



COMILLAS

UNIVERSIDAD PONTIFICIA

ICAI

MASTER´S DEGREE IN INDUSTRIAL ENGINEERING

MASTER´S FINAL THESIS

**ADVANCED NEURAL NETWORKS
ARCHITECTURES RESEARCH -
FORECASTING
RECOMMENDATIONS**

Author: Santiago Rilo Sánchez

Supervisors:

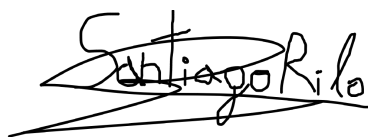
Jose Portela González

Jaime Pizarroso Gonzalo

Madrid

August 2020

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título *Advanced Neural Networks Architectures Research – Forecasting Recommendations* en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2019/2020 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.



Fdo.: Santiago Rilo Sánchez Fecha: 30/ 08/ 2020

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Fdo.: José Portela González Fecha: 30/ 08/ 2020



Fdo.: Jaime Pizarroso Gonzalo Fecha: 30/ 08/ 2020

*A mis padres por su apoyo, comprensión y cariño en estos años de Universidad.
A todos mis amigos dentro y fuera del ICAI que me han acompañado y apoyado,
haciendo este proyecto posible*

Acknowledgment

To the Chair of Connected Industry (Cátedra de Industria Conectada) and the CIC LAB.

Special thanks to Dr. José Portela and Jaime Pizarroso for guiding me and supporting me in this endeavour.

To Álvaro López for starting this initiative, trusting and encouraging research and fostering student-led projects.

To Mónica López-Tafall and Daniel Elechiguerra for their meaningful contributions.

To Dr. Jaime Boal for his help and expertise in Latex.

Contents

Abstract	xi
Resumen del proyecto	xv
1. Introduction	1
1.1. A Brief History of Machine Learning and Deep Learning	1
1.2. Objectives and scope	2
1.3. Basic Concepts	2
2. State of the art	5
2.1. Neural Architectures	5
2.1.1. Multilayer Perceptron	5
2.1.2. Convolutional Neural Networks	7
2.1.3. Recurrent Neural Networks	9
2.1.3.1. Vanilla Recurrent Neural Networks	11
2.1.3.2. Long-Short Term Memory Recurrent Neural Networks	12
2.1.3.3. Gated Recurrent Units Neural Networks	13
2.1.3.4. Bi-directional RNNs and Bi-directional LSTMs	14
2.1.3.5. Hopfield Networks	14
2.1.3.6. Boltzmann Machine	15
2.1.3.7. Deep Belief Network	16
2.1.4. Deep Auto-encoders	18
2.2. Activation functions	19
2.2.1. Sigmoid	19
2.2.2. Softmax	20
2.2.3. Linear	20
2.2.4. Hyperbolic tangent (tanh)	21
2.2.5. Rectified Linear Unit (ReLU)	21
2.2.6. Softplus	22
2.2.7. Leaky ReLU and Generalized ReLU	23
2.2.8. Exponential Linear Unit (ELU)	23
2.2.9. Scaled Exponential Linear Unit (SELU)	23
2.2.10. Maxout	24
2.2.11. Swish	24
2.3. Optimization algorithms	25
2.3.1. Gradient Descent	25
2.3.2. Stochastic Gradient Descent	25

2.3.3. Mini-Batch Gradient Descent	25
2.3.4. Momentum	26
2.3.5. Nesterov Momentum	26
2.3.6. AdaGrad	26
2.3.7. AdaDelta	27
2.3.8. RMSProp	28
2.3.9. Adam	28
2.3.10 Nadam	29
2.4. Loss functions	29
2.5. Optimization techniques	30
2.5.1. Regularization	31
2.5.2. Parameter initialization	31
2.6. Use Cases	32
2.6.1. Classification Problems	32
2.6.2. Regression Problems	32
2.6.3. Forecasting Problems	32
2.6.4. Unsupervised Learning	32
3. Case Study	33
3.1. Problem Statement	33
3.2. LSTM and GRU networks comparison	35
3.2.1. First LSTM-GRU Test	35
3.2.2. Second LSTM-GRU Test	36
3.2.3. Optimized algorithms and analysis with outliers	36
3.2.4. Optimized algorithms and analysis without outliers	38
3.2.5. Model dynamic response	42
3.3. Convolutional Neural Network predictor	43
4. Conclusions	47
4.1. Key takeaways	47
4.2. Recommendations	48
Sustainable Development Goals	49
References	51

List of Figures

Figure 1.1. Deep Learning timeline	2
Figure 2.1. Neuron Diagram	6
Figure 2.2. Multilayer Perceptron Diagram	7
Figure 2.3. Steps in Backpropagation [Dos19a]	7
Figure 2.4. Convolutional Neural Network LeNet-5	7
Figure 2.5. Visual representation of a convolution [Ste19b]	8
Figure 2.6. Common Pooling techniques [Ste19b]	9
Figure 2.7. Flattening step example in a Convolutional Neural Network [Ano18a]	9
Figure 2.8. Convolutional Neural Networks Layers Connection [Ara18]	10
Figure 2.9. Flattened version of a Recurrent Neural Network [Sal97]	10
Figure 2.10. Recurrent Neural Network Diagram untangled in time [PG17]	11
Figure 2.11. Vanilla RNN and equations [LJY17]	11
Figure 2.12. Basic architectures of RNN, GRU and LSTM cells [Rat18]	12
Figure 2.13. LSTM cell model and equations [Rat18]	13
Figure 2.14. GRU cell model and equations [Rat18]	13
Figure 2.15. Simple bidirectional RNN network [Raj19]	14
Figure 2.16. 3 neuron Hopfield Network with 8 possible states, two stable [Raj19]	15
Figure 2.17. Hopfield network acting on degraded images from noisy or partial inputs [Lev02]	15
Figure 2.18. Energy equation of a Boltzmann Machine [19]	16
Figure 2.19. CRepresentation of a Deep Believe Network [Póc17]	17
Figure 2.20. Comparisson between pre-trained or not pretrained DBN [Póc17]	17
Figure 2.21. Deep Autoencoder Structures [Le15]	18
Figure 2.22. Sigmoid or Logistic Function [Mal18]	20
Figure 2.23. Softmax transform [Uni20]	20
Figure 2.24. Representation of a linear activation function [Gup20]	21
Figure 2.25. Hyperbolic tangent function [V17]	21
Figure 2.26. ReLU Function [Mem16]	22
Figure 2.27. Softplus function [Ser17]	22
Figure 2.28. Rectified Linear Functions [Ste19a]	23
Figure 2.29. Exponential Linear Unit	23
Figure 2.30. Scaled Exponential Linear Unit	24
Figure 2.31. Maxout Function and equation [Ste19a]	24
Figure 2.32. Swish function and derivative [Ste19a]	24
Figure 2.33. Optimization algorithms with (right) and without (left) a momentum term	26

List of Figures

Figure 2.34. Nesterov Momentum Equations [Sin17]	27
Figure 3.1. Different types of inputs for the forecasting problem	34
Figure 3.2. Power generated Output	34
Figure 3.3. Results of the first LSTM-GRU test	35
Figure 3.4. Results of the second LSTM-GRU test	36
Figure 3.5. GRU model avoiding an outlier in time-step 791	37
Figure 3.6. Normalized residuals for the two advanced neural network models	37
Figure 3.7. Sample of output timeseries without outliers	38
Figure 3.8. Learning curve from optimized GRU	38
Figure 3.9. Learning curve from optimized LSTM	39
Figure 3.10. Sample of the fitted models without outliers	40
Figure 3.11. Fitted models, output and benchmarks	40
Figure 3.12. Q-Q Plot of the neural network predictors	40
Figure 3.13. Plot of the normalized residuals for the LSTM and the GRU	41
Figure 3.14. Histogram the normalized residuals for the LSTM and the GRU	41
Figure 3.15. LSTM partial and complete auto-correlation plots	41
Figure 3.16. GRU partial and complete auto-correlation plots	41
Figure 3.17. Validation dataset with steps implemented	42
Figure 3.18. Time response of the downwards step	42
Figure 3.19. Plot of the upwards step	43
Figure 3.20. First training run of the uni-dimensional convolutional neural network	44
Figure 3.21. Second training run of the uni-dimensional convolutional neural network	44
Figure 3.22. Comparison plot between optimized CNN and GRU predictors	45
Figure 3.23. Q-Q Plot for the residuals of the CNN predictor	45
Figure 3.24. Plot of normalized residuals of CNN and GRU models	46
Figure 3.25. Partial and total auto-correlation graphs for the CNN	46

Abstract

This project contains a comparison between different neural network architectures with the goal of evaluating the effect a given architecture has in a wind power generation forecasting problem. It includes an extensive state of the art and a use case. The project concludes the GRU neural networks are successful at tackling timeseries forecasting, the CNN should be used for large datasets and that several alternative activation functions can outperform the ReLU.

Keywords: Machine Learning, GRU, CNN, LSTM, Artificial Intelligence.

Introduction

The main purpose of this final thesis is to come up with a series of recommendations to select the right architecture for a neural network for an specific problem, focusing on timeseries forecasting problems. The project involves an extensive state-of-the-art section in which a review of current and past trends is carried out in order to explore the most popular deep neural network architectures.

The second part of the project involved selecting, designing and analyzing networks that were tested using a real data set from a wind energy forecasting competition containing wind power generation data.

Case study

The dataset to be analyzed comes from a machine learning contest and it contains recorded values of wind power generation with 90 different input variables. The available explanatory variables include wind speed, wind direction and temperatures in a given area. Overall, this problem accounts with over 35,000 samples.

As a way to test the value added with the trained models, they are compared with two benchmarking algorithms. The first benchmark is a simple one-day forecasting window using the value of the previous datapoint. It is a common proxy used in several industries such as the financial services and it is an easy way to set a target for any forecasting model. The second benchmark model is a simpler neural network, a multilayer perceptron.

The first round of experiments revolved around the Long-Short Term Memory architecture versus the Gated Recurrent Units architecture comparison.

The first experiment aimed at checking the behaviour of both algorithms with the same structure and testing out common alternatives of activation functions and optimization functions. ReLU, SELU, Tanh, maxout and sigmoid were used as activation functions. Adam and Nadam were used as optimization algorithms. All the networks had 3 layers and 20 neurons, that were picked as common initial values. 5 simulations were carried out for each model.

In the second experiment, the ELU activation function was included and the Nadam algorithm was discarded. The same network structure than the first experiment was kept. In this case, the number of simulations was 30 to make the results more statistically reliable with a larger sample.

In order to compare the best performing versions of the MLP benchmark, the GRU and the LSTM, a hyperparameter optimizer was used.

The third experiment consisted in observing how the models behave when facing an outlier. Under these circumstances some models learned how to avoid individual outliers. The tuned hyperparameters in this experiment were the model, activation function, number of layers, neurons per layer, learning rate and optimization algorithm. The optimization process used 200 iterations in each case.

The fourth experiment was a comparison between models with no outliers in the data. The objective was to fully compare the models with each other without the influence of the outliers, that were interpolated.

The fifth experiment consisted in observing the time response of the neural network architectures when a step is simulated in the output variable. The neural networks are capable of mimicking dynamics, and the step experiment is a way of showing what dynamics were learned.

In the final experiment, the one-dimensional convolutional neural network was explored. The main objective was to observe differences in the predictions compared with the most successful recurrent neural network. In this model the output is modelled using pattern recognition and extraction, so an improvement over the benchmark was expected beforehand.

Results

The GRU algorithm outperformed to LSTM algorithm in three out of the four tested cases in the first experiment, a 100% of cases in the second experiment, as well as in the optimization tests.

In the first experiment the ReLU activation function was outperformed by SELU. In the second ELU outperformed ReLU in all cases. The ReLU activation function was never the chose none for any Bayesian optimization except for the CNN.

The Convolutional Neural Network obtained the second lowest MSE score forthe whole experiment, also being the simplest and fastest model to run.

A sample of the results of the best performing models without outliers is shown:

Model	MSE	MAE
Benchmark	115.815	7.443
Optimized MLP	125.495	7.949
Optimized GRU	91.154	6.617
Optimized LSTM	109.571	7.356
Optimized CNN	98.204	6.902

Table 1. Summary table of the results of the project

Conclusions

Several conclusions can be outlined from the project.

Recurrent neural networks are a powerful tool to address timeseries forecasting problems. Their internal memory captures information from past timesteps and allowed them to beat both benchmarks. A prove that the models replicate the dynamic response of the ouput is the fact that they do not follow the permanent value of an arbitrary step in the validation dataset. Better dynamics is not the only quality in favour of RNNs, they can also learn how to avoid outliers.

When talking about RNN alternatives, the LSTM is usually the first one mentioned. However, in this project the GRU algorithm outperformed to LSTM algorithm in most cases.

Alternatively from RNNs, The optimized CNN proved to be a very lightweight model that was trained in under 10 seconds each iteration. It is the model with the second most accurate result and with the fastest training time.

On the activation functions side, the ReLU was surpassed by modified versions, such as the ELU activation function. In the first experiment the ReLU activation function was outperformed by SELU. In the second ELU outperformed ReLU in all cases. The ReLU activation function was never the chose none for any Bayesian optimization except for the CNN.

Resumen del proyecto

En este proyecto se ha realizado una comparación entre diversas arquitecturas de red neuronal para evaluar el efecto del tipo de arquitectura en un problema de predicción de generación de energía eólica. El trabajo incluye un estado del arte extenso sobre la materia y un caso práctico. Se concluye que la red neuronal GRU es la más adecuada, que la CNN se debe utilizar en sets de datos grandes y que existen alternativas eficaces a la función de activación ReLU.

Palabras Clave: Inteligencia artificial, GRU, CNN, LSTM

Introduction

El principal objetivo de este proyecto de fin de máster es crear una serie de recomendaciones sobre la selección de la arquitectura de red neuronal para un problema específico, centrando las conclusiones en problemas de predicción de series temporales. El proyecto incluye un extenso estado del arte donde se lleva a cabo una revisión de las tendencias más influyentes y de las redes neuronales más utilizadas.

En segunda parte del proyecto se seleccionan, diseñan y analizan redes neuronales para probar su eficacia en un caso de estudio real de generación de energía eólica.

Caso de estudio

El conjunto de datos a utilizar proviene de un concurso de machine learning y contiene valores de generación de viento con 90 variables diferentes. Las variables explicativas disponibles incluyen velocidad del viento, dirección del viento y temperaturas de la zona a analizar. En total, cuenta con más de 35.000 datos.

Como manera de comparar el valor aportado por los diferentes modelos de predicción, se utilizaron dos estimadores de referencia. El primer estimador de referencia es un retardo de un solo intervalo temporal del histórico de generación eléctrica. Es común usarlo para ver si el modelo es capaz de vencer a un predictor eficaz para todos disponible disponible. El segundo modelo de referencia será un perceptrón multicapa.

La primera ronda de experimentos tiene como objetivo comparar los modelos de red neuronal "Long-Short Term Memory" "Gated Recurrent Unit".

El primer experimento pretende observar el comportamiento de ambos algoritmos al testar alternativas comunes a la función de activación y a los algoritmos de optimización. Para las funciones de activación se exploraron la ReLU, SELU, tanh, maxout y sigmoide. Para los algoritmos de optimización se utilizaron Adam y Nadam. Todas las redes contaban con 3 capas y 20 neuronas, que se escogieron como valores iniciales. Se realizaron 5 simulaciones por modelo.

En el segundo experimento, se incluyó la función de activación ELU y el algoritmo de optimización Nadam fue descartado. Se mantuvo la misma estructura de arquitectura que en el primer test. En este caso, el número de simulaciones se aumentó a 30 para obtener un resultado más fiable estadísticamente. Para obtener y comparar las mejores versiones del MLP, la red GRU y la red LSTM, se utilizó un optimizador de hiperparámetros bayesiano.

El tercer experimento consistió en observar el comportamiento de los modelos ante la existencia de outliers. Bajo las circunstancias adecuadas, algunos modelos aprendieron a evitar outliers. Los hiperparámetros explorados en este experimento fueron el tipo de algoritmo, la función de activación, el número de capas, el número de neuronas por capa y la tasa de aprendizaje. Para cada modelo se utilizaron 200 iteraciones en el proceso de optimización.

El cuarto experimento fue una comparación entre los modelos sin outliers en el conjunto de datos. El objetivo final era comparar los resultados sin la influencia de los outliers, que fueron interpolados.

El quinto experimento consistió en analizar la respuesta temporal de los modelos cuando se introducía un escalón arbitrario en el output. Las redes neuronales son capaces de aprender dinámicas de comportamiento y en este experimento se buscaba explorar que dinámicas habían aprendido.

En el experimento final, se exploró el modelo de red convolucional unidimensional. El objetivo principal era comparar la calidad de sus predicciones en contraposición a la red neuronal recurrente con menor error. Estos modelos estiman la salida utilizando reconocimiento y extracción de patrones, por lo que una mejora sobre los resultados de referencia era de esperar.

Resultados

El algoritmo GRU superó al algoritmo LSTM en tres de cuatro ocasiones en el primer experimento, en el 100 % de los casos en el segundo experimento, así como nuevamente en la totalidad de los experimentos de optimización bayesiana.

En el primer experimento la función de activación ReLU tuvo un rendimiento menor que la función de activación SELU. En el segundo test ELU superó a la ReLU en todos los casos. La función de activación ReLU nunca fue la escogida tras un proceso de optimización de hiperparámetros bayesiano salvo en el caso de la CNN.

La red neuronal convolucional obtuvo el segundo menor resultado con respecto al error cuadrático medio en el global de los tests del proyecto. También fue el modelo con menor tiempo de entrenamiento.

A continuación se muestra un resumen de los mejores modelos sin outliers:

Model	MSE	MAE
Estimador de referencia	115.815	7.443
MLP optimizada	125.495	7.949
GRU optimizada	91.154	6.617
LSTM optimizada	109.571	7.356
CNN optimizada	98.204	6.902

Tabla 2. Tabla resumen de los resultados del proyecto

Conclusiones

Se pueden sacar varias conclusiones del trabajo realizado en este proyecto.

Las redes neuronales recurrentes son una herramienta eficaz para pronosticar series temporales. Su memoria interna captura información pasada y les permitió batir ambos modelos de referencia de este proyecto. Un síntoma de que pueden replicar la respuesta dinámica de la salida real es que no siguen el valor permanente de un escalón arbitrario cuando este se introduce en el conjunto de datos de validación. Replicar respuestas temporales de forma acertada no es la única ventaja de las redes neuronales recurrentes, también se ha podido observar como son capaces de detectar y evitar outliers.

A la hora de hablar de alternativas dentro de las redes neuronales recurrentes, el modelo LSTM es normalmente el primero en ser mencionado. Sin embargo, en este proyecto se ha observado como el modelo GRU superó en rendimiento a la LSTM en la práctica totalidad de casos.

A parte de las redes neuronales recurrentes, la red convolucional optimizada demostró ser un model muy ligero y preciso que entrenó en menos de 10 segundos en cada iteración. Fue el modelo con el segundo mejor resultado del proyecto y con el tiempo de entrenamiento más bajo.

Desde el punto de vista de funciones de activación, la función ReLU fue superada por la SELU en el primer experimento. En el segundo experimento, la ELU obtuvo mejor resultado que la ReLU en todos los casos. La función de activación ReLU no fue la escogida en los procesos de optimización de hiperparametros bayesianos, salvo en el caso de la CNN.

1

Introduction

The story so far: in the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move
Douglas Adams (1952–2001)

This first chapter introduces the rationale behind this thesis as well as its main objectives. In addition, it provides the reader with a general overview of the organization and the outline of the dissertation in order to make it easier to follow.

1.1. A Brief History of Machine Learning and Deep Learning

The history of Machine Learning is a history of exploration and continuous improvement. Machine Learning took its first steps when neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper called “A Logical Calculus of the Ideas Immanent in Nervous Activity” in 1943. It described the mathematical workings of a biological neuron. To support it, they created a model using an electrical circuit and the first artificial neuron was born.

The first perceptron model appeared in 1957, designed by Frank Rosenblatt in his paper “The Perceptron: A Perceiving and Recognizing Automaton”. The first backpropagation model is dated from 1960. The multilayer perceptron appeared in 1965 and the first deep neural network was invented by Alexey Grigoryevich in 1971. Most theoretical bases of machine learning and deep learning are not novel, they have been known and studied for decades. Deep learning is the branch of machine learning that involves artificial neural networks for representation learning. It can be supervised, semi-supervised or unsupervised. It can achieve better results than other families inside machine learning, but it requires more computational power, time and data.

Most theoretical bases of machine learning and deep learning are not novel, they have been known and studied for decades. However, most deep learning real life applications and

world changing uses have occurred since 2008. This was due to the advent of the age of information with its impact on the availability and explosion of data, plus the exceptional increase of computing power at an affordable price and unprecedented scale. Great success stories and billion-dollar companies have followed. In 2014 Facebooks DeepFace technology enabled face identification with human-like accuracy. In 2016, Alpha Go defeated the 9 dan professional Go player Lee Sedol. In 2020, 7 out of the top 10 most valuable companies in the world are technological companies that use deep learning in their services or for their operations [Pra19].

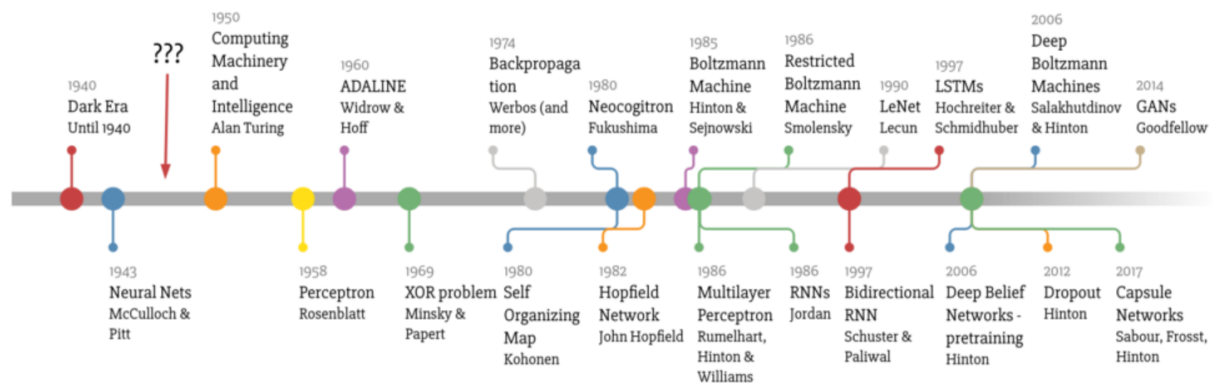


Figure 1.1. Deep Learning timeline.

1.2. Objectives and scope

The main purpose of this final thesis is to come up with a series of recommendations to select the right architecture for a neural network for an specific problem, focusing on timeseries forecasting problems. The project involves an extensive state-of-the-art section in which a review of current and past trends is carried out in order to explore the most popular deep neural network architectures. The second part of the project involved selecting, designing and analyzing networks that were tested using a real data set from a wind energy forecasting competition containing wind power generation data. The models will be created using public access machine learning and deep learning environments such as TensorFlow and coded in Python.

1.3. Basic Concepts

In order to fully grasp the systems that will be introduced in the state-of-the-art section, the reader needs to be familiar with some basic vocabulary.

- Supervised learning: it is a machine learning paradigm in which the desired answer for the algorithms is known. This is so by having a labeled data set that serves as ground truth. The algorithm is trained and rated depending on how distant its predictions come when compared with the ground truth.
- Unsupervised learning: contrary to the previous definition, this type of machine learning problem does not have a labelled data set. The inputs are given to the algorithm without any additional information on how the output should look like. The objective of the fitting process is to find meaningful ways to explore the data.

- **Sample:** single row of data which is fed into an algorithm. Databases are made up of many samples, this is, many rows of data.
- **Hyperparameter:** it is a variable that is tuned before the learning process starts. It defines the fundamental architecture of the system. It affects the behaviour of the network as it can be the size, the activation function, the optimization function used as well as many others case-specific.
- **Batch:** hyperparameter that defines the number of samples processed by an algorithm before the internal model parameters are updated.
- **Epoch:** it is an hyper-parameter that defines the number of times that the learning algorithms will work through the training dataset.

2

State of the art

*If I have seen further than others,
it is by standing upon
the shoulders of giants.*
Sir Isaac Newton (1642–1727)

The purpose of the following sections is to introduce some of the most common neural network architectures, exposing their inner workings, most common problems and giving an overview of its most extended applications in the fields of machine learning and deep learning.

2.1. Neural Architectures

In this part of the thesis a detailed summary of the most prominent neural network families is presented, focusing on those with an application in forecasting solutions. These include multilayer perceptrons, convolutional neural networks, several variations of recurrent neural networks and auto-encoders.

2.1.1. Multilayer Perceptron

This was the first neural network model developed. As the name suggests, it is composed of different layers of perceptrons. A perceptron is a model for supervised learning, this means the model is given pairs of input-output couples in the training process, of binary classifiers. The first prototype was developed at the Cornell Aeronautical Laboratory in 1957 [Jus17]. It is inspired in the way biological neurons work. The neuron is given a set of training data that allows it to adjust the inner parameters to find the best fit possible.

Within the perceptron we can distinguish:

- **Input:** available information for the perceptron to create a map (set of solutions) and solve the supervised learning problem. There is a special type of input called the bias. In Figure 2.1 the inputs are represented by x_i and the weight associated to the bias by w_0 .

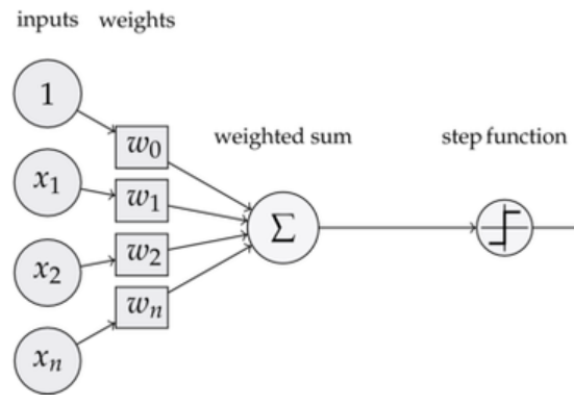


Figure 2.1. Neuron Diagram.

- **Weights:** values that are updated in iteration during the training process. They are responsible for the fitting process. They are represented by $[w_1, w_2, w_3 \dots]$.
- **Bias:** allows to shift the origin of the activation function either to more negative or to more positive values. It enables an extra degree of freedom, making more precise fits possible during the training process.
- **Weighted sum:** it is the result you get after the multiplication of each weight by each feature value. Result of summing the products of the input values by the weight of their corresponding connection.
- **Activation function:** mathematical model that determines how much the output is activated depending on the value received from the weighted sum. There are many different options depending on the application, from a simple step function, to a sigmoid, a linear or a ReLU, that will be further explained in Section 2.2.
- **Output:** result of passing the weighted sum value to the activation function.

$$Output = f\left(\sum_{i=0}^n w_i * Input_i\right) \quad (2.1)$$

A multilayer perceptron is made up of an input layer, hidden layers and output layer. Each layer is composed by a previously fixed number of neurons. In a classic multilinear perceptron, the hidden layer nodes are interconnected with the outputs of the input layer neurons. A hidden layer consists of neuron nodes stacked in that do not directly connect with inputs and outputs. It enables the model with extra parameters that allow for fitting in more complex solutions. Consequently, it increases the training time of the model due to the rise in computation requirements. The dimension of the input layer depends on the dimension of the training data. The number of neurons in the hidden layer is one of the hyper-parameters of the model and should be decided by the user. The dimension of the output layer depends on the application.

In order to understand the training process of a multilayer perceptron we need to introduce the concept of backpropagation [Leo18]. It is a machine learning training iterative algorithm that consists of adjusting the weights of the model so as to minimize the difference between actual output and desired output. The weights of the network are updated using gradient descent computing the derivative of the error with respect to the weights. It starts in the last

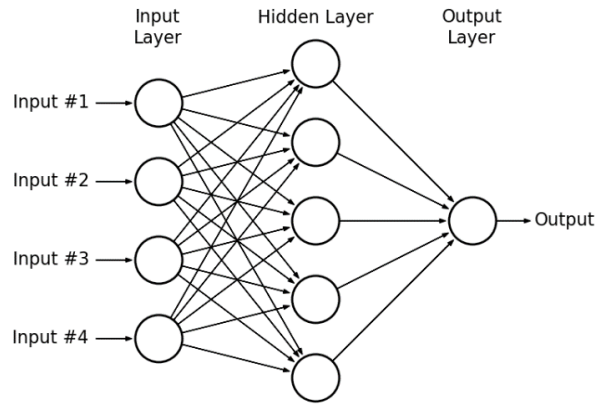


Figure 2.2. Multilayer Perceptron Diagram.

layer of the model and partial computations of the gradient from one layer are reused in the computation of the gradient for the previous layer. This backwards flow of the error information allows for efficient computation of the gradient at each layer versus the naive approach of calculating the gradient of each layer separately [MSW20].

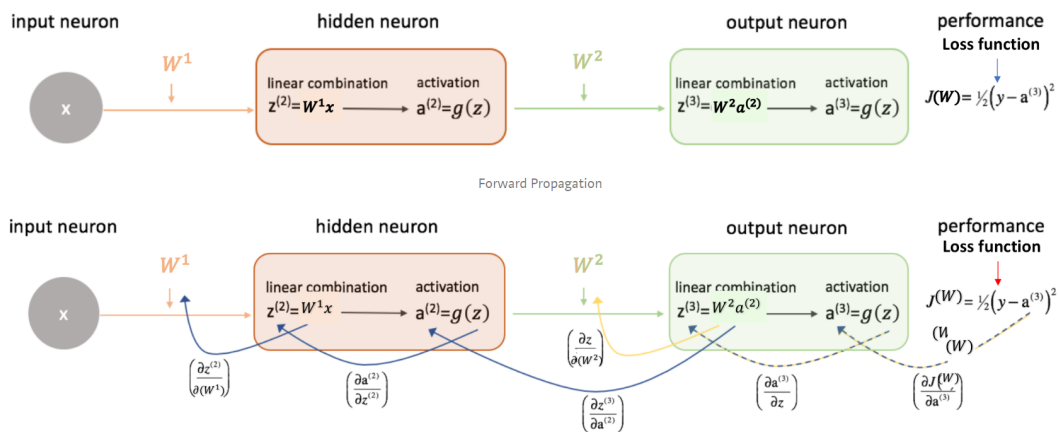


Figure 2.3. Steps in Backpropagation [Dos19a].

The multilayer perceptron algorithm can be tuned in several ways depending on the application. It is common practice to use different activation functions (see Section 2.2) for the output depending on the machine learning problems addressed. The number of hidden layers is also dependent of the application.

2.1.2. Convolutional Neural Networks

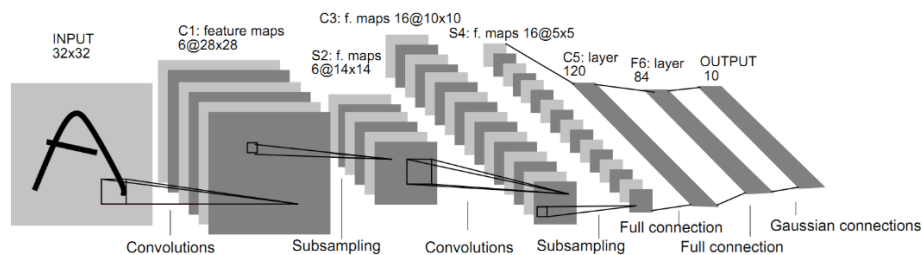


Figure 2.4. Convolutional Neural Network LeNet-5.

Convolutional neural networks are a class of deep, feed-forward artificial neural networks that are mainly applied to analyzing visual imagery (commonly known as machine vision).

These systems are inspired by the organization of the human Visual Cortex [Ste19a]. One main task of CNN's is to reduce the images into a form which is easier to process, without losing critical information to obtain a good prediction. This is referred as feature extraction or feature recognition. A usual requirement in CNN's is that they must be computationally viable in massive data sets.

CNN's can be broadly divided into two parts, the feature recognition part and the classification part. The classification part is usually similar to a multilayer perceptron. It connects the detected visual features to the desired output.

When dealing with image applications it is common to divide the images into the three RGB colour channels.

Some common types of layers that are present in convolutional neural networks are:

- Convolutional layers: a filter or Kernel is applied to a matrix or 2D input. This Kernel is a moving window and computes calculations that result in another matrix, usually of a smaller size, although a bigger size or the same size can also be obtained. The objective of the convolution operation is to extract high-level features such as edges from the input image. Several features can be applied simultaneously [Ste19b].

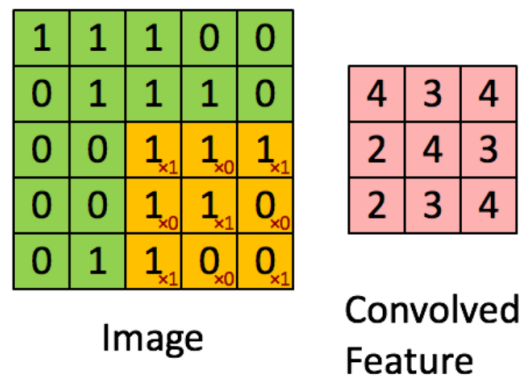


Figure 2.5. Visual representation of a convolution [Ste19b].

- Pooling layer: the result of a convolutional layer is an activation map. The objective of a pooling layer is to provide a non-linear downsampling for those activation maps. It is considered an aggressive operation as it can discard useful information. The two most popular techniques are max pooling, which takes the maximum value of a region and pass it over to the next layer, and average pooling, which instead computes the average. Max pooling is used to removed noise from the inputs, as the average pooling is affected by all the values of the input matrix. The pooling layer introduces zero parameters as it computes a fixed function of the input.
- Flattened Layer: the purpose of this part of the network is to condense the information into a 1-dimensional array to pass it to the final stage of the Convolutional Neural Network, the Fully Connected Layer. As an ordinary artificial neural network, flattening the data is a requirement given the underlying architecture of training with one array at a time [Jeo19]. There is no information loss at this step, it is only a reshaping of the data structure.

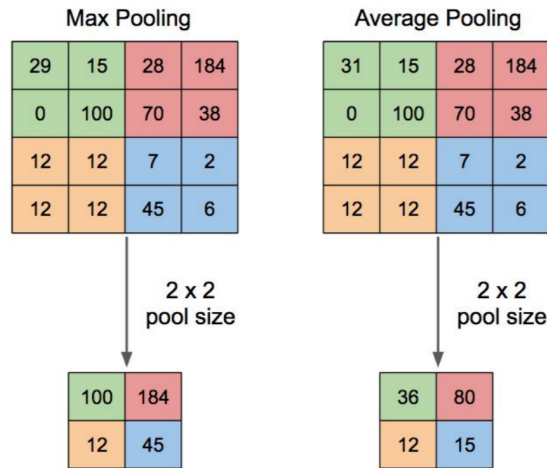


Figure 2.6. Common Pooling techniques [Ste19b].

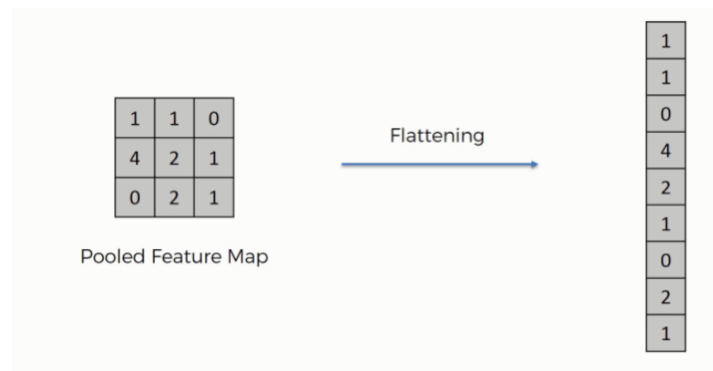


Figure 2.7. Flattening step example in a Convolutional Neural Network [Ano18a].

- **Fully Connected Layer:** It receives the flattened image as a column vector. This flattened output is fed to a feed-forward neural network. Over the training process, this part of the network learns to distinguish between dominating and low-level features. The common output is a vector, which is commonly passed through a softmax activation function to represent the confidence of classification.

Convolutional neural networks are usually trained with very large datasets. In order to avoid overfitting, several techniques are commonly used, such as dropout, batch normalization, gradient clipping, and max-norm constraint (see Section 2.3).

2.1.3. Recurrent Neural Networks

Recurrent neural networks are part of the family of feedforward neural networks. However, they can send information over time-steps. Recurrent neural networks are considered Turing complete and can simulate arbitrary programs, i.e., they can store information of previous steps to modify its behaviour in the future. This means they can compute everything that could be desired to be computed [De 18]. Most of the programming languages are also Turing complete.

Recurrent neural networks are a special kind inside neural networks, as they are usually referred to as systems with memory. This is due to the way they compute their output, using not only the inputs of each sample in isolation, but also some internal variables (the number and

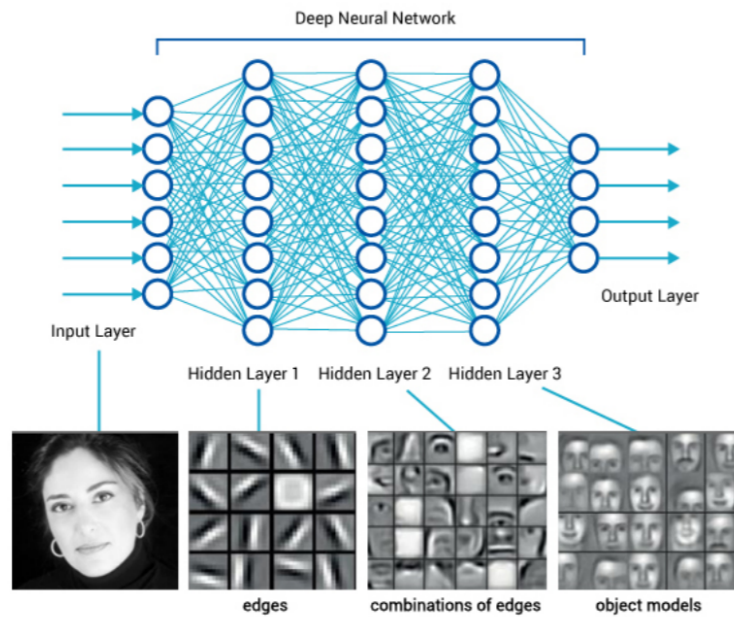


Figure 2.8. Convolutional Neural Networks Layers Connection [Ara18].

characteristics depends on the RNN type) that are influenced by the values of the inputs of previous timesteps.

In order to visualize a RNN it is good practice to first observe the perpendicular vision of an MLP presented in Figure 2.2, comparing it with the “flattened” version shown below in Figure 2.9. In the last example, we can imagine one of the two perpendicular directions as normal to our vision plain. The loops represent the recurrent connections.

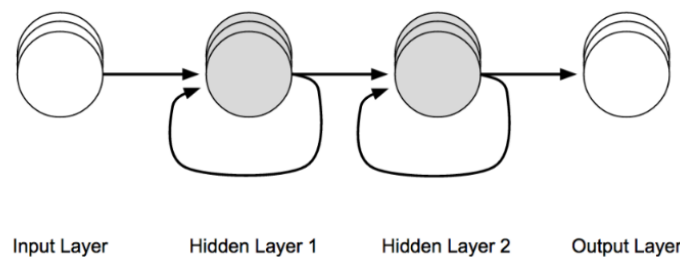


Figure 2.9. Flattened version of a Recurrent Neural Network [Sal97].

In the next step we add the time variable, untangling the recurrent connections and the result is the scheme presented in Figure 2.10.

Recurrent neural networks commonly use backpropagation through time for the training process. Backpropagation through time is fundamentally the same idea as standard backpropagation, a chain rule is applied to calculate the derivatives based on the connection structure of the network. However, the loss is calculated in a forward motion before calculating the gradient. It can be computationally expensive. Alternatively, truncated backpropagation through time can also be used. In this algorithm, the sequence is split into parts, running forward and backwards through slices of the sequence instead of the whole one. This implies a smaller computational burden.

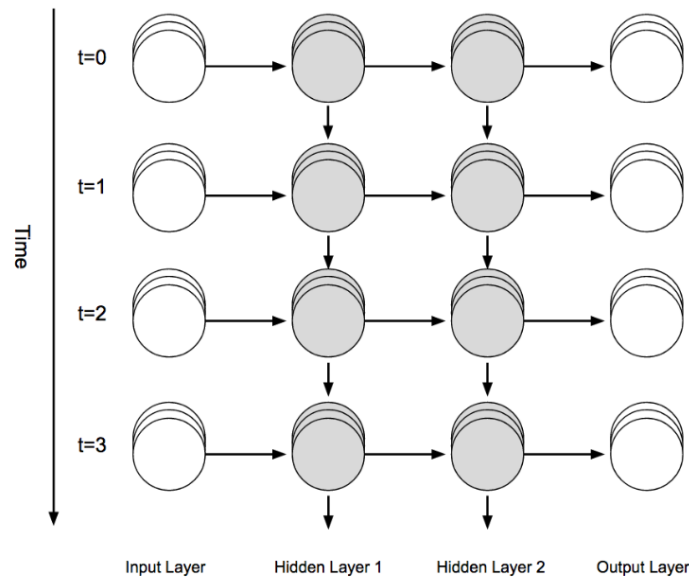


Figure 2.10. Recurrent Neural Network Diagram untangled in time [PG17].

One characteristic feature of the recurrent neural networks is that they accept several input and output data format, depending on its configuration. In the following classification, the first word refers to the length of the input and the last one to the length of the output:

- One to one: called the vanilla recurrent neural network. It consists in just one single hidden vector.

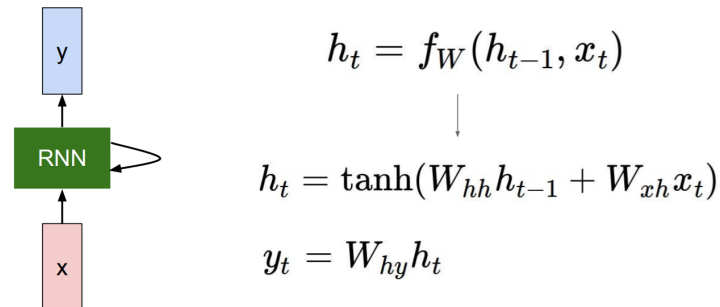


Figure 2.11. Vanilla RNN and equations [LJY17].

- One to many: applications such as image captioning. It takes an image and outputs a set of words describing it.
- Many to one: designed for sequence inputs. Useful for sentiment classification [PGN18].
- Many to many: it can be applied to video classification where you label each frame. Some architectures are suitable for machine translation tasks [LGS16].

There are several variants of recurrent neural network that will be further explained. What changes between every RNN variation is the internal cell structure:

2.1.3.1. Vanilla Recurrent Neural Networks

As shown in Figure 2.12, the simple RNN cell (or Vanilla RNN) is a basic model in which there is a multiplication of the input by the previous output. In this particular example, it is later passed through a hyperbolic tangent activation function [Rat18].

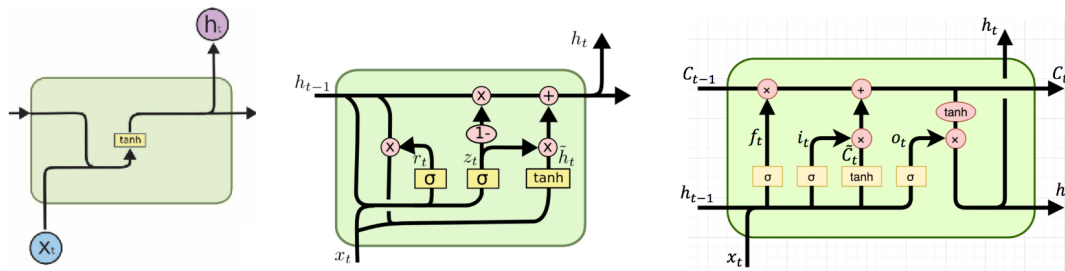


Figure 2.12. Basic architectures of RNN, GRU and LSTM cells [Rat18].

Vanilla RNN are a prime example of a model affected by the two most prominent problems related to the training phase of recurrent neural networks:

- **Vanishing gradients:** In the process of training the recurrent neural network, the weights are updated using the values of the gradient and multiplying it to a learning rate, that is usually a small number, between 0,01 and 0,001. The gradient is calculated in a chain, being the gradient for the first weights of the network the result of the multiplication of all the previous partial derivatives later in the network. The vanishing gradient problem takes place when the derivatives are smaller than one, and as a result the update speed of the weights turns very slow. This problem can grow in every iteration and because of the chain calculations it more severely impacts the early weights in the network. This issue can slow the training process to a point where after many iterations the weights are no where near the optimum value. Computationally it could make the computer crash if it surpasses the number of decimal points available. Common tactics to solve this problem in recurrent neural networks are truncated backpropagation and gradient clipping [Ano18b].
- **Exploding gradients:** this phenomenon shares the same nature as the vanishing gradients issue. Due to the chain calculations, if several partial derivatives turn out to be higher than one, the first weights of the network will have a large gradient which will update its value. If this situation is repeated many times during the training process you may end up with a situation in which during each iteration the weight is modified in great way, resulting in a value that distances itself more and more from the optimum. Same as in the vanishing gradients, it could make the computer crash if the values are greater than the maximum the datatype supports in that memory spot. Common tactics to deal with this problem include weight initialization or using echo state networks [Bro19].

Regarding these two problems, LSTMs have proven like an effective way of successfully solving them due to their cell state feature that is constantly being updated and regulated so these two phenomena do not hurt the performance.

2.1.3.2. Long-Short Term Memory Recurrent Neural Networks

The Long-Short Term Memory cell is exposed in Figure 2.13 [Ngu19]. The model was first proposed in 1997 by Stepp Hochreiter and Jürgen Schmidhuber. The LSTM cell has what is called a cell state. This is usually referred as the memory of the cell and it passes across the cell (represented as the top way). It allows information from earlier timesteps to affect the output, reducing the effect of the short-term memory. It also has a hidden state that comes from previous timesteps and that is represented with the letter h in the diagram. As the cell state goes through the cell, information is added or removed from it by the gates.

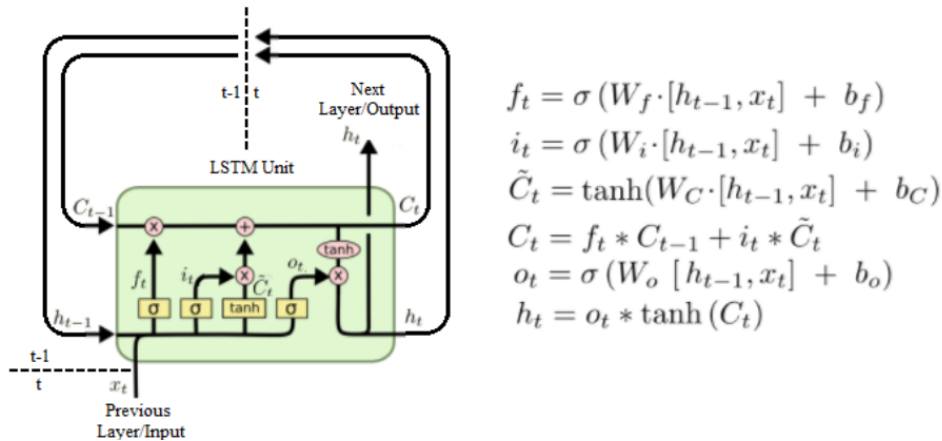


Figure 2.13. LSTM cell model and equations [Rat18].

The gates are different neural networks that decide which information is allowed on the cell state. LSTM cells have a forget gate (first equation), an input gate (second and third equations) and an output gate (fifth equation). The forget gate decides whether if information from the hidden state and from the previous gate should be kept or thrown away. It uses a sigmoid function, multiplying this information by a number from 0 to 1. The input gate decides how relevant information from the current step is, it will transform it into a number between 0 and 1 (sigmoid) and multiply it for a number between -1 and 1 to regulate it (tanh). The output gate decides what the next hidden state should be. The hidden state is also used for predictions. It uses information from the previous hidden state, the input and the cell state.

2.1.3.3. Gated Recurrent Units Neural Networks

Next variation is a Gated Recurrent Unit (GRU). This was a posterior development to the LSTM. It was proposed by Kyunghyun Cho in 2014 and it intended to be similarly powerful but lighter than LSTM cells. It is considered a valid alternative to LSTMs as it is more powerful than vanilla RNNs and it is comparatively lighter to train. The GRU cell has no cell state, and the gates directly modify the hidden state and use it to pass information. GRU cells have two gates: the reset gate (second equation) and the update gate (first equation). The update gate works in a similar way as the input gate in an LSTM. It decides what information to keep or discard. The reset gate decides how much past information to forget.

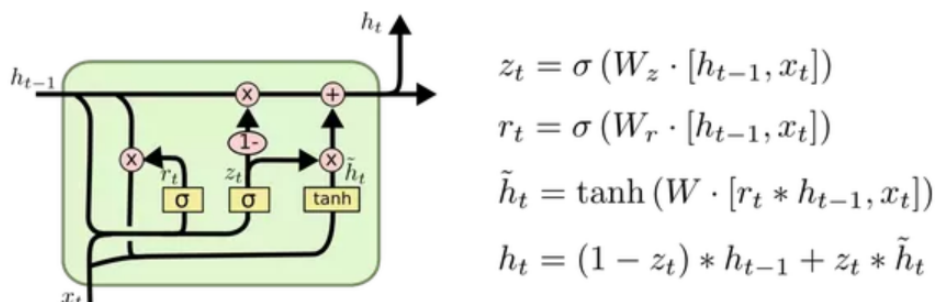


Figure 2.14. GRU cell model and equations [Rat18].

The more complex the individual cell becomes, the more matrixes are involved in solving its equations and the heavier the computational needs become. Vanilla RNNs are simple but they struggle with passing information after a few timesteps due to variability in the gradients (vanishing gradients and exploding gradients problems explained above). Both GRU and LSTM

are widely used nowadays. GRU is significantly lighter to compute and performs better in some small size datasets [Chu+14]. LSTM networks need to train numerous parameters, but they are usually more robust than the other variants.

2.1.3.4. Bi-directional RNNs and Bi-directional LSTMs

These networks explicitly model a simple idea: just as the future can be predicted from the past, the past can be deduced from the future. Bi-directional RNN structures are models in which the same input is introduced in order and in reverse to different first layers of the neural network. Then, the final output of the model is computed using the states of both normal and reverse parts of the network [Raj19].

In a Bidirectional LSTM the building blocks of the network are LSTM memory cells. Deep BLSTM networks are obtained by stacking layers in between the input and the output. In BRNN, backpropagation through time must be computed separately on both the forward and the backward networks.

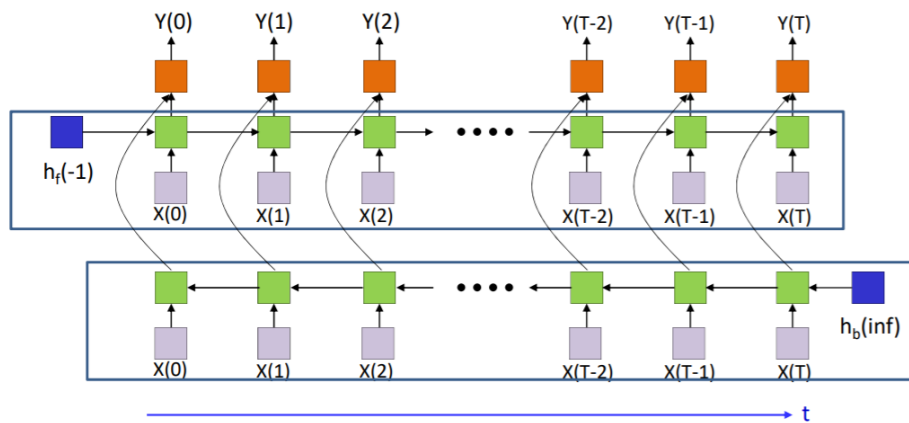


Figure 2.15. Simple bidirectional RNN network [Raj19].

To summarize, recurrent neural networks are very powerful for machine learning applications due to their characteristics:

- Distributed hidden state that allows them to store information about the past efficiently.
- Non-linear dynamics that allow them to update their hidden state in complicated ways in order to adapt their predictions to the dynamic of the real output.

They could potentially learn to implement many different small-size programs that each capture a nugget of knowledge and run them in parallel, interacting to produce very complex effects. However, on the minus, side the computational requirements are significantly higher compared to a simple multilayer perceptron.

Additionally, we can find other types of recurrent neural networks such as:

2.1.3.5. Hopfield Networks

Defined as a “loopy binary network with symmetric connections” [Raj19], they are neural networks that can mimic the inner workings of the human memory. These networks are made up by nodes and connections between the nodes. Each node is updated by a calculation that involves the value of the node (typically 1 or -1) and the values of the connections to the

surrounding nodes multiplied by the values of the surrounded nodes. If the result is contrary to the sign of the node value (threshold), the value of that node is flipped (now it has the opposite value). As a result, the “energy” of the network is modified. The energy of the network is a property of Hopfield Networks that can be calculated taking into consideration all nodes and all connections [Den97].

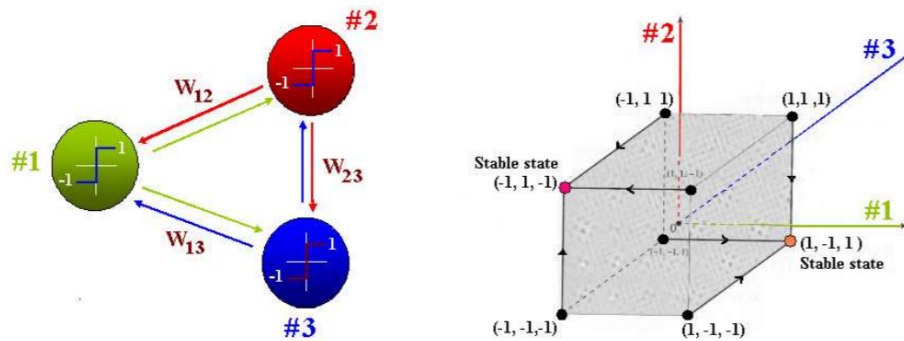


Figure 2.16. 3 neuron Hopfield Network with 8 possible states, two stable [Raj19].

The nodes are updated either in a synchronous or in a random way and the updates lead to a smaller energy value. This way, the Hopfield Networks are guaranteed to reach a local minimum, although not the global minimum. Instead of -1 and 1, it is common for the Hopfield Networks nodes to have an activation function, such as tanh. These types of networks are appropriate to create content addressable memory systems for very high-speed searching applications (they can recover storage information from partial or corrupt inputs) [Raj19]. There is a theoretical limit (by using the common technique known as Hebbian Learning) of how many “patterns” or pieces of information we can store with less than a 0,4% chance of failure in a HN: $0.14 \times N$ being N the number of nodes in the network. These networks can also be used to construct interpretations of sensory input and reconstruct information from degraded input [Bhi17].

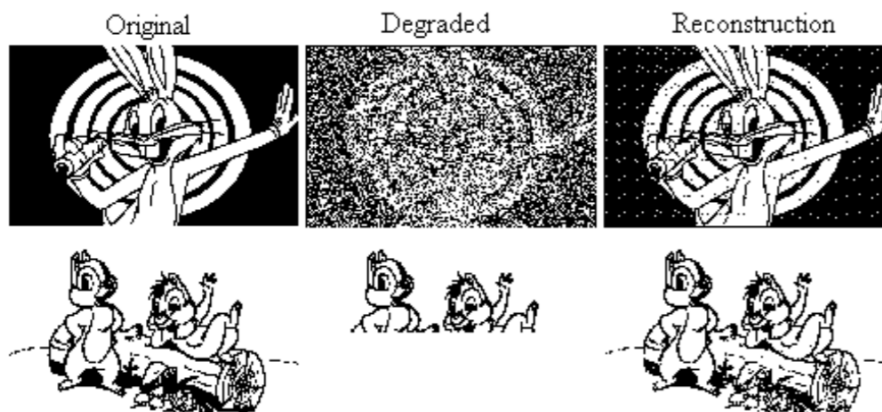


Figure 2.17. Hopfield network acting on degraded images from noisy or partial inputs [Lev02].

2.1.3.6. Boltzmann Machine

They appeared after investigating ways of improving the Hopfield Networks. Using several optimization techniques, a Hopfield Network can ensure N stable patterns stored for an N -size network [19]. A Boltzmann Machine is a Hopfield Network that has $N + K$ neurons, where N are the visible neurons (the ones that will store the actual patterns of interest) and K are

hidden neurons (useful to increase the capacity of the network but whose actual values are not important) [19]. Boltzmann Machines are stochastic recurrent neural networks and Markov random field [Pra19]. The value of each node is not deterministic but based on a probability distribution function. In a Boltzmann Machine the probability of generating a “visible” vector (the desired output) is defined in terms of the energies or joint configurations of the visible and hidden units. Another way of saying this, the energy of a pattern is the analogue of the negative log-likelihood of a Boltzmann distribution (a Boltzmann distribution is the probability distribution that a system will be in a certain state as a function of energy and temperature, for this model $T=1$).

$$p(v, h) \propto \exp -E(v, h) \quad (2.2)$$

$$-E(\mathbf{v}, \mathbf{h}) = \sum_{i \in \text{vis}} v_i b_i + \sum_{k \in \text{hid}} h_k b_k + \sum_{i < j} v_i v_j w_{ij} + \sum_{i, k} v_i h_k w_{ik} + \sum_{k < l} h_k h_l w_{kl}$$

binary state of unit i in \mathbf{v}
bias of unit k

Energy with configuration \mathbf{v} on the visible units and \mathbf{h} on the hidden units
indexes every non-identical pair of i and j once
weight between visible unit i and hidden unit k

Figure 2.18. Energy equation of a Boltzmann Machine [19].

The Markov Chain Monte Carlo (MCMC) is used to get samples from the model starting from a random global configuration [Sha17]. The MCMC algorithm is used to estimate a posterior probability distribution for the neuron states. The visible units are clamped to the given data vector, only hidden units are allowed to change states. Each hidden configuration is an “explanation” of a visible configuration that has been observed. The better the explanation, the lower the energy [Den97].

Apart from filling out patterns, denoising patterns and computing conditional probabilities of patterns, Boltzmann machines are also used for classification problems [AMR14]. Boltzmann machines are limited due to an extensive training time and therefore its limited to small problems [Her18]. To solve these issues, the alternatives are Restricted Boltzmann machines [Say18], Deep Boltzmann Machines [Kha18] and the Helmholtz machines [Kir06].

2.1.3.7. Deep Belief Network

Deep Belief Networks are a generative graphical model. Generative means that it not only focusses on the distribution of our output variable given our input variables, but that it can also learn the distribution of the inputs [Póc17]. Back-propagation is considered the standard method in artificial neural networks to calculate the error contribution of each neuron after a batch of data is processed. However, it comes with its own problems and limitations and Deep Belief Networks were a proposed solution for it. The main problems of back propagation are that it requires labelled data (when many data sources are unlabelled), the learning time does not scale well when you had multiple hidden layers and the training process can stop in a poor local minimum.

Deep Belief Networks contain many layers of hidden variables. Each layer captures high-order correlations between the activities of hidden features in the layer immediately below, performing a de facto feature extraction. The superior two layers of DBNs form a restricted Boltzmann Machine. The lower layers form a directed sigmoid belief network [Gan+15].

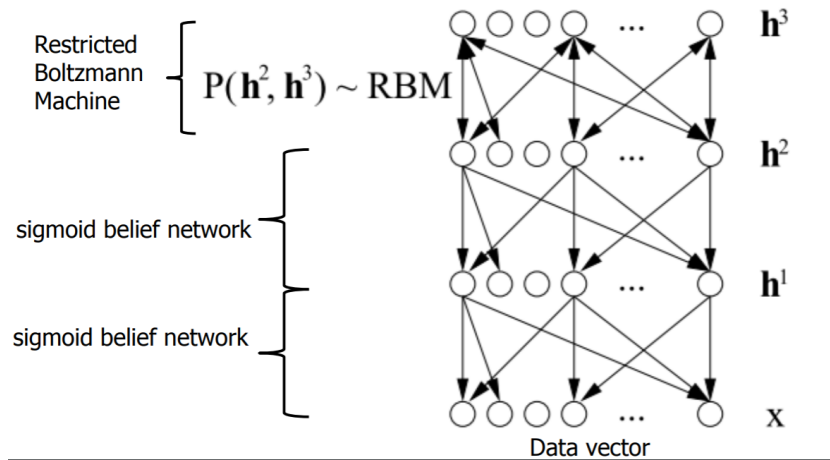


Figure 2.19. Representation of a Deep Believe Network [Póc17].

During the training process in a Deep Belief Network it is common to pre-train each layer with an unsupervised learning algorithm, sequentially across layers, starting in the first layer [Póc17]. This pretraining method allows for better results, as it performs a feature extraction method to the input data in an unsupervised way and facilitates the supervised training of the model. After having initialized these layers, the neural network can be used for a supervised training process as usual. The initialization process is called greedy layer-wise unsupervised leaning. It substitutes the random initialization of multi-layer networks.

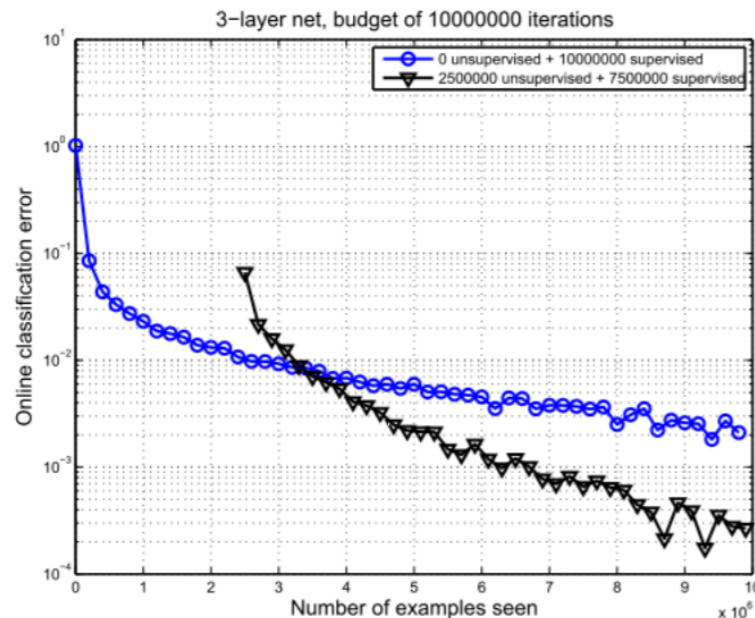


Figure 2.20. Comparisson between pre-trained or not pretrained DBN [Póc17].

In Figure 2.20, we can see the comparative results from a scientific paper in which they trained a deep belief network with and without pretraining it. In the black dotted line, a quarter of the observations where used for this purpose. From a data management standpoint, training

the first layers in an unsupervised way requires less labeled data, what is advantageous and opens the use of Deep Belief Networks to more applications.

Deep Belief Networks face two main problems:

- The inference problem: how to infer the states of the unobserved variables.
- The learning problem: how to adjust the interactions between variables to make the network more likely to generate the training data.

Since Deep Belief Networks are trained in an unsupervised manner, typical problems that affect other neural network structures, such as vanishing and exploding gradients, are not an issue.

2.1.4. Deep Auto-encoders

An autoencoder is a type of neural network designed to learn data codings in an unsupervised manner. The input is the same as the output. Autoencoders are able to compress the input into a lower dimensional code and then reconstruct the output from this representation [Der17].

An autoencoder consists on three different parts, an encoder, a code and a decoder. The encoder processes the inputs and produces the code, while the decoder uses the code to reconstruct the input. In order for an autoencoder to run we need a loss function comparing the output with the target. Autoencoders are trained the same way as other feedforward neural networks, using backpropagation.

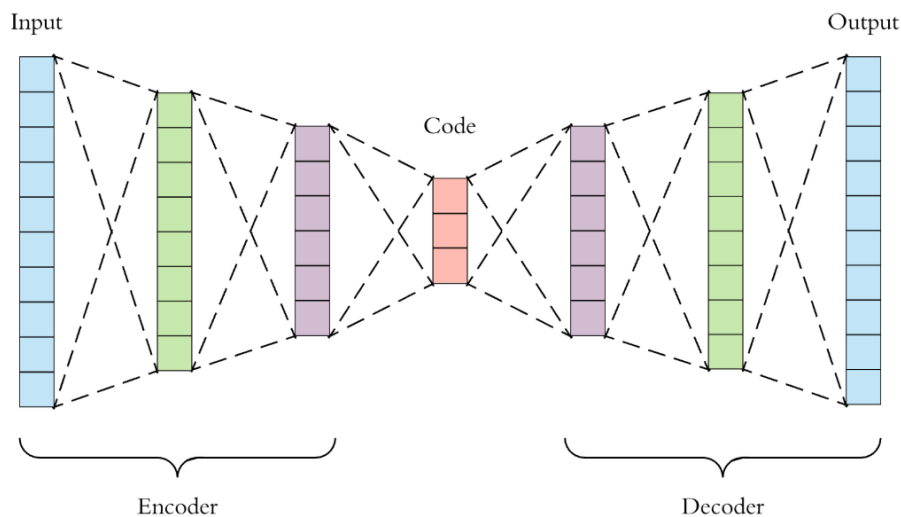


Figure 2.21. Deep Autoencoder Structures [Le15] .

Deep autoencoders consist on several layers in the encoder and decoder part (Figure 2.21 shows two hidden layers in each case). It is important to mention that decoders are usually a mirror image of the encoders, although it does not need to be that case necessarily.

Hyperparameters of the model are the nodes per layer, number of layers and the code size [Le15]. However, making the structure of the autoencoder too extense can sometimes lead to overfitting. It is a common practice to leave the code size small, that way the algorithm will be forced to learn smart ways of compressing the data and overfitting will be prevented.

Autoencoders are deliberately incomplete. They miss some information in order to be lightweight and useful. They should not be used if you have an application that needs perfect compression. Autoencoders are common in dimensionality reduction problems and data denoising [WYZ15].

The decoder part of the autoencoder can be used on itself for generative purposes once it is already trained. By exciting the layer connected to the code you can generate outputs related to what the autoencoder was coding. This approach is known as dictionary-based techniques [Bhi11].

There is a more complex version of autoencoders called Variational Autoencoders [Sch19]. It has a probabilistic approach, with a probabilistic encoder and decoder. It assumes the input variables follow a probabilistic distribution and it parametrizes those distribution in order to achieve the conditional distribution of the output. It opens new applications compared with vanilla autoencoders. They have been used to generate images of fictional celebrities and high-resolution digital artwork [Ha16].

All the previous neural network models have inside them a key component that affects the way they learn and behave: the activation function.

2.2. Activation functions

Activation functions play an essential role in an artificial neural network. They are the mathematical equivalent of the electrical potential that builds up in biological neurons which then fire when a given threshold is reached. Its job is determining the magnitude of the output after receiving the inputs of the neurons multiplied by the weights as an argument (potentially internal variables too, as in recurrent neural networks).

An important feature in activation functions is non-linearity. Otherwise, our neural network will compute regular linear functions resulting in a polynomial fit. The purpose of using neural networks is to take advantage of the universal approximation property derived from its non-linearity [McN17]. However, in some cases a linear activation function can be used, such as in a regression or forecasting problem for the last layer.

Another fundamental property that an activation function should have is differentiability. It is a necessary condition to calculate the gradients of the error with respect to the weights while performing backpropagation [Ste19a].

As it is the case with neural networks in general, activation functions are a topic of heavy research. In this section a selection of the most used ones and the most promising ones will be presented.

2.2.1. Sigmoid

The sigmoid activation function is a typical choice for applications in which a probability prediction is desired [RZL17]. This function is capped between 0 and 1 and it is monotonic. However, its derivative is not monotonic and that can cause problems in the training process. Other problems in the training phase include being affected by the vanishing gradients problem and having a slow convergence (the derivative in the extremes of the functions is remarkably flat) [Ste19a].

$$f(x) = \frac{1}{1 + \exp -x} \quad (2.3)$$

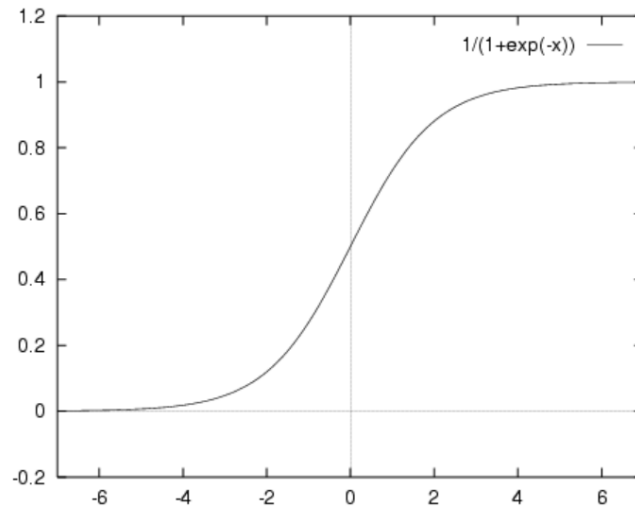


Figure 2.22. Sigmoid or Logistic Function [Mal18].

2.2.2. Softmax

The softmax functions turns numeric values into a probability that add to one. Given this property, it is common practice to use a softmax in the last layer of a multiple classification problem [Uni20].

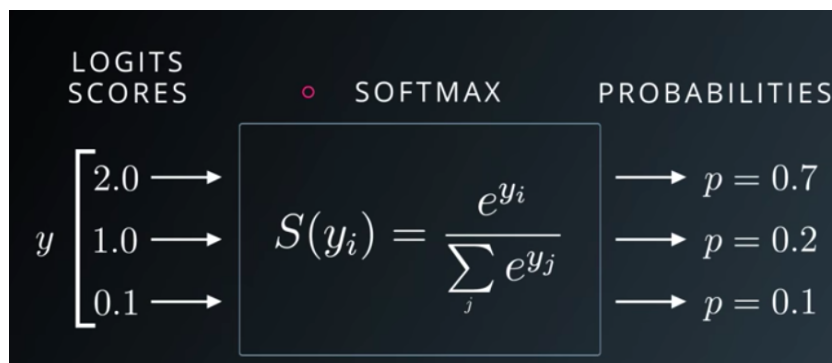


Figure 2.23. Softmax transform [Uni20].

2.2.3. Linear

Linear activation functions are of limited application for the reasons explained in previous paragraphs of this section. Additionally, it is important to notice that the derivative of a linear function is a constant and therefore, gradient descent is impeded for a neural network with only this activation function in all its layers (in practice as long as a layer of linear activation functions is preceded by a non-linear layer there are ways of applying gradient descent learning [V17]). It is commonly used in some cases, for example, it can be an option in the last layer of a regression or forecasting problem.

$$f(x) = x \quad (2.4)$$

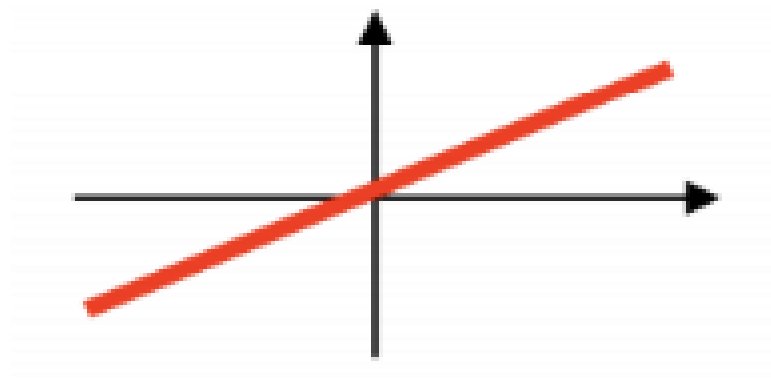


Figure 2.24. Representation of a linear activation function [Gup20].

2.2.4. Hyperbolic tangent (tanh)

In this activation function, there is a useful characteristic to highlight in comparison to the sigmoid function, it is zero centered. The gradient is also steeper compared to a sigmoid activation function, making it better for training. Still, it has issues with the derivative, such as the vanishing gradient problem [V17].

$$f(x) = \tanh(x) \quad (2.5)$$

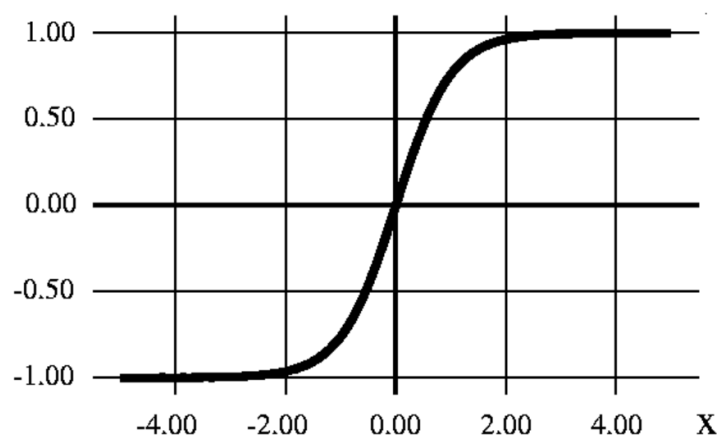


Figure 2.25. Hyperbolic tangent function [V17].

2.2.5. Rectified Linear Unit (ReLU)

The rectified linear unit is one of the simplest non-linear functions. It returns zero as an output if the input value is below zero. It returns the input when the input is above zero [Ste19a]. It is computationally lightweight, and it is very extended for deep learning applications, specially when applied to the hidden layers.

The flat region of the ReLU has a negative impact on the training process of our network. As the derivative in that region is also zero, through gradient descent some neurons can be

stopped from updating and end up being underused. These are commonly referred to as “dead neurons”, and result in an excessively large and underperforming network. Several activation functions have been developed to surpass the main issues affecting the rectified linear unit [Dan17].

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \quad (2.6)$$

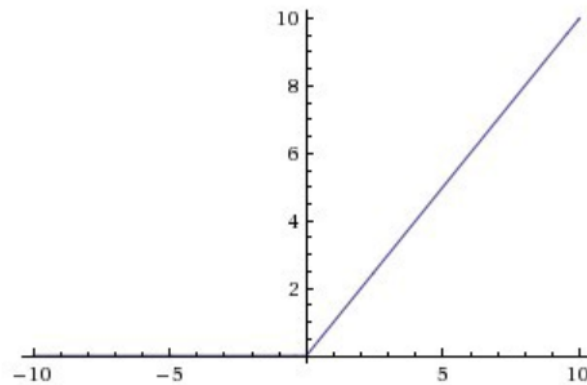


Figure 2.26. ReLU Function [Mem16] .

2.2.6. Softplus

The softplus activation function is very similar to a ReLU, but it presents two distinct differences that are advantageous. First is the soft transition close to the origin, that avoids a discontinuity. Second is that the region below zero is no longer flat, avoiding the dead neurons issue. However, the equation involves a logarithm and an exponential, so it is more computationally costly than the ReLU.

$$f(x) = \ln(1 + \exp x) \quad (2.7)$$

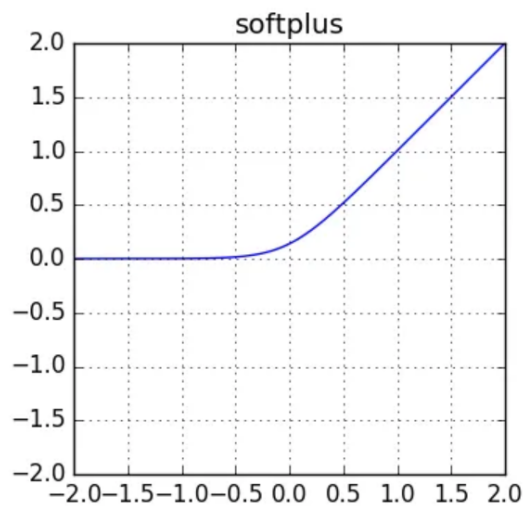


Figure 2.27. Softplus function [Ser17] .

2.2.7. Leaky ReLU and Generalized ReLU

The following two functions are a variation of the simple ReLU function. The Leaky ReLU is similar to the ReLU but it counts with a slight negative slope on the negative part. This negative slope is aimed at eliminating the dead neuron problem. The Leaky ReLU is, in fact, one of the multiple activation functions included in the generalized ReLU family. All the functions belonging to this family are computationally similar to the original ReLU.

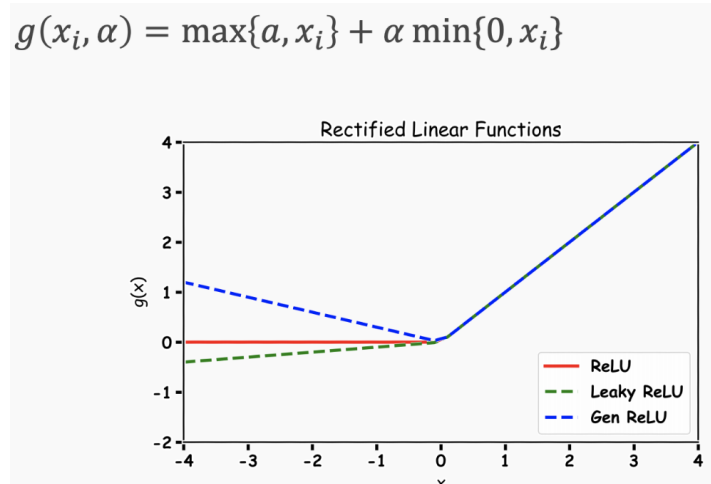


Figure 2.28. Rectified Linear Functions [Ste19a] .

2.2.8. Exponential Linear Unit (ELU)

It is very similar to the ReLU function, but it avoids the zero values when the inputs are negative. The extra parameter alpha should always be a positive number. ELU smoothen slowly until its output equal to -alpha.

$$f(x) = \begin{cases} x & x > 0 \\ \alpha (\exp x - 1) & x \leq 0 \end{cases} \quad (2.8)$$

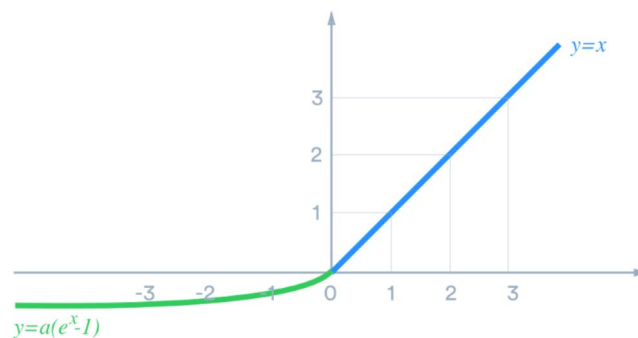


Figure 2.29. Exponential Linear Unit.

2.2.9. Scaled Exponential Linear Unit (SELU)

Evolves on the idea of the ELU function, including an offset with an extra parameter. It is a relatively new activation function that has not being tested extensively. More research with recurrent and convolutional neural networks is necessary before confirming an improvement over ELU.

$$f(x) = \gamma \begin{cases} x & x > 0 \\ \alpha (\exp x - 1) & x \leq 0 \end{cases} \quad (2.9)$$

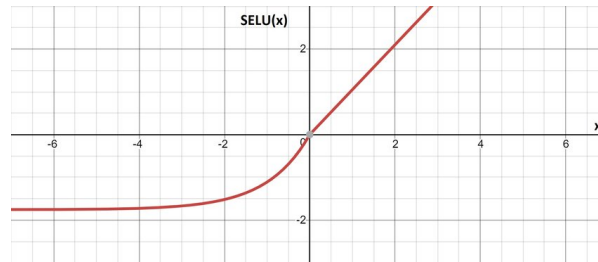


Figure 2.30. Scaled Exponential Linear Unit.

2.2.10. Maxout

It is the maximum of k linear functions. It is a learnable activation function. Both ReLU and leaky ReLU are special cases included in Maxout. It can benefit from the strengths of these functions without the drawbacks. However, it doubles the total number of parameters that must be learnt for each neuron [Jai19].

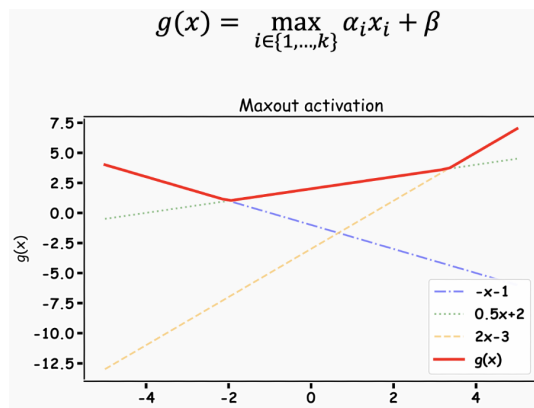


Figure 2.31. Maxout Function and equation [Ste19a] .

2.2.11. Swish

A self-gated activation function. It is a smooth and non-monotonic curve. It consists in the multiplication of a sigmoid by the input (x). Swish was developed by Google in 2017. It outperforms ReLU in multiple studies recently developed [Ste19a]. In very deep networks, Swish outperforms ReLU by a large margin in the range between 40 and 50 layers [Din+17].

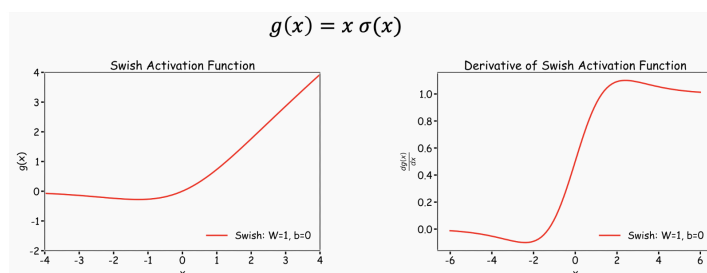


Figure 2.32. Swish function and derivative [Ste19a] .

2.3. Optimization algorithms

Optimization involving neural networks means non-convex optimizations. In order to tackle this problem, there are two main approaches, the first order optimization algorithms and the second order optimization algorithms. First order optimization algorithms minimize a loss function using the values of the gradient with respect to the parameters. Second order algorithms compute the second order derivative to minimize the loss function.

2.3.1. Gradient Descent

It is the most known technique and the foundation to the ones developed in this section. It uses the gradient of the cost function in order to update the parameter and obtain a new configuration that leads to a reduce loss [Sin17].

$$\theta = \theta - \eta \cdot \nabla \cdot J(\theta) \quad (2.10)$$

The problem with gradient descent is that it tends to excessively oscillate whenever there is a slope in two axes. It also struggles whenever there are saddle points [Sin17]. Many algorithms have been developed to tackle these issues.

2.3.2. Stochastic Gradient Descent

This technique performs a parameter update for every training example, updating the values one example at a time [Sin17]. There is a learning rate parameter (η). The stochastic gradient descent has the characteristic that it fluctuates during the updates. This fluctuation is positive in the sense that it enables for better minima to be discovered, but it also complicates the convergence to the optimal minimum, given that overshooting is common.

$$\theta = \theta - \eta \cdot \nabla \cdot J(\theta; x(i); y(i)) \quad (2.11)$$

Where $x(i)$, $y(i)$ are the training examples.

2.3.3. Mini-Batch Gradient Descent

This optimization algorithm proposes a way of solving the high variance and unstable convergence of the stochastic gradient descent. It performs an update for every batch including the n data points selected as a parameter. As a result, this technique usually provides a more stable convergence pattern, as it reduces the variance in the updates [Sin17]. Typical mini-batch sizes vary from 50 to 256, but it depends on the case. This algorithm is commonly used in neural network applications nowadays.

A key challenge for both Stochastic Gradient Descent and Mini-Batch Gradient Descent is stopping the training phase in a local minimum. Non-optimal local minima are not the main issue but saddle points, when one direction slopes upwards and another one backwards. The saddle points form a plateau of same error values, making it particularly hard for SGD or Mini-Batch GD to successfully move out of the zone.

2.3.4. Momentum

Momentum is an added term to the optimization equation. It attempts to replicate the concept of momentum in physics, as an object slides down a slope its velocity increases and it becomes harder for it to change directions [Sin17]. It results in faster and more consistent optimization with less oscillations.

Compute gradient estimate:

$$g = \frac{1}{m} \cdot \sum_{i=0}^n \nabla_{\theta} \cdot L((x(i); \theta), y(i)) \quad (2.12)$$

Update velocity:

$$v = \alpha \cdot v - \varepsilon \cdot g \quad (2.13)$$

Update parameters:

$$\theta = \theta + v \quad (2.14)$$

The momentum parameter (v) is commonly known as velocity and it updates the value in every iteration. The internal parameter (α) is set between 0 and 1 [Ste19a].



Figure 2.33. Optimization algorithms with (right) and without (left) a momentum term.

2.3.5. Nesterov Momentum

Following the analogy with the concept of momentum in traditional physics, gradient descent with momentum methods are equivalent to a dummy ball that slides down a curved terrain. However, a researcher named Yurii Nesterov realized this model is not the optimum for optimization algorithms. An excessive accumulated momentum could make our fictional dummy ball to miss a minima solution. Nesterov proposed an optimization algorithm where our fictional ball computes lookaheads of the slope of the trajectory and reduces the momentum term in case it recognises a “hill” is in front [Sin17]. In practice it means computing an anticipatory correction for the momentum parameter.

2.3.6. AdaGrad

This algorithm is an abbreviation for adaptive gradient. The main focus is to allow the learning rate to adapt based on the parameters. The algorithm uses a different learning rate for each parameter at a given time step based on the past gradients which were computed for each parameter.

Apply an **interim** update:

$$\tilde{\theta} = \theta + v$$

Perform a correction based on gradient at the interim point:

$$g = \frac{1}{m} \sum_i \nabla_{\theta} L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$$

$$v = \alpha v - \epsilon g$$

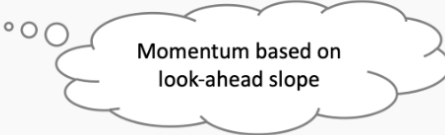
$$\theta = \theta + v$$


Figure 2.34. Nesterov Momentum Equations [Sin17].

The most significant advantage of AdaGrad is that it eliminates the need to manually tune the learning rate [Ste19a]. Most optimization algorithms simply use a value of about 0,01. The main issue with the algorithm is that, as it performs an accumulated sum of squared errors in the denominator, the learning rate is ever decreasing, coming to a point in which the model can no longer pursue any significant additional learning. This is called the decaying learning rate problem [Sin17].

Accumulate squared gradients:

$$r_i = r_i + g_i^2 \quad (2.15)$$

Update each parameter:

$$\theta_i = \theta_i - \frac{\epsilon}{\delta + \sqrt{r_i}} \cdot g_i \quad (2.16)$$

2.3.7. AdaDelta

The main objective of AdaDelta is to improve AdaGrad by tackling the decaying learning rate problem. Instead of accumulating the totality of previous squared gradient, AdaDelta works with a window of the last w gradients [Sin17]. Instead of storing the past w squared gradients, it computes the decaying mean for the squared gradients. The parameter γ of the decaying average is usually set around 0,9. The parameter ϵ is a small value added to avoid division by zero [Gyl18].

$$E[g^2]_t = \gamma \cdot E[g^2]_{t-1} + (1 - \gamma) \cdot g_t^2 \quad (2.17)$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \quad (2.18)$$

However, the authors of the AdaGrad algorithm noticed that the theoretical units of their update function did not match the theoretical units of the parameters (as it is common

optimization techniques described before such as SGD or AdaGrad). It was their decision to correct the unit of the parameter updates with hessian approximations [Gyl18].

The result of their solution is presented in Equation 2.18. It has the additional advantage that it does no longer require a learning rate (it was substituted).

2.3.8. RMSProp

RMSprop is another algorithm intended to solve the decaying learning rate problem of AdaGrad [Rud16]. It is an unpublished adaptive learning rate method proposed by Geoff Hinton [Rud16]. It was developed simultaneously and independently as AdaGrad. In fact, the update equation is the same as the one presented in AdaGrad if we do not correct the units of the parameters, also computing a moving average of the square gradients of each parameter for every iteration. The learning rate is usually set around 0,001.

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} \cdot g_t \quad (2.19)$$

2.3.9. Adam

Adam stands for adaptive moment estimation. It is one of the most popular optimization algorithms nowadays [Kar17]. It is a combination of RMSprop and standard Momentum. As other optimization algorithms exposed before, Adam computes adaptive learning rates for each parameter. It calculates an exponential decaying average for past squared gradients (like RMSprop or AdaGrad), as well as an exponential decaying average of past gradients (like Momentum).

Estimate first momentum:

$$v_i = \rho_1 \cdot v_i + (1 - \rho_1) \cdot g_i \quad (2.20)$$

Estimate second momentum:

$$r_i = \rho_2 \cdot r_i + (1 - \rho_2) \cdot g_i^2 \quad (2.21)$$

Update parameters:

$$\theta_i = \theta_i - \frac{\epsilon}{\delta + \sqrt{r_i}} \cdot v_i \quad (2.22)$$

Suggested values for the moving average parameters, ρ_1 and ρ_2 , are 0,9 and 0,999 respectively. The ϵ parameter is usually set to a value of 10^{-8} [Kar17]. The vectors of moving averages are initialized with zeros in the first iteration [Raj19].

Further developments have been proposed starting from the Adam algorithm, such as Adamax [Chu+14] (a variation in the update rule is introduced), Nadam [Kha19] and AMSGrad [RKK18] (uses the maximum of past squared gradients rather than the exponential average to update the parameters) [Gyl18].

2.3.10. Nadam

It was proposed to improve the Adam algorithm. It is a combination of Nesterov's momentum and Adam [Kha19]. This algorithm is taking advantage of the theoretical superiority of Nesterov's momentum versus the regular "vanilla" momentum.

The only significant change is that the momentum is computed using interim parameters, instead of current parameters. It applies the acceleration to the parameters before computing the gradients, then do the update with the gradients calculated.

Estimate first momentum:

$$v_{i+1} = \rho_1 \cdot v_i + (1 - \rho_1) \cdot g_{NAG} \quad (2.23)$$

Estimate second momentum:

$$r_i = \rho_2 \cdot v_i + (1 - \rho_2) \cdot g_{NAG}^2 \quad (2.24)$$

Update parameters:

$$\theta_{i+1} = \theta_i - \frac{\varepsilon}{\delta + \sqrt{r_i}} \cdot v_{i+1} \quad (2.25)$$

2.4. Loss functions

Another relevant aspect of training a neural network is computing how much the predicted and the real output differ, in order to update the network parameters to reduce this difference. A loss function (or cost function) is a measure of how the neural network performs with respect to the ideal behaviour given the application. Therefore, it is dependent on the end use and the output of the network. It can be a function of the output, the input, the weights and the biases.

Binary Cross Entropy

Also known as Bernoulli negative log-likelihood. Well suited to binary classification problems where the last layer is a sigmoid activation function [Bus18].

$$C_{CE}(W, B, S^r, E^r) = - \sum_j (E_j^r \cdot \ln a_j^L + (1 - E_j^r) \cdot \ln(1 - a_j^L)) \quad (2.26)$$

Cross Entropy

Also known as softmax loss. As the name suggests, it is targeted at multiple class classification problems where the last layer uses softmax as an activation function [Góm18].

$$CE = \frac{1}{M} \sum_p - \log\left(\frac{\exp S_p}{\sum_p \exp S_j}\right) \quad (2.27)$$

Mean Squared Error (MSE)

Also known as maximum likelihood or sum squared error. Common in regression and forecasting problems, it has the issue that individual extreme values can significantly taint the resulting value.

$$C_{CE}(W, B, S^r, E^r) = \frac{1}{n} \sum_j (a_j^L - E_j^r)^2 \quad (2.28)$$

Root Mean Squared Error (RMSE)

The root mean squared error applies the square root to the formula shown in Equation 2.28. This measure of accuracy is frequently found in forecasting and regression problems.

$$C_{CE}(W, B, S^r, E^r) = \sqrt{\frac{1}{n} \sum_j (a_j^L - E_j^r)^2} \quad (2.29)$$

Mean Absolute Error (MAE)

This measure of accuracy is frequently found in forecasting and regression problems. It measures the average magnitude of the errors in a set of predictions, without considering their direction.

$$C_{CE}(W, B, S^r, E^r) = \frac{1}{n} \sum_j |a_j^L - E_j^r| \quad (2.30)$$

Exponential Cost

It requires tuning the parameter τ .

$$C_{EXP}(W, B, S^r, E^r) = \tau \exp \frac{1}{\tau} \sum_j (a_j^L - E_j^r)^2 \quad (2.31)$$

Huber Loss

It is a robust loss function for regression and forecasting problems. It is less sensitive to outliers than the mean squared error cost function [Dra19].

$$L(y, \hat{y}) = \begin{cases} (y - g^{-1}(\hat{y}))^2 & \dots \quad |g(y) - \hat{y}| \leq \alpha \\ |y - g^{-1}(\hat{y})| & \dots \quad |g(y) - \hat{y}| > \alpha \end{cases} \quad (2.32)$$

2.5. Optimization techniques

In order to improve overall training and avoiding issues such as overfitting, several optimization techniques are commonly used. The most relevant ones are the following:

2.5.1. Regularization

- Dropout: during the training process, some activations are randomly ignored with a pre-set probability p . This effectively prevents overfitting by reducing the correlation between neurons [Hin+12].
- Batch normalization: it is a technique that makes networks robust to bad initialization of weights. Usually inserted right before activation layers. It reduces covariance shift by scaling and normalizing inputs [LJY19].
- Data augmentation: it consists on enriching the existing data to allow for better generalization skills of the network. It involves applying known transformations to the data set, either in a fixed or a random proportion. There is a very clear case for convolutional neural networks, as images can be horizontally flipped, translated, rotated, stretched, the colour can be modified etc [Res20].
- Dropconnect: it involves neutralizing some connections during the training process, setting its weights to zero [LJY19].

2.5.2. Parameter initialization

- Zero initialization: initializing bias and weights with initial value zero. It is more a theoretical experiment than a useful initialization technique. In some cases, if the activation function has a symmetric derivative (i.e. sigmoid) a zero weight initialization can lead to symmetric hidden units that cannot outperform a linear fit (all the weights are trained symmetrically, having the same parameter updates, resulting in redundancy) [Res20].
- Random initialization: it brings an improvement with respect to the zero initialization. Weights are initialized randomly and the biases are initialized to zero. When initializing the values, it is important to avoid one of the two classic gradients problems, exploiting and vanishing gradients. A too large or a too small initialization can lead to these problems respectively, so values need to be carefully capped [Bus18].
- He normal initialization: consists on multiplying the random initialization by a factor [Bus18]. This technique works well when using ReLUs as activation function [Dos19b].
- Xavier initialization: in the same way as the He normal initialization does, it involves multiplying a randomly generated set of numbers by a factor [Bus18]. The Xavier initialization is intensively used when dealing with hyperbolic tangents as the activation function [Dos19b].
- Bias initialization: it refers to the way the bias vector is initialized. The most common tactic is to initialize these values to zero, as the asymmetry breaking is provided by the normal values assigned to the weights initially [Res20]. However, in case of the ReLU activation function, some practitioners prefer to initialize the bias value higher than zero (typically 0,01) to ensure the activation of the neurons in the first iterations to prevent dead neurons [Res20].
- Pre-initialization: this technique involves using the weight values of a previous trained network as the initial parameters. It can only be applied when the original network had a similar function to the one set to train. This concept is related to the idea of transfer learning [Res20].

2.6. Use Cases

Given the diversity of options presented in the state of the art of this project, it was considered a requirement to organize and structure the different alternatives available when tackling a Machine Learning problem. From the state-of-the-art analysis and the literature review, an overwhelming majority of use cases fell into one of the three categories of supervised learning. Although this classification cannot be considered complete, it is provided in this project to shed some light on the question of what type of network should be used for other applications that do not resemble the one discussed in Chapter 3.

2.6.1. Classification Problems

- Text processing tasks like sentiment analysis, parsing or named entity recognition.
 - Recurrent Neural Networks in the character level.
- Image recognition
 - Convolutional Neural Networks (very effective with the filter scanning algorithms, specially with augmented data).
 - Deep Belief Networks (powerful when dealing with unlabeled data sets).
- Object Recognition
 - Convolutional Neural Networks
- Speech Recognition
 - Recurrent Neural Networks (effective structures to analyze sequential data and relevant information over timesteps).
- More general tasks
 - Deep Belief Network
 - Multi-Layer Perceptron

2.6.2. Regression Problems

- Multi-Layer Perceptron
- Recurrent Neural Networks

2.6.3. Forecasting Problems

- Recurrent Neural Networks

2.6.4. Unsupervised Learning

- Restricted Boltzmann Machines
- Autoencoder

3

Case Study

Whatever you can do or dream, begin it.

Johann Wolfgang von Goethe
(1749–1832)

On this chapter several alternative neural network architectures will be tested and their performance will be compared using a wind energy generation dataset. This case study intends to simplify the architecture selection process, given the multiple alternatives available in the machine learning toolbox

3.1. Problem Statement

The dataset to be analyzed comes from a machine learning contest and it contains recorded values of wind power generation with 90 different input variables. The available explanatory variables include wind speed, wind direction and temperatures in a given area. Overall, this problem accounts with over 35,000 samples.

In Figure 3.1 the three different types of input variable are visible. In the top chart the temperatures are displayed in degrees Celsius. The second chart shows the aggregate of the wind direction variables. This can oscillate from -180° to 180° . In the last graph the wind speed variables are shown and how they evolve through time. The last plot shows the evolution of wind speed variables through time.

Figure 3.2 shows the output, which is the total electric power (MW) generated in the wind power station at a given moment.

As a way to test the value added with the trained models, they will be compared with two base benchmarking algorithms.

The first benchmark (from now on just benchmark) is a simple one-day forecasting window using the value of the previous datapoint. It is a common proxy used in several industries such as the financial services and it is an easy way to set a target for any forecasting model.

$$y_{pred}[t] = y[t - 1] \quad (3.1)$$

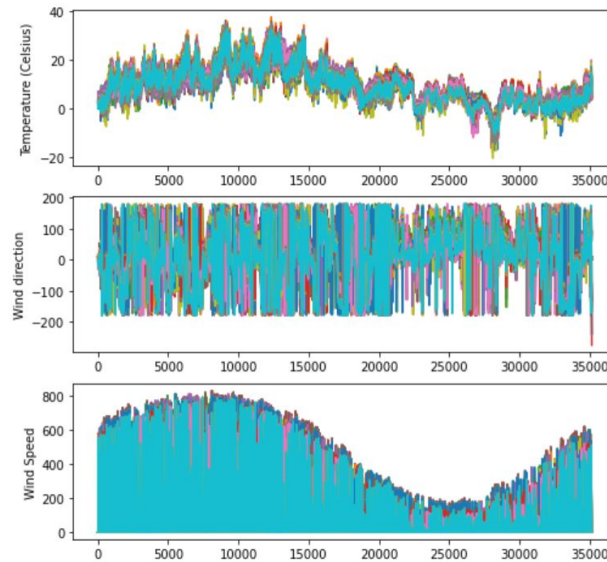


Figure 3.1. Different types of inputs for the forecasting problem.

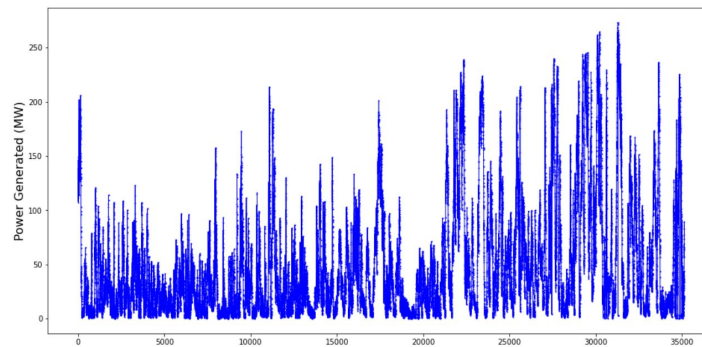


Figure 3.2. Power generated Output.

The one-timestep lag error with outliers is shown in Table 3.2. The one-timestep lag error without outliers is shown in Table 3.4.

The second benchmark model (from now on MLP) is a simpler neural network, a multilayer perceptron. One of the objectives of the project is to show how this more complex models contribute to a better explanation of a time series forecasting problem. This are common initial values in similar applications. The MLP will change the structure in every experiment, in order to have a similar structure as the GRU and LSTM models that will be tested, layer and neuron-wise. This is done to fairly compare all models and isolate the effect that the chosen model type (cell structure) has on the model predictions.

The parameter for batch size was selected at 50 and the number of iterations was initially set to 30. This are common initial values in similar applications. In order to capture the output time evolution, MLP uses 5 backsteps of the output as input variables and the prediction is in a one timestep horizon.

The training-validation split is 80% training, 20% validation. The validation set corresponds with the last part of the dataset in order, given the sequential nature of the problem.

The error measures shown in every single one of the following experiments are the errors of the validation dataset, as it shows if the models are capable of generalizing and correctly leverage the relationships between outputs and inputs. The optimal model would be the one

with lowest validation error, as this would indicate which model generalizes better when faced with new data.

3.2. LSTM and GRU networks comparison

The first task this thesis will take on is a performance comparison between two popular recurrent neural networks, the Long-Short Term Memory and the Gated Recurrent Unit. The later is a posterior an simplified version of the first one and it was introduced with the purpose of substituting the LSTM in applications that do not need such heavy calculations to achieve an accurate fit.

The cell state present in the LSTM algorithm is a powerful feature that allows it to control the information flow from past timesteps to influence the output. However, the GRU was created under the believe that a simpler algorithm could not only improve the running time, but also the performance scores of the LSTMs.

3.2.1. First LSTM-GRU Test

The first experiment was carried out using a random neural network architecture with 3 layers and 20 neurons per layer for each case. The epochs were set to 20 while the batch size was fixed at 50.

The rationale behind the experiment was to check the behaviour of both algorithms with the same structure and testing out common alternatives of activation functions and optimization functions. Tanh, maxout and sigmoid were used in this first test as well.

Code	Algorithm	Opt. algorithms	Activation	Average MSE	Local Ranking	Total Ranking
1	LSTM	Adam	ReLU	249.20	2	5
2	LSTM	Adam	SeLU	247.42	1	4
3	LSTM	Nadam	ReLU	351.16	4	8
4	LSTM	Nadam	SeLU	266.13	3	6
5	GRU	Adam	ReLU	237.22	3	3
6	GRU	Adam	SeLU	221.72	2	2
7	GRU	Nadam	ReLU	279.10	4	7
8	GRU	Nadam	SeLU	211.69	1	1

Figure 3.3. Results of the first LSTM-GRU test.

The average validation MSE value in Figure 3.3 refers to the MSE score achieved by the different algorithms average over the five simulations carried out.

As we can observe in Figure 3.3, the alternatives tested were some of the most promising activation function and optimization algorithms as seen in Chapter 2. This includes the ReLU, SELU and ELU activation functions and the Adam and Nadam optimization algorithms.

The main conclusion of this initial test are that the SELU activation function is a promising rival for the ReLU, outperforming it in several cases. At this level of complexity, the GRU algorithm outperformed the LSTM algorithm in 3 out of 4 cases. No definitive conclusions are driven from the comparison between the Adam and Nadam optimization algorithms.

3.2.2. Second LSTM-GRU Test

As learned in the previous section, the activation functions do impact in a notable way the performance of the algorithm. Therefore, it was decided to keep the SELU and the ReLU options and to include the ELU as an alternative as well.

Given that Adam is simpler and more common in the literature than Nadam, and that there was no significant difference in behaviour found in the first LSTM-GRU test, the Adam algorithm was selected as the only choice for the second LSTM-GRU test in order to decrease the computational requirements of these experiments.

In this case, the number of simulations was 30 to make the results more reliable with a larger sample. This way there is a higher statistical certainty of the veracity of the results, as the results are an aggregate and not just a single data point. All the networks had 3 layers and 20 neurons, as in the previous experiment. Epochs were kept to 20 and batch size to 50.

Code	Algorithm	Activation	Average MSE	Local Ranking	Total Ranking
1	LSTM	ReLU	266.49	2	5
2	LSTM	eLU	256.28	1	4
3	LSTM	SeLU	310.37	3	6
4	GRU	ReLU	201.41	2	2
5	GRU	eLU	207.93	1	1
6	GRU	SeLU	226.49	3	3

Figure 3.4. Results of the second LSTM-GRU test.

The results of this test indicate that, at this level of complexity the GRU algorithm is more powerful than the LSTM algorithm. The ReLU activation function turned out to be better than the SELU, however, the simpler ELU was the best activation function referring to its average MSE validation values.

3.2.3. Optimized algorithms and analysis with outliers

In order to compare the best performing versions of the MLP benchmark, the GRU and the LSTM, a hyperparameter optimizator was used. In this case, it is a Bayesian optimizator, which creates a statistical model that evaluates the predictors as a black box. By updating the believes after each tested point, via multiple iterations, it trains the model in each iteration with the hyperparameter configuration which should have the lowest error predicted by the statistical model [Res20].

Using the Bayesian enables to find optimized solutions faster than by manually fixing several parameters. The tuned hyperparameters in this experiment were the model, activation function, number of layers, neurons per layer, learning rate and optimization algorithm. The optimization process used 200 iterations for each model.

For this experiment, the hyperparameters of the functions were the following:

The interesting side of this experiment is seen when observing how the models behave when facing an outlier. Under these circumstances some models have learned how to avoid individual outliers. This is the case for the Optimized GRU (hyperparameters shown in Table 3.1, results shown in Table 3.2) and it can be observed in Figure 3.5.

Model	Activation	Number of layers	Neurons per layer	Learning rate	Optimizer
MLP	SELU	2	34	0.0009746	Adam
LSTM	ELU	2	50	0.0010384	Adam
GRU	ELU	3	56	0.0007395	Adam

Table 3.1. Hyperparameters values for the best neural network models

Model	MSE	MAE
Benchmark	272.079	8.340
Optimized MLP	236.645	7.961
Optimized LSTM	210.550	7.872
Optimized GRU	178.295	7.574

Table 3.2. Results of the prediction models

In this experiment, very prominent outliers can be observed when plotting the residuals with respect to time (See Figure 3.6). However, there is a difference between the optimized GRU algorithm and the other models, as this one learns to avoid some outliers.

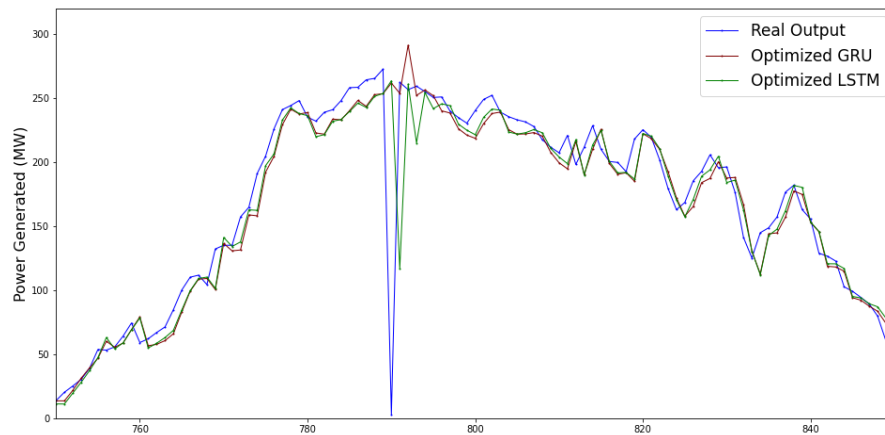


Figure 3.5. GRU model avoiding an outlier in time-step 791.

As observed in the previous image, the LSTM can also recognise it is facing an outlier, reducing the drop to a little over half of the step, however there is a ripple effect that persists for the following timesteps (Figure 3.5).

When analyzing the normalized residuals of the fitted optimized LSTM and GRU, it can be observed how there are multiple outliers not avoided in the validation set (See Figure 3.6).

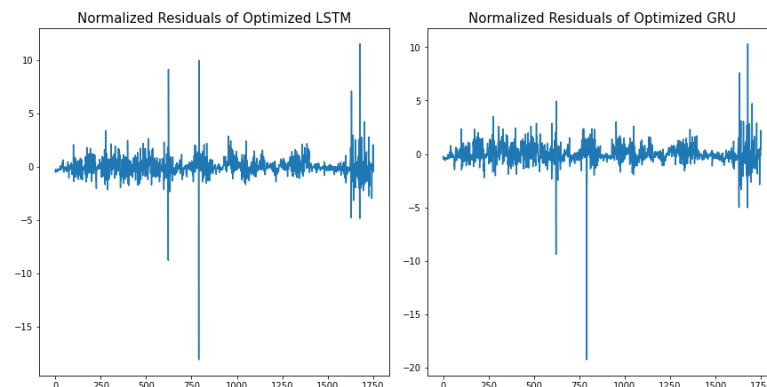


Figure 3.6. Normalized residuals for the two advanced neural network models.

3.2.4. Optimized algorithms and analysis without outliers

For the second experiment in this series, the power output function was cleaned to make sure no outliers were passed to the algorithms in the inputs. The results can be shown in the following image Figure 3.7.

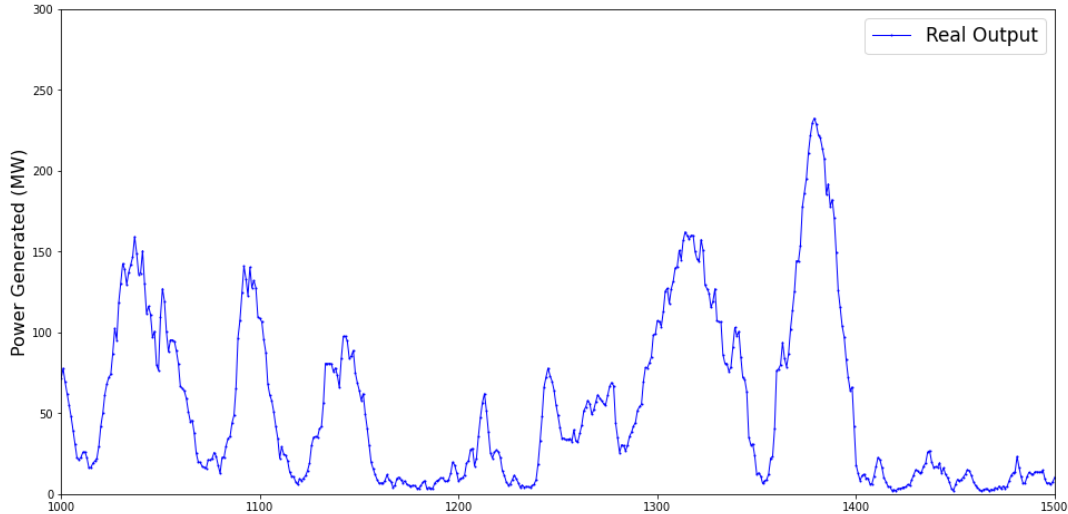


Figure 3.7. Sample of output timeseries without outliers.

The timeseries was corrected both in the training and validation datasets. The number of outliers located in the validation dataset far exceeds the ones found in the training dataset. Despite being just 20% of the data, it contained 13 outliers versus 4 outliers found in the rest of the set. It is also a region with significant higher values and higher variance than the training one.

In this experiment, the models were first over-trained to get a rough concept of where the optimum training should end. After training the algorithms for 150 epochs, it was determined that the optimal epoch value was close to 25 in both cases. The final learning curve for the three models differed significantly, specially between the GRU (Figure 3.8) and the LSTM (Figure 3.9) ones.

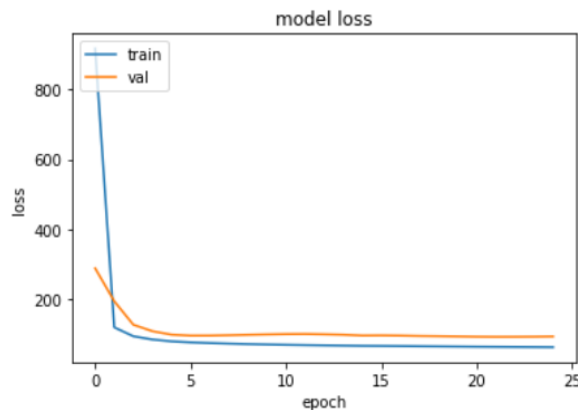


Figure 3.8. Learning curve from optimized GRU.

The final learning curve for the GRU algorithm is remarkably stable, without big changes in the slope and tracing a smooth curve. This was attributed to the small learning rate value. In contrast, Figure 3.9 shows a more erratic behaviour that keeps improving on average but it oscillates.

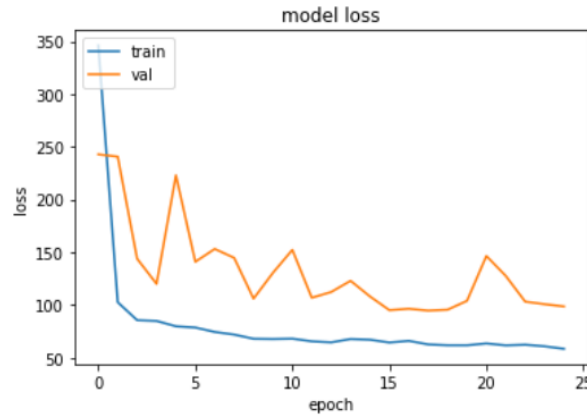


Figure 3.9. Learning curve from optimized LSTM.

The Bayesian algorithm was used again for hyperparameter optimization purposes with 200 iterations for each model:

Model	Activation	Number of layers	Neurons per layer	Learning rate	Optimizer
MLP	SELU	3	50	0.0010145	Adam
LSTM	ELU	2	50	0.0007182	Adam
GRU	ELU	4	62	0.0000965	Adam

Table 3.3. Hyperparameters values for the neural network models trained without outliers

Model	MSE	MAE
Benchmark	115.815	7.443
Optimized MLP	125.495	7.949
Optimized LSTM	109.571	7.356
Optimized GRU	91.154	6.617

Table 3.4. Results of the prediction models

Several relevant observations can be made from the tables above. On one hand, none of the optimized models choose the ReLU option. On the other hand, the GRU model is deeper than the rest, using 4 layers. It is also the best performer and it has a particularly small learning rate.

We can observe the next fit of the models to the problem in a region that was heavily affected by outliers before in Figure 3.10. There is little to no influence of the past outliers in the next predictions. It is possible to observe a similar behaviour in all neural networks, following the 1 lag reference closely and trying to approach closer the output at several points of the graph. The GRU solution seems to better fit the output in some regions (e.g. from 1665 to 1670).

It is interesting to observe that the influence of the one timestamp lag grows further when the output value changes abruptly. Most of the time the predictors revolve around the output value and the benchmark. However, when the slope is too steep they give up additional efforts and opt instead to become a follower of the one time period delay (Figure 3.10).

In Figure 3.11 one could observe a sample of the fitted models in the same region that used to host an outlier (See Figure 3.5). The behaviour of the models has improved without the outliers, it seems like interpolate the outliers was the right thing to do to improve the predictions.

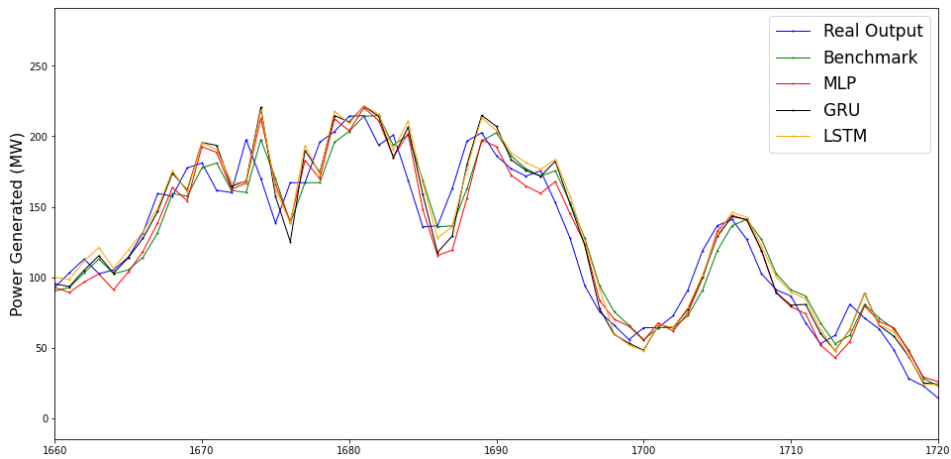


Figure 3.10. Sample of the fitted models without outliers.

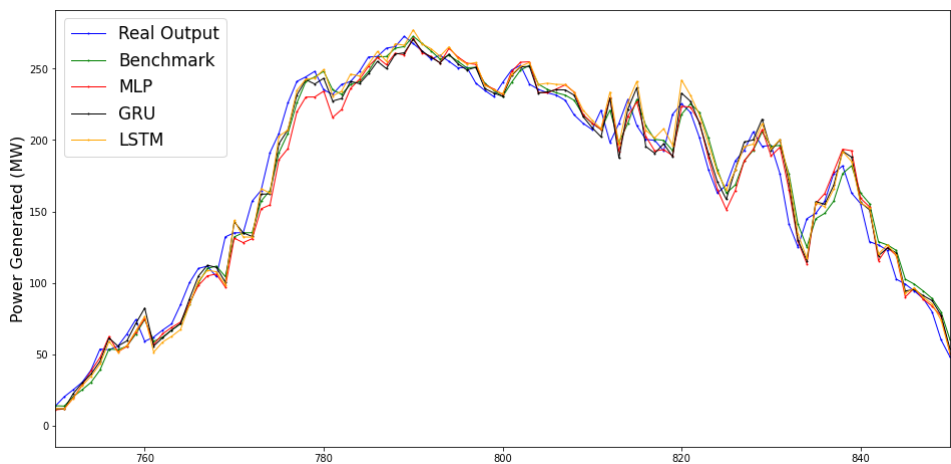


Figure 3.11. Fitted models, output and benchmarks.

In Figure 3.14, it is possible to see the contrast when the outliers were removed (See Figure 3.6). Regarding the residuals, they do not follow a normal distribution for any of the adjusted neural network models (See Figure 3.12). This implies that the model is not explaining all the error possible, as the result is not white noise.

Despite not following a normal distribution, the residuals of the optimized GRU model are promising since it beats the MSE score of the one lag benchmark by a 22.3% and the MSE score of the MLP by a 27.4%.

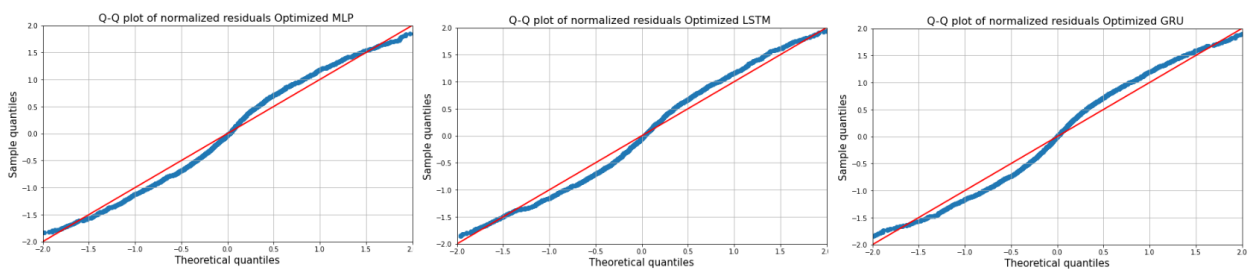


Figure 3.12. Q-Q Plot of the neural network predictors.

The histogram of the distributions confirms the residuals follow a normal in none of the cases. To start with, no distribution is fully symmetrical.

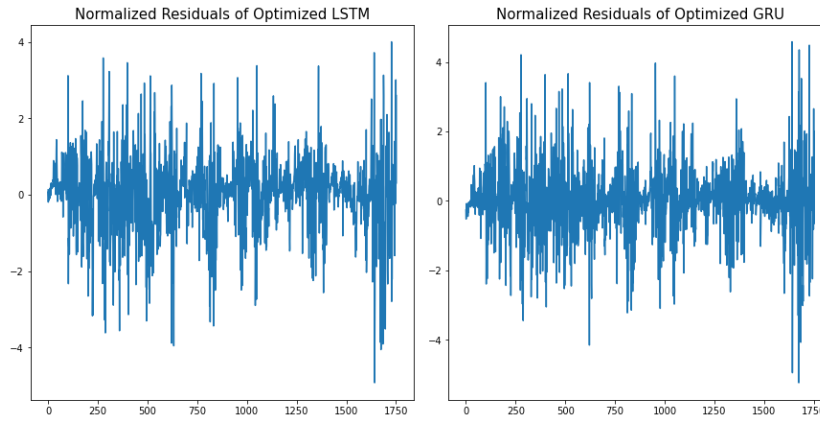


Figure 3.13. Plot of the normalized residuals for the LSTM and the GRU.

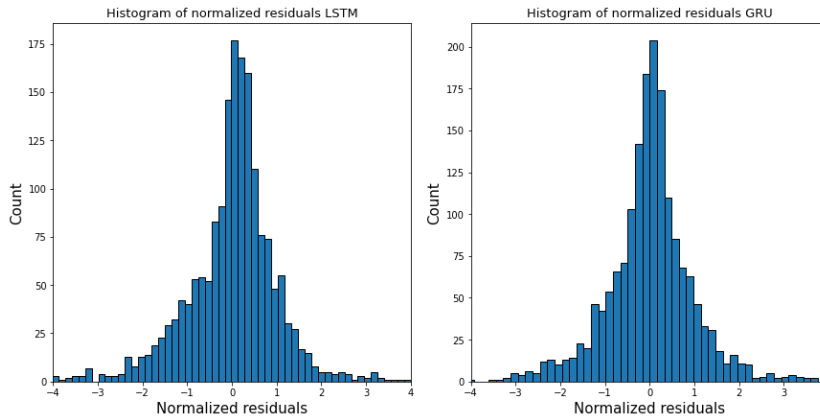


Figure 3.14. Histogram the normalized residuals for the LSTM and the GRU.

Furthermore, the partial and total autocorrelation plots show the optimized GRU can explain significant information from previous lags, specially when compared with the ACF and PACF plots of the closest performing algorithm, the LSTM. (Figure 3.15 and Figure 3.16). In this respect, the GRU architecture reduces significantly the explanatory information lost from previous time-steps.

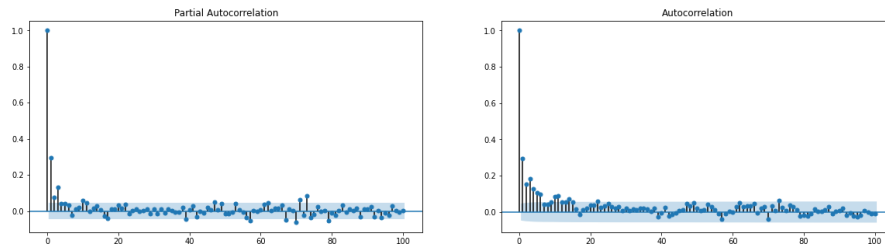


Figure 3.15. LSTM partial and complete auto-correlation plots.

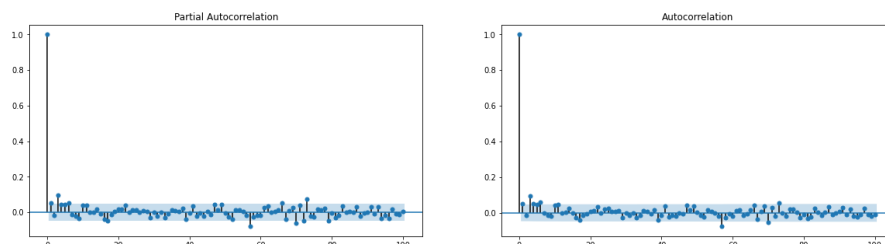


Figure 3.16. GRU partial and complete auto-correlation plots.

3.2.5. Model dynamic response

After training the models and computing the errors, the next part of the project consisted in observing the time response of the neural network architectures when a step is simulated in the output variable. Neural networks are capable of mimicking dynamics, and the step experiment is a way of showing what dynamics were learned.

In order for this experiment to be considered valid it was important for the inputs not to affect the dynamics when subject to the sharp increase of the step. To ensure this to the greatest extent, two semi-flat regions of the validation training set were modified. (See Figure 3.17). In each of the two areas the procedure was the same. First, an area of 50 time-steps is leveled using the average value of the time-period to obtain a flat plain. Second, in the middle of the newly created flat area a step of 50 MW (decreasing or increasing, depending on the case) is applied for 15 time-periods, enough time for the systems to reach a permanent state, if the case.

As previously mentioned, the models are already trained and the steps do not affect the training dataset, but only the validation one. The resulting validation dataset with the two steps implemented is shown in Figure 3.17.

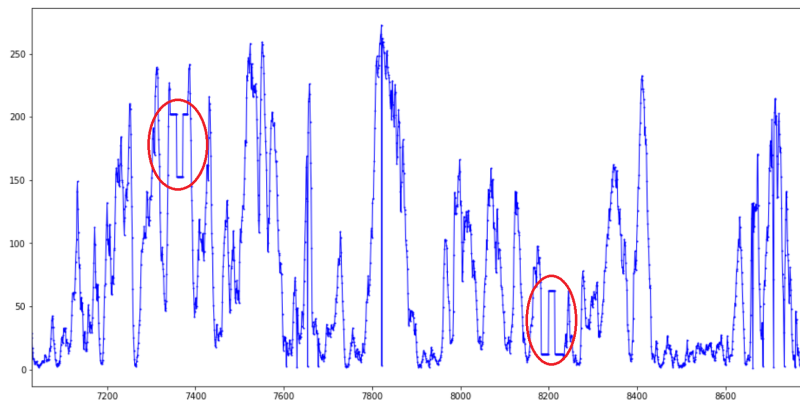


Figure 3.17. Validation dataset with steps implemented.

In the downwards step between 205 MW and 155 MW several different behaviours can be observed. In the first place, the three models follow the the one timestep lag immediately. In this sense, they all behave the same way. However, the LSTM model seems to have a positive offset. The GRU model follows the step better, and it reaches the permanent value at the end of the plain. The MLP has a good response, but it does not reach the required level in the last stage. Both the LSTM and the GRU overshoot when during the upwards part of the step.

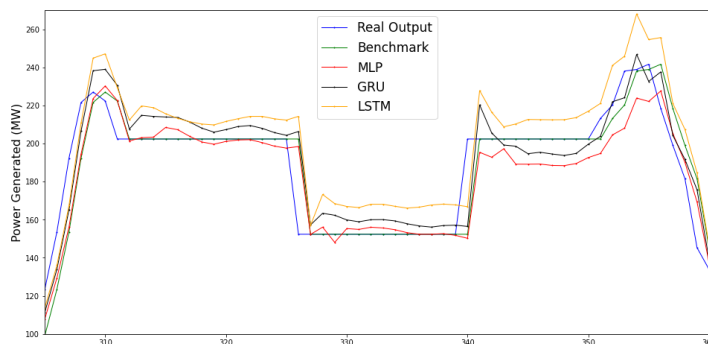


Figure 3.18. Time response of the downwards step.

When analyzing the time response of the upwards step, the one starting in a low value around 12 MW and then rising to 62 MW, we see a different behaviour. In this case, the GRU model is the one showing the most overshoot. Again, the LSTM model is the slowest in reacting and it's the furthest away in the first timestamps. It is important to notice how none of the predictors stabilize in the high value of the upwards step. The reason behind this seems to be that the inputs are clearly indicating that the output should be significantly smaller, so the learning process was successful. It also shows that the predictors rely on big changes in the 1 day lag for the shortterm predictions, but this influence wears down as the time-steps pass.

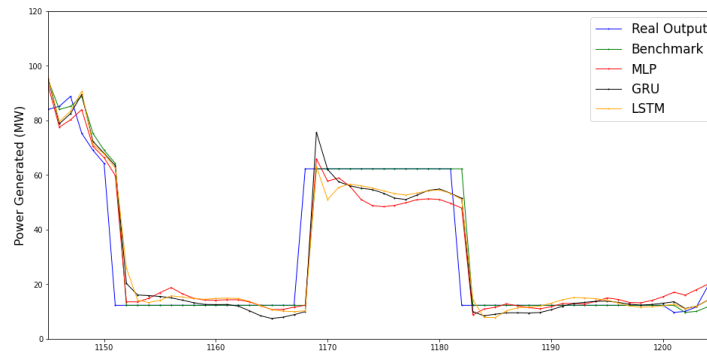


Figure 3.19. Plot of the upwards step.

3.3. Convolutional Neural Network predictor

In the final experiment of the project, the one-dimensional convolutional neural network (CNN from now onwards) will be explored. The main objective is to observe differences in the predictions compared with the most successful recurrent neural network. In this model the output is modelled using pattern recognition and extraction, so an improvement over the benchmark is expected beforehand.

The way of approaching the training of the CNN was to run a Bayesian hyperparameter optimizer in two steps. The first one with 200 iterations and a wide hyperparameter space so the algorithm could explore different and diverse solutions.

After obtaining the solution, a second optimization process was launched with 50 iterations and with the space of the hyper-parameters more restricted around the values that rose in the first process. It is very relevant to find the optimized function this way, given that the CNN has a larger number of hyper-parameters, including activation function, kernel size, pool size, dropout, neuron layers, neurons per layer, number of filter layers, filters per layer and others.

Iteration	Activation	Filter layers	Filter per layer	Neuron layers	Neurons per layer
First	ELU	1	28	2	20
Second	ReLU	1	48	1	44

Table 3.5. Main hyper-parameters of the Convolutional Neural Network's optimization

In Table 3.5 we can observe how the structure of the proposed solutions changes when subject to the second optimization process. The network reduces the number of neuron layers, but the one remaining layer has 20 filters per layer more and 20 additional neurons per layer. It is worth mentioning that the activation function changes back to ReLU, an activation function that was discarded most of the times when a recurrent neural network had to choose between

Model	MSE	MAE
Benchmark	115.815	7.443
Optimized GRU	91.154	6.617
Optimized CNN	98.204	6.902

Table 3.6. Results of the prediction models

this one and a ELU or SELU alternative. The resulting kernel size was of 4 and the pooling layer size was of 1.

The training process can be observed bellow. In Figure 3.20, it is possible to observe that the optimized CNN reaches an optimum solution with regards to the validation score and then starts getting away from it as the epochs go by.

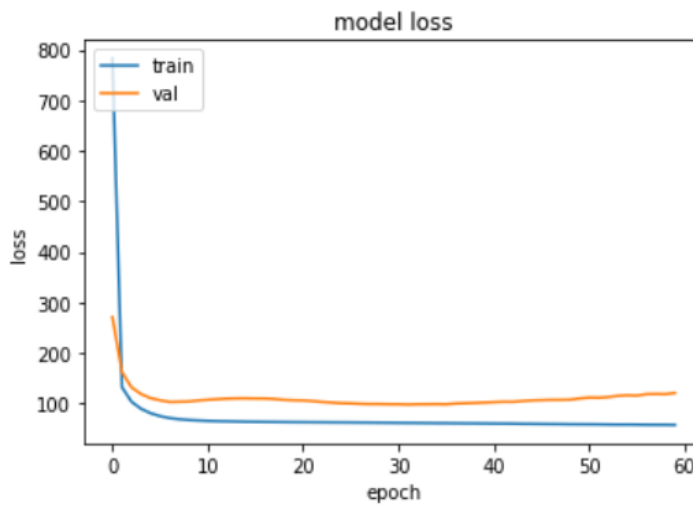


Figure 3.20. First training run of the uni-dimensional convolutional neural network.

Meanwhile, the training error continues decreasing. This is a clear example of over fitting. To prevent this from happening, the epochs were restricted to a number of 30 in the resulting Figure 3.21, no signs of over fitting are appreciated.

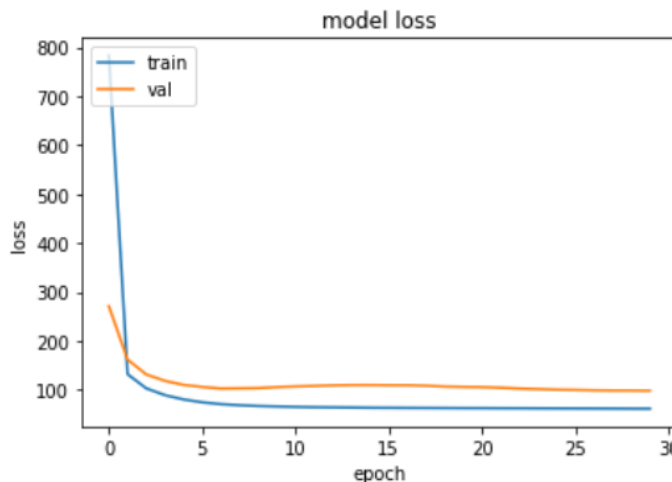


Figure 3.21. Second training run of the uni-dimensional convolutional neural network.

The training of the novel CNN model leaves several points to be commented on. It is interesting to observe the small variability in the results, in a similar fashion to what observed

training the GRU predictor. Once again, a low learning rate is to blame here. At a first glance, the model beats the results of the benchmark as well as the optimized LSTM. The last part of this experiment will be the comparison with the GRU architecture.

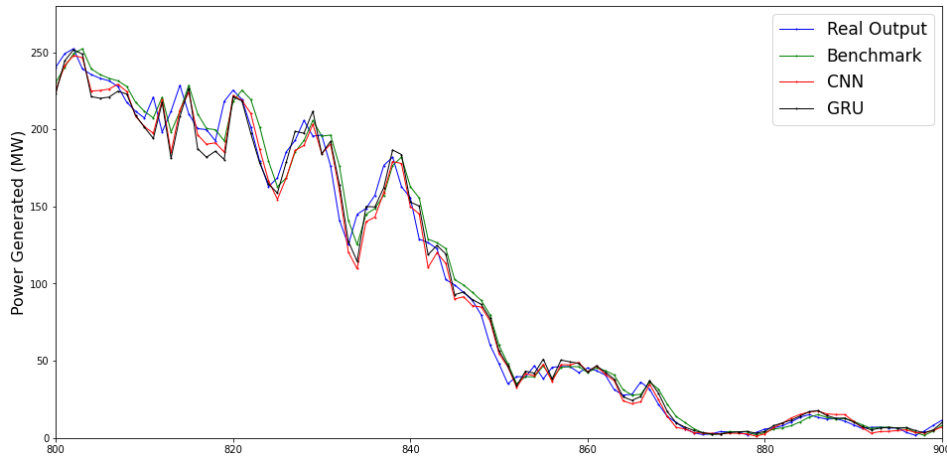


Figure 3.22. Comparison plot between optimized CNN and GRU predictors.

In figure Figure 3.22 several observations can be made. The GRU and the CNN model alternate between a close follow up of the output and a follow up of the one timestep lag. It is only when having a look at the results in Table 3.6 that the best neural network is shown; in this case the GRU holds the first position in both metrics.

The final part of the experiments consists in observing the residuals to determine if there are significant improvements either in the normality of the residuals, the value of the residuals plotted against time or in the partial and total auto-correlation graphs when it comes to the CNN model. For this experiment, the residuals did not follow a normal distribution (Figure 3.23).

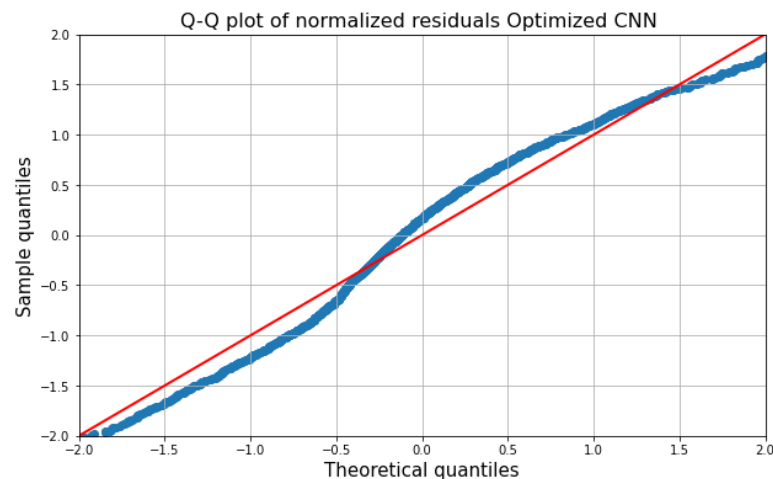


Figure 3.23. Q-Q Plot for the residuals of the CNN predictor.

In Figure 3.23 the actual distribution of residuals in quantiles crosses two times the theoretical line that normal distributions would follow. Together with the average separation of the empirical line, it is safe to conclude the residuals do not correspond with the result expected in case of a normal distribution.

When plotting the residuals together in two adjacent graphs (See Figure 3.24) the images are very similar. The outliers usually revolve around the same areas, however, the GRU model

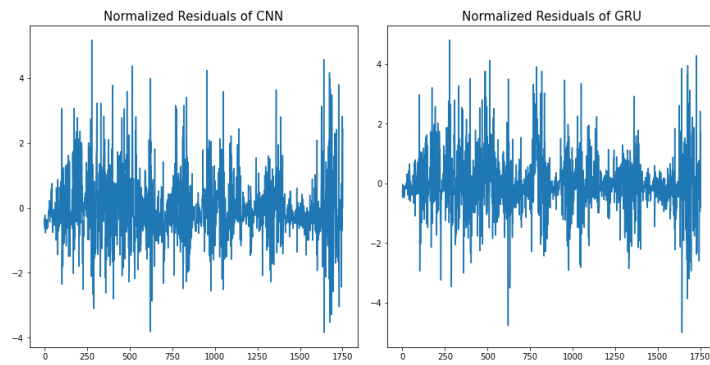


Figure 3.24. Plot of normalized residuals of CNN and GRU models.

seems to behave better in several areas, especially in the last part of the curve. It is important to look at the actual values of the residuals given that the scales are different in each graph.

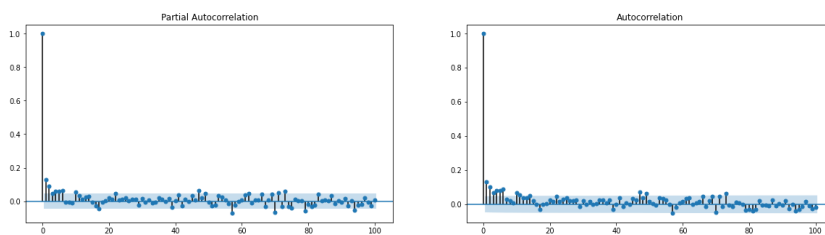


Figure 3.25. Partial and total auto-correlation graphs for the CNN.

With respect to the autocorrelation plots, the CNN predictor obtained a good result, eliminating many significant lags. However, the GRU predictor still seems to better capture the influence of lag and inputs to reconstruct the output (See Figure 3.16).

4

Conclusions

*The end is in the beginning
and lies far ahead.*

Ralph Waldo Ellison (1913–1994)

4.1. Key takeaways

After carrying out a thoughtful and in-depth analysis of the multiple experiments in the project, several conclusions are drawn:

As suspected from the State of the Art section and empirically confirmed, recurrent neural networks are a powerful tool to address timeseries forecasting problems. Their internal memory captures information from past timesteps and allowed them to beat both benchmarks, the one day window and the MLP. The ability to better interpret and replicate the dynamics of the output result in better predictions. A prove that the models replicate the dynamic response of the output is the fact that they do not follow the permanent value of an arbitrary step in the validation dataset. Better dynamics is not the only quality in favour of RNNs, they can also learn how to avoid outliers; however, it is not reliable as not all the outliers are avoided (some are just not detected). This behaviour was observed in a GRU algorithm.

When talking about RNN alternatives, the LSTM is usually the first one mentioned. However, in this project the GRU algorithm outperformed to LSTM algorithm in three out of the four tested cases in the first experiment, a 100% of cases in the second experiment, as well as in the optimization tests.

Alternatively from RNNs, The optimized CNN proved to be a very lightweight model that was trained in under 10 seconds each iteration. It is the model with the second most accurate result and with the fastest training time.

On the activation functions side, the ReLU was surpassed by modified versions, such as the ELU activation function. In the first experiment the ReLU activation function was outperformed by SELU. In the second ELU outperformed ReLU in all cases. The ReLU activation function was never the chose none for any Bayesian optimization except for the CNN.

4.2. Recommendations

- The GRU model should be the first one used for a timeseries forecasting problem given the good results. It provides the most accurate results with smaller datasets and its training is lighter than more complex models, like the LSTM.
- Alternatives to ReLU should always be tested. ReLU is only recommended for very complex networks or CNNs.
- Try convolutional neural networks when dealing with forecasting problems and very large datasets. They deliver an attractive results vs training time trade-off.
- Have a look at partial and total autocorrelation plots when choosing between different neural networks. Apparently similar looking models may have quite different ACF and PACF plots. These graphs help determine whether the results could be further improved using information from previous timesteps.

Sustainable Development Goals

After running the research and deriving the main conclusions, the last step is to address how the findings in this document can be applied to enable and promote the Sustainable Development Goals set by the United Nations.

The goals that fit the most with the characteristics of this project are the following:

- **Goal 3: Good Health and Well-being.** By putting into practice the learnings of this document, better predictive systems can be created, particularly for wind energy forecasting, such as in the use case. This will promote the creation of new wind power stations and, as a by product, will reduce our dependency on fossil fuels and other non-sustainable sources of energy. Limiting the emissions of fuels such as coal has a long lasting impact on the health of the inhabitants of a given area.
- **Goal 4: Quality education.** The end goal of this document was to come up with guidelines for future developments in the field of timeseries forecasting. This project will serve as a reference point for future research and developments, therefore contributing to an enhanced education environment for all.
- **Goal 7: Affordable and clean energy.** The forecasting techniques and architectures discussed in this paper can lead to improvements in the forecasting on energy available, what could result in a better management of it. This will make energy more affordable for all. As the use case revolves around wind energy, it will make it cleaner as well.
- **Goal 8: Decent work and economic growth.** The project promotes the development of machine learning algorithms, which are at the front of innovation and economic growth nowadays. Increasing the research in this highly technological fields will result in the creation of high skilled jobs and economic growth.
- **Goal 9: Industry, innovation and infrastructure.** This project on itself is the results of the effort for continuous improvement and innovation that is in the center of the scientific endeavour. It includes recommendations that have the goal of fostering innovation in the technological and energy industries.

References

- [19] *Lecture 21 | Hopfield Nets and Boltzmann Machines (Part 1)*, Nov. 2019. [Online]. Available: <https://www.youtube.com/watch?v=ZnB8MMjg1mA> (visited on 04/12/2020).
- [AMR14] J. A. Koziol, E. M. Tan, and P. Ren, *Restricted Boltzmann Machines for Classification of Hepatocellular Carcinoma*, Mar. 2014. [Online]. Available: <https://www.hindawi.com/journals/cbj/2014/418069/> (visited on 04/05/2020).
- [Ano18a] Anonymous, *Convolutional Neural Networks (CNN): Step 3 - Flattening - Blogs SuperDataScience - Big Data | Analytics Careers | Mentors | Success*, 2018. [Online]. Available: <https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening> (visited on 04/05/2020).
- [Ano18b] —, *Recurrent Neural Networks (RNN) - The Vanishing Gradient Problem - Blogs SuperDataScience - Big Data | Analytics Careers | Mentors | Success*, en, Library Catalog: www.superdatascience.com, Aug. 2018. [Online]. Available: <https://www.superdatascience.com/blogs/recurrent-neural-networks-rnn-the-vanishing-gradient-problem> (visited on 04/05/2020).
- [Ara18] L. Araujo, *Neural Networks · Artificial Inteligence*, 2018. [Online]. Available: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/neural_networks.html (visited on 04/05/2020).
- [Bhi11] R. Bhiksha, *Index of /~tom/10701_sp11/slides*, 2011. [Online]. Available: https://www.cs.cmu.edu/~tom/10701_sp11/slides/ (visited on 04/05/2020).
- [Bhi17] —, *Index of /~bhiksha/courses/deeplearning/Fall.2015/slides*, 2017. [Online]. Available: <https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2015/slides/> (visited on 04/05/2020).
- [Bro19] J. Brownlee, *How to Avoid Exploding Gradients With Gradient Clipping*, Feb. 2019. [Online]. Available: <https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/> (visited on 04/05/2020).
- [Bus18] V. Bushaev, *Adam — latest trends in deep learning optimization*. en, Library Catalog: towardsdatascience.com, Oct. 2018. [Online]. Available: <https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c> (visited on 04/05/2020).
- [Chu+14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”, *arXiv:1412.3555 [cs]*, Dec. 2014, arXiv: 1412.3555. [Online]. Available: <http://arxiv.org/abs/1412.3555> (visited on 04/05/2020).
- [Dan17] L. Danqing, *A Practical Guide to ReLU - Danqing Liu - Medium*, 2017. [Online]. Available: <https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7> (visited on 04/05/2020).
- [De 18] L. De Mol, *Turing Machines (Stanford Encyclopedia of Philosophy)*, Sep. 2018. [Online]. Available: <https://plato.stanford.edu/entries/turing-machine/> (visited on 04/05/2020).

References

- [Den97] S. Dennis, *The Hopfield Network: Descent on an Energy Surface*, 1997. [Online]. Available: <http://staff.itee.uq.edu.au/janetw/cmc/chapters/Hopfield/> (visited on 04/12/2020).
- [Der17] A. Dertat, *Applied Deep Learning - Part 3: Autoencoders*, en, Library Catalog: towardsdatascience.com, Oct. 2017. [Online]. Available: <https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798> (visited on 04/05/2020).
- [Din+17] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio, “Sharp Minima Can Generalize For Deep Nets”, *arXiv:1703.04933 [cs]*, May 2017, arXiv: 1703.04933. [Online]. Available: <http://arxiv.org/abs/1703.04933> (visited on 04/05/2020).
- [Dos19a] N. Doshi, *Deep Learning Best Practices (1) — Weight Initialization*, en, Library Catalog: medium.com, May 2019. [Online]. Available: <https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94> (visited on 04/05/2020).
- [Dos19b] —, *Deep Learning Best Practices (1) — Weight Initialization*, en, Library Catalog: medium.com, May 2019. [Online]. Available: <https://medium.com/usf-msds/deep-learning-best-practices-1-weight-initialization-14e5c0295b94> (visited on 04/05/2020).
- [Dra19] D. Draxler, *Generalized Huber Regression - Towards Data Science*, Jun. 2019. [Online]. Available: <https://towardsdatascience.com/generalized-huber-regression-505afaff24c> (visited on 04/05/2020).
- [Gan+15] Z. Gan, R. Henao, D. Carlson, and L. Carin, “Learning Deep Sigmoid Belief Networks with Data Augmentation”, en, p. 9, 2015.
- [Góm18] R. Gómez, *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names*, May 2018. [Online]. Available: https://gombro.github.io/2018/05/23/cross_entropy_loss/ (visited on 04/05/2020).
- [Gup20] D. Gupta, *Fundamentals of Deep Learning - Activation Functions and their use*, Library Catalog: www.analyticsvidhya.com Section: Deep Learning, Jan. 2020. [Online]. Available: <https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/> (visited on 04/12/2020).
- [Gyl18] R. Gylberth, *Continuing on Adaptive Method: ADADELTA and RMSProp*, en, Library Catalog: medium.com, Jul. 2018. [Online]. Available: <https://medium.com/konvergen/continuing-on-adaptive-method-adadelta-and-rmsprop-1ff2c6029133> (visited on 04/05/2020).
- [Ha16] D. Ha, *Generating Large Images from Latent Vectors* |, Apr. 2016. [Online]. Available: <https://blog.otoro.net/2016/04/01/generating-large-images-from-latent-vectors/> (visited on 04/12/2020).
- [Her18] J. A. Hertz, *Introduction To The Theory Of Neural Computation*, en. CRC Press, Mar. 2018, Google-Books-ID: NwpQDwAAQBAJ.
- [Hin+12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors”, *arXiv:1207.0580 [cs]*, Jul. 2012, arXiv: 1207.0580. [Online]. Available: <http://arxiv.org/abs/1207.0580> (visited on 04/05/2020).
- [Jai19] P. Jain, *Complete Guide of Activation Functions - Towards Data Science*, 2019. [Online]. Available: <https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044> (visited on 04/05/2020).
- [Jeo19] J. Jeong, *The Most Intuitive and Easiest Guide for Convolutional Neural Network*, Jan. 2019. [Online]. Available: <https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480> (visited on 04/05/2020).

- [Jus17] JustinB, *Introduction to Perceptron: Neural Network*, en-US, Library Catalog: blog.knoldus.com Section: ML, AI and Data Engineering, Sep. 2017. [Online]. Available: <https://blog.knoldus.com/introduction-to-perceptron-neural-network/> (visited on 04/05/2020).
- [Kar17] A. Karpathy, *A Peek at Trends in Machine Learning*, en, Library Catalog: medium.com, Apr. 2017. [Online]. Available: <https://medium.com/@karpathy/a-peek-at-trends-in-machine-learning-ab8a1085a106> (visited on 04/05/2020).
- [Kha18] R. Khandelwal, *Deep learning — Deep Boltzmann Machine (DBM)*, en, Library Catalog: medium.com, Dec. 2018. [Online]. Available: <https://medium.com/datadriveninvestor/deep-learning-deep-boltzmann-machine-dbm-e3253bb95d0f> (visited on 04/05/2020).
- [Kha19] —, *Overview of different Optimizers for neural networks*, en, Library Catalog: medium.com, Feb. 2019. [Online]. Available: <https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3> (visited on 04/05/2020).
- [Kir06] K. G. Kirby, “A Tutorial on Helmholtz Machines”, en, p. 26, 2006.
- [Le15] Q. V. Le, “A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks”, en, p. 20, 2015.
- [Leo18] J. Leonel, *Multilayer Perceptron - Jorge Leonel - Medium*, 2018. [Online]. Available: <https://medium.com/@jorgesleonel/multilayer-perceptron-6c5db6a8dfa3> (visited on 04/05/2020).
- [Lev02] I. Levner, “Quantum Computing (Recent Developments in Quantum Computational Intelligence)”, 2002.
- [LGS16] B. P. Lokhande, S. S. Gharde, and M. E. Student, “Video Classification with Recurrent Neural Network”, en, vol. 4, no. 1, p. 8, 2016.
- [LJY17] F.-F. Li, J. Johnson, and S. Yeung, “Lecture 10: Recurrent Neural Networks”, en, p. 105, 2017.
- [LJY19] —, “Administrative: Project Proposal”, en, p. 80, 2019.
- [Mal18] K. Maladkar, *Types Of Activation Functions in Neural Networks and Rationale behind it*, en-US, Library Catalog: analyticsindiamag.com Section: Developers Corner, Jan. 2018. [Online]. Available: <https://analyticsindiamag.com/most-common-activation-functions-in-neural-networks-and-rationale-behind-it/> (visited on 04/12/2020).
- [McN17] D. McNeela, *The Universal Approximation Theorem for Neural Networks*, 2017. [Online]. Available: https://mcneela.github.io/machine_learning/2017/03/21/Universal-Approximation-Theorem.html (visited on 04/05/2020).
- [Mem16] H. Memo, *Rectified Linear Unit (ReLU) | An old brother's memo*. Jul. 2016. [Online]. Available: [https://sohero.github.io/2016/07/14/Rectified%20Linear%20Unit%20\(ReLU\)/](https://sohero.github.io/2016/07/14/Rectified%20Linear%20Unit%20(ReLU)/) (visited on 04/12/2020).
- [MSW20] J. McGonagle, G. Shaikouski, and C. Williams, *Backpropagation | Brilliant Math & Science Wiki*, 2020. [Online]. Available: <https://brilliant.org/wiki/backpropagation/> (visited on 04/05/2020).
- [Ngu19] M. Nguyen, *Illustrated Guide to LSTM's and GRU's: A step by step explanation*, en, Library Catalog: towardsdatascience.com, Jul. 2019. [Online]. Available: <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21> (visited on 04/05/2020).
- [PG17] J. Patterson and A. Gibson, *Major Architectures of Deep Networks - Deep Learning*, en, Library Catalog: www.oreilly.com, 2017. [Online]. Available: <https://www.oreilly.com/library/view/deep-learning/9781491924570/ch04.html> (visited on 04/05/2020).

References

- [PGN18] S. Pal, S. Ghosh, and A. Nag, “Sentiment Analysis in the Light of LSTM Recurrent Neural Networks”, *International Journal of Synthetic Emotions*, vol. 9, pp. 33–39, Jan. 2018.
- [Póc17] B. Póczos, *Advanced Introduction to Machine Learning, CMU-10715*, Sep. 2017. [Online]. Available: <https://www.cs.cmu.edu/~epxing/Class/10715/lectures/DeepArchitectures.pdf>.
- [Pra19] A. Prasad, *Conditional Random Fields Explained - Towards Data Science*, 2019. [Online]. Available: <https://towardsdatascience.com/conditional-random-fields-explained-e5b8256da776> (visited on 04/05/2020).
- [Raj19] B. Raj, *11-785 Deep Learning*, 2019. [Online]. Available: <https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Spring.2019/archive-f19/www-bak11-22-2019/> (visited on 04/05/2020).
- [Rat18] S. Rathor, *Simple RNN vs GRU vs LSTM :- Difference lies in More Flexible control*, en, Library Catalog: medium.com, Jun. 2018. [Online]. Available: <https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57> (visited on 04/05/2020).
- [Res20] M. S. Researcher PhD, *Neural Network Optimization*, en, Library Catalog: towardsdatascience.com, Jan. 2020. [Online]. Available: <https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0> (visited on 04/05/2020).
- [RKK18] S. J. Reddi, S. Kale, and S. Kumar, “ON THE CONVERGENCE OF ADAM AND BEYOND”, en, p. 23, 2018.
- [Rud16] S. Ruder, *An overview of gradient descent optimization algorithms*, en, Library Catalog: ruder.io, Jan. 2016. [Online]. Available: <https://ruder.io/optimizing-gradient-descent/> (visited on 04/05/2020).
- [RZL17] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for Activation Functions”, *arXiv:1710.05941 [cs]*, Oct. 2017, arXiv: 1710.05941. [Online]. Available: <http://arxiv.org/abs/1710.05941> (visited on 04/05/2020).
- [Sal97] J. Salerno, “Using the particle swarm optimization technique to train a recurrent neural model”, in *Proceedings Ninth IEEE International Conference on Tools with Artificial Intelligence*, ISSN: 1082-3409, Nov. 1997, pp. 45–49.
- [Say18] A. Sayantini, *Restricted Boltzmann Machine Tutorial | Deep Learning Concepts*, en-US, Library Catalog: www.edureka.co Section: Artificial Intelligence, Nov. 2018. [Online]. Available: <https://www.edureka.co/blog/restricted-boltzmann-machine-tutorial/> (visited on 04/05/2020).
- [Sch19] D. Schwartz, “Variational Autoencoders”, en, p. 76, Nov. 2019.
- [Ser17] S. Serengil, *Softplus as a Neural Networks Activation Function*, en-US, Library Catalog: sefiks.com Section: Machine Learning, Aug. 2017. [Online]. Available: <https://sefiks.com/2017/08/11/softplus-as-a-neural-networks-activation-function/> (visited on 04/05/2020).
- [Sha17] B. Shaver, *A Zero-Math Introduction to Markov Chain Monte Carlo Methods*, 2017. [Online]. Available: <https://towardsdatascience.com/a-zero-math-introduction-to-markov-chain-monte-carlo-methods-dcba889e0c50> (visited on 04/05/2020).
- [Sin17] A. Singh Walia, *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent*, Jun. 2017. [Online]. Available: <https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f> (visited on 04/05/2020).
- [Ste19a] M. Stewart, *Intermediate Topics in Neural Networks - Towards Data Science*, Jun. 2019. [Online]. Available: <https://towardsdatascience.com/comprehensive-introduction-to-neural-network-architecture-c08c6d8e5d98> (visited on 04/05/2020).

- [Ste19b] —, *Simple Introduction to Convolutional Neural Networks*, Feb. 2019. [Online]. Available: <https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac> (visited on 04/12/2020).
- [Uni20] Uniqtech, *Understand the Softmax Function in Minutes*, en, Library Catalog: medium.com, Jan. 2020. [Online]. Available: <https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d> (visited on 04/05/2020).
- [V17] A. S. V, *Understanding Activation Functions in Neural Networks*, en, Library Catalog: medium.com, Mar. 2017. [Online]. Available: <https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0> (visited on 04/05/2020).
- [WYZ15] Y. Wang, H. Yao, and S. Zhao, “Auto-Encoder Based Dimensionality Reduction”, *Neurocomputing*, vol. 184, Nov. 2015.

