$ (Figure 2.20c). After that, the long-term state $\mathbf{C}_t$ is ready to be output, but before that, its state is copied to be passed to a than activation function and to be filtered by the last gate, which is the *output gate*, controlled by the third sigmoid layer's output o_t (Figure 2.20d). This operation is essential in order to produce the short-term state $\mathbf{h}_t$ to output.

To sum up the fully-connected layer's roles:

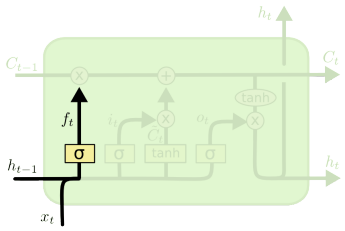
- The tanh layer that outputs $\tilde{C}_t$ takes as inputs $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ and it is the same that we find in a basic recurrent cell. The result is partially stored as new information in the long-term state.
- The sigmoid layer that outputs f_t takes as inputs $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ and it controls the forget gate. Based on the result, some parts of the long-term state are erased.
- The sigmoid layer that outputs i_t takes as inputs $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ and it controls the input gate. Based on the result, some parts of the new information in $\tilde{C}_t$ are added to the long-term state.
- The sigmoid layer that outputs o_t takes as inputs $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ and it controls the output gate. Based on the result, some parts of the long-term state are read and output as $\mathbf{h}_t$.

All the equations for the different outputs can be found in Figure 2.20.

Through this process, the LSTM model is able to preserve important information through time and to understand which are the essential inputs to keep and what to discard. This allows the network to learn long-term pattern in the data.

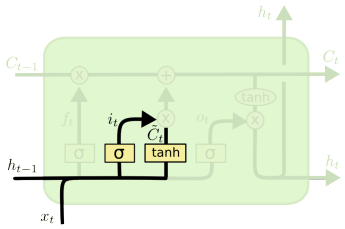


(a)



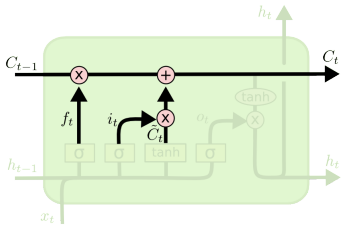
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

(b)



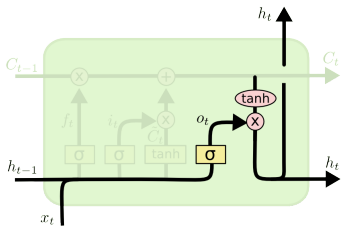
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

(c)



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

(d)



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

(e)

Figure 2.20. Elements in an LSTM cell step-by-step (from *colah's blog*)

2.8 Functional connectivity and graph representation

2.8.1 Functional connectivity

2.8.2 Graph representation

2.9 Graph-based deep learning models

2.9.1 Graph neural network

2.9.2 Edge-conditioned convolution

2.9.3 Global pooling

2.9.4 Graph-based LSTM and Convolutional neural networks

Chapter 3

Methods

In this chapter ...

Chapter 4

Implementation

In this chapter ...

Chapter 5

Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Notes

Topics

- Intro (big summary of the thesis)
    - Problem statement
    - What (GNN, LSTM, etc.)
- Background (theoretical explanation of models used and SOTA)
    - Machine learning methods
    - Neural Networks
    - Dense
    - LSTM
    - Conv
    - GNN
    - Seizure prediction
    - EEG
    - Functional connectivity network
    - State of the art
- Methods (description of all the work done and motivations, but no numbers)
    - Baseline
    - Graph / contribution
- Experiments/Implementation (technical description of all the work done with parameters and numbers)
    - Environment (software, server, framework)

Data + preprocessing

Architectures (params)

Results

- Conclusions

  Comments

  Future work

Project steps

- Baseline

  Detection

  — Random Forest

  — Gradient Boosting

  — SVM

  — Dense

  — LSTM

  — Convolution

  Prediction

  — LSTM

  — Convolution

  —— Stride > 1 and bigger look_back

  —— Parameters search

- Graph Neural Networks

  GNN + LSTM

  GNN + Conv1D

  TODO Pooling

  — **TODO** Hierarchical (Rex Ying)

  — **TODO** Decimation (Bianchi)

- Baseline

  TODO Cross Validation on 3 seizures

  TODO Bayesian optimization

  TODO Different functional connectivity nets

  TODO Big data

  TODO Structural connectivity (KNN)

5.1 General considerations

5.1.1 IEEG plots

The ieeg plots give an overall idea of the type of data to deal with and of the shape of the patient's seizures. On 24 hours of recording, there are three seizures in the first 3 hours. The data is divided in clips of 1 hour each, containing 1800000 timestamps each (the signal is sampled at 500Hz). The three seizures found have a duration between 13000 and 15000 timestamps each (between 26 and 30 seconds).

The ieeg doesn't explicitly show the presence of a seizure; indeed the dynamic of the electrodes signals doesn't change in correspondence of the beginning of a seizure. For example, in the first seizure the signals dynamic remains pretty stable during the first half of the seizure and suddenly changes at the beginning of the second half of the seizure. This behaviour suggests that the seizure is not directly represented by the dynamic of the signals, but it is represented by the non-linear relations between the electrodes behaviours.

The ieeg plots show an interesting thing: in all the three seizures, there are two electrodes that are more dynamic exactly during the seizure. In the plots they are always represented with green colour.

- classic plot (with y values): electrodes signals positioned at -5000 and 2500.

- norm plot (y normalised): electrodes signals positioned at 8th position from the top and 33th position from the bottom.

5.2 Machine learning classic models

5.2.1 Experiments with classic methods

The three machine learning classic methods used to perform experiments are random forest, gradient boosting and support vector machine. All the three methods have been implemented using scikit learns modules; for gradient boosting also xgboost library has been used. For all the experiment just portions containing seizures of the entire dataset have been used to train and to test data in order to speed up the process and to reduce the huge difference of availability of data between the two classes.

Classic methods have been used only for the detection task, since they are not powerful enough in order to deal with the prediction task, at least with this kind of data.

Parameters:

- random forest:

  number of estimators: 100

  maximum depth: 10

  both balanced and non-balanced class weight

- gradient boosting:

  number of estimators: 100

  maximum depth: 10

- svm:

  gamma: scale

  both weighted and non-weighted classes

In all the experiments, seizure 2 and 3 have been used as training set, while seizure 1 has been used as test set.

Maybe try to cross-validate by exchanging datasets for training and testing.

Results: All the experiments had almost the same results:

- Loss: around 0.15
- Accuracy: around 0.85

- Roc auc: around 0.65

Only the unbalanced random forest was able to get almost decent results:

- Loss: 0.11
- Accuracy: 0.85
- Roc auc: 0.73

The results are very bad and show that the classic methods are too weak to be able to solve this problem having to deal with such a complex and unbalanced dataset. Indeed, the area under the Roc curve is around 0.65 for the majority of the models, which means that the predictions are made almost in a random way.

Maybe try using some different metric (ex. F1)

The predictions have also been plotted in order to have a better idea of what was predicted wrong and where and also the plots show a non-logical distribution, even if in some cases (for example in the non-balanced experiment with the random forest) the predictions in the seizure area seem to match with the targets. Also in the gradient boosting experiment, the model seems to start predicting 1 (there is a seizure) in the same timestamps in which the signals in the ieeg start to be more dynamic (almost in the middle of the seizure 1).

5.3 Deep learning classic models

5.3.1 Dense neural network

A dense neural network have been built in order to deal with the seizure detection task. After some hyperparameters tuning, the final structure of the network was the following:

Parameters: These are the parameters used in the network's layers.

- activation: tanh
- kernel regularisation: L2(5e-4)
- class weight: {0: len(y_train) / n_negative, len(y_train / n_positive}

Structure: These are the layers used to build the network.

```
Dense(units: 512)
Dropout(0.5)
Dense(units: 512)
Dropout(0.5)
Dense(units: 256)
Dropout(0.5)
Dense(units: 1, activation: 'sigmoid')
```

In order to compile the model, binary cross-entropy has been used as loss and Adam as optimiser.

Initially the structure was composed by less layers and each one with less units (ex. 128, 156), but the model wasn't powerful enough so both the number of layers and units have been increased. The kernel regularisation and the dropout layers have been added in order to avoid overfitting on training data. For the activation of the dense layers, both tanh and ReLu have been tried, but for some reason tanh works way better. Since the two classes of the problem, that are "seizure" (1) and "not-seizure" (0), are very unbalanced due to the small presence of seizure timestamps with respect to the number of non-seizure timestamps, class weight has been used to train the network. The weight assigned to each class is inversely proportional to the number of timestamps of that class with respect to the total number of timestamps.

Initially, in order to preprocess the data, the MinMaxScaler from scikit-learn library was used, but then it was substituted by the StandardScaler because this one led to better performances. The data are also shuffled in order to help the network to generalise.

Initially, as for the machine learning classic methods, just portions containing seizures of the entire dataset have been used to train and to test data. In order to do some test, a modification of the initial dataset has been tried: the dataset has been modified so that each clip is trimmed ending with the end of the seizure and starting 100,000 timestamps before the end of the seizure. The intention of this test was to eliminate the useless bias generated by the timestamps after the seizure. However, after testing it, for some reason the results got a lot worse, so the old portions of the dataset have been used for the next experiments.

Results After all the tuning described above, the best results that the dense network got on the detection task were the following:

- Loss: 0.87

- Accuracy: 0.79
- Roc auc: 0.74

These results are not extremely good but they are ok considering the complexity of the problem. The dense network behaved in a similar way to the unbalanced random forest, looking at the results. Looking at the plots of the predictions, they don't seem to be very promising, but in order to better understand the behaviour of the prediction, also the running mean have been plotted and it explicitly shows that the network is able to understand where is the seizure.

A strange thing happens in the predictions: in the test prediction plot, there is a sort of a "hole" of zero predictions right after the end of the seizure.

5.3.2 LSTM neural network

I built an LSTM network in order to deal both with the detection and the prediction tasks.

For this network, I introduced subsampling to the data preprocessing. The subsampling function take as input the subsampling factor, which represent how many negative examples we want to keep with respect to the positive ones. In this way, we can consistently reduce the disparity in the amount of positive and negative examples, preserving all the positive ones. The subsampling was performed both for the detection and the prediction task. The subsampling factor was set to 2, so that for each positive samples there are two negative ones. The stride in the subsampling was set to 1 in order to keep all the (positive) samples.

Another function was used in order to generate the sequences of data and labels to give as input to the network. It takes the parameters "look_back", which is the length of the input sequence (how many samples do I look behind), and "target_steps_ahead", which represents how many steps ahead to predict (how many samples there are between the end of the input sequence and the future sample I want to predict). Given this two parameters, it creates the pairs input_sequence - target accordingly.

For the detection task, the parameter "target_steps_ahead" was set to 0, that is the target corresponding to the last sample of the input sequence. For the prediction task, obviously the parameter was set to some values higher than 0, as we want to predict the target of a sample in the future, that wasn't inside the input sequence.

Detection task For the detection task, after some hyperparameter search and tuning, the best result was given by these parameters (experiment 147):

```
epochs:         10
batch_size:     64
depth_lstm:     1
depth_dense:    2
units_lstm:     256
reg:             l2(5e-1)
activation:     relu
batch_norm:     True
dropout:         0.4
class_weight:   {0: 0.4682, 1: 3.0}
look_back:      100
stride:         1
predicted_timestamps:    1
target_steps_ahead:      0
subsampling_factor:      2
```

The network was build in this way:

```
LSTM(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 100 were:

- Loss: 0.2632
- Accuracy: 0.9059
- Roc auc: 0.9263

Prediction task For the prediction task, after some hyperparameter search and tuning, the best result was given by these parameters (experiment 20):

```
epochs:         10
batch_size:     64
depth_lstm:     1
depth_dense:    2
units_lstm:     256
```

```
reg:             l2(5e-1)
activation:    relu
batch_norm:    True
dropout:        0.4
class_weight:  {0: 1.1561, 1: 7.4074}
look_back:     200
stride:        1
predicted_timestamps:   1
target_steps_ahead:     2000
subsampling_factor:     2
```

The network was build in this way:

```
LSTM(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 100 and a target steps ahead of 2000 were:

- Loss: 0.3098
- Accuracy: 0.9454
- Roc auc: 0.9337

The network that performed better was the same for both the detection and prediction tasks, even with the same regularisation parameters. What changed was, of course, the distance of the sample predicted and the length of the input sequence. The LSTM, in both the situation, wasn't able to deal with more than 200 samples in the input sequence (the results were way worse with a higher value). For the prediction task, the network was able to successfully predict until 2000 steps ahead, which correspond to 4 seconds. Higher than that, the prediction became almost random.

5.3.3 Convolutional neural network

I built a CNN in order to deal both with the detection and the prediction tasks.

For this network, as for the LSTM, I introduced subsampling to the data preprocessing (see LSTM).The subsampling was performed both for the detection and the prediction task. The subsampling factor was set to 2, so that for each positive samples there are two negative ones. Another function was used in order to generate the sequences of data and labels to give as input to the network (see LSTM).

Detection task For the detection task, we used the same parameter we found for the LSTM, except for the look_back, that in the case of the convolutional can be much higher. The best result was given by these parameters (experiment 2):

```
epochs:         10
batch_size:     64
depth_conv:     5
depth_dense:    2
filters:         512
kernel_size:    5
reg:             l2(5e-1)
activation:     relu
batch_norm:     True
dropout:         0.4
class_weight:  {0: 1.1561, 1: 7.4074}
look_back:      1000
stride:         1
predicted_timestamps:   1
target_steps_ahead:     0
subsampling_factor:     2
```

Run again experiments for detection with CNN using less filters and smaller kernel size

The network was build in this way:

```
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
Conv1D(filters: 512)
MaxPooling1D()
```

```
Flatten()
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 1000 were:

- Loss: 0.1550
- Accuracy: 0.9374
- Roc auc: 0.9870

Prediction task For the prediction task, I began with some hyperparameter search and tuning using the same parameters used for the LSTM as initial model. The best result was given by these parameters (experiment 29):

```
epochs:        10
batch_size:    64
depth_conv:    5
depth_dense:   2
filters:        256
kernel_size:   5
reg:            l2(5e-1)
activation:    relu
batch_norm:    True
dropout:        0.4
class_weight:  {0: 1.1560, 1: 7.4074}
look_back:     1000
stride:        1
predicted_timestamps:   1
target_steps_ahead:     2000
subsampling_factor:     2
```

The network was build in this way:

```
Conv1D(filters: 256)
MaxPooling1D()
Conv1D(filters: 256)
MaxPooling1D()
Conv1D(filters: 256)
MaxPooling1D()
```

```
Conv1D(filters: 256)
MaxPooling1D()
Conv1D(filters: 128)
MaxPooling1D()
Flatten()
Dense(units: 256)
BatchNormalization()
Dropout(0.4)
Dense(units: 1, activation: 'sigmoid')
```

The results on an input sequence of length 100 and a target steps ahead of 2000 were:

- Loss: 0.2598
- Accuracy: 0.9314
- Roc auc: 0.9274

For the prediction task, the network was able to successfully predict until 2000 steps ahead, which correspond to 4 seconds. Higher than that, the prediction became almost random.

CNN notes

Bibliography

[1] B.S. Chang and Lowenstein D.H. Epilepsy. *The New England Journal of Medicine*, 349(13):1257–66, September 2003. doi: 10.1056/NEJMra022308. PMID: 14507951.

[2] R.S. Fisher et al. Ilae official report: a practical clinical definition of epilepsy. *Epilepsia*, 55(4): 475–82, April 2014. doi: 10.1111/epi.12550. PMID: 24730690.

[3] World Health Organization (WHO). Epilepsy, February 2018. URL `http://www.who.int/news-room/fact-sheets/detail/epilepsy`. [Online; accessed 1-November-2018].

[4] D.L. Longo. *369 Seizures and Epilepsy - Harrison's principles of internal medicine*. McGraw-Hill, 18th edition, 2012. ISBN 978-0-07-174887-2.

[5] M.J. Eadie. Shortcomings in the current treatment of epilepsy. *Expert Review of Neurotherapeutics*, 12(12):1419–27, December 2012. doi: 10.1586/ern.12.129. PMID: 23237349.

[6] Patrick Kwan et al. Definition of drug resistant epilepsy: Consensus proposal by the ad hoc task force of the ilae commission on therapeutic strategies. *Epilepsia*, 51(6):1069–1077, June 1st 2010. ISSN 1528-1167. doi: 10.1111/j.1528-1167.2009.02397.x. PMID: 19889013.

[7] Patrick Kwan and Martin J. Brodie. Early identification of refractory epilepsy. *The New England Journal of Medicine*, 342(5):314–319, February 3rd 2000. ISSN 0028-4793. doi: 10.1056/NEJM200002033420503. PMID: 10660394.

[8] M. Bishop Cristopher. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN 978-0-387-31073-2.

[9] Emerj. Machine learning healthcare applications – 2018 and beyond, February 2019. URL `https://emerj.com/ai-sector-overviews/machine-learning-healthcare-applications`. [Online; accessed 7-March-2019].

[10] K. Bergey Gregory. Neurostimulation in the treatment of epilepsy. *Experimental Neurology*, 244: 87–95, June 2013. doi: 10.1016/j.expneurol.2013.04.004. PMID: 23583414.

[11] Epilepsy Foundation. Neurostimulation in the treatment of epilepsy, May 31st 2017. URL `https://www.epilepsy.com/article/2017/5/neurostimulation-treatment-epilepsy`. [Online; accessed 25-April-2019].

[12] M.J. Cook, T.J. O'Brien, S.F. Berkovic, et al. Prediction of seizure likelihood with a long-term, implanted seizure advisory system in patients with drug-resistant epilepsy: a first-in-man study. *The Lancet Neurology*, 12(6):563–571, June 2013. doi: 10.1016/S1474-4422(13)70075-9.

[13] B. Litt and K. Lehnertz. Seizure prediction and the preseizure period. *Curr Opin Neurol.*, 15(2): 173–177, April 2002. PMID: 11923631.

[14] Mormann Florian, G. Andrzejak Ralph, E. Elger Christian, and Lehnertz Klaus. Seizure prediction: the long and winding road. *Brain*, 130:314–333, 2007. doi: 10.1093/brain/awl241.

[15] K. Lehnertz, F. Mormann, H. Osterhage, et al. State-of-the-art of seizure prediction. *J Clin Neurophysiol*, 24:147–153, May 2007. doi: 10.1097/WNP.0b013e3180336f16.

[16] K. Lehnertz, F. Mormann, T. Kreuz, et al. Seizure prediction by nonlinear eeg analysis. *IEEE Engineering in Medicine and Biology Magazine*, 22(1):57–63, April 2003. doi: 10.1109/MEMB.2003.1191451.

[17] G. Andrzejak Ralph, Chicharro Daniel, E. Elger Christian, and Mormann Florian. Seizure prediction: Any better than chance? *Clinical Neurophysiology*, 120(8):1465–1478, August 2009. doi: 10.1016/j.clinph.2009.05.019.

[18] Gadhoumi Kais, Lina Jean-Marc, Mormann Florian, and Gotman Jean. Seizure prediction for therapeutic devices: A review. *Journal of Neuroscience Method*, 260:270–282, February 2016. doi: 10.1016/j.jneumeth.2015.06.010.

[19] Saad Zaghloul Zaghloul and Bayoumi Magdy. Early prediction of epilepsy seizures vlsi bci system, June 2019. URL `https://arxiv.org/abs/1906.02894`. arXiv:1906.02894 [eess.SP].

[20] Usman Syed Muhammad, Usman Muhammad, and Fong Simon. Epileptic seizures prediction using machine learning methods. *Comput Math Methods Med.*, December 2017. doi: 10.1155/2017/9074759. PMID: 29410700.

[21] DR Freestone, PJ Karoly, and MJ Cook. A forward-looking review of seizure prediction. *Curr Opin Neurol.*, 30(2):167–173, April 2017. doi: 10.1097/WCO.0000000000000429. PMID: 28118302.

[22] Hügle Maria et al. Early seizure detection with an energy-efficient convolutional neural network on an implantable microcontroller, June 2018. URL `https://arxiv.org/abs/1806.04549`. arXiv:1806.04549 [stat.ML].

[23] Haddad Tahar et al. Epilepsy seizure prediction using graph theory. *IEEE*, June 2014. doi: 10.1109/NEWCAS.2014.6934040.

[24] DuyTruong Nhan et al. Convolutional neural networks for seizure prediction using intracranial and scalp electroencephalogram. *Neural Networks*, 105:104–111, September 2018. doi: 10.1016/j.neunet.2018.04.018.

[25] Hussein Ramy, Osama Ahmed Mohamed, Ward Rabab, Jane Wang Z., Kuhlmann Levin, and Guo Yi. Human intracranial eeg quantitative analysis and automatic feature learning for epileptic seizure prediction, April 2019. URL `https://arxiv.org/abs/1904.03603`. arXiv:1904.03603 [cs.NE].

[26] Duy Truong Nhan, Kuhlmann Levin, Reza Bonyadi Mohammad, and Kavehei Omid. Semi-supervised seizure prediction with generative adversarial networks, June 2018. URL https://arxiv.org/abs/1806.08235. arXiv:1806.08235 [cs.CV].

[27] Epilepsy Society. About epilepsy, January 2017. URL https://www.epilepsysociety.org.uk/epilepsy#.XRJN-5MzZ24. [Online; accessed 25-June-2019].

[28] Géron Aurélien. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc., March 2017. ISBN 978-1-491-96229-9.

[29] P. Kingma Diederik and Ba Jimmy. Adam: A method for stochastic optimization, December 2014. URL https://arxiv.org/abs/1412.6980. [Online; accessed 30-June-2019].

[30] Ruder Sebastian. An overview of gradient descent optimization algorithms, September 2016. URL https://arxiv.org/abs/1609.04747. [Online; accessed 30-June-2019].

[31] Bosagh Zadeh Reza and Ramsundar Bharath. *TensorFlow for Deep Learning*. O'Reilly Media, Inc., March 2018. ISBN 978-1-491-98044-6.

[32] Hochreiter Sepp and Schmidhuber Jürgen. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.