Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

# Power Amplifier Behavioral Modeling Through Convolutional Neural Networks

Autor: Gonzalo Chávez Romero

Tutores: Juan Antonio Becerra González, María José Madero Ayora

**Dpto. Teoría de la Señal y Comunicaciones**
**Escuela Técnica Superior de Ingeniería**
**Universidad de Sevilla**

Sevilla, 2020

Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

# Power Amplifier Behavioral Modeling Through Convolutional Neural Networks


Autor:

Gonzalo Chávez Romero

Tutores:

Juan Antonio Becerra González

Profesor Sustituto Interino

María José Madero Ayora

Profesora Titular

Dpto. Teoría de la Señal y Comunicaciones

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla



Sevilla, 2020

Trabajo Fin de Grado: Power Amplifier Behavioral Modeling Through Convolutional Neural Networks

Autor: Gonzalo Chávez Romero
Tutores: Juan Antonio Becerra González, María José Madero Ayora

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

# Agradecimientos

En mi primer lugar, me gustaría destacar lo especialemnte priviliegiado que me siento de tener familiares que me han apoyado no solo moralmente durante el transcurso de la carrera, sino también académicamente. Es mucho más fácil afrontar los complicados retos de la carrera sabiendo que los tengo a mi lado y por ello estoy muy agradecido.

A continuación, me gustaría destacar la ayuda y compañía de mis compañeros de clase, sobre todo Vicente, Manu, Juan Antonio, Rafa y Luis. Es increíble lo amena que se hace la carrera con personas tan inteligentes y agradables a mi alrededor. Todos y cada uno de ellos han hecho la rutina del día a día realmente cómoda y me llevo muy buenas memorias con cada uno de ellos, como los desayunos con Jota o el día que Vicente vino con un zapato de cada color. Realmente me entristece pensar que es muy poco probable que vuelva a verles día a día y sin duda son la razón por la que echaré de menos este periodo universitario.

Por último, me gustaría agradecer al profesor Juan Antonio Becerra su ayuda y motivación durante la carrera. Ahora mismo soy un apasionado por el Machine Learning y probablemente sea gracias a él. Antes de conocerle y ponerme a trabajar en este TFG jamás me habría planteado investigar en el campo de la Teoría de Señal y Comunicaciones, ya que era el que menos me apasionaba de toda la carrera, pero aquí estamos, realizando un TFG sobre predistorsión digital, todo gracias a Juan Antonio, que me hizo ver que aunque mi primera impresión no fuera la mejor, realmente tenía líneas de investigación muy interesantes.

# Abstract

In this document we present an approach to characteristic modeling of power amplifiers using convolutional neural networks. Radio Frequency Power Amplifiers suffer from distortions when they approach high efficiency levels, and by having a model of how the amplifier is going to distort radiofrequency signals, we can pre-distort the signal before hand with the inverse characteristic to mitigate the distortion the amplifier applies.

To do so, we propose a convolutional neural networks. Neural network behave like universal function approximators given enough data. We present that this architecture is fairly successful when modeling nonlinearities on three different amplifiers while keeping its complexity and number of parameters to reasonable levels.

Lastly, we replicate results on similar research showing that feeding certain transformations of the inputs to the network leads to better modeling of the amplifier without raising the number of parameters much. Because of the inherent performance of convolutional neural networks, we can augment the input data both memory-wise and order-wise without adding many parameters to the model.

# Index

# Notation

| | |
|---|---|
| $a$ | Scalar |
| $\mathbf{v}$ | Vector |
| $\mathbf{A}$ | Matrix |
| $A$ | Tensor |
| $\mathbb{R}$ | Set containing all the real numbers |
| $\{0, 1, ..., n\}$ | Set containing all the numbers between 0 and $n$ |
| $\det(\mathbf{A})$ | Determinant of matrix $\mathbf{A}$ |
| $\mathbf{A}^\top$ | Transpose of $\mathbf{A}$ |
| $\mathbf{A}^{-1}$ | Inverse of $\mathbf{A}$ |
| $a_i$ | Element $i$ of vector $\mathbf{a}$ |
| $A_{i,j}$ | Element $i$, $j$ of tensor A |
| $A_{i,j,k}$ | Element $i$, $j$, $k$ of 3D tensor A |
| $A_{i,:}$ | Row $i$ of tensor A |
| $A_{:,j}$ | Column $j$ of tensor A |
| $A_{:,:,k}$ | 2D slice $k$ of tensor A |
| $\frac{\partial y}{\partial x}$ | Partial derivative of $y$ with respect to $x$ |
| $\frac{dx}{dy}$ | Derivative of $y$ with respect to $x$ |
| $\Delta_x y$ | Gradient of $y$ with respect to $x$ |
| $\Delta_{\mathbf{X}} y$ | Matrix derivatives of $y$ with respect to $\mathbf{X}$ |
| $\mathbf{J}$ | Jacobian Matrix |
| $\|\|\mathbf{x}\|\|_p$ | $L^p$ norm of $\mathbf{x}$ |
| $\|\|\mathbf{x}\|\|$ | $L^2$ norm of $\mathbf{x}$ |
| $x^{(i)}$ | The $i$-th example of a dataset |
| $MSE$ | Mean square error |
| $RMSE$ | Root Mean square error |
| $\leqslant$ | Less or equal to |
| $\geqslant$ | Greater or equal to |
| e | number e |
| $\mathbb{R}$e | Real component |
| $\mathbb{I}$m | Imaginary component |
| sen | Sine |
| tg | Cosine |
| $\sin^y x$ | Sine of $x$ to the $y$-th power |
| $\cos^y x$ | Cosine of $x$ to the $y$-th power |
| $ReLU$ | Rectified Linear Unit Function |

# 1 Introduction

Wireless communication are growing quickly both in complexity and number of application, with users demanding near-perfect transmission and reception of data. Clients demand higher datarates for their streaming and video services and range to provide content all around the globe. However, the signal propagation distance depends on the frequency and power of those signals. Because of that, wireless signals often fall into the Radio Frequency range, from a few hundred MHz to a few GHz, since these frequencies provide sufficient range with a reasonable power that allows mobile systems to receive them. Radio Frequency Power Amplifiers become then a key component in wireless communication, boosting a signal strength to a desired power level to reach the receiving system.

Amplifiers are the main consumer of power not only on base station but also on mobile devices, so we want them to be as efficient as possible, that is, we want to make the best use of the power provided to the amplifier, but often, higher efficiency comes with distortions on the output signal. Since amplifiers operate the most efficient while in the saturation zone, they start introducing distortions or *nonlinearities* on the output signal, which leads to errors at the receiving end.

To mitigate this distortions, digital pre-distortion systems are designed to allow amplifiers to behave as linearly and thus efficiently as possible. Neural networks are one of the most promising techniques on digital pre-distortion, because of their flexibility and the vast amount of research on them, bringing groundbreaking techniques to better model and optimize them at a really high pace. To be able to pre-distort signals, we need models of the amplifiers that tells us how and why the aforementioned distortions happen, this models are called behavioral models.

On Chapter 2, we introduce the neural network concept and most of the mathematical tools we are going to use to design, create and test our neural network. On Chapter 3, we explain the problem of behavioral modeling and digital pre-distortion and previous work on neural networks as behavioral models. On Chapter 4 we present the experiemnts we are going to perform and the code used in them, while in Chapter 5 we provide some insights on the results of the experiments and the neural network learning process. Then, on the last Chapter, we write the conclusions from our investigation and add two appendices to provide more knowledge on our two main tools we used to model each experiment.

## 1.1 Objectives

On this document, we investigate on the use of one particular neural network architecture, called Convolutional Neural Network, applying it as a behavioral model of a variety of amplifiers with different power at their input signal. We test here if the flexibility of similarly complex networks can

provide us with an efficient pre-distortion tool. This application is motivated by recent success of Convolutional Neural Networks on other time-related and forecasting problems, and their simplicity, scalability and paralellization potential compared to Recurrent Neural Networks make them a very attractive option for behavioral modeling.

We test the networks performance with a variety of real-world, lab-acquired signals. We also explore which architecture performs the best for each experiment, confirming previous research on related topics.

## 1.2 Methods and Materials

Input and output pairs of signals are obtained using amplifiers at the lab or on online laboratories such as the one provided by Chalmers University of Technology. Neural networks architecture are defined using Python and a variety of machine networks libraries, mostly Pytorch, SKLearn and Optuna and experiments run on a Google Colab environment, which provides us with high-end Graphics Processing Units for quicker computations and optimization loops.

# 2 Neural Networks

This section aims to explain what neural networks are and what problems they aim to solve. Neural networks are widely used nowadays for a variety of purposes, it can be either self driving cars, image recognition, natural language processing, stock market prediction and a great amount of different applications are discovered each year for neural networks, but what are they, really?

Neural Networks are certainly not the first ever learning algorithm created, but since they have shown to be flexible in a variety of different scenarios, they are raising a great interest in the scientific community. Neural networks' first developments come motivated by neuroscientific research that tried to model the brain behaviour as early as 1943 [40]. Even though neuroscience is regarded as a source of inspiration for current neural networks, it does not serve as its guide, and modern research on the machine learning field diverges from the neuroscience path. Neural networks have however become an important technology fairly recently, and that has to do with two main factors: the amount of data available to us and the computational power, specifically in current GPUs (Graphic Processing Units).

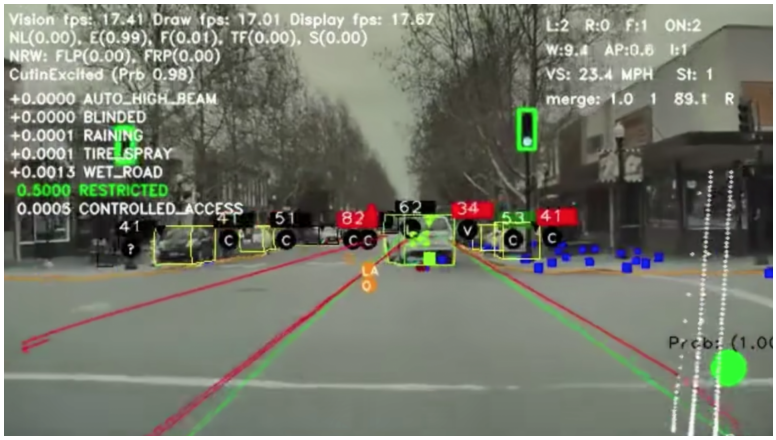



**Figure 2.1** Tesla's Autopilot interpretation of its environment. Here, the input is the different images captured by the cameras and the networks output is represented by the different lines painted over the road identifying each object and path .

We can then think of neural networks as universal function approximators, that is, with enough data and computational power, a neural net will be able to find a function that maps from a given input to a desired output, not matter the dimensionality and complexity of the problem at hand. Now,

if we come back to one of the aforementioned examples, like the self driving car, we can see when the neural net comes into play: the network maps from a number of inputs (cameras and a number of sensors) to a desired output (steer the driving wheel a number of degrees, activate the brakes, increase acceleration and so on). Of course, the network at hand here becomes fairly complicated, as it has more parts and different layers than the simple one used in this document, but it exemplifies the level of abstraction a neural net can achieve and the vast amount of situation where we can make use of an universal function approximator.

## 2.1 Overview of Neural Networks

The quintessential architecture of neural networks is called Feedforward Neural Net of Multilayer Perceptron. They are called feedforward because the input flows through the different functions of the network to the output, without any feedback of the network outputs to itself (unlike, for example, a Recurrent Neural Net, outside of the scope of this document but widely used nonetheless).

As a quick note, we will be using the word tensor throughout the document. Tensor is the abstraction of the matrix or vectors, that is, for example, a vector is a 1-dimensional tensor, and a matrix is a 2 dimensional tensor. Further on, we will talk about 3-dimensional tensors as inputs for our neural networks.

The building block of neural networks is called a neuron [18]. Neurons are small components that take a number of inputs, sum them and apply a defined function.

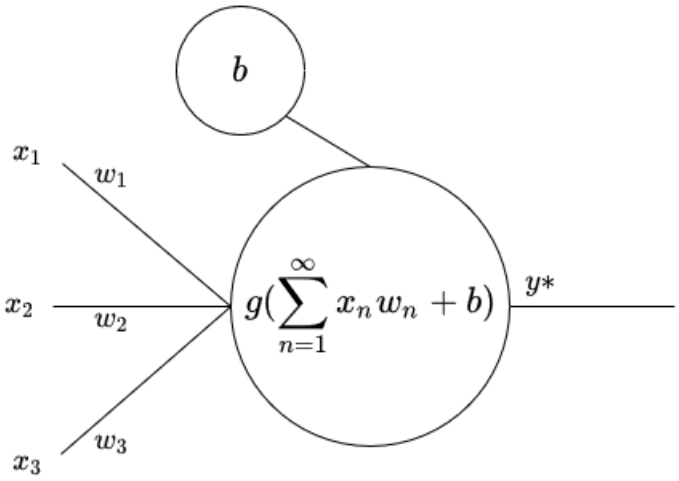



**Figure 2.2** Diagram of a single neuron. The inputs are represented by the **x** vector and the output is represented by the $y*$ scalar. Any given neuron can have as many inputs as desired. The **w** vector represents the weights associated with the neuron, and $b$ represents the bias of the neuron .

For a given neuron, we can express the output as:

$$y* = g(\sum_{n=1}^{\infty} x_n w_n + b)$$

Where g(x) is called Activation Function, and they serve a double purpose: to prevent gradient vanishing and gradient exploding, which we'll explain later in the learning section, and to introduce non-linearity in the network. The parameter $b$ here is called bias, and gives each neuron a constant regulated by the weight of the corresponding connection in case we need to shift the activation function to produce better results.

Neurons are arranged in layers inside a neural network, and these layers are connected by synapses governed by weights. The diagram below exemplifies this:

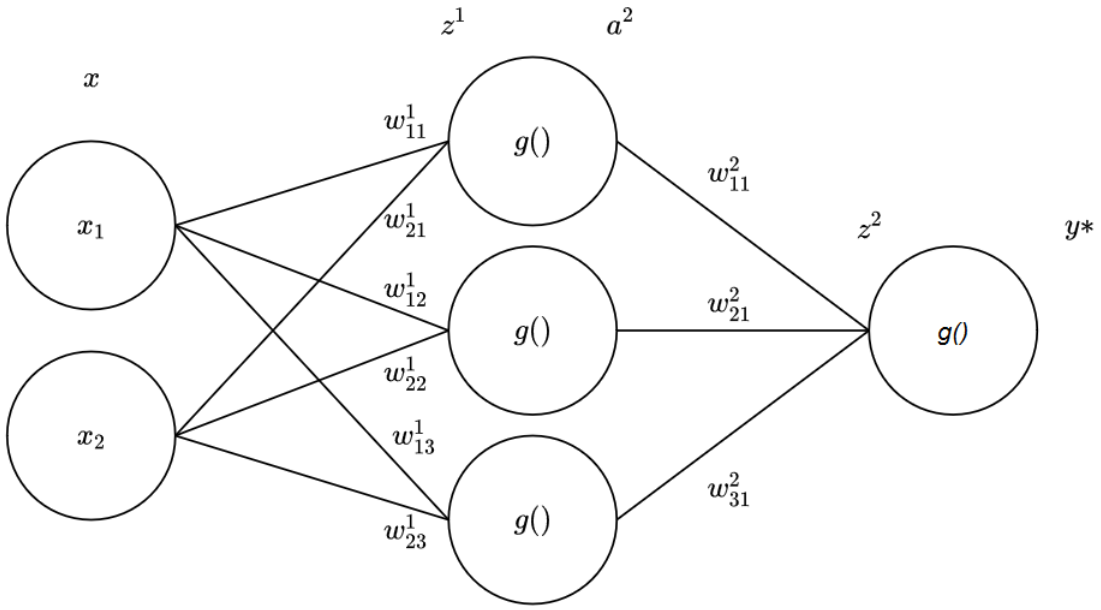



**Figure 2.3** Diagram of a simple neural network. All the neurons are connected by the weights parameters, the bias $b$ here is removed for clarity .

Where the parameters $w^l_{nm}$ are the weights. Each connection on a neural network has a number associated with it called weight. Weights are multiplied by the input on that connection and are the parameters we will adjust during the training process of a neural network. Each element the vector $z^1$ represents each input of the corresponding neuron:

$$z^1 = \begin{bmatrix} x_1 w^1_{11} + x_2 w^1_{21} + b w_{b1} \\ x_1 w^1_{12} + x_2 w^1_{22} + b w_{b2} \\ x_1 w^1_{13} + x_2 w^1_{23} + b w_{b3} \end{bmatrix}$$

And $a^2$ represents the vector where each element is the output of each neuron:

$$a^2 = \begin{bmatrix} g(z^1_1) \\ g(z^1_2) \\ g(z^1_3) \end{bmatrix}$$

The network above represents a very simple feedforward neural network. More complicated variations exist, where different kind of layers and neurons are used, such as Recurrent Neural Networks using LSTM (Long Short Term Memory) [24] or Convolutional Neural Networks [34].

## 2.2 Forward propagation on Neural Networks

When we provide neural networks with an input and it makes the computation necessary to produce and output is called a Forward Pass, because the information is flowing forward from input to output. On this section we are going to explain the computations a neural net follows to produce its outputs.

First, the input to every neural network is in the form of a multidimensional matrix or a tensor. In most circumstances the information is going to take the form of a 2D matrix, where each row represents an example and each column represents the information about the abovementioned example we want the network to learn, also called 'features'. Here's an example of a dataset of different NBA players where each row represents a different player and each column represent the statistics of each player:

| | League | Season | Stage | Player | Team | GP | MIN | FGM | FGA | 3PM | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NBA | 2009 - 2010 | Regular_Season | Kevin Durant | OKC | 82 | 3239.3 | 794 | 1668 | 128 | ... |
| 1 | NBA | 2009 - 2010 | Regular_Season | LeBron James | CLE | 76 | 2965.6 | 768 | 1528 | 129 | ... |
| 2 | NBA | 2009 - 2010 | Regular_Season | Dwyane Wade | MIA | 77 | 2792.4 | 719 | 1511 | 73 | ... |
| 3 | NBA | 2009 - 2010 | Regular_Season | Dirk Nowitzki | DAL | 81 | 3038.8 | 720 | 1496 | 51 | ... |
| 4 | NBA | 2009 - 2010 | Regular_Season | Kobe Bryant | LAL | 73 | 2835.4 | 716 | 1569 | 99 | ... |

5 rows × 31 columns

**Figure 2.4** Simple Dataset of the NBA players. The Dataset contains 31 columns, that is, 31 features for each player. .

We could use the above dataset for, for example, train a network to predict the number of 3-pointers a player is going to score given the different information on the dataset. For our example of a forward computation we are going to use however a simple, 2-feature input and a neural net with only one hidden layer and output layer, as the network on Figure 1.3.

Given an example input like:

| $x_1$ | $x_2$ | y |
|---|---|---|
| 1 | 6 | 50 |
| 2 | 7 | 78 |
| 3 | 5 | 15 |

Here, $x_1$ and $x_2$ are our input vectors or features and we want to train the network to predict $y$ given $x_1$ and $x_2$. Now, we can write the matrix resulting of the first layer computations as:

$$z^1 = \begin{bmatrix} 1 & 6 \\ 2 & 7 \\ 3 & 5 \end{bmatrix} \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \end{bmatrix} = \begin{bmatrix} 1w^1_{11} + 6w^1_{21} & 1w^1_{12} + 6w^1_{22} & 1w^1_{13} + 6w^1_{23} \\ 2w^1_{11} + 7w^1_{21} & 2w^1_{12} + 7w^1_{22} & 2w^1_{13} + 7w^1_{23} \\ 3w^1_{11} + 5w_{21^1} & 3w^1_{12} + 5w^1_{22} & 3w^1_{13} + 5w^1_{23} \end{bmatrix}$$

Note that $\mathbf{Z^1}$ is now a matrix instead of a vector, since we are feeding several examples at the same time. The number of examples (rows) on a forward computation is called batch size. The batch size is one of the many parameters that we have to adjust empirically to find the best results. This

parameters are called hyperparameters. Now, if we name each matrix and ignore the bias, we can write:

$$\mathbf{X} \cdot \mathbf{W^1} = \mathbf{Z^1}$$

Where $\mathbf{Z^1}$ has as many rows as the input matrix and as many columns as the number of neurons on the hidden layer. Then, following this notation, we can write the 4 equations that govern the forward pass on this neural network:

$$\mathbf{Z^1} = \mathbf{X} \cdot \mathbf{W^1}$$
$$\mathbf{A^1} = g(\mathbf{Z^1})$$
$$\mathbf{Z^2} = \mathbf{A^1} \cdot \mathbf{W^2}$$
$$\mathbf{Y}* = g(\mathbf{Z^2})$$

Where $g()$ is the activation function present on each neuron. One of the most widely used activation function is the ReLU function:

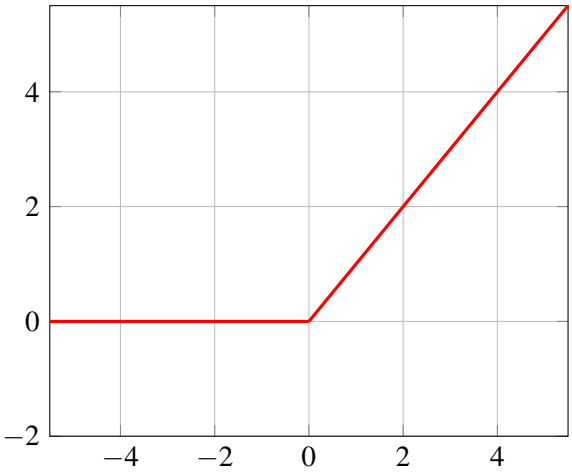



**Figure 2.5** ReLU activation function.

Other activation functions are represented below. What activation function we use is again one of the hyperparameters of the neural network. We may have to try different activation functions for the neurons until we arrive at the best performing one.

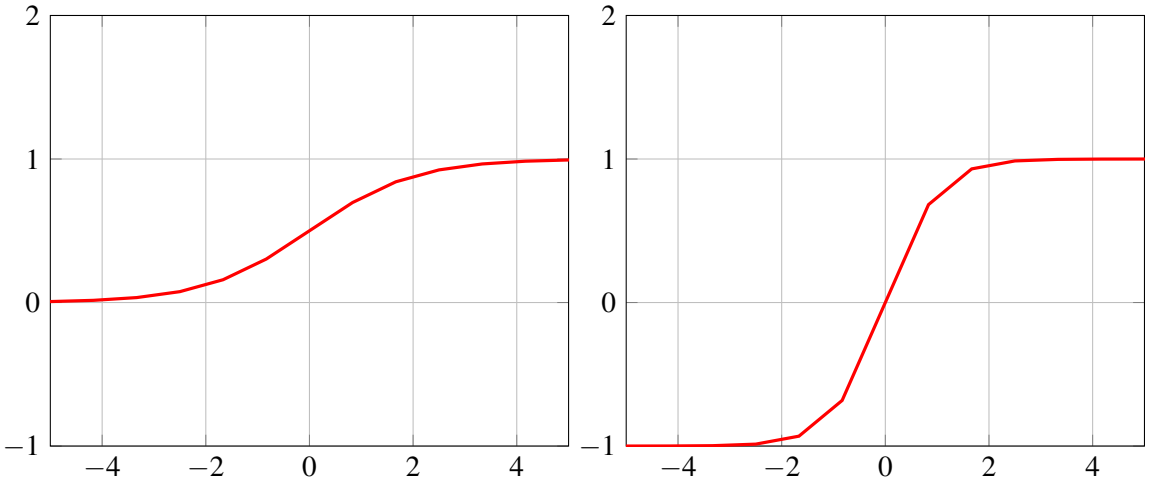



**Figure 2.6** Left: Sigmoid activation function. Right: Tanh activation function.

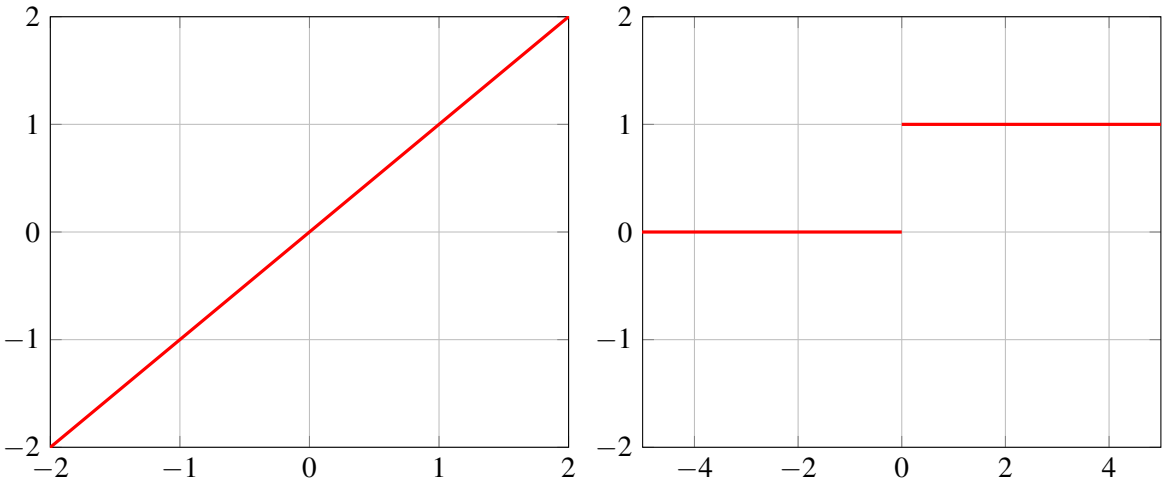



**Figure 2.7** Left: Identity activation function. Right: Binary Step activation function.

When we first create the neural network, the weights are randomly initialized, since the network doesn't know yet the importance of each feature. As we train the network, the values of these weights are going to change accordingly to produce better results. To train the network, we need a function to measure the error of a forward pass of the neural network to know how to adjust the weights. This function is called the Cost function (also called Loss function). One Loss function we can use is the Root Mean Squared Error (RMSE) to compute the error between the desired output and the actual output of the net. RMSE has the following formula:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - y_i*)^2}{n}}$$

That was the simple walk-through of the neural net forward pass, and now we have a way to measure the error of its output. We mentioned before that activation functions introduce non-linearity on the neural networks outputs. To expand on this explanation, we can consider an example where the activation function is the linear function as in Figure 1.7. Given the activation function, we can express the output of the neural network like this:

$$\mathbf{Z^1} = \mathbf{W^1} \cdot \mathbf{X} + b^1$$
$$\mathbf{A^2} = \mathbf{Z^1}$$
$$\mathbf{Z^2} = \mathbf{W^2} \cdot \mathbf{A^2} + b^2$$
$$\mathbf{y}* = \mathbf{Z^2}$$

Then:

$$\mathbf{y}* = \mathbf{W^2} \cdot [\mathbf{W^1} \cdot \mathbf{X} + b^1] + b^2$$
$$\mathbf{y}* = [\mathbf{W^2} \cdot \mathbf{W^1}] \cdot \mathbf{X} + [\mathbf{W^2} \cdot b^1] + b^2$$

Let:

$$\mathbf{W} = [\mathbf{W^2} \cdot \mathbf{W^1}]$$
$$b = [\mathbf{W^2} \cdot b^1] + b^2$$

Then the output is:

$$\mathbf{y}* = \mathbf{W} \cdot \mathbf{X} + b$$

Which is again a linear function. We can say then that if no activation functions are going to be used, we might as well not add more layers to a neural net, since all the layers are going to behave the same way, transforming the data on a linear fashion and not providing any further insight on the data at hand, since the addition of two linear functions is another linear function. Our network couldn't then approximate any non-linear function, such as the one below, making the network a pretty weak function approximator. By adding nonlinear activation function, we allow the neural network to transform the input data on nonlinear representations to gain insight on the input/output relationship. A really good example of this is the XOR function approximation step-by-step example on [18]

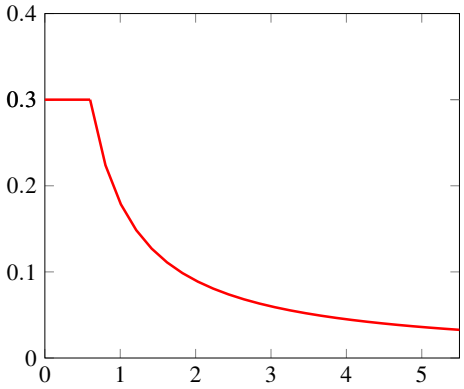



**Figure 2.8** Example of a non-linear function that a neural network with no activation function couldn't approximate.

## 2.3 Backpropagation and Gradient Descent

Now that we have the difference between the output we want and the output the network gave us, we need to adjust the weights of the network to reduce the error. To calculate how to change the weights to produce outputs with less error we use backpropagation or backprop [52]. To explain how backprop works, we can plot the RMSE against one of the parameters, for example $w^1_{11}$, for the given input $\mathbf{X}$:

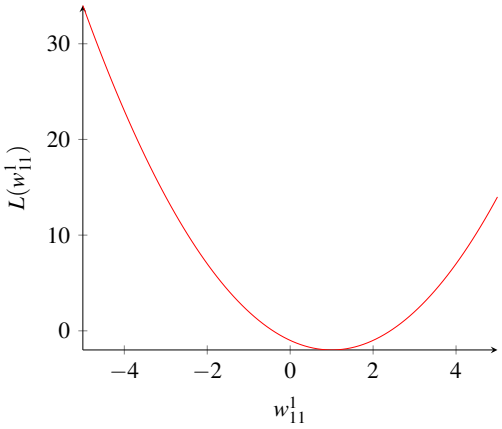



**Figure 2.9** Hypothetical plot of the Loss function and $w_{11}^1$ .

In this graph, we would want the error to be as low as possible when given X or inputs similar to X to the neural network. To do so, we'd need to change the value of $w_{11}^1$ in the direction where the cost function decreases. To achieve that, we calculate the derivative of the cost function with respect to $w_{11}^1$ to obtain the slope of the cost function. Now that we know the slope of the cost function, we know in which direction the cost function decreases and increases, so we just need to adjust $w_{11}^1$ in the direction to decrease the RMSE. This process of updating every weight parameter of a neural network can then be written as [18, 35]:

$$w_{11}^{1+} = w_{11}^1 - \varepsilon \frac{dLoss}{dw_{11}^1}$$

Where $\varepsilon$ is the learning rate parameter. The learning rate is once again one of the hyperparameters of the network. The higher the learning rate, the "faster" the network is going to learn, but if the learning rate is too high, we can overshoot the optimum value of the weight.

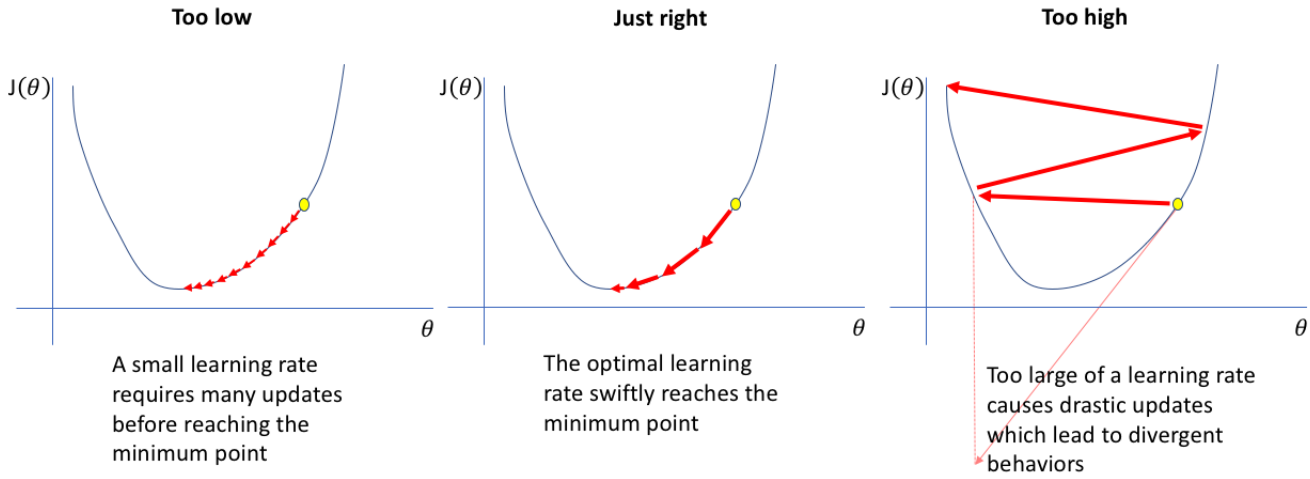



**Figure 2.10** Very good example of how different learning rates affect the learning process. $J(\theta)$ represents the Loss and $\theta$ represents the weight parameter. Source [27] .

As we explained above, the process of calculating each derivative of every parameter on the neural

net is called backprop. Backprop makes use of the chain rule of calculus, which is used to compute the derivative of functions formed by other functions. In neural networks, a computing graph is built for every parameter on the network (like the one below) and we just need to plug the specific numbers on the derivatives to obtain the slope of the cost function on that weight parameter.

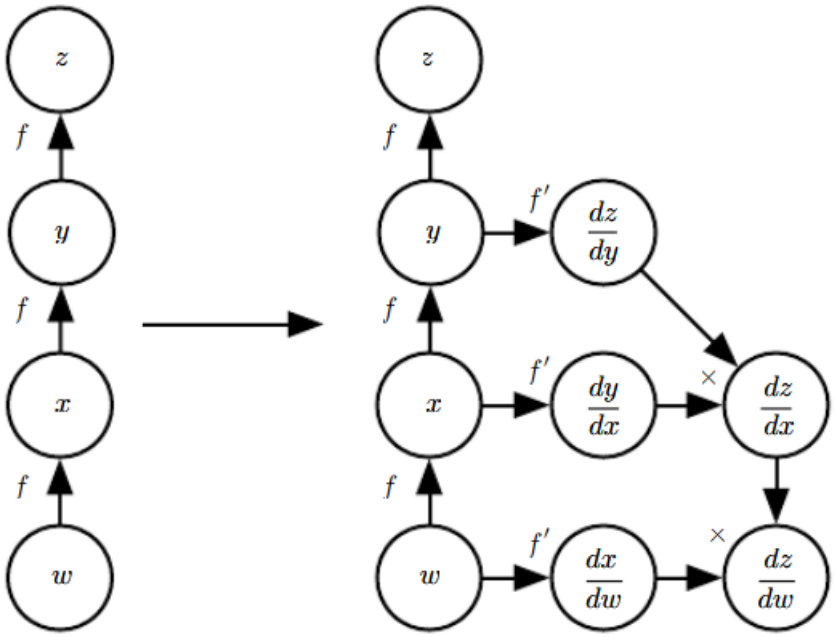



**Figure 2.11** When calculating the derivatives, we make use of the chain rule. The backprop algorithm simply creates computing nodes describing how to calculate the derivatives. Source [18] .

Adjusting weights is how neural networks learn, a good way of thinking about it is that the network is finding the neuron connections and transforming the inputs that produce the best results, that is, the results that lead to the lowest RMSE across all inputs.

## 2.4 Good practices and classical problems of neural nets

The first good practice technique on machine learning is to split the dataset at hand. See, for a given set of examples, we are going to divide them in two groups, a train set to train the neural network and a test set to evaluate the performance of the neural network. We want our network not only to perform well on inputs from the train set, but also to perform well when given previously unseen inputs, that is, we want our network to generalize well. We will consider our model successful when the training loss is low but also when the test error is low as well. We have to be careful however when performing the split because data may be imbalanced, that is, what if our train set is only a subset of too similar examples? When performing the split we need to keep an eye on the variance of the dataset and each train and test subset to make sure that high variance is mantained across all variables on each subset. Cross Validation, however may help prevent this. Using K-Folds Cross Validation, we split the train data on K subsets, train on K-1 sets and use the last subset as test data. We then average the model against each of the folds. Lastly we test the model with the test data to evaluate the performance.

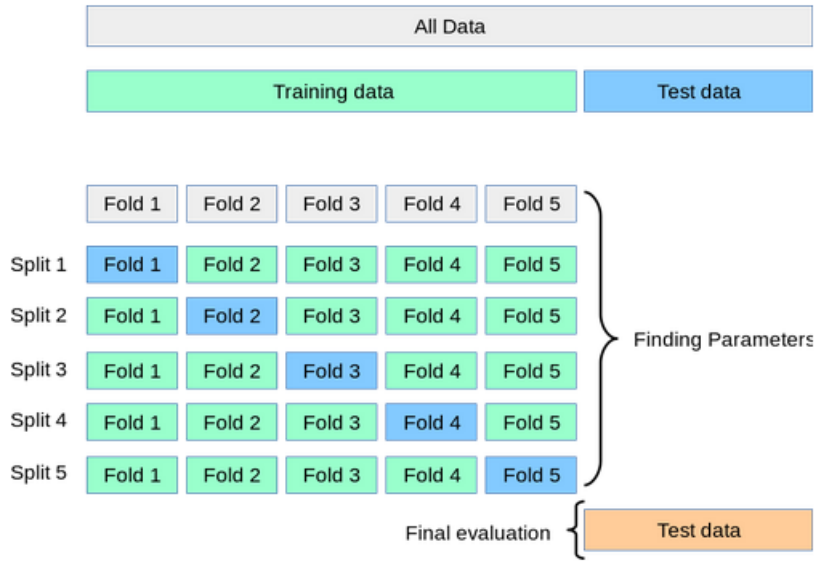



**Figure 2.12** K-Folds Cross Validation visualization [47] .

Overfitting and underfitting are classical problems of machine learning. We say that a model is overfitting when the model "memorizes" a training set. When overfitting, model's training error is very low, but test error remains high regardless of the epochs. The model is then failing to generalize by catching patterns on the training data that are not useful for generalization. The model predictions have then very high variance.For the given data [7]:

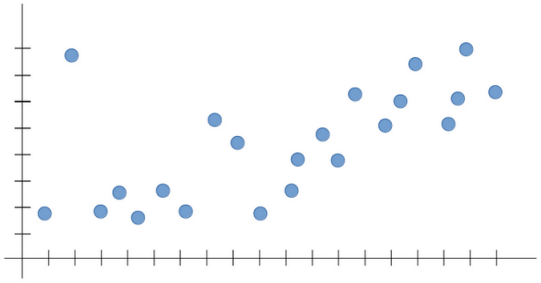


**Figure 2.13** Random Dataset plot [7] .

A correctly trained model would find the following underlying function:

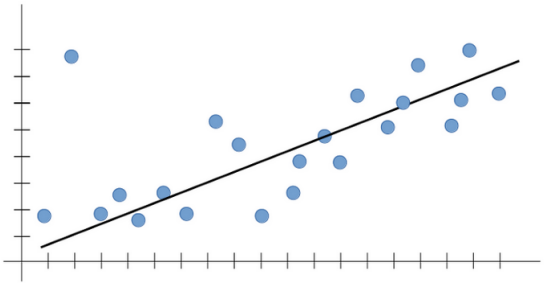


**Figure 2.14** Good fit of the dataset [7] .

While an overfit model would look like:

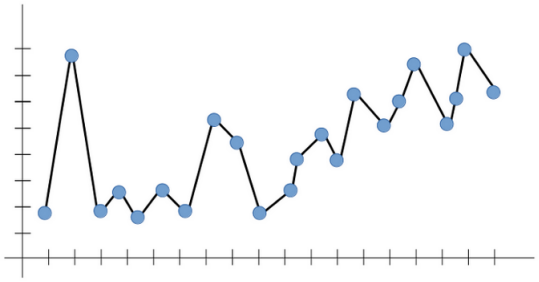


**Figure 2.15** Overfitting model [7] .

Some causes of overfitting may be that the model is overly complex for the underlying function, that is, the model has too many neurons/layers for the problem at hand or that we have trained for too many epochs. To prevent overfitting, we can reduce the number of epochs or complexity, or we can perform Early Stopping. During early stopping, we test the model regularly during the training loop and stop training when the test error starts to increase. On Convolutional Neural Networks, Dropout is a popular technique to prevent overfitting as well. With Dropout, connections between neurons are removed during training to force the model to find different connections to reduce the error and generalize well, effectively reducing the model complexity [18].

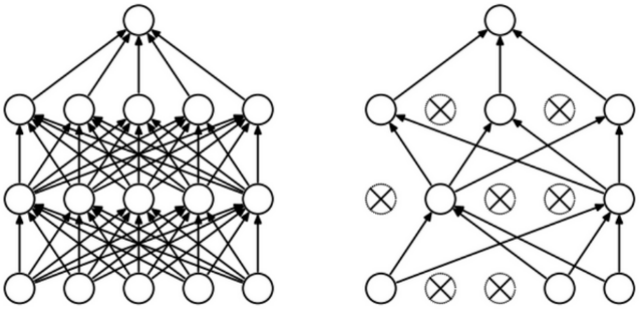



**Figure 2.16** Dropout visualization [14] .

We say that a model underfits when the contrary is happening. The model is not complex enough or hasn't trained for enough epochs to capture the underlying function and is highly biased.

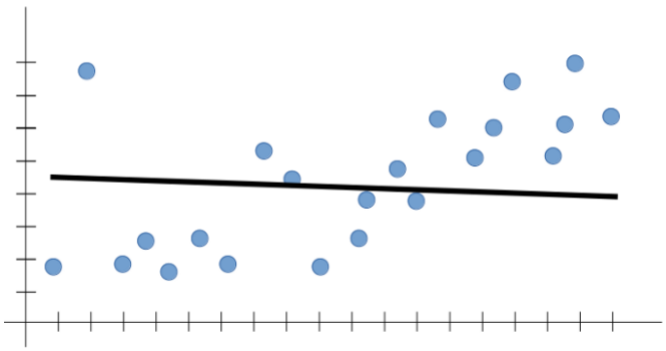


**Figure 2.17** Underfitting model [7] .

Other classical problem when working with neural networks are vanishing and exploding gradients. Up until the release of ResNet [21], vanishing gradients was the biggest problem when working with deep networks. As we explained before, networks learn by backpropagating the errors by using the chain rule and adjusting the weights accordingly. For a network of $N$ hidden layers, $N$ derivatives will be multiplied together. If those derivatives are large, gradients will increase exponentially until they "explode". High gradients equals large steps when adjusting the weights, which leads to unstable training and performance of the model as explained before. On the other hand, when we have small derivatives, gradients will decrease exponentially until weights don't get adjusted anymore, which lead to dead neurons and the network is not able to learn at all. To prevent gradient vanishing/exploding, we may reduce the number of layers of our network, clip the gradients so they are inside a given range or initialize or weights carefully, more on this later.

## 2.5 ADAM algorithm for gradient descent

ADAM is the state-of-the-art optimization algorithm in deep learning. As per [28], ADAM is an adaptative learning rate method, in that it computes individual learning rates for every parameter in the network. Adam uses estimators or first and second moments of gradients to change the learning rate on the fly during the training process. A moment is defined as:

$$m_n = E[X^n]$$

Since we can considerate the gradients of the cost function as random variables, we can think of the first moment as the mean and the second moment as the uncentered variance. To estimate the moments, ADAM uses exponentially moving averages, like this:

$$m_t = (1-\beta_1)\sum_{i=0}^{i} \beta_1^{t-i} \Delta Loss_i$$

$$v_t = (1-\beta_2)\sum_{i=0}^{i} \beta_2^{t-i} \Delta Loss_i^2$$

Where $m$ and $v$ are the moving averages and t represents the current batch. $\beta$ is an hyperparameter, with values around 0.9. ADAM then updates each wight as:

$$w_t = w_{t-1} - \varepsilon \frac{\tilde{m_t}}{\sqrt{\tilde{v_t}} + \mu}$$

Where $\varepsilon$ is the learning rate, $\mu$ is another hyperparameter to avoid zero division and $\tilde{m_t}$ and $\tilde{v_t}$ are respectively:

$$\tilde{m_t} = \frac{m_t}{1-\beta_1^t}$$

$$\tilde{v_t} = \frac{v_t}{1-\beta_2^t}$$

ADAM uses the combined average of past gradients to incorporate momentum. By adding fractions of previous updates to the current update, ADAM increases the updates for a given parameter if several updates in a row push the gradient in the same directions, giving more weight

to more recent updates thanks to the decay parameter $\beta$. ADAM outperforms traditional SGD algorithms as well as more sophisticated AdaGrad and RMSProp [51]

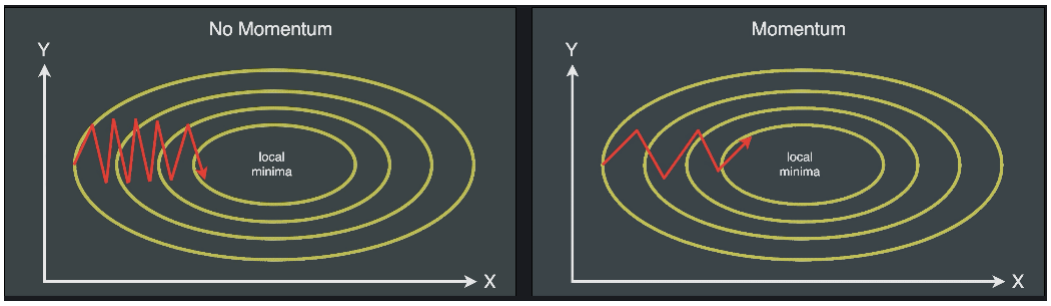



**Figure 2.18** Momentum on ADAM. ADAM performs larger weight updates when the gradients points to the same directions over several batches [**?**] .

## 2.6 Convolutional Neural Networks

In this section we will explain the basics of the Convolutional Neural Networks (CNNs). Convolutional neural networks are famously used for image recognition and time-series data prediction. Convolutional networks [34] became a superior architecture in image recognition in [32, 22] with the MNIST [33] data set. Since 2012, and because of its performance on the ImageNet Large Scale Visual Recognition Challenge ILSVRC, they have become the standard for image recognition and classification. Since then, every winner of the ILSVRC competition has been some form of CNN, with ZFnet [59] and Google's GoogLeNet [54] winning in the following years.

Although time-series prediction has been dominated by Recurrent Neural Networks architectures, recently Convolutional Architectures [9] combined with residual blocks [29] has seen success modeling long-term dependencies on a given sequence and producing sequence-to-sequence predictions outperforming most of the time RNNs while also having the advantage of being less computationally expensive.

CNNs contain one or more layers that perform convolutions on a given data input, regardless of the input's dimensionality.

The discrete convolution operation can be defined as:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

Where $x$ denotes the input, $s(t)$ or output is often called feature map and the function $w$ is called kernel or filter. The kernel is the interesting part here, since it's a multidimensional array of learning parameters that change overtime in the training process. Since we are going to work with 1D inputs, consider the discrete function in vector form:

| 1 | 5 | 3 | 5 | 6 | 2 | 4 | 1 | 2 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|

**Figure 2.19** Example input for 1D convolution .

We can see that the sequence length is 11, as in 11 elements in the vector. With the number of features being 1 as well, since the input is 1-dimensional, the kernel interacts with the input like this:

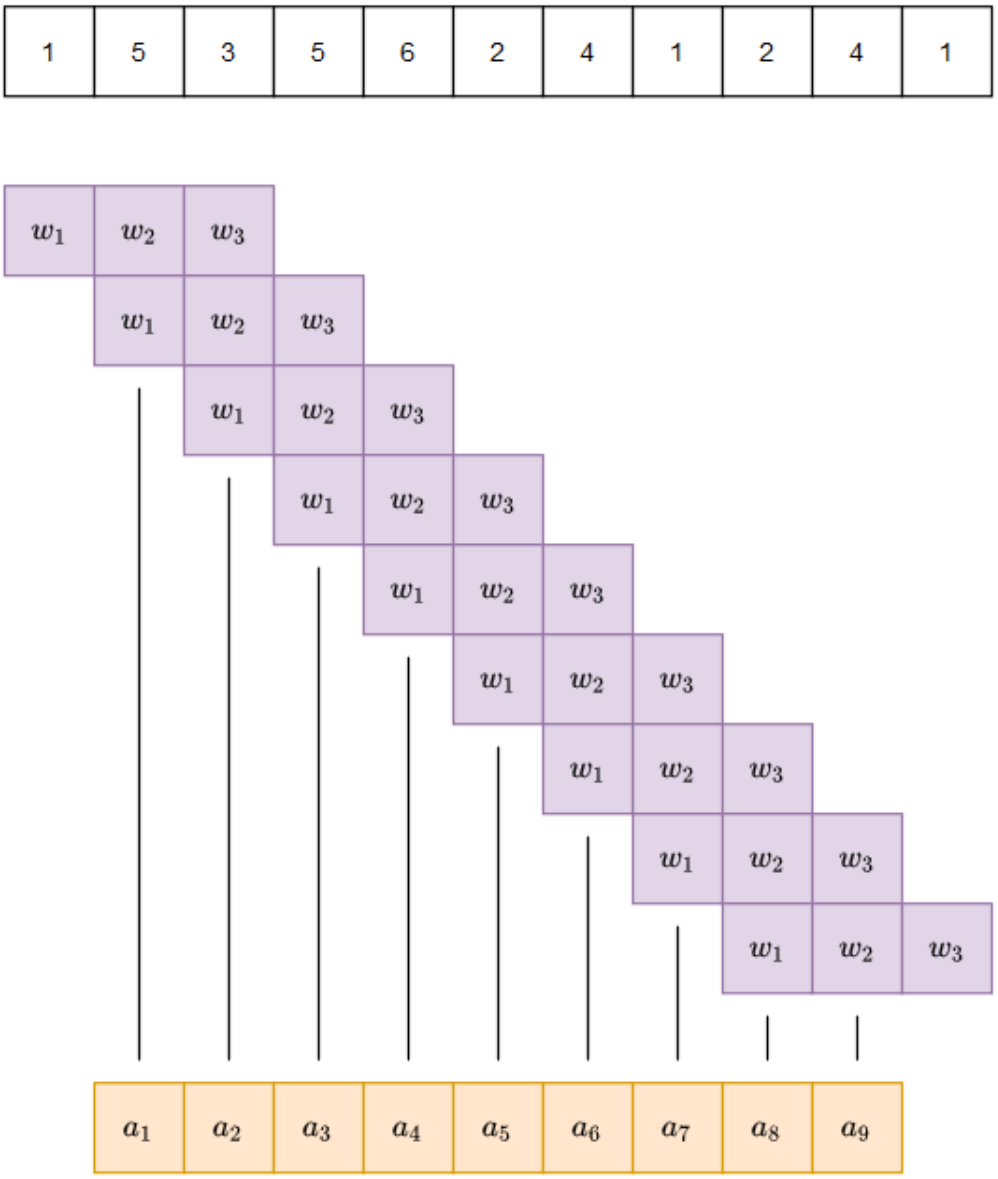



**Figure 2.20** Kernel calculating the feature map for the vector input .

Here, the kernel is represented in purple and it has a kernel size of 3, and the feature map is represented in orange. In the example above, each row of the kernel represents a neuron in the convolutional layer. The kernel behaves like a sliding window calculating the dot product between its weights ($w_1$, $w_2$, $w_3$) and the input vector at the top. We can express each output as:

$$a_1 = g(1w_1 + 5w_2 + 3w_3)$$
$$a_2 = g(5w_1 + 3w_2 + 5w_3)$$

$$a_3 = g(3w_1 + 5w_2 + 6w_3)$$

and so on.

Where $a_n$ represents the output of that neuron, $g(x)$ is the activation function and $w_n$ represent the weights of the kernel. It's very important to note that for a given kernel, the weights on each neuron are going to change in the same manner, that is, when we update the weights on the learning process, $w_1$, $w_2$ and $w_3$ are going to be the same across all neurons. With this, the kernel is learning common features or patterns on an input signal across all the length of the sequence.

Normally, we will have more than one kernel, that is, we are going to be looking for more than one pattern on the input signal. In the learning process, the network will adjust the weights on the kernels to reduce the overall RMSE when performing the convolution with the input. The input tensor we are going to feed the convolutional layer is going to have the shape:

$$(N, CH_{in}, L_{in})$$

Where N represents the batch size, that is, each individual example of data we are going to feed to the neural network, $CH_{in}$ represents the number of input features and $L_{in}$ is the length of the input sequence. The output tensor is going to have the shape:

$$(N, CH_{out}, L_{out})$$

Where $CH_{out}$ represents the number of kernels, or the number of different patterns we want to find on the sequence, and $L_{out}$ represents the feature map length. We can then express the output of a given kernel as [46]:

$$out(N_i, Ch_{out_j}) = bias(Ch_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_out_j, k) * input(N_i, k)$$

With that, weights in the kernel are adjusted in the same fashion as described above by the backprop algorithm.

## 2.7 ResNET and Temporal Convolutional Networks

As we mentioned before, a large amount of layers usually leads to gradients exploding or vanishing, but if the function to approximate is very complex, a higher number of layers might allow the network to find complex patterns on data. We can then say that a higher number of layers does not come with better performance as shown in the image below from [21]:

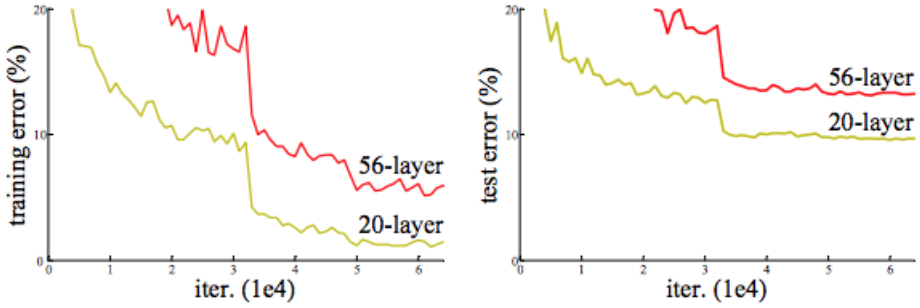



**Figure 2.21** Performance of different models on the image-recognition CIFAR-10 dataset .

As we can see, regardless of the regularization or dropout or gradient-improving technique we choose, usually shallow networks outperform very deep networks, mostly due to vanishing gradients. The layers closer to the output are improving their weights while the layers closer to the input are not learning at all. To resolve this, the Residual Block is introduced:

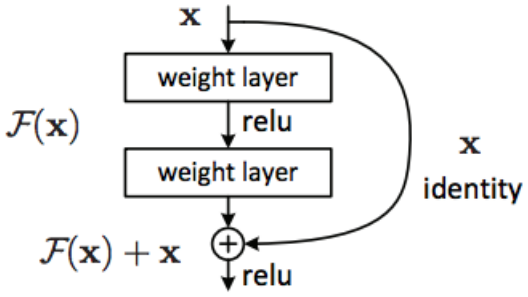



**Figure 2.22** Residual block, the main component of the ResNET [21] .

We are forcing the network to learn the residual mapping from the input to the output instead of transforming the input data through several layers to get the output. This allows the ResNet to learn the most effective *depths* for a given input. Now we can have networks with hundreds of layers that does not suffer from vanishing/exploding gradient and that is very computationally attractive, since the identity mapping does not introduce extra parameters or computations. Variable-depth networks become very potent, specially in convolution-related tasks, since higher level of abstraction is needed, leading to ResNET to win the ImageNet competition on 2015.

Temporal Convolutional Networks (TCN) are popular architecture when working with time-series datasets. They implement several layers of convolutional layers to capture long time dependencies and use residual blocks to prevent vanishing gradient. TCNs are designed for sequence modeling, that is, a TCN outputs:

$$y*_0, \ldots, y*_T = f(x_0, \ldots, x_T)$$

Where the ground truth $y_t$ depends only on $x_0, \ldots, x_t$ and not on future inputs $x_{t+1}, \ldots, x_T$. TCN prevents leakage from the future, TCN uses 1D convolutional layers where each layer has the same length as the input, and it performs dilated convolution:

$$F(s) = (x *_d f)(s) = \sum_{i=0}^{k-1} f(i) x_{s-di}$$

Where d is the dilation factor, k is the filter size and $s - di$ points to the direction of the past. Using larger dilations allows TCN to look further into the past by expanding each neuron receptive field

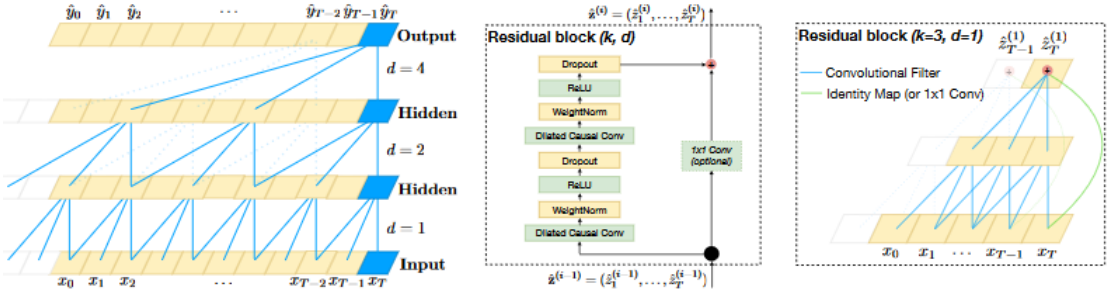



**Figure 2.23** Building blocks of TCN. *Left* 3 dilated convolutional layers with $d = 1, 2, 4$ and kernel size $k = 3$. *Middle* TCN residual block architecture for TCN. *Right* Residual connection on TCNs[9].

TCN are able to train in parallel, unlike RNNs, where past predictions are needed in the training. TCN also allow for variable receptive field by varying the number of convolutional layers, changing the dilation parameter $d$ or increasing the filter size. TCNs outperform most RNNs architectures on a variety of benchmark datasets as shown in [9]

# 3 Radio Frequency Power Amplifiers

Radio Frequency Power Amplifiers (RFPA) are electronic devices that boost a given signal to the according level so it reaches the receiver. An ideal RFPA would take the already modulated signal at a given frequency and output a boosted undistorted signal, that is, we want the amplifier to behave as linearly as possible, but that's not the case. Amplifiers power consumption often surpasses every other component on a given radio system, so optimizing the efficiency of the amplifier is key to reduce battery consumption or operating costs. Nowadays, circuit design must confront tougher and tougher linearity specifications, to reduce nonlinear distortions as much as possible.

However, as we just mentioned, amplifiers are not perfect, and the function that models its behaviour is certainly not linear, several nonlinearities are introduced on the system in the form of distortions. These nonlinearities cause interference on the receiving end.

RFPAs are divided into different categories depending on their efficiency: A, B, AB, etc… [58] and there's a tradeoff between efficiency and linearity on a given amplifier. With linearity, we mean that the output is a linear transformation of the input to the amplifier, such as:

$$y(t) = Gx(t)$$

Where $y(t)$ represents the output of the amplifier, $x(t)$ its input and $a$ is a constant parameter.

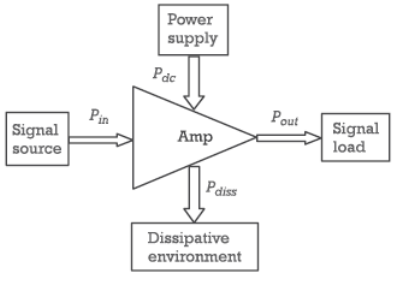



**Figure 3.1** Power inputs and output on an amplifier. Source [48] .

On the figure above, $P_{in}$ and $P_{out}$ represent the input and output power of the signals through the amplifier, $P_{dc}$ represents the DC power supply to the amplifier and $P_{diss}$ is the power dissipated as heat from the amplifier. We can rewrite the equation above as:

$$G = \frac{P_{out}}{P_{in}}$$

And we can write

$$P_{out} + P_{diss} = P_{in} + P_{dc}$$

Then

$$G = 1 + \frac{P_{dc} - P_{diss}}{P_{in}}$$

Thus the total power at the output of the amplifier is limited by the DC supply. We can represent this nonlinearity as:

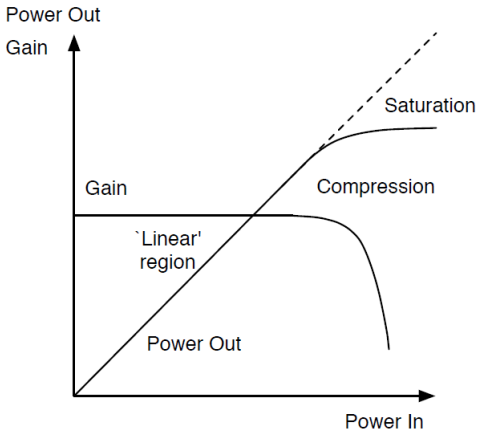



**Figure 3.2** Power in the input plotted against the output power and the gain. Source [58] .

This effect is called gain compression. The input signal amplitude is affecting the output signal amplitude in a non-linear way. On lower power inputs we can see how both the Gain and the output power follow the linear "ideal" behaviour we want from the RFPA, with this ideal gain being represented by the dashed line. However, at higher powers we can see how both move away from the linear behaviour and enter the saturation zone. At higher power inputs, we won't get higher power outputs past a certain point. This makes sense, since the amplifier will not be able to create power from nowhere. The maximum power the RFPA will be able to output is going to be close to the power provided to the amplifier as supply, with a small portion of the power dissipated as heat. The thing is, we want the amplifier to operate around this nonlinear zone, since we want to be as efficient as possible, thus making the best use of the power fed to the amplifier.

## 3.1 Nonlinear distortions

To get an idea of how nonlinear system affects the given inputs [48], let's work with an example for a bit. Given an typical stimulus to a telecommunications environment, we can write an input baseband signal as:

$$x(t) = A(t)cos[\omega_c t + \theta(t)]$$

Given a nonlinear system restricted to the third polynomial:

$$y_{NL} = a_1 x(t-\tau_1) + a_2 x(t-\tau_2)^2 + a_3 x(t-\tau_3)^3$$

The response of a linear system would be:

$$y_L = a_1 A(t-\tau_1) cos[\omega_c t + \theta(t-\tau_1) - \phi_1]$$

And the response of the nonlinear system would be:

$$\begin{aligned} y_{NL} &= a_1 A(t-\tau_1) cos[\omega_c t + \theta(t-\tau_1) - \phi_1] \\ &+ a_2 A(t-\tau_2)^2 cos[\omega_c t + \theta(t-\tau_2) - \phi_2]^2 \\ &+ a_3 A(t-\tau_3)^3 cos[\omega_c t + \theta(t-\tau_3) - \phi_3]^3 \end{aligned}$$

By using the following trigonometric relations:

$$cos(\alpha)^2 = \frac{1}{2} + \frac{1}{2} cos(2\alpha)$$

$$cos(\alpha)^3 = \frac{3}{4} cos(\alpha) + \frac{1}{4} cos(3\alpha)$$

We can rewrite the nonlinear response as:

$$\begin{aligned} y_{NL} &= a_1 A(t-\tau_1) cos[\omega_c t + \theta(t-\tau_1) - \phi_1] \\ &+ \frac{1}{2} a_2 A(t-\tau_2)^2 + \frac{1}{2} a_2 A(t-\tau_2)^2 cos[2\omega_c t + 2\theta(t-\tau_2) - 2\phi_2)] \\ &+ \frac{3}{4} a_3 A(t-\tau_3)^3 cos[\omega_c t + \theta(t-\tau_3) - \phi_3] \\ &+ \frac{1}{4} a_3 A(t-\tau_3)^3 cos[3\omega_c t + 3\theta(t-\tau_3) - 3\phi_3] \end{aligned}$$

Where $\phi_1 = \omega_c \tau_1, \phi_2 = \omega_c \tau_2, \phi_3 = \omega_c \tau_3$

Here, several interesting terms appeared. We have a number of nonlinear responses on different frequencies. This phenomenon is called *spectral regrowth*. Though we can filter this frequencies out, other components such as the $\frac{1}{2} a_2 A(t-\tau_2)^2$ term introduce the so called DC bias. Lastly, now the amplitude of our baseband signal has been distorted as well, becoming $a_1 A(t-\tau_1) + \frac{3}{4} a_3 A(t-\tau_3)^3$, as well as the phase of the baseband signal being distorted. This effects are known as AM/AM and AM/PM distortions respectively, meaning that the input amplitude modulation creates output amplitude and phase modulation on the output signal.

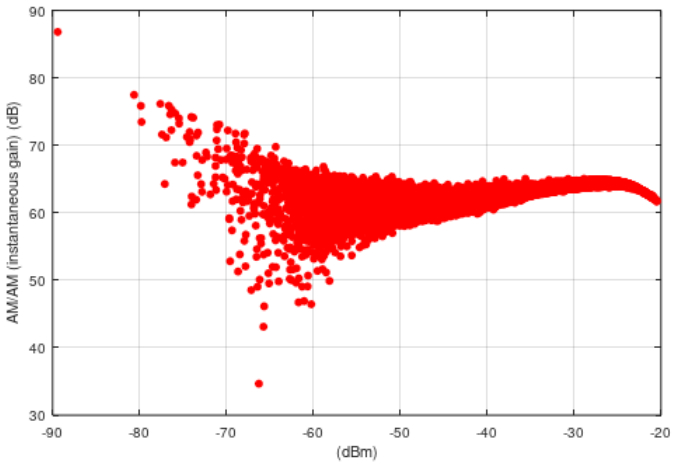



**Figure 3.3** AM/AM distortion on a class J amplifier. X axis represents the input power (dBm) while the Y axis represents the AM/AM gain (dB). We can clearly see the nonlinear behaviour of the amplifier, especially towards higher power inputs .

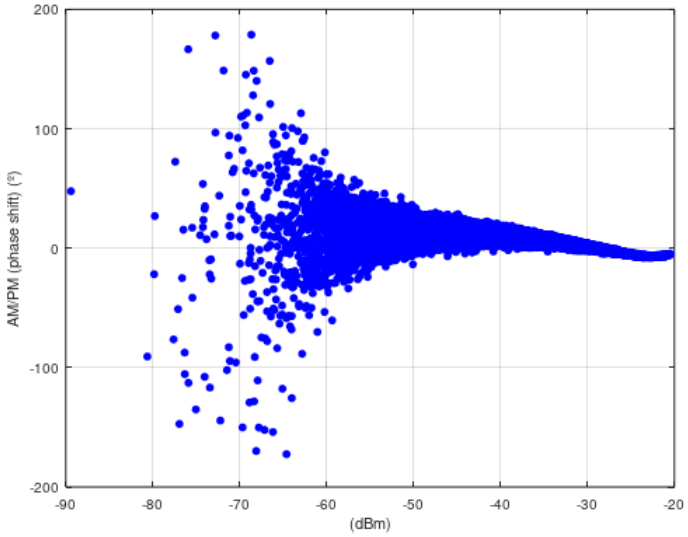



**Figure 3.4** AM/PM distortion on a class J amplifier. X axis represents the input power (dBm) while the Y axis represents the AM/PM phase shift (dB). Again, the system behaves in a nonlinear fashion more clearly seen on higher input powers .

## 3.2 Other memory-related distortions

As per [58], memory effects means past inputs of a given signal influence influence the value of the signal in the current timestep. The number of samples from the past that influence the current timestep is known as memory depth.

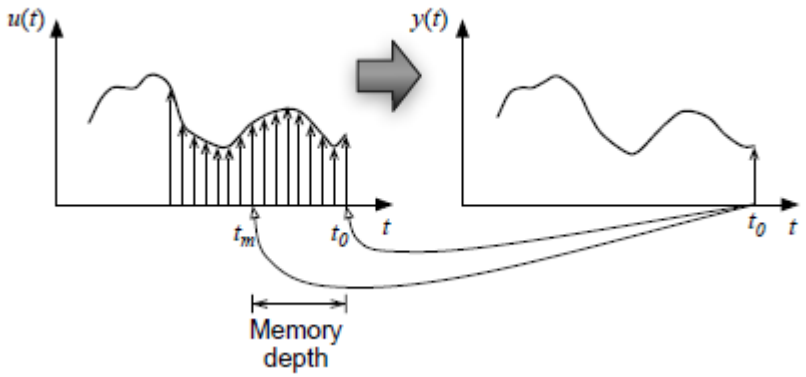



**Figure 3.5** Past signal influence current-time values on systems with memory [58] .

Memory effects can be caused by a variety of reasons, such as charge being built-up on capacitors. On amplifiers, the memory effects affect the rise and fall times, creating different output voltages for the same input voltage.

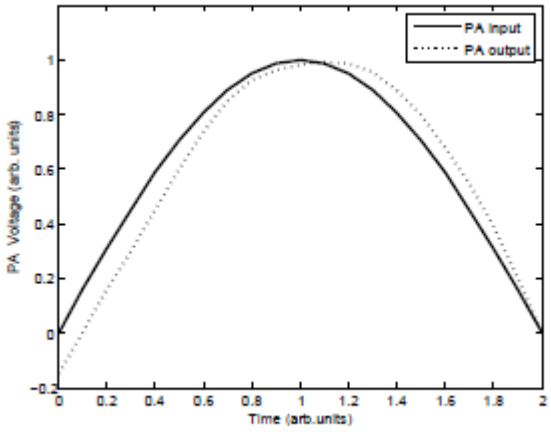



**Figure 3.6** Rise and fall times behave differently because of memory effects [58] .

Memory effects on amplifiers can be classified on Short-Term Memory Effects (STM) or Long-Term Memory Effects (LTM). STM effects are produced by matching networks, that is, components such as capacitors or inductors built into the amplifier and capacitances inherent to the transistor, where build-up. transport and decay of charge create time delays.

LTM effects have affect across timesteps much bigger than the carrier frequency of the input signal. They include thermal effects, inherent to the flow of thermal energy within the components of the amplifier, and can manifest itself as a small change in the gain of the amplifier after the signal has passed. Other LTM effect is the charge trapping, where imperfect components of the amplifier store charge and release it continually. DC supply may also affect the modulation as a LTM effect introducing bias voltage at the modulation frequency affecting the gain of the amplifier. Lastly, if the amplifier contains control features for automatic bias and gain control, the time associated with early mentioned features may introduce LTM effects that can be difficult to correct.

We can then rewrite this memory dependencies onto our nonlinear system model as:

$$y_{NL} = f_{NL}(u[t], u[t-1], u[t-2], ..., u[t-m])$$

## 3.3 Behavioral modeling and Digital Predistortion

To predict and quantify the distortions on a given RFPA and design circuits to better mitigate the aforementioned distortions, behavioral models are required. The advantage of building this behavioral models is that we do not need to know exactly how every single component on a complex systems interacts with each other, we could treat the system as a black box and model its input/output relationship through a number of techniques. The performance of a behavioral model resides then in the accuracy of our observations of the signals and the choice of formulation that describes the amplifier itself.

There are many options to choose from to model an input-output relationship such as polynomial approximation and Fourier series, but one of the most popular techniques on PA behavioral modeling is by using Volterra series [61]. Introduced in 1887 by Vito Volterra, they use a combination of linear convolution and nonlinear power series to describe an input/output relationship of a system with fading memory, the general equation of discrete Volterra series is:

$$y(n) = \sum_{k=1}^{K} \sum_{i_1=0}^{M} \dots \sum_{i_p=0}^{M} h_p(i_1, \dots, i_p) \prod_{j=1}^{k} x(n-1)$$

Where $h_p(i_1, \dots, i_p)$ represents the parameters of the Volterra model, the Volterra kernels, K is the nonlinearity order and M is the memory depth. Given as the input and output of an amplifier:

$$x(t) = \Re[\tilde{x} e^{j\omega_0 t}]$$
$$y(t) = \Re[\tilde{y} e^{j\omega_0 t}]$$

Where $\omega_0$ is the carrier frequency and $\tilde{x}$ and $\tilde{y}$ are the envelope of the input and output signal respectively, a discrete Volterra model of $K = 3$ is given as:

$$\tilde{y}(n) = \sum_{i=0}^{M} \tilde{h_1}(i)\tilde{x}(n-i)$$

$$+ \sum_{i_1=0}^{M} \sum_{i_2=0}^{M} \sum_{i_3=0}^{M} \tilde{h_3}(i_1, i_2, i_3)\tilde{x}(n-i_1)\tilde{x}(n-i_2)\tilde{x}^*(n-i_3)$$

$$+ \dots$$

Where $\tilde{h_p}(i_1, \dots, i_p)$ is the complex Volterra kernel and $()^*$ represent the conjugate transpose. Even order kernels and the items associated with kernel symmetry are removed on the equation for clarity.

With conventional Volterra series, the number of parameters increases drastically with the nonlinearity order and the memory depth. To fix this, several techniques have been proposed to simplify the Volterra model such as [60, 10].

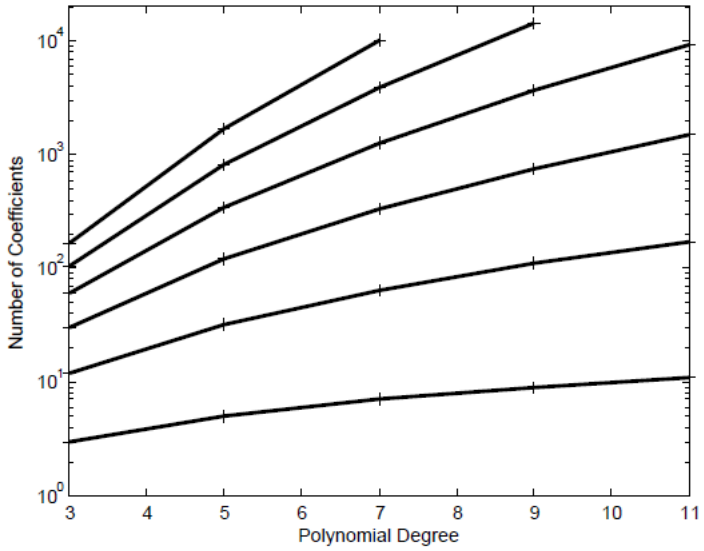



**Figure 3.7** Coefficient exponential growth on Volterra series [58] .

The Memory Polynomial Model (MP) the simplest form of the Volterra series, which can be described as a Taylor series with memory:

$$y(n) = \sum_{j=1}^{K} \sum_{i=0}^{M} a_{ij} x(n-i)|x(n-i)|^{j-1}$$

Where $x(n)$ is the input signal, $a_{ij}$ are the coefficients of the model, M represents the memory depth of the model, and K represents the nonlinearity polynomial order.

Some forms of the MP, with varying complexity, have been proposed in the literature, such as Generalized Memory Polynomial Models (GMP) or orthogonal polynomial models [42]. Comparison between model is however delicate, with several metrics being proposed for model comparison such as Normalized Mean Squared Error (NMSE), memory effects modeling ratio (MEMR) and weighted error-to-signal power ratio (WESPR), with the NMSE metrics being the most prominently used in research.

Although Volterra series and its forms perform very well, it suffers from limitations and tradeoffs we have to choose from including complexity, speed, ease of identification, etc… This, and because of the exponential increase of the polynomials, makes calculating strong nonlinear, highly polynomial order models computationally hard. Often, even-order terms of Volterra series are ignored since their effects fall outside of the frequency band if interest, however, these components may generate bias in the PA bias and supply line and create memory effects[58].

The aim with NNs and their Universal Approximation Theorem is to create an unified model able to tackle all the PAs nonlinearities, dc offset and IQ imbalance included, while keeping the complexity of the model to a reasonable level to allow the computations to be fast on DPD systems.

We perform all this modeling for digital-predistortion purposes (from now DPD), where distortion is added to the input signal to cancel the inherent nonlinearities of the PA. We can represent a linear amplifier based on digital pre-distortion as:

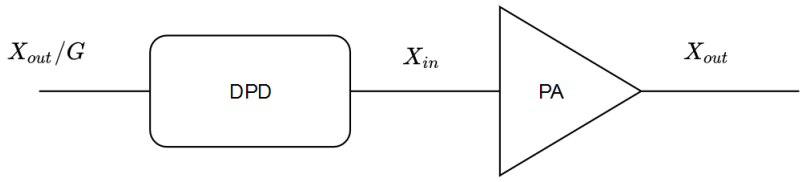



**Figure 3.8** DPD component with a RFPA .

In [17] there's a great explanation of DPD's function. While the behavioral modeling of the PA consist of finding $f_{pa}$ where $f_{pa}(X_{in} = X_{out})$, the DPD component is responsible of applying $f_{dpd}$ such that $f_{dpd}(X_{out}(n)/G) = X_{in}(n)$.

We could then follow two approaches to find DPD's function: one would be studying and inverting the model of the PA to be linearized, while the other approach consists of modeling its behaviour since we can measure its inputs and outputs. This second approach is called indirect learning technique, proposed in [15].

The figure below shows a fairly simple predistortion technique where an AM/AM distortion is applied (DPD Expansion, where we increase the signal amplitude) in the input signal to mitigate the gain compression of the amplifier achieving a linear behaviour (dashed line).

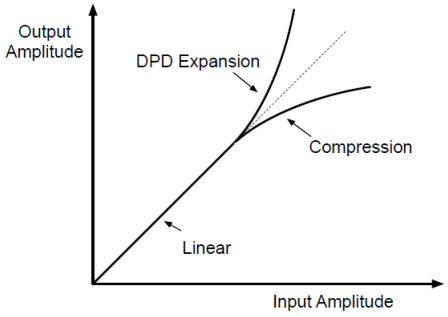



**Figure 3.9** Simple Digital pre-distortion. Source [58] .

## 3.4 Previous work with Neural Networks as behavioral models

In this section we will introduce some popular neural net architectures from previous research which have been fairly successful at modeling the nonlinearities of the RFPA. Following [39], using Neural Nets to model RFPAs is not a new idea, and as early as 2003 with [44] we can find research combining neural nets and RFPAs. Most notably, we take a look at the Augmented Real-Valued Time-Delay Neural Networks (ARVTDNN) [56] and its predecessor the Real-Valued Focused Time-Delay Neural Network (RVFTDNN) [50], where the aim is to model the relationship between the I/Q input and the I/Q outputs, where I represents the in-phase and Q represents the quadrature signal branches of the modulator. Both architectures are real-valued, that is, the inputs, parameters and outputs of the neural net are real values. We could also choose to give the network complex input and have the weights be complex parameters [55], but if we want to take advantage of modern machine learning techniques, such Kaiming weight initialization [22], Adam SGD [28] or robust backpropagation, we have to treat the single complex number input as two separate features. By

making this change no information is lost, since the network is going to find relationships between both inputs regardless. Another option would be to use the polar topology [11], figure below, where two neural nets are used to separately model the amplitude and phase distortions, however, having two separates branches for each component may create imbalance since they may not converge at the same time [26].

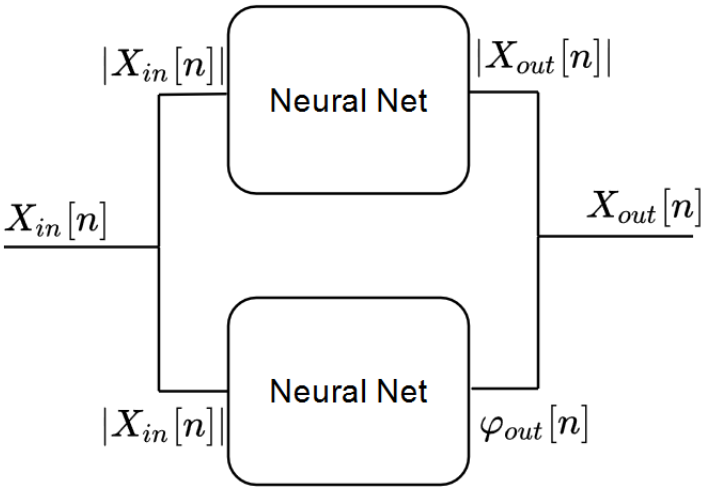



**Figure 3.10** Polar neural network .

The Cartesian topology is then preferred. In this topology, one neural network is used with a 2-dimensional input and a 2-dimensional output, thus avoiding the imbalance problem since there's only one branch of the learning process [49].

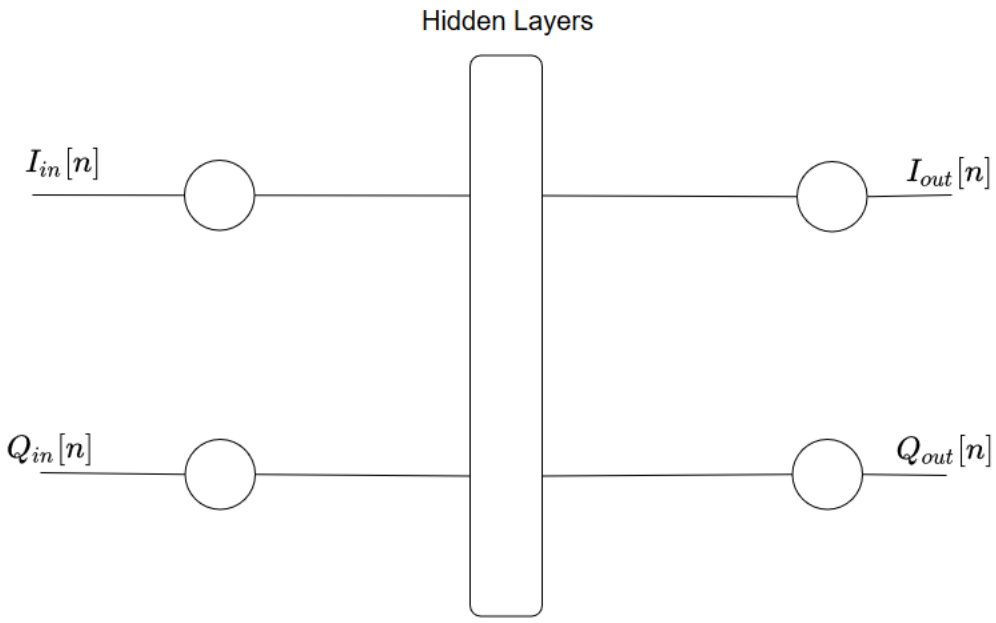



**Figure 3.11** Cartesian neural network .

This architecture is the one followed by the RVFTDNN, which in more detail looks like this:

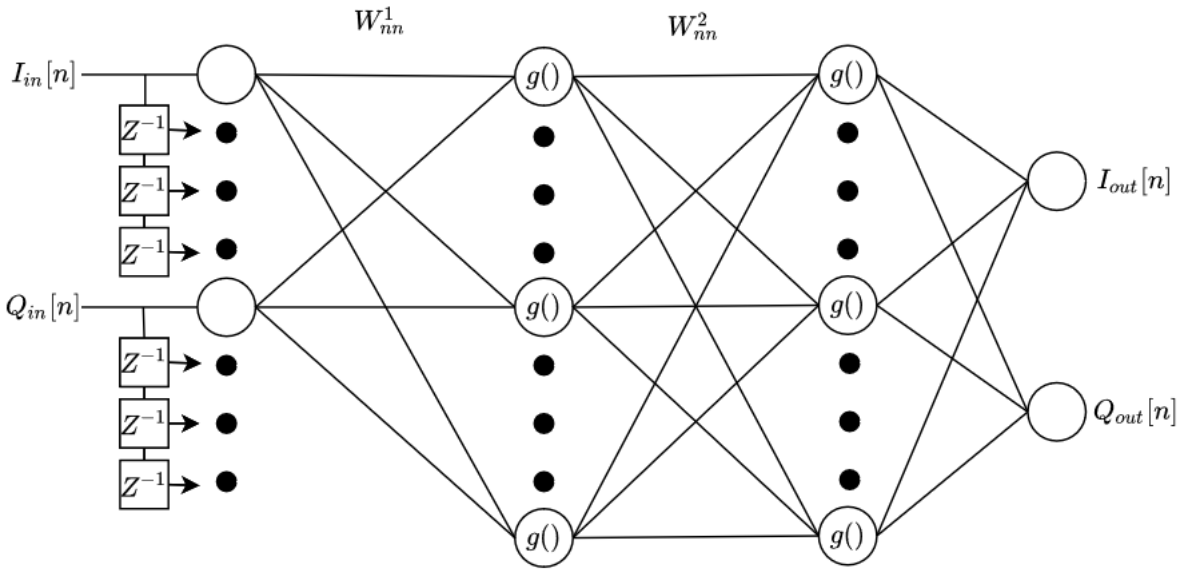



**Figure 3.12** Cartesian neural network .

Where the inputs matrix's shape is:

$$(N,(2m+2))$$

Where $N$ denotes the total samples and $m$ the memory depth we want the system to consider. Thus, a single row from the input is in the form:

$$X = [I_{in}[n], I_{in}[n-1], I_{in}[n-2], \ldots I_{in}[n-m], Q_{in}[n], Q_{in}[n-1], Q_{in}[n-2], \ldots Q_{in}[n-m]]$$

In Figure 2.12, $Z^{-1}$ represents the unit-delayed response of the input $x(n-1)$. $W^k_{nn}$ represents the weight matrices of the connections of the neurons. Biases are removed from the diagram for clarity. Weights are initialized randomly. If we follow the forward computation as explained in chaper 1, the output of a given neuron can be represented as:

$$y* = g(\sum_{n=1}^{\infty} x_n w_n + b)$$

Where the activation function $g()$ is the linear function in the input and output layers and the tansig function on the hidden layers to introduce nonlinearity. The Loss function proposed is the Half Mean Squared Error, which accommodated for the double output can be written as:

$$L = \frac{1}{2N}\sum_{n=1}^{N}(I_{out}[n] - I*_{out}[n])^2 + (Q_{out}[n] - Q*_{out}[n])^2$$

Where $N$ denotes the batch size, and I* and Q* denote the output of the network. The weights are then adjusted by the Lavenberg-Marquardt algorithm [19], instead of SGD or one of its variants:

$$\Delta W = [J^T(W) + J(W) + \mu I]^{-1} J^T(W) e(W)$$

Where J(W) is the Jacobian matrix:

$$J(W) = \begin{bmatrix} \frac{\delta e_1(W)}{\delta W_1} & \frac{\delta e_1(W)}{\delta W_2} & \cdots & \frac{\delta e_1(W)}{\delta W_n} \\ \frac{\delta e_2(W)}{\delta W_1} & \frac{\delta e_2(W)}{\delta W_2} & \cdots & \frac{\delta e_2(W)}{\delta W_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta e_N(W)}{\delta W_1} & \frac{\delta e_N(W)}{\delta W_2} & \cdots & \frac{\delta e_N(W)}{\delta W_n} \end{bmatrix}$$

W represents the weights and bias matrix and e(W) is the error matrix:

$$e(W) = [e_I(1), e_Q(1), e_I(2), e_Q(2), \ldots, e_I(N), e_Q(N)]$$

The complete description of the Levenberg-Marquardt algorithm can be found at [19]. In short, the Levenberg-Marquardt algorithm can shift between Gauss-Newton for small values of $\mu$, and as $\mu -> \infty$ the equation becomes the SGD weight update described in the earlier chapters of this document. Details on the performance of the architecture can be found in [49]. In conclusion, RVFTDNNs outperforms other techniques when I/Q imbalances and dc offsets are not already compensated for.

In 2019, a more recent architecture was proposed, the ARVTDNN [56]. This new architecture is a hybrid between the polar and the cartesian architectures described above. It looks like this:

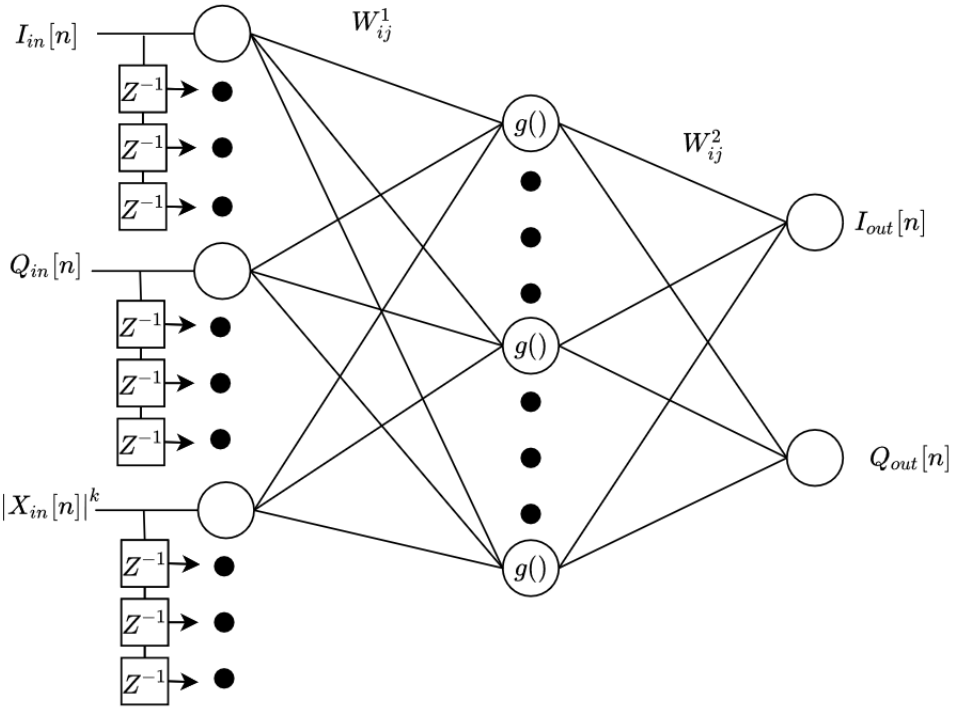



**Figure 3.13** Diagram of ARVTDNN .

Where now we are not only feeding the I/Q components and their delayed values, but also:

$$|X[n]|^k = ((I[n]^2 + Q[n]^2)^{1/2})^k$$

This technique of feeding complex transformations of the input data is well known in the data science and machine learning fields and it's known as data augmentation. In the paper [56] there's the full mathematical explanation why feeding this new information makes the network more accurate, in short, by giving this information to the network it is able to compute more complex transformation for the data at hand, since otherwise it would need to find by itself terms such as $|X_{in}[n]|^k$ in case they were beneficial for the behavioral model. As per [56], the ARVTDNN performs best with the *tansig* activation function and they perform hyperparameter search [20] to find the optimal number of neurons for the network as well as the number of hidden layers. The network has a single hidden layer and they use the half MSE as a cost function as well, with a memory depth $M = 3$. They then compare the results with some other popular modeling options such as MPM Volterra by calculating the NMSE as described in [41]:

$$NMSE_{db} = 10log_{10}\frac{\frac{1}{N}\sum_{j=1}^{N}|y*[n] - y[n]|^2}{\frac{1}{N}\sum_{j=1}^{N}|y[n]|^2}$$

All in all, previous work on neural nets and RFPAs proves that NN can be a one-step solution not only for amplifier distortions but also for dc offset and I/Q imbalance. ARVTDNN outperforms RVFTDNN by 2 to 4 dB with a very similar model complexity.

# 4 Experimental design

As we mentioned before, 1D convolutional neural networks have achieved state-of-the-art performance in several applications such as biomedical data classification and motor-fault detection, while keeping the complexity and hardware requirements low because of the light requirements of compact 1d CNNs. With the success of signal processing CNNs in the biomedical field [30] and civil engineering field [8], it's worth researching if the use of CNNs might be a fit for our behavioral modeling problem.

With that said, we propose the following architecture:

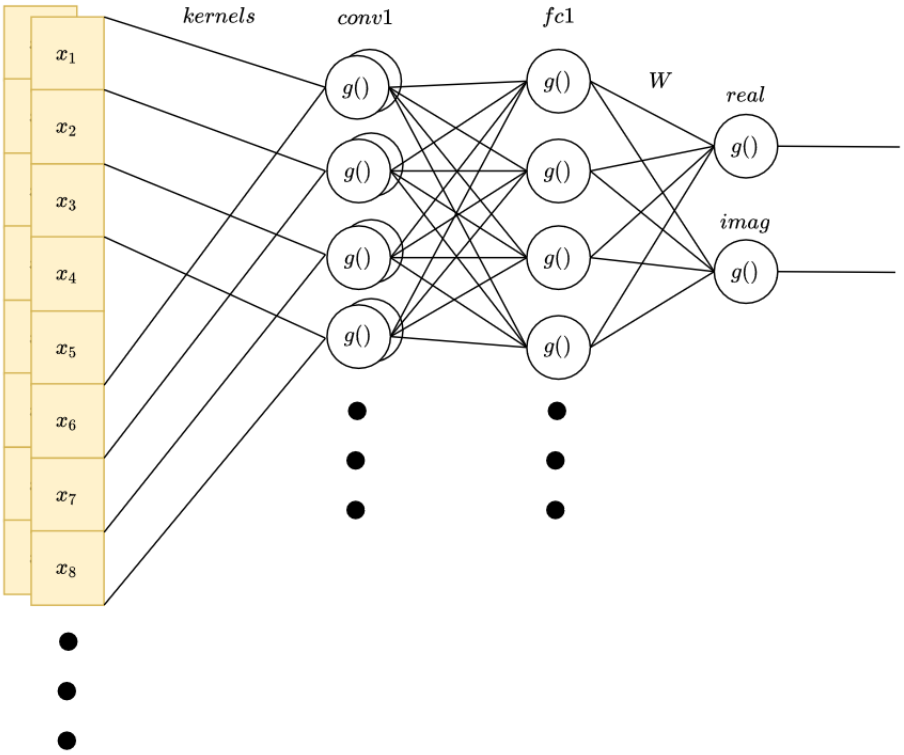



**Figure 4.1** Proposed 1D-CNN architecture for behavioral modeling .

The training process is identical for all the datasets investigated. The training of the models was performed on Google's Colaboratory (Colab) [2]. Colab allows its customers to execute Python and R

code on the cloud, backed by GPUs and with a cell-like structure a lot like Jupyter's Notebooks [31]. Training was performed with NVIDIA Tesla K80 GPUs across all models. GPUs are significantly faster when training neural networks, specifically CNNs, because they allow for parallel computation on their CUDA cores of tensor operations. Training on GPUs was on our experience more than two times faster than using CPUs.

The libraries we used for the experiment are:

```
import numpy as np
import pandas as pd
import seaborn as sns
%matplotlib inline
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import glob
from sklearn.preprocessing import MinMaxScaler
```

Input data is stored on CSV files with each column representing the components of a given input, such as:

| | real | imag |
|---|---|---|
| **0** | 0.005323 | 0.023368 |
| **1** | 0.004153 | 0.025691 |
| **2** | 0.002178 | 0.026179 |
| **3** | -0.000515 | 0.024898 |
| **4** | -0.003793 | 0.022030 |
| **...** | ... | ... |
| **1105915** | -0.016707 | 0.003077 |
| **1105916** | -0.018949 | -0.006080 |
| **1105917** | -0.020247 | -0.015539 |
| **1105918** | -0.020569 | -0.024644 |
| **1105919** | -0.019958 | -0.032762 |

1105920 rows × 2 columns

**Figure 4.2** Example of Pandas Dataframe .

Data handling is performed using the Pandas library [45]. Pandas is a very potent Python library for data analysis and handling. Pandas stores data in a Dataframe object, which looks a lot like a matrix, and allows fast, efficient and complex computation over the data. It allows slicing just like in Numpy or Matlab, indexing and subsetting the dataset and a very robust group by functionality to define our own functions to split and process data. Although Pandas is a Python library, it does most of the heavy-lifting on C. Lastly, Pandas allows writing and reading data to and from a variety of different file types. Pandas is the industry standard in machine learning when it comes to tabular data, data is, 2D data.

| | Unnamed: 0 | Unnamed: 0.1 | Company Name | Location | Datum | Detail | Status Rocket | Rocket | Status Mission |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | SpaceX | LC-39A, Kennedy Space Center, Florida, USA | Fri Aug 07, 2020 05:12 UTC | Falcon 9 Block 5 \| Starlink V1 L9 & BlackSky | StatusActive | 50.0 | Success |
| 1 | 1 | 1 | CASC | Site 9401 (SLS-2), Jiuquan Satellite Launch Ce... | Thu Aug 06, 2020 04:01 UTC | Long March 2D \| Gaofen-9 04 & Q-SAT | StatusActive | 29.75 | Success |
| 2 | 2 | 2 | SpaceX | Pad A, Boca Chica, Texas, USA | Tue Aug 04, 2020 23:57 UTC | Starship Prototype \| 150 Meter Hop | StatusActive | NaN | Success |
| 3 | 3 | 3 | Roscosmos | Site 200/39, Baikonur Cosmodrome, Kazakhstan | Thu Jul 30, 2020 21:25 UTC | Proton-M/Briz-M \| Ekspress-80 & Ekspress-103 | StatusActive | 65.0 | Success |
| 4 | 4 | 4 | ULA | SLC-41, Cape Canaveral AFS, Florida, USA | Thu Jul 30, 2020 11:50 UTC | Atlas V 541 \| Perseverance | StatusActive | 145.0 | Success |

**Figure 4.3** example of Pandas Dataframe with the “All space missions" Dataset.

Along with Pandas, we are using the library Seaborn and Pyplot [25, 57] for data visualization. Seaborn is built on top of Pyplot and allows fancy graphs of data, although we are only going to use it to plot hyperparameter search data and loss graphs.

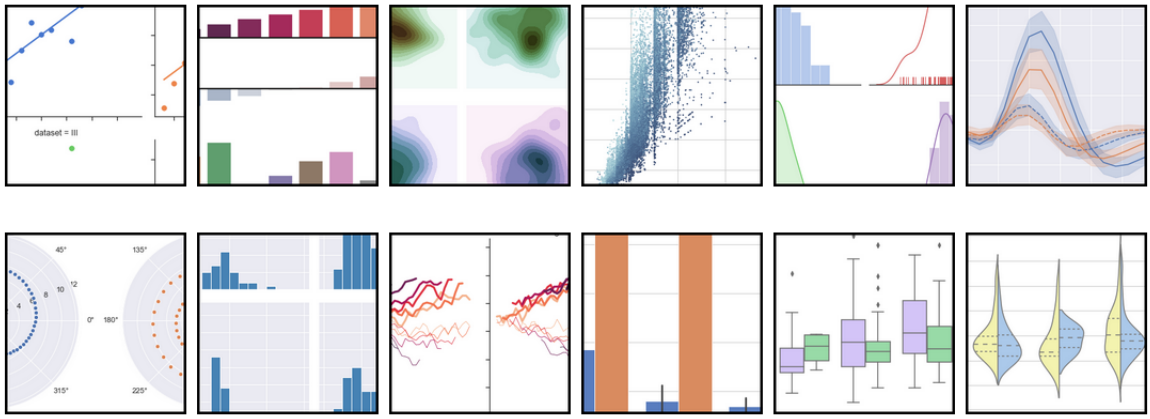



**Figure 4.4** Some of the plots available on Seaborn.

To perform the data preprocessing we will need to do, we will use the library scikit-learn [47], a very powerful Python library with lots of different tools for data analysis and prediction. It comes with some built-in complex models such as Random Forests [5], Gradient Boosting [4] and tools for data preprocessing and classification. As mentioned though, we are only going to make use of the component *model_selection* and the *MinMaxScaler* to scale our input data between $[0, 1]$, more explanation on this further into the document.

Our Deep Learning framework of choice is Pytorch, developed and maintained by Facebook [46], and there’s a few reasons behind it. First and foremost, is the framework I have already experienced with, having used it before for a few deep learning projects of my own. Also, Pytorch is considerably more popular than its main competitor, Google’s Tensorflow, on the academic field [3]:

In the plot above, we can see the ratio between Pytorch papers and papers that use either Tensorflow or Pytorch at the top research conferences on the AI industry. The lines having a positive slope mean that there was a majority of Pytorch papers in each conference. Pytorch has experienced an exponential growth compared to Tensorflow in the past few years and it is for a number of reasons:

Pytorch is very similar to Python and Numpy, making it very easy to understand and allowing users to tune a variety of parameters of their models in a familiar way, while Tensorflow behaves a lot like a black-box, where interaction with the model is restricted and calling a method may

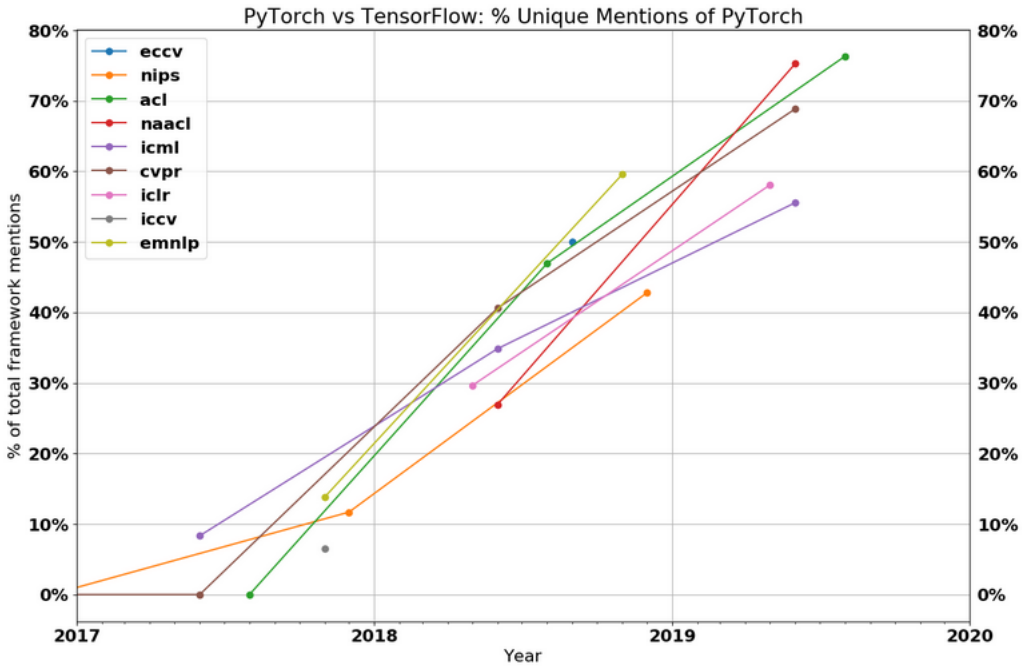



**Figure 4.5** Mentions of Pytorch on research papers [3] .

| CONFERENCE | PT 2018 | PT 2019 | PT GROWTH | TF 2018 | TF 2019 | TF GROWTH |
|---|---|---|---|---|---|---|
| CVPR | 82 | 280 | 240% | 116 | 125 | 7.7% |
| NAACL | 12 | 66 | 450% | 34 | 21 | -38.2% |
| ACL | 26 | 103 | 296% | 34 | 33 | -2.9% |
| ICLR | 24 | 70 | 192% | 54 | 53 | -1.9% |
| ICML | 23 | 69 | 200% | 40 | 53 | 32.5% |

**Figure 4.6** Growth of Pytorch and Tensorflow mentions on research [3] .

be doing several operations in the background the user is not aware of. Pytorch allows to create complex models from the ground up with it's "Layer" system and backpropagation mechanics are easy to understand and customize. Finally, Pytorch seems to be as fast or faster than Tensorflow [53]. To learn a bit more about how Pytorch operates, please check appendix 1.

The last library we are going to use in our experiments for hyperparameter search is Optuna. As we mentioned, on a given NN there are a few parameters that have to be chosen empirically to find the best NN configuration for a desired problem, there's no rule of thumb to choose the optimal number of neurons on a given layer. Some of this hyperparameters are:

- Number of neurons on a layer
- Depth of the network (number of layers)

- Learning rate
- Size of the kernel on a CNN
- Number of filters on a CNN

We have to run experiments with different values of those parameters to choose the best architecture, and Optuna is designed for just that. Optuna samples from given ranges of hyperparameters and searches for the ones with better results given a specific metric. More on Optuna on the second appendix, but here's a snippet of how Optuna works. We only need to define an objective function with the code to optimize and the possible values and Optuna will run it finding the best match

```
import optuna

def objective(trial):
    x = trial.suggest_uniform('x', -10, 10)
    return (x - 2) ** 2

study = optuna.create_study()
study.optimize(objective, n_trials=100)

study.best_params  # E.g. {'x': 2.002108042}
```

Now that we have explained what each library is for, let's start with the code rundown. After loading the data using Pandas, we have the input data on a Pandas Dataframe called *X* and the output data in the Dataframe *y*, which for our first experiment look like this:

```
X.head()
```

| | real | imag |
|---|---|---|
| 0 | 0.005323 | 0.023368 |
| 1 | 0.004153 | 0.025691 |
| 2 | 0.002178 | 0.026179 |
| 3 | -0.000515 | 0.024898 |
| 4 | -0.003793 | 0.022030 |

```
y.head()
```

| | real | imag |
|---|---|---|
| 0 | 1.67240 | 9.6376 |
| 1 | 1.23060 | 10.7480 |
| 2 | 0.53956 | 11.0360 |
| 3 | -0.58778 | 10.6440 |
| 4 | -2.06680 | 9.4430 |

**Figure 4.7** Input and output Dataframes.

The .head() method allows us to see the first entries on a given Dataset. Now, since we are going to make use of some transformations of the input data to train our network, we can add the columns like this:

```
X['mode1'] = ((np.sqrt((X['real']**2)+(X['imag']**2))))
X['mode2'] = ((np.sqrt((X['real']**2)+(X['imag']**2)))**2)
X['mode3'] = ((np.sqrt((X['real']**2)+(X['imag']**2)))**3)
X['mode4'] = ((np.sqrt((X['real']**2)+(X['imag']**2)))**4)
X['mode5'] = ((np.sqrt((X['real']**2)+(X['imag']**2)))**5)
```

Here, we are creating a column on the DataFrame $X$ and assigning values to it. We are performing operations with the entire columns here, which means, Pandas is performing the operations element-wise. On that snippet of code, the columns created are the ones corresponding to:

$$X[\text{"}mode1\text{"}] = |X_{in}|$$
$$X[\text{"}mode2\text{"}] = |X_{in}|^2$$
$$X[\text{"}mode3\text{"}] = |X_{in}|^3$$
$$X[\text{"}mode4\text{"}] = |X_{in}|^4$$
$$X[\text{"}mode5\text{"}] = |X_{in}|^5$$

And the Dataframe with the extra columns looks like this:

| | real | imag | mode1 | mode2 | mode3 | mode4 | mode5 |
|---|---|---|---|---|---|---|---|
| **0** | 0.005323 | 0.023368 | 0.023967 | 0.000574 | 0.000014 | 3.299364e-07 | 7.907476e-09 |
| **1** | 0.004153 | 0.025691 | 0.026025 | 0.000677 | 0.000018 | 4.587024e-07 | 1.193751e-08 |
| **2** | 0.002178 | 0.026179 | 0.026269 | 0.000690 | 0.000018 | 4.762131e-07 | 1.250984e-08 |
| **3** | -0.000515 | 0.024898 | 0.024903 | 0.000620 | 0.000015 | 3.846177e-07 | 9.578261e-09 |
| **4** | -0.003793 | 0.022030 | 0.022354 | 0.000500 | 0.000011 | 2.497063e-07 | 5.581962e-09 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **1105915** | -0.016707 | 0.003077 | 0.016988 | 0.000289 | 0.000005 | 8.328592e-08 | 1.414863e-09 |
| **1105916** | -0.018949 | -0.006080 | 0.019901 | 0.000396 | 0.000008 | 1.568425e-07 | 3.121258e-09 |
| **1105917** | -0.020247 | -0.015539 | 0.025523 | 0.000651 | 0.000017 | 4.243240e-07 | 1.082984e-08 |
| **1105918** | -0.020569 | -0.024644 | 0.032100 | 0.001030 | 0.000033 | 1.061746e-06 | 3.408205e-08 |
| **1105919** | -0.019958 | -0.032762 | 0.038362 | 0.001472 | 0.000056 | 2.165814e-06 | 8.308572e-08 |

1105920 rows × 7 columns

**Figure 4.8** Dataset of inputs with the extra columns .

The next step we are going to perform is data preprocessing. More specifically we are going to re-scale our data to the $[0,1]$ range, column by column. Scikit-learn maxes this operation easy by the MinMaxScaler object:

```
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()
X = scaler_x.fit_transform(X.values)
y = scaler_y.fit_transform(y.values)
```

Each value on a given column can be expressed as:

$$\frac{X_i - min(X)}{max(X) - min(X)}$$

Where $min(X)$ and $max(X)$ are the minimum and the maximum value on a given column. Performing this operation is good practice in machine learning. We perform this operation to stop the magnitude of a feature from disturbing the error calculation. Imagine that we have a feature on an example whose value is '5Kg', but we decide to write it as '5000g'. The vector representing that example on the feature space is going to be governed by the 5000g feature if the rest of the features have small values, and since we are going to use the euclidean distance to calculate the error, the network is going to focus on fixing the high magnitude feature instead if giving equal importance to all the features of the dataset. Here, feature scaling is specially important since as seen on Figure 3.8, the added features are exponentially smaller because they are powers of the I/Q components.

With that said, the Dataframe now looks like this:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | 0.361690 | 0.501862 | 0.203033 | 0.041387 | 0.008420 | 1.712912e-03 | 3.484727e-04 |
| **1** | 0.389392 | 0.530896 | 0.171798 | 0.029659 | 0.005108 | 8.796911e-04 | 1.515000e-04 |
| **2** | 0.431149 | 0.556134 | 0.150608 | 0.022813 | 0.003446 | 5.204400e-04 | 7.860738e-05 |
| **3** | 0.481407 | 0.575675 | 0.168127 | 0.028409 | 0.004788 | 8.070874e-04 | 1.360350e-04 |
| **4** | 0.533128 | 0.588569 | 0.220091 | 0.048615 | 0.010719 | 2.363436e-03 | 5.211105e-04 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **344059** | 0.478920 | 0.503634 | 0.028101 | 0.000817 | 0.000023 | 6.687172e-07 | 1.912287e-08 |
| **344060** | 0.482277 | 0.530643 | 0.080902 | 0.006621 | 0.000539 | 4.383963e-05 | 3.567250e-06 |
| **344061** | 0.488479 | 0.553685 | 0.127266 | 0.016310 | 0.002083 | 2.660143e-04 | 3.397280e-05 |
| **344062** | 0.494507 | 0.573302 | 0.167053 | 0.028048 | 0.004697 | 7.867202e-04 | 1.317575e-04 |
| **344063** | 0.497285 | 0.589978 | 0.199977 | 0.040154 | 0.008046 | 1.612333e-03 | 3.230863e-04 |

344064 rows × 7 columns

**Figure 4.9** Dataset scaled to the range [0, 1] .

Next, we define the neural network. We do so by using the *torch.nn* module from Pytorch and creating a class for it:

```
l_in = 10
class neuralNet(nn.Module):
    def __init__(self, n_channels, n_hidden, k_size, n_dimensions_in):
        super(redNeuronal, self).__init__()
        self.n_channels = n_channels
        self.k_size = k_size
        self.l_out = (l_in-k_size)+1
        self.conv1 = nn.Conv1d(n_dimensions_in, n_channels,
            k_size, padding=(k_size-1))
        self.fc1 = nn.Linear((self.l_out+2*(k_size-1))*n_channels, n_hidden)
        self.real = nn.Linear(n_hidden, 1)
        self.imag = nn.Linear(n_hidden, 1)
    def forward(self, X):
        x = F.leaky_relu(self.conv1(X))
        x = F.leaky_relu(self.fc1(x.view(-1, (self.l_out+2*(self.k_size-1))
```

```
            *self.n_channels)))
        out_real = F.leaky_relu(self.real(x))
        out_imag = F.leaky_relu(self.imag(x))
        return out_real, out_imag
```

First we define the layers our network is going to have. Our network is formed by a layer of convolutional neurons, followed by a fully connected layer and it's going to have two outputs, one for each $I$ and $Q$ component of the output of the power amplifier. The parameters we define on the initialization method are the following:

- `n_channels`: number of kernels we want the network to learn from the input data.
- `k_size`: depth of the network (number of layers)
- `l_out`: length of the output of each kernel. The length of each input *l_in* is 10, that is, we are taking into account the present sample and 10 past samples to train the neural network.
- `conv1`: the convolutional layer. We use $torch.nn.Conv1d$ to create it, and it requires as inputs the dimensions of the input, the number of kernels to use, the size of each kernel and we add padding to the input so its length is consistent. Padding is just adding a set of '0' at the beginning and end to the input vector so the convolution is consistent.
- $fc1$, *real*, *imag*: these are the fully connected layers we define for the neural network. We need to define the dimensions of the input and output of each one of them. On the first one, the number of dimensions in is going to be:

$$(l_out + padding * 2) \cdot n_{channels}$$

  And the output dimensionality is going to be determined by the paramete *n_hidden*. The other two layers simply define each output of our neural network, with *n_hidden* as their input dimensionality and 1 as their output. Here, we could also use one Linear layer instead of *real* and *imag* but with 2 as the output dimensionality, the result is exactly the same.

We choose as activation function the leaky ReLU function. We do so to prevent the dying ReLU problem [38], where if the input is too negative, the gradient is going to be 0 and the neuron is going to produce 0 as output for ever, leading to an useless neuron. The leaky ReLU function is the following:

$$f(x) = \begin{cases} \alpha x & x < 0 \\ x & x > 0 \end{cases}$$

Where $\alpha$ is a small constant. The plot of the leaky ReLU is the following:

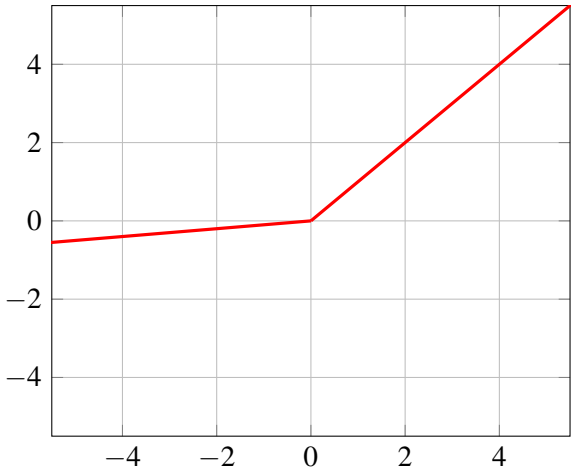



**Figure 4.10** Leaky ReLU plot.

The class *neuralNet* returns then two vectors of shape $(batchsize, 1)$ representing the real and imaginary value for each example we feed in. The decisions on this architecture are made empirically. We chose the depth of the architecture by performing several experiments with varying depth and at high depths the network experienced vanishing gradients, so the weights on the layers near the output are not being updated [18]. We also chose the length of the input data by following [56], where the memory depth is 3, being inputs further back less relevant to the behavioral modeling. Length 10 was a number with high enough impact for us to measure as well as keeping the complexity and number of parameters reasonable. As we see, that leaves the number of channels and number of hidden neurons as variables of our architecture. The optimal values of these variables vary from experiment to experiment, so we are using hyperparameter tuning to optimize these values, see appendix 2.

The next step then is to define the Dataset class, which is going to help us iterate over the Dataset. We define the class as follows:

```
class _data_(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return self.X.shape[0]
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
batch_size = 220
```

We can find more information about the Dataset class on appendix 1, but we are basically defining here how to return the data and how to calculate the total length of the dataset. This is required by Pytorch to iterate over the data. Here we also define the batch size. We chose the batch size to be 220 since it gave us a big enough set of examples to perform gradient steps without the loss oscilating too much.

Now we define the core functions for the experiments. The first function is the training loop for the network, which looks like this:

```
def train_network(network, optimizer, scheduler, X_train, y_train,
    total_epochs, print_epoch=False):
```

```
  datos = _data_(X_train, y_train)
  loader = DataLoader(dataset=datos, batch_size=batch_size, shuffle=True)
  train_loss = []
  for epoch in range(total_epochs):
    for x, y_true in loader:
        network.cuda()
        x, y_true = x.cuda(), y_true.cuda()
        optimizer.zero_grad()
        out_real, out_im = network(x.float())
        loss = ((((out_real-y_true[:, 0].view(-1, 1))**2)+(
            (out_im-y_true[:, 1].view(-1, 1))**2)).mean())/2
        loss.backward()
        optimizer.step()
        train_loss.append(loss.detach().cpu().numpy())
    scheduler.step()
    if print_epoch and epoch%50==0:
      print(epoch)
```

This function takes as inputs the network to train, an optimizer, an scheduler, the input and output sets and the number of epochs to train the network for. First, it creates the loader to iterate over the input/output sets, more information on loaders on appendix 1. Then, we set the network and input/output tensors on CUDA mode for training on GPUs, clear all the gradients and compute the outputs of the neural network. After that, we calculate the half MSE and store its value on a single-valued tensor called loss. We call the backward method on the tensor, which calculates all the gradients of the network and then call *optimizer.step*() to update all the weights on the neural network. Then, we save the loss on an *Arraylist* in case we want to plot it for model evaluation. Lastly, we call *scheduler.step*() to lower the learning rate of the optimizer. We do this to prevent overshooting the minima of the loss function. Pytorch has scheduler objects which, given an optimizer will gradually reduce the learning rate over the epochs so we get finer adjustments on the later stages of the training loop.

The next function we define is the test function. The test function evaluates the network on previously unseen examples to test its generalization capabilities, that is, to see how well a trained network performs:

```
def test_network(network, X_test, y_test):
  with torch.no_grad():
    preds = network(torch.from_numpy(X_test).cuda().float())
    preds = np.concatenate(
        (preds[0].detach().cpu().numpy(),
            preds[1].detach().cpu().numpy()), axis=1)
  preds = scaler_y.inverse_transform(preds)
  y_test = scaler_y.inverse_transform(y_test)
  y_test = pd.DataFrame(y_test, columns=["real", "imag"])
  y_test['real_pred'] = preds[:, 0]
  y_test['imag_pred'] = preds[:, 1]
  y_test['error_abs'] = np.sqrt(((y_test['real']-y_test['real_pred'])**2)+(
    (y_test['imag']-y_test['imag_pred'])**2))
  y_test['error'] = (y_test['error_abs']**2).mean()
  y_test['norma_abs'] = np.sqrt((y_test['real']**2)+(y_test['imag']**2))
  y_test['norma'] = (y_test['norma_abs']**2).mean()
```

```
  y_test['error/norma'] = y_test["error"] / y_test['norma']
  y_test['nmse'] = 10*np.log10(y_test['error/norma'])
  return y_test['nmse'].mean()
```

Given a network and an input/output sets, the function first disables the gradients with the line with $torch.no_grad()$. Pytorch keeps track of all the computations on tensors to compute the gradients later, and by setting that clause we are telling Pytorch to ignore the operations made there, since we are testing the network, not training it. Next, we apply the inverse transformation on the data that we scaled between $[0, 1]$ to calculate the NMSE. Then, we create a DataFrame to hold the values of errors and absolute values of the NMSE function and make the calculations column-wise. Lastly, we return the total NMSE for that sample.

The last helper function we are going the define is called *hyperparameter_search*, and it looks like this:

```
def hyperparam_search(network, optimizer, scheduler, trial, X_train, y_train):
  X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train, shuffle=False, train_size=0.90)
  all_nmse = []
  for epoch in range(15):
    train_network(network, optimizer, scheduler, X_train, y_train, 1, False)
    nmse = test_network(network, X_val, y_val)
    all_nmse.append(nmse)
    trial.report(np.mean(all_nmse), epoch)
    if trial.should_prune():
        raise optuna.TrialPruned()
    if epoch%25==0:
      print(epoch)
  print(all_nmse)
  return np.mean(all_nmse)
```

This function first splits the given X and y sets into a training and validation set. We train the network on a set and for each epoch we test the network on the validation set and add the value to an *Arraylist*. On this method there's also defined the $trial.report$ and $trial.should_prune()$ needed for a trial pruning of the Optuna module. Basically Optuna can stop a trial for its hyperparameter search when it's not successful enough to save computation time, and it requires both methods to be called on the hyperparameter search loop. The function lastly returns the mean of all the values stored on the *Arraylist* for hyperparameter selection.

Now, onto the main loop of the experiments, which looks like this:

```
columns_for_experiments = [[0, 1], [0, 1, 2], [0, 1, 3], [0, 1, 4],
[0, 1, 2, 3], [0, 1, 2, 4], [0, 1, 2, 3, 4], [0, 1, 2, 4, 6],
[0, 1, 2, 3, 4, 5]]
for elem in series_no_indexes:
    print("With columns:")
    print(elem)
    data_to_input = np.zeros((X.shape[0]-l_in, len(elem), l_in))
    counter = 0
    for column in elem:
        for index in range(X.shape[0]-l_in):
            data_to_input[index, counter, :] = X[index:index+l_in, column].transpose()
        counter += 1
    X_train, X_test, y_train, y_test = train_test_split(data_to_input, y[:-l_in],
        shuffle=True, train_size=0.90)
```

```
#hyperparameter tuning
def objective(trial):
  n_channels = trial.suggest_int('n_channels', 50, 120, 10)
  n_hidden = trial.suggest_int("n_hidden", 10, 40, 10)
  network = neuralNet(n_channels, n_hidden, 5, data_to_input.shape[1]).cuda()
  optimizer = torch.optim.Adam(network.parameters(), lr=1e-3)
  scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size = 2, gamma=0.9)
  return hyperparam_search(network, optimizer, scheduler, trial, X_train, y_train)

study = optuna.create_study(direction="minimize",
    pruner=optuna.pruners.MedianPruner())
study.optimize(objective, n_trials=15)
print(study.best_params)

#Now we train the proper network
network = neuralNet(study.best_params['n_channels'],
    study.best_params['n_hidden'], 5, data_to_input.shape[1]).cuda()
optimizer = torch.optim.Adam(network.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size = 2, gamma=0.9)

train_network(network, optimizer, scheduler, X_train, y_train, 100, True)

torch.save(network.state_dict(), "./weights" + str(elem))
print(test_network(network, X_test, y_test))
```

On this loop, first we define the columns we are going to performs the experiments with. Then, iterating over the experiments, we create an the *data_to_input* tensor holding the transformed input data in a 3D tensor, the first dimension being each single example, the second dimension is the different features and then last dimension is the past samples for memory. Then, we split the tensor into a train and test set. Next, we define the objective function that Optuna is going to call to perform the hyperparameter search. We define the ranges we want Optuna to look into, create a neural network with the sampled parameters, along with an optimizer and scheduler and call the *hyperparameter_search* function, which will train the network and test it, and Optuna will choose the networks that performs the best on a given set of columns. We perform 15 trials on each experiment, then choose the best parameters Optuna found and create the final neural network to be trained for 100 epochs. We found that a *kernel_size* of 5 performed the best, along with a learning rate of 0.001, when training for 100 epochs. We chose 100 epochs since they are enough epochs to allow the model to converge and find the minima and it kept the training computation time to a reasonable length. We then save the model weights on a file to later use and show on screen the final performance value of the network on each experiment.

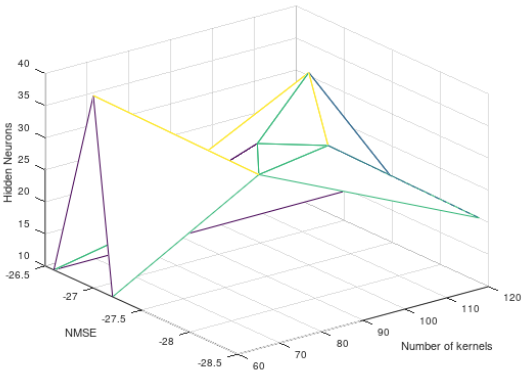
# 5 Experiment Results

In this section we will show how the CNN performs on a group of Datasets. Each Dataset contains approximately the same number of sampled $I$ and $Q$ components of the input/output of a RFPA. Performance varies notably between Datasets, with the CNN performing the worst for less accurate samples. The parameters used to train the network are exactly the same across all experiments, and are the ones explained in the previous chapter. We let then Optuna free to choose the best performing parameters for each experiment on each Dataset. All the Datasets contain a little over 360000 samples of input/output signals.

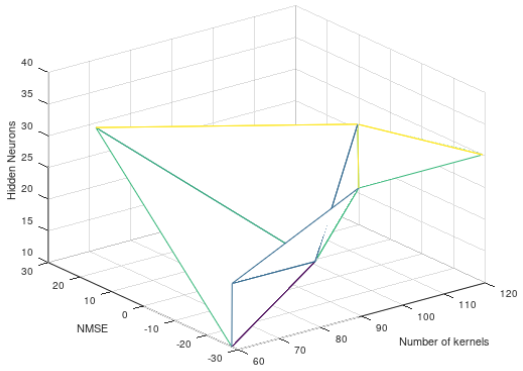
## 5.1 Experiment 1 - 15MHz 5G-NR at -23 dBm

On this experiment, we train the network on a 15MHz 5G-NR(New Radio) signal from the Chalmers Online Laboratory. The Chalmers weblab was launched on 2014 and it's used worldwide for research and education. Users can submit input signals and the Weblab will return the distorted output signals from a variety of amplifiers. We test a 5G-NR signal. 5G-NR is designed to be the global standard for 5G communication, with the bulk of users being cellphone users, we can see then the interest in increasing the amplifiers efficiency to improve battery life and build a reliable communication system with DPD. More information for state-of-the-art research on DPD for 5G-NR signals can be found in [13].

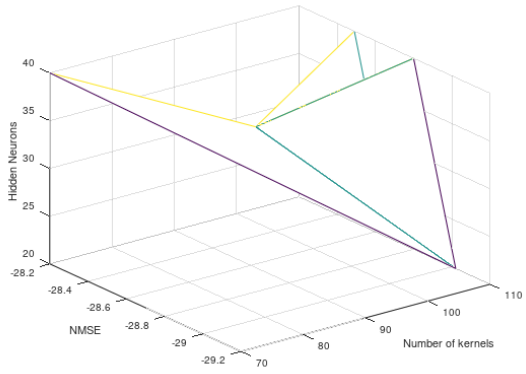
The results of the Hyperparameter search can be found on the plots below. The plots represent the surfaces of the different values for each input data augmentation to find the best NMSE. The plots are created by triangulating the points returned by Optuna with the `trimesh`function of Octave. Here we can see how the performance for the first epochs varies about 1 to 2 dB depending on the parameters chosen.



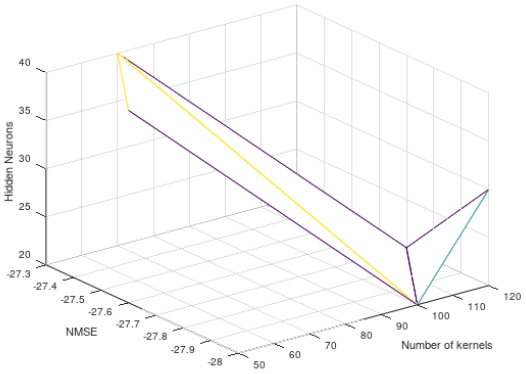

**(1)** With no additional inputs.



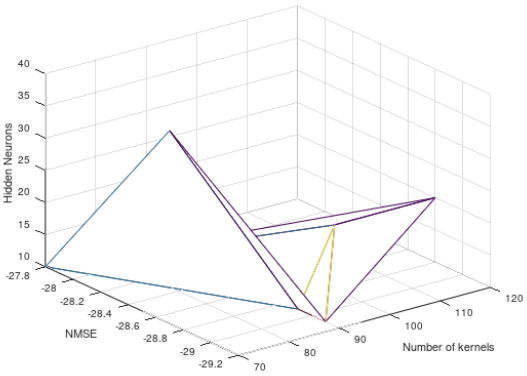

**(2)** With $|X(n)|$.



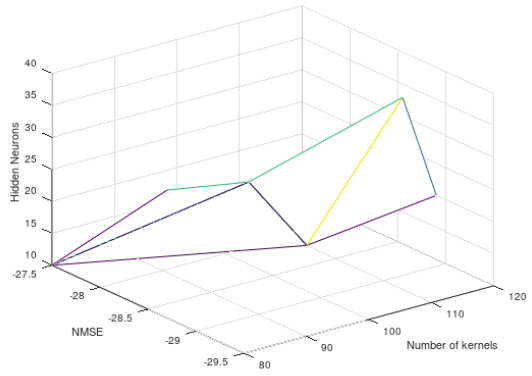

**(3)** With $|X(n)|^2$.



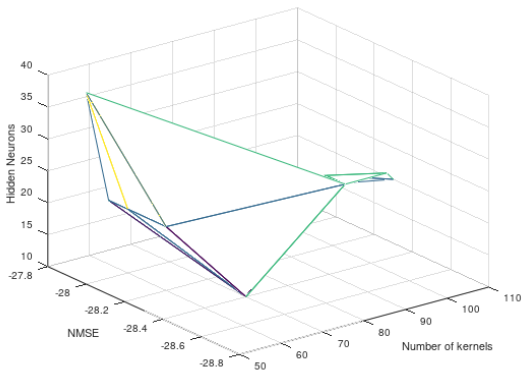

**(4)** With $|X(n)|^3$.



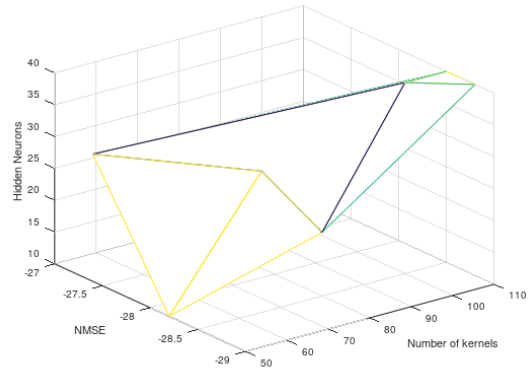

**(5)** With $|X(n)|, |X(n)|^2$.



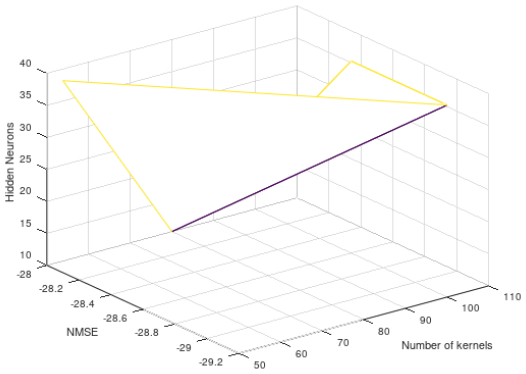

**(6)** With $|X(n)|, |X(n)|^3$.




**(7)** With $|X(n)|, |X(n)|^2, |X(n)|^3$.




**(8)** With $|X(n)|, |X(n)|^3, |X(n)|^5$.




**(9)** With $|X(n)|, |X(n)|^2, |X(n)|^3, |X(n)|^4$.

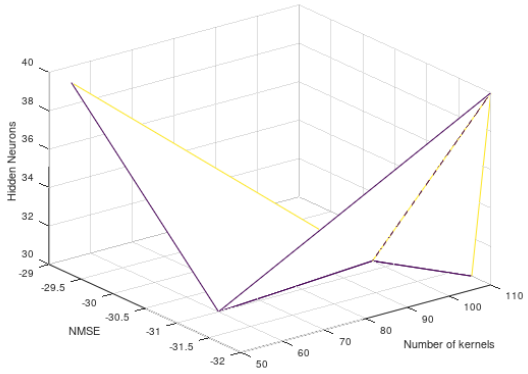
Performance of the CNN is shown on the table below:

| Amplitude Terms (In addition to $I$ and $Q$) | Optimal No. of kernels | Optimal No. of neurons on hidden layer | NMSE(dB) |
|---|---|---|---|
| None | 120 | 20 | -31.01 |
| $\|X(n)\|$ | 120 | 30 | -32.70 |
| $\|X(n)\|^2$ | 110 | 20 | -31.51 |
| $\|X(n)\|^3$ | 120 | 30 | -31.83 |
| $\|X(n)\|, \|X(n)\|^2$ | 90 | 10 | -32.92 |
| $\|X(n)\|, \|X(n)\|^3$ | 110 | 40 | -32.85 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3$ | 90 | 30 | -33.42 |
| $\|X(n)\|, \|X(n)\|^3, \|X(n)\|^5$ | 110 | 40 | -34.21 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3, \|X(n)\|^4$ | 100 | 40 | -34.19 |

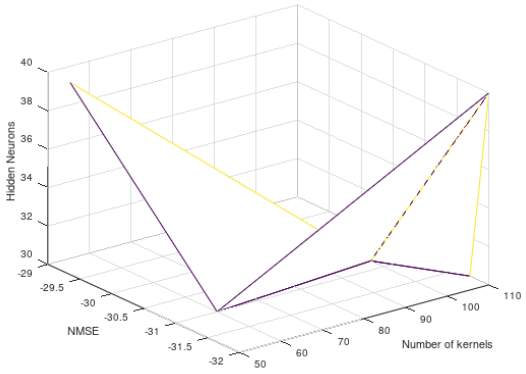
## 5.2 Experiment 2 - Class J PA at -31 dBm

Class J PA were proposed in 2003 in [16], they provide similar efficiency and linearity as the Class-AB and Class-B but across a broader range of frequencies.

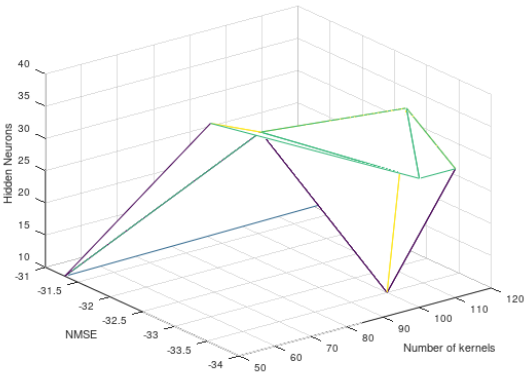
The results of the Hyperparameter search can be found below.



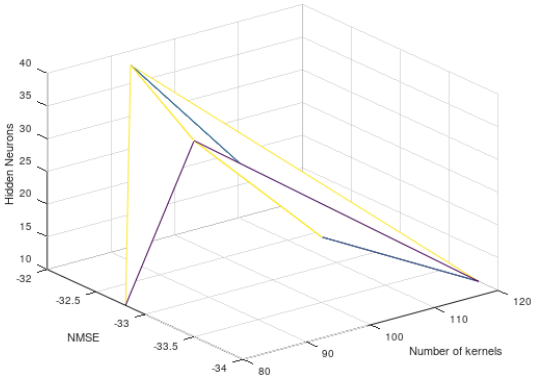

**(10)** With no additional inputs.



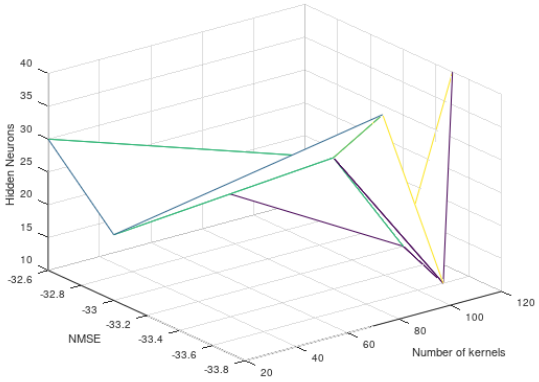

**(11)** With $|X(n)|$.



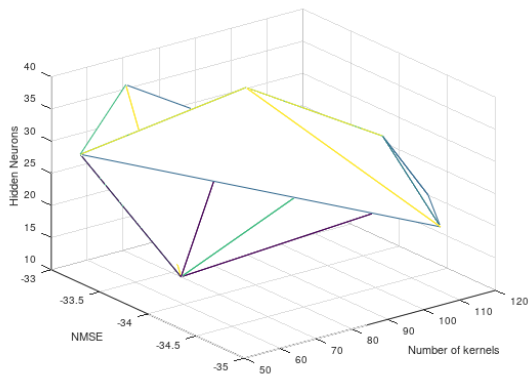

**(12)** With $|X(n)|^2$.



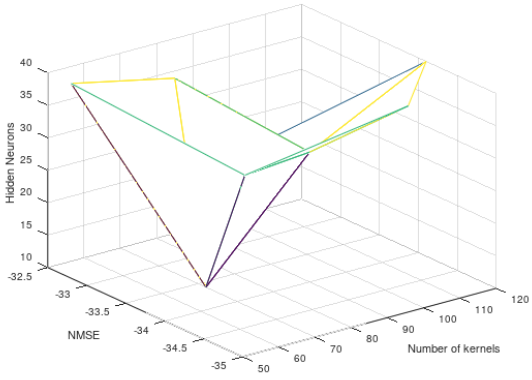

**(13)** With $|X(n)|^3$.



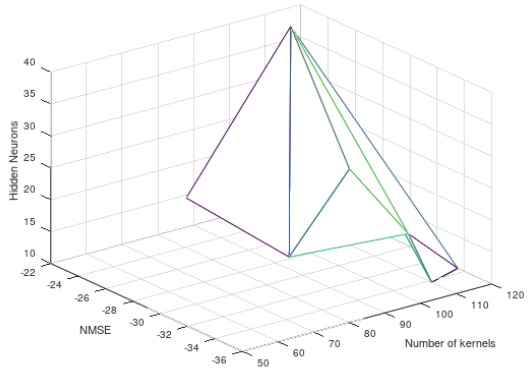

**(14)** With $|X(n)|, |X(n)|^2$.



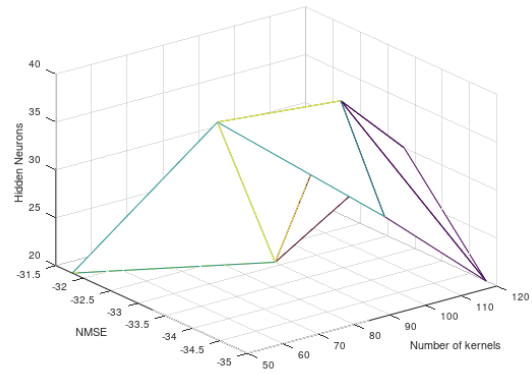

**(15)** With $|X(n)|, |X(n)|^3$.




**(16)** With $|X(n)|, |X(n)|^2, |X(n)|^3$.




**(17)** With $|X(n)|, |X(n)|^3, |X(n)|^5$.



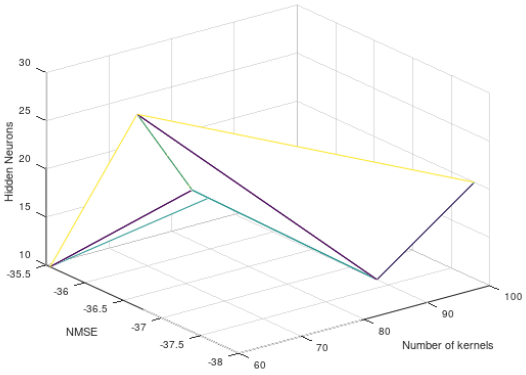

**(18)** With $|X(n)|, |X(n)|^2, |X(n)|^3, |X(n)|^4$.

Performance of the CNN is shown on the table below:

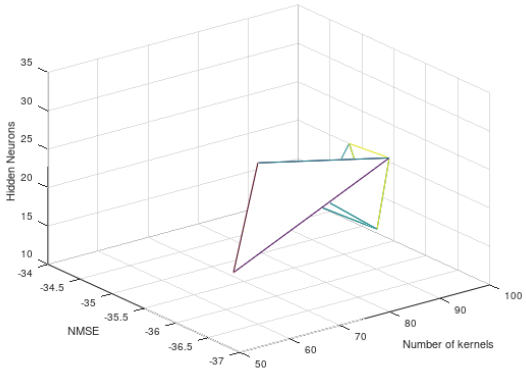
| Amplitude Terms (In addition to $I$ and $Q$) | Optimal No. of kernels | Optimal No. of neurons on hidden layer | NMSE(dB) |
|---|---|---|---|
| None | 110 | 30 | -40.00 |
| $\|X(n)\|$ | 120 | 30 | -41.00 |
| $\|X(n)\|^2$ | 100 | 30 | -40.46 |
| $\|X(n)\|^3$ | 120 | 10 | -39.97 |
| $\|X(n)\|, \|X(n)\|^2$ | 80 | 40 | -41.19 |
| $\|X(n)\|, \|X(n)\|^3$ | 110 | 20 | -41.25 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3$ | 100 | 40 | -41.55 |
| $\|X(n)\|, \|X(n)\|^3, \|X(n)\|^5$ | 100 | 20 | -41.67 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3, \|X(n)\|^4$ | 90 | 30 | -42.02 |

## 5.3 Experiment 3 - LTE at 15MHz at -22.5 dBm

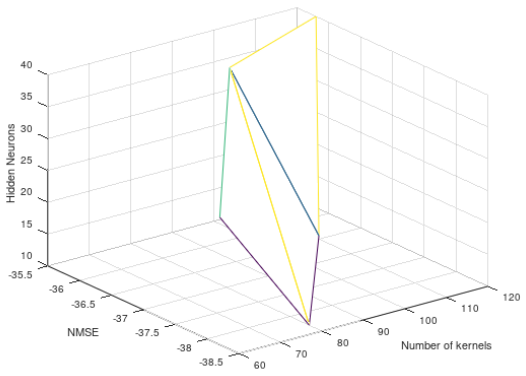
LTE signals where generated as in [10]. The amplifier here is the Cree amplifier, a Class-AB amplifier at -22.5 dBm.



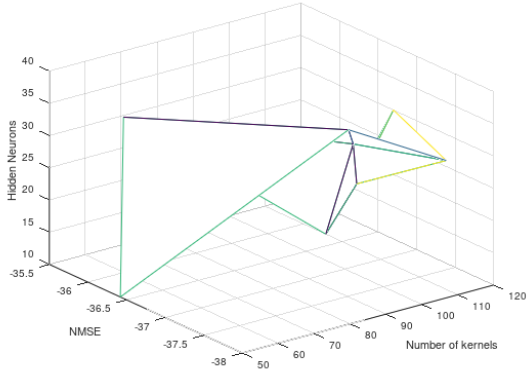

**(19)** With no additional inputs.



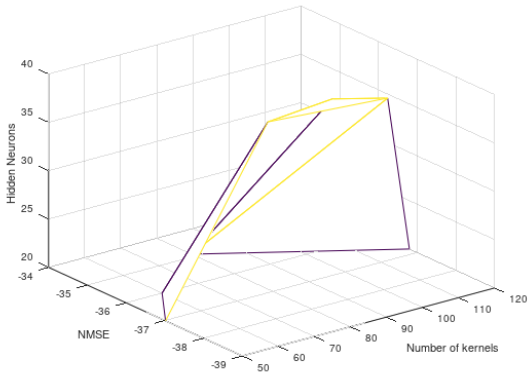

**(20)** With $|X(n)|$.



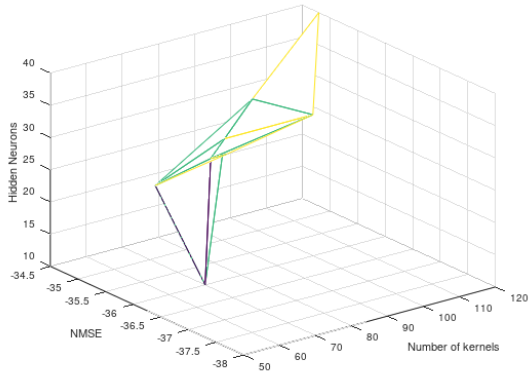

**(21)** With $|X(n)|^2$.



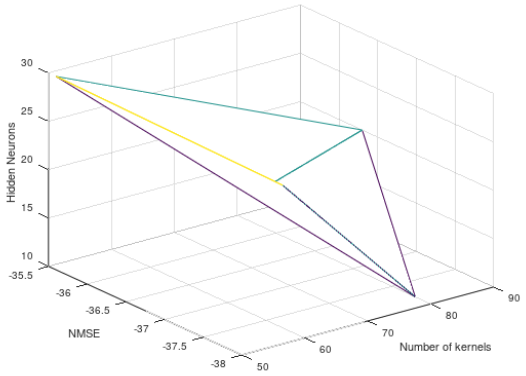

**(22)** With $|X(n)|^3$.



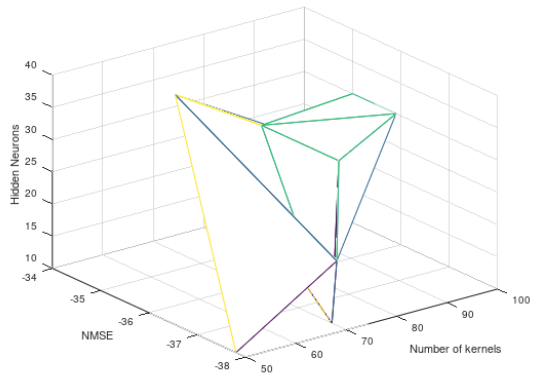

**(23)** With $|X(n)|, |X(n)|^2$.



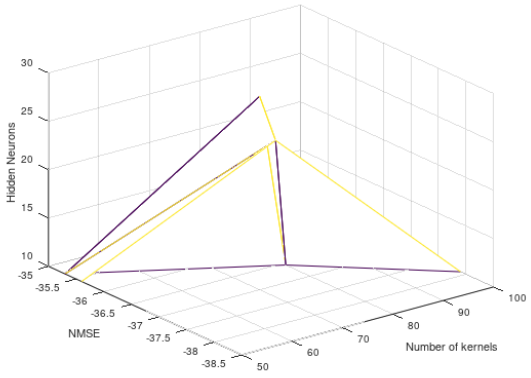

**(24)** With $|X(n)|, |X(n)|^3$.




**(25)** With $|X(n)|, |X(n)|^2, |X(n)|^3$.




**(26)** With $|X(n)|, |X(n)|^3, |X(n)|^5$.



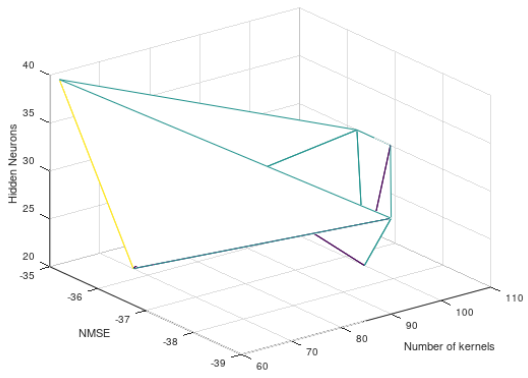

**(27)** With $|X(n)|, |X(n)|^2, |X(n)|^3, |X(n)|^4$.

Performance of the CNN is shown on the table below:

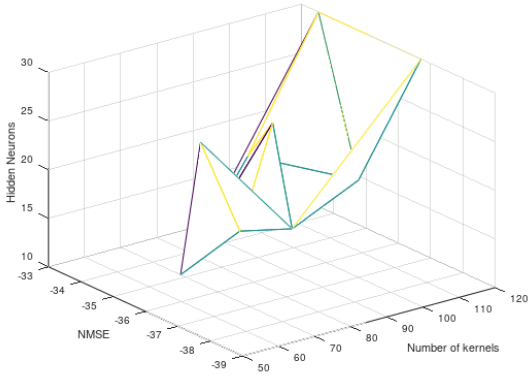
| Amplitude Terms (In addition to $I$ and $Q$) | Optimal No. of kernels | Optimal No. of neurons on hidden layer | NMSE(dB) |
|---|---|---|---|
| None | 100 | 20 | -48.36 |
| $\|X(n)\|$ | 80 | 30 | -49.22 |
| $\|X(n)\|^2$ | 80 | 10 | -48.59 |
| $\|X(n)\|^3$ | 80 | 40 | -48.27 |
| $\|X(n)\|, \|X(n)\|^2$ | 100 | 40 | -49.54 |
| $\|X(n)\|, \|X(n)\|^3$ | 50 | 40 | -49.15 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3$ | 80 | 10 | -50.19 |
| $\|X(n)\|, \|X(n)\|^3, \|X(n)\|^5$ | 50 | 10 | -50.20 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3, \|X(n)\|^4$ | 60 | 30 | -50.25 |

## 5.4 Experiment 4 - LTE at 15MHz at -35.5 dBm

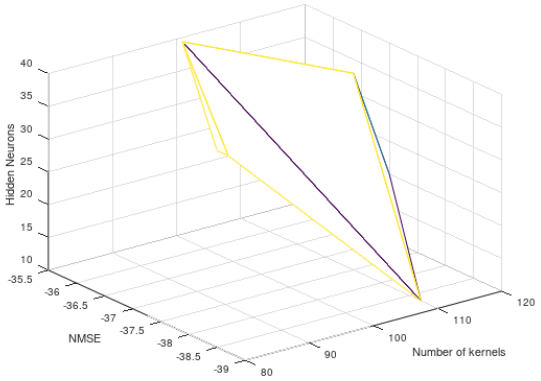
Again, LTE signals where generated as in [10]. The same amplifier as in Experiment 3 is used but now the power is lowered to -35.5 dBm



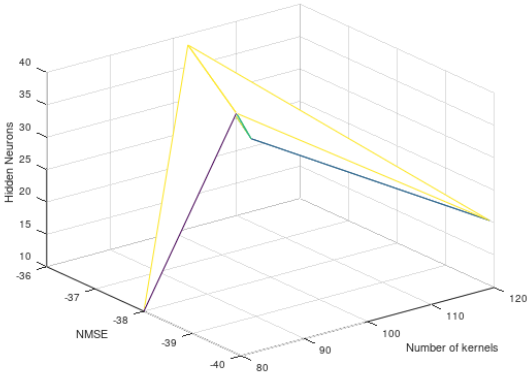

**(28)** With no additional inputs.



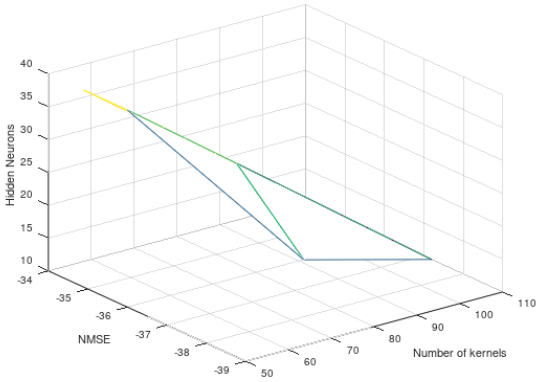

**(29)** With $|X(n)|$.



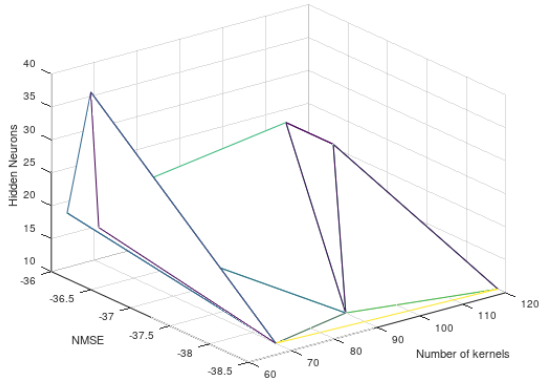

**(30)** With $|X(n)|^2$.



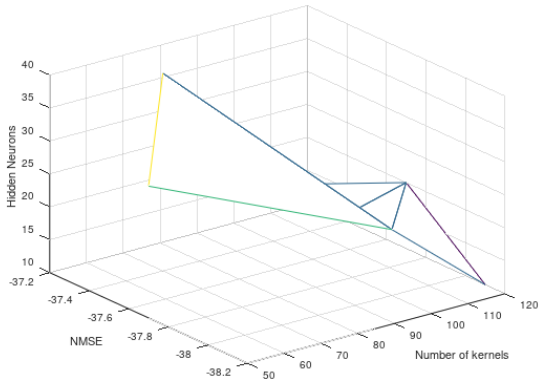

**(31)** With $|X(n)|^3$.



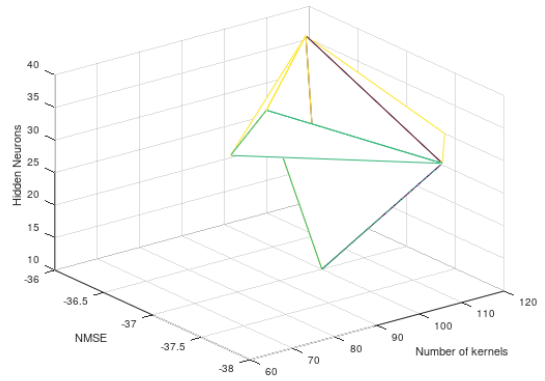

**(32)** With $|X(n)|, |X(n)|^2$.



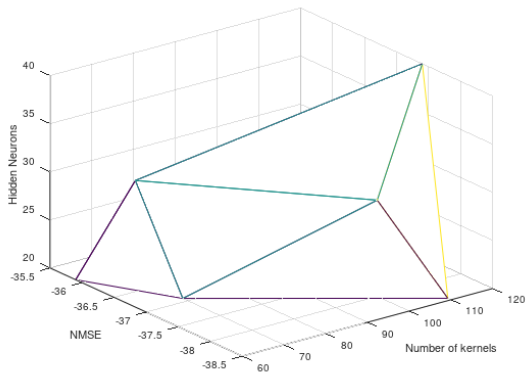

**(33)** With $|X(n)|, |X(n)|^3$.




**(34)** With $|X(n)|, |X(n)|^2, |X(n)|^3$.




**(35)** With $|X(n)|, |X(n)|^3, |X(n)|^5$.




**(36)** With $|X(n)|, |X(n)|^2, |X(n)|^3, |X(n)|^4$.

Performance of the CNN is shown on the table below:

| Amplitude Terms (In addition to $I$ and $Q$) | Optimal No. of kernels | Optimal No. of neurons on hidden layer | NMSE(dB) |
|---|---|---|---|
| None | 90 | 30 | -50.64 |
| $\|X(n)\|$ | 70 | 20 | -51.21 |
| $\|X(n)\|^2$ | 80 | 40 | -50.95 |
| $\|X(n)\|^3$ | 120 | 20 | -50.83 |
| $\|X(n)\|, \|X(n)\|^2$ | 70 | 20 | -52.09 |
| $\|X(n)\|, \|X(n)\|^3$ | 120 | 10 | -51.98 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3$ | 120 | 10 | -52.02 |
| $\|X(n)\|, \|X(n)\|^3, \|X(n)\|^5$ | 110 | 30 | -52.23 |
| $\|X(n)\|, \|X(n)\|^2, \|X(n)\|^3, \|X(n)\|^4$ | 110 | 20 | -52.76 |

## 5.5 Comments on the experiments

Results of the presented experiments confirm that adding more terms to the input of the CNN does come with a small increase in performance, as referenced in [56]. Furthermore, the inclusion of the term $|X[n]|$ allows the network to generate the even-order intermodulation terms, reaching lower NMSE scores, but adding redundant terms to the network such as $|X[n]|^2$, $|X[n]|^3$ does not improve the networks performance as much. However, the more combinations we add to the input network,

the better it performs, as we can see in the last rows of the experiments'tables, bringing around -2dB of improvement compared with feeding the network only the I/Q components.

We can also observe that an increase in the networks complexity, or the number of parameters and kernels, does not come with an increase in performance. We can see that a higher input dimensionality does not imply a higher complexity, with simpler networks architecture outperforming more complex ones with different input dimensionality. As we can see, hyperparameter selection on deep learning is still an empirical approach, but recent research on AutoML [23] may help find a more comfortable solution to current machine learning pipelines.

# 6 Conclusion

On this document we observed that CNNs can model a variety of amplifiers and powers regardless of where distortions come from, all while mantaining the number of learning parameters low. The network architectures explored on this document are not very complex, which allows us to fine-tune them to adapt better to each experiment, as we saw on the hyperparameter search section. We also confirmed previous research that a higher number of feature transformations of the input, more specifically adding $|X(n)|^k$ terms, greatly increase the accuracy of networks with similar architectures.

## 6.1 Future lines of investigation

More sophisticated architectures of CNNs, such as TCN [9], which include the use of Residual blocks and several convolutional layers may have greater success on RFPA behavioral modeling comparable or surpassing current numerical approaches. Although they include a higher number of parameters and overall complexity, their ability to model accurately memory effects by causal convolution may come with better overall accuracy. More research will be needed on this topics to compare performances.

# Appendix A
# Pytorch

In this section we are going to explain a bit what Pytorch is more in depth and how it works. Pytorch is our library of choice for the neural network architecture we proposed and its training loop.

The building block of Pytorch is the tensor. A tensor can be described as a multidimensional matrix. Pytorch provides several methods to create tensors, such as the *.zeros*() method, *.randn*() method or *.empty*() method, and also creating tensors from Numpy arrays.

```
x = torch.zeros(5, 3)
print(x)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

Performing operations on tensors is really simple, it looks a lot like performing operations with Numpy arrays, so it's very 'pythonic':

```
x = torch.zeros(3, 3)
y = torch.randn(3, 3)
z = x+y
print(z)
```

```
tensor([[-0.1215,  0.0835, -0.7474],
        [-0.1995, -0.4273, -0.3058],
        [-0.0596, -0.1303,  0.5063]])
```

When it comes to performance and fast computation, Pytorch allows tensors to be run on CUDA devices. CUDA (Compute Unified Device Architecture) [43] is a parallel computing platform developed by NVIDIA. CUDA devices are essentially NVIDIA GPUs highly optimized for Deep Learning and tensor operations. CUDA-enabled GPUs offers a set of low-level, high performance instructions and exposes said API to a variety of programming languages. When we want operations to be performed by a CUDA-enabled device, we just have to call *.cuda*() on a given tensor or model. This will tell Pytorch to move those objects to the GPU as well as all the tensors resulting from their operations.

Since performing operations with CUDA is really straight-forward, we can write code that runs either on GPU or CPU seamlessly, we just need to check if a CUDA device is available to the system and call the *.cuda()* method on our tensors and models:

```
if torch.cuda.is_available():
  device = 'cuda'
else:
  device = 'cpu'

x = torch.randn(4, 4, device=device)
print(x)
```

```
tensor([[ 0.4288, -0.2511, -1.3960, -1.4391],
        [ 0.7458, -0.0637,  0.4903, -0.6889],
        [-1.0640,  0.5303, -2.5384, -0.2693],
        [-0.6191,  0.1749, -0.5994, -0.2116]], device='cuda:0')
```

Pytorch CUDA implementation offers some other low-level operations to the user to fine tune different aspects such as memory allocation and distributed computation across several GPUs.

Pytorch's autograd package is the core package when working with neural networks. Autograd provides automatic differentiation for all operations on tensors. All operations we perform on a tensor are tracked by Pytorch. When we calculate the Loss function and call *backward()* on the resulting tensor, all the gradients are computed automatically and stored on an attribute called grad on each tensor. We can choose not to track operations by using the *detach()* method on a tensor or using the sentence with $torch.no_grad()$.

```
  with torch.no_grad():
    preds = network(torch.from_numpy(X_test).cuda().float())
    preds = np.concatenate((preds[0].detach().cpu().numpy(),
        preds[1].detach().cpu().numpy()), axis=1)
```

Pytorch also stores the operation on the tensor on an attribute called `grad_fn`that stores the Function reference that created the tensor:

```
x = torch.zeros(2, 2, requires_grad=True)
y = x+2
print(y)
```

```
tensor([[2., 2.],
        [2., 2.]], grad_fn=<AddBackward0>)
```

Autograd is an engine for computing vector-Jacobian products. Given $\vec{y} = f(\vec{x})$, the gradient of $\vec{y}$ with respect to $\vec{x}$ is the Jacobian matrix:

$$J = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \cdots & \frac{\delta y_1}{\delta x_n} \\ \vdots & \ddots & \vdots \\ \frac{\delta y_n}{\delta x_1} & \cdots & \frac{\delta y_m}{\delta x_n} \end{bmatrix}$$

For a given vector:

$$v = (v_1, v_2, \ldots, v_m)^T$$

Autograd valculates $v^T * J$. If v is the gradient of a scalar function, that is, $i = g(\vec{y})$, then:

$$v = (\frac{\delta i}{\delta y_1}, \ldots, \frac{\delta i}{\delta y_m})^T$$

Then the vector-Jacobian dot product would be the gradient of i with respect to  would be:

$$J^T * v = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \cdots & \frac{\delta y_1}{\delta x_n} \\ \vdots & \ddots & \vdots \\ \frac{\delta y_n}{\delta x_1} & \cdots & \frac{\delta y_m}{\delta x_n} \end{bmatrix} \begin{bmatrix} \frac{\delta i}{\delta y_1} \\ \vdots \\ \frac{\delta i}{\delta y_m} \end{bmatrix} = \begin{bmatrix} \frac{\delta i}{\delta x_1} \\ \vdots \\ \frac{\delta i}{\delta x_m} \end{bmatrix}$$

Thus building computing graphs and its derivatives becomes really easy using vector-Jacobian products. More information about specifics on Autograd can be found on [1].

To build neural networks architectures, Pytorch provides the nn.Modules library and a method forward(input) that produces an output. The general training procedure of a neural network goes like this:

```
defineNetwork();
for number of epochs do
    for batch of data do
        output = network(input)
        error = calculateLoss(output, realValues)
        calculateGradients(error)
        adjustWeights()
    end
end
```

**Algorithm 1:** Training loop of neural network

To define a network on Pytorch, we have to create a class which inherits from *nn.Module* and define two methods: the constructor or initialization method were we'll define the layers of our network, and the forward method, where we'll define the order of computation of the layers of the neural network. We can take the example of convnet from Pytorch's documentation:

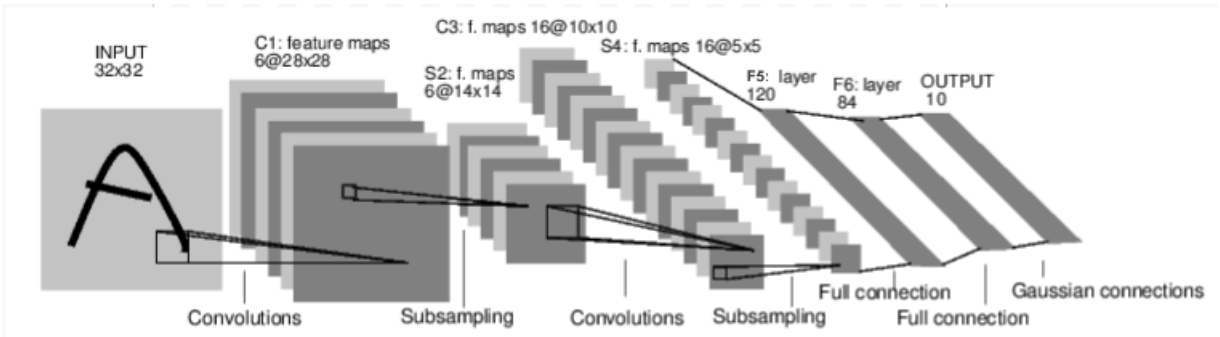



**Figure A.1** Convnet general architecture .

The network is a 2D CNN for image recognition. Two convolutional layers are needed, along with two fully connected layers and two pooling layers. The pooling layers basically downsize the feature maps to capture more general features. The code to define the neural network above would be:

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):

        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, 16*6*6)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

The layers are defined on the *__init__* (constructor) function. On the *torch.nn* submodule we can find the different layers Pytorch offers. Each layer and its inputs vary from type to type, for example, the Linear layer represents a fully connected standard layer, where we need to provide the number of dimensions in and out of the layer. The convolutional layers on the other hand require you to define other aspects such as the kernel sizes and number of kernels. Then, once we have our layers in place, we need to define the forward function. The forward function defines the order of computations of the layers. On the example above, we apply first the conv1 layer, then apply the activation function to all the outputs of the conv1 by using the *torch.Functional* module, which contains the activation functions and other utilities, and then apply the pool function. We do the same for the conv2 layer and then reshape the x tensor, since it has more than 2 dimensions, to a 2-dimensional one to apply the linear functions to it.

The backward function is then automatically created by Autograd with all the gradient computation and the network can be called as Net(input).

To feed data into a given network, pytorch provides the Dataset and Dataloader modules. To create a Dataset, we must subclass the *torch.utils.data.Dataset* class and overwrite two methods the *__getitem()__* method, where we define how the data is returned when called, and the *__len__()* method, where we define how the total length of the dataset is going to be computed. We can take our Dataset as an example:

```
class _data_(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return self.X.shape[0]
```

```
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

Once we have our Dataset defined, we can create a loader that will return an iterable over the Dataset. While creating the Dataloader, we can define the batch size and if the data is to be shuffled (randomized) when feeding the data. We define the Dataloader as below, and we can iterate over the dataset with a simple for loop.

```
X_train = torch.randn(2048, 3)
y_train = torch.randn(2048, 1)
class _data_(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return self.X.shape[0]
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
datos = _data_(X_train, y_train)
loader = DataLoader(dataset=datos, batch_size=100, shuffle=True)
for x, y in loader:
  print(x.shape)
  print(y.shape)
  break
```

```
torch.Size([100, 3])
torch.Size([100, 1])
```

Lastly, Pytorch offers several optimizers on the *torch.optim* module. Most of the state-of-the-art algorithms are implemented there. To use an optimizer, we must instantiate it with the parameters (weights) of a neural network, the desired learning rate and other parameters specific to each algorithm. For example:

```
    optimizer = torch.optim.Adam(network.parameters(), lr=1e-3)
```

Here we are creating an ADAM optimizer [paper adam] instead of an SGD optimizer and giving it a learning rate of 0.001. Once we have our optimizer created, we need to call its method step() after calculating the gradients, like so

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

# Appendix B
# Optuna

As we explained, when building and training a neural network, there are many parameters that have to be chosen empirically for the problem at hand, called hyperparameters. Optuna is a framework that facilitates running several experiments with different hyperparameters to find the parameters that produce the best results. First, we need to define and objective function to be optimized, imagine we have a function $train(network, X\_train, y\_train)$ that returns the train error after training the network, the objective function can then be:

```
def objective(trial):
  n_channels = trial.suggest_int('n_channels', 50, 120, 10)
  n_hidden = trial.suggest_int("n_hidden", 10, 40, 10)
  network = neuralNet(n_channels, n_hidden, 5, 3)
  return train(network, X, y)
```

On the function above, first we are setting the intervals of the variables we are going to optimize. For example, we set the number of hidden neurons $n\_hidden$ to be in the range $[10, 40]$ with increments of 10. After creating the ranges, we instantiate the network and call the function train, and then return the error. The goal of Optuna here is to find then the parameters $[n\_channels, n\_hidden, k\_size]$ that minimize the training error on a neural net. A trail object corresponds to a single execution of the objective function. To start the optimization, we create an study object and pass the objective function we just defined:

```
study = optuna.create_study(direction="minimize",
    pruner=optuna.pruners.MedianPruner())
study.optimize(objective, n_trials=15)
```

Here, we create the study object with several parameters:

- `Direction`: if we want to maximize or minimize the returned value of the objective function. If we were training for a different metric, say, accuracy, we would need to change the direction to "maximize" since we want the accuracy to go as high as possible. Since we are optimizing with the prediction error, we want to "minimize" here.
- `Pruner`: if Optuna finds that a trial is not very successful at the beginning, it may prune or stop said trial to save computing time. We will explain a bit more on pruners below.
- `n_trials`: the total number of trials we want Optuna to invest on finding the best parameters, not here, that maybe some of those trials may be pruned thus wont be completed totally.

When choosing which parameters to run a trial with, we can tell Optuna which sampler to use when searching over the parameter space. There are some basic samplers like GridSampler, which searches for all parameters in order, or RandomSampler, which searches randomly, but there are more sophisticated samplers such as the TPESampler, a sampler using the Tree-structured Parzen Estimator [12]. About pruners, to enable them on Optuna, we have to make use of two special methods, the *trial.report*() method for returning information about the training status to Optuna and the method *optuna.TrialPruned*() to stop a trial in case the reports are not successful enough, in an example:

```
for epoch in range(15):
  train_network(network, optimizer, scheduler, X_train, y_train, 1, False)
  nmse = test_network(network, X_val, y_val)
  all_nmse.append(nmse)
  trial.report(np.mean(all_nmse), epoch)
  if trial.should_prune():
      raise optuna.TrialPruned()
  if epoch%25==0:
    print(epoch)
```

We are reporting the mean of the NMSE while training the model on all the epochs. When Optuna detects that the mean of the NMSE is not low enough, it stops the training by calling *optuna.TrialPruned*() and opens the next trial with different parameters. To decide if a metric is good enough for pruning a trial, Optuna offers a variety of pruners, such as MedianPruner, which after a number of trials may prune following trials if the reported value is below the median of the past trials, percentile pruners, threshold pruners and more sophisticated pruners such as Successive Halving [36] or Hyperband [37].

After finishing the hyperparameter exploration, we can ask Optuna for the hyperparameter importance, that is, which hyperparameter has more impact when reducing the error. The method *get_param_importance*() returns a dictionary where the keys are the different parameters to optimize and the values are each parameters importance given by floating point numbers that sum to 1, being the higher the number, the higher the importance. Lastly, Optuna also offers some visualization tools to aid in the hyperparameter search, for example, by calling *optuna.visualization.plot_param_importances*() we get something like

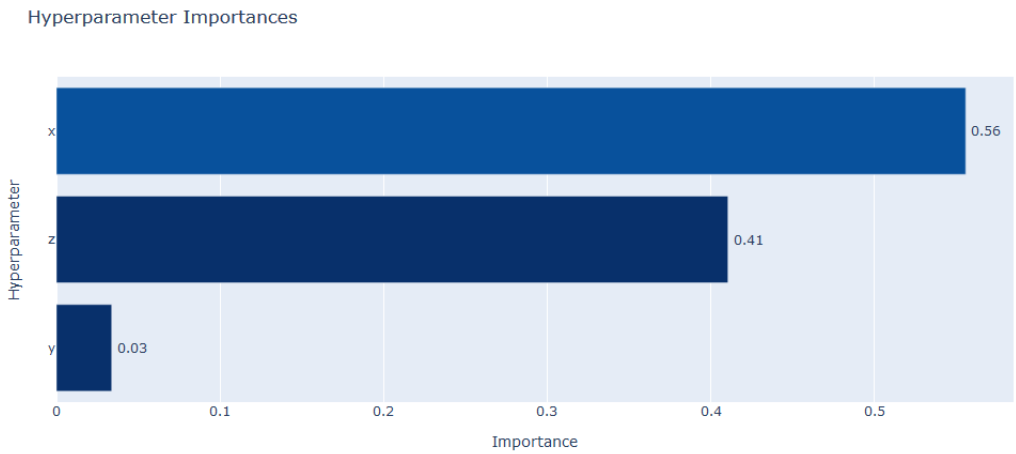



**Figure B.1** Hyperparameter importance [6] .

We can also plot multidimensional plots to represent the relationship between several hyperparameters and the objective function by calling $plot\_countour()$:

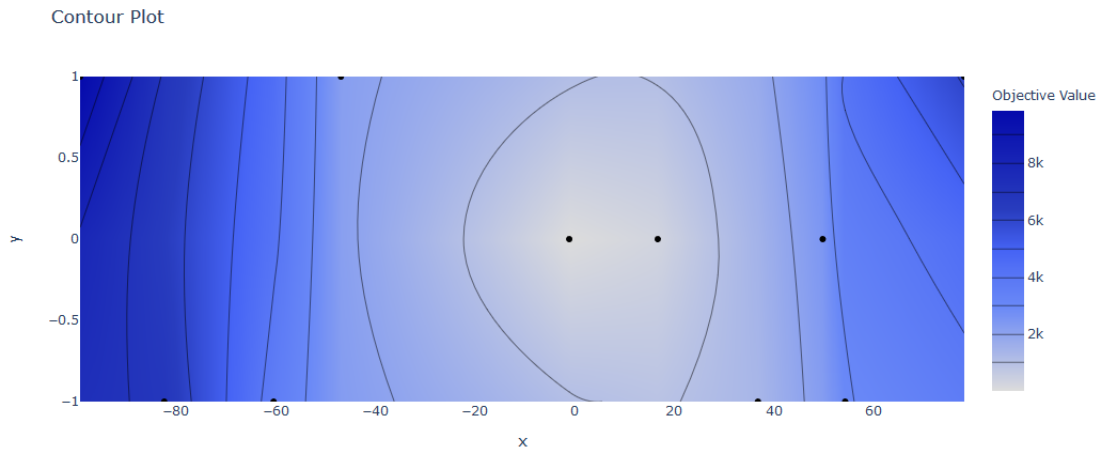



**Figure B.2** Contour plot [6] .

Or plot the history of the trials on a given study:

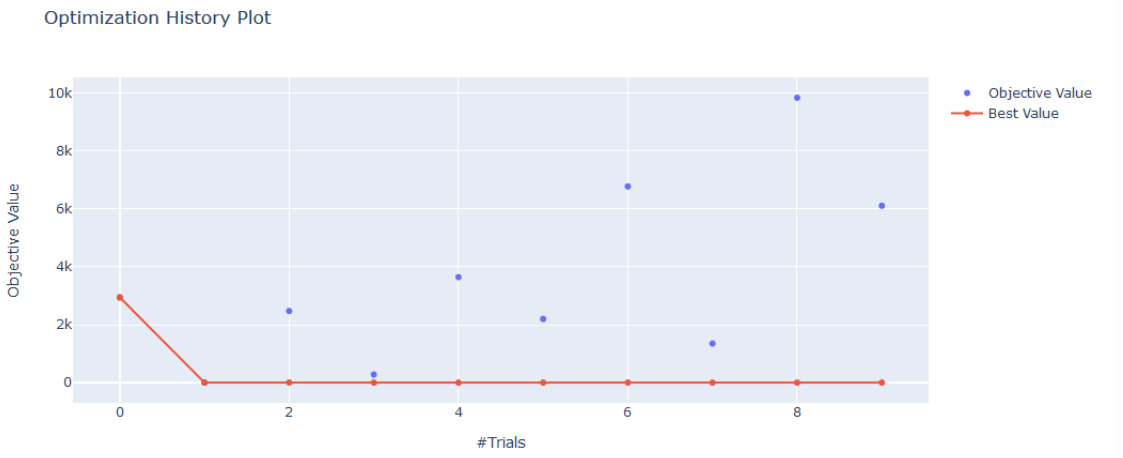



**Figure B.3** Contour plot [6] .

# Figures

# Bibliografía


[1] *Pytorch autograd*, https://pytorch.org/tutorials/beginner/blitz/autograd$_t$*utorial.html*.

[2] *Google colaboratory*, https://research.google.com/colaboratory/faq.html.

[3] *Deep learning frameworks*, https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.

[4] *Gradient boosting*, https://towardsdatascience.com/understanding-gradient-boosting-machines-9be756fe76ab.

[5] *Random forests*, https://towardsdatascience.com/understanding-random-forest-58381e0602d2.

[6] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama, *Optuna: A next-generation hyperparameter optimization framework*, Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2019.

[7] Anas Al-Masri, *What are overfitting and underfitting in machine learning?*, https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690.

[8] Onur Avci, Osama Abdeljaber, Serkan Kiranyaz, and Daniel Inman, *Structural damage detection in real time: Implementation of 1d convolutional neural networks for SHM applications*, Structural Health Monitoring & Damage Detection, Volume 7, Springer International Publishing, 2017, pp. 49–54.

[9] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun, *An empirical evaluation of generic convolutional and recurrent networks for sequence modeling*.

[10] J. A. Becerra, M. J. Madero-Ayora, and C. Crespo-Cadenas, *Comparative analysis of greedy pursuits for the order reduction of wideband digital predistorters*, IEEE Transactions on Microwave Theory and Techniques **67** (2019), no. 9, 3575–3585.

[11] N. Benvenuto, F. Piazza, and A. Uncini, *A neural network approach to data predistortion with memory in digital radio systems*, Proceedings of ICC '93 - IEEE International Conference on Communications, IEEE.

[12] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl, *Algorithms for hyper-parameter optimization*, Advances in Neural Information Processing Systems 24 (J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, eds.), Curran Associates, Inc., 2011, pp. 2546–2554.

[13] Alberto Brihuega, Lauri Anttila, Mahmoud Abdelaziz, Fredrik Tufvesson, and Mikko Valkama, *Digital predistortion for multiuser hybrid mimo at mmwaves*.

[14] Amar Budhiraja, *Dropout in (deep) machine learning*, https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5.

[15] Changsoo Eun and E. J. Powers, *A new volterra predistorter based on the indirect learning architecture*, IEEE Transactions on Signal Processing **45** (1997), no. 1, 223–227.

[16] Steve C. Cripps, *Rf power amplifiers for wireless communications*, Artech House Publishers, 2006.

[17] F. M. Ghannouchi and O. Hammi, *Behavioral modeling and predistortion*, IEEE Microwave Magazine **10** (2009), no. 7, 52–64.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, *http://www.deeplearningbook.org*.

[19] M. T. Hagan and M. B. Menhaj, *Training feedforward networks with the marquardt algorithm*, IEEE Transactions on Neural Networks **5** (1994), no. 6, 989–993.

[20] Simon Haykin, *Neural networks: A comprehensive foundation*, Prentice Hall, 1999.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep residual learning for image recognition*.

[22] _____, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*.

[23] Xin He, Kaiyong Zhao, and Xiaowen Chu, *Automl: A survey of the state-of-the-art*.

[24] Sepp Hochreiter and Jürgen Schmidhuber, *Long short-term memory*, Neural computation **9** (1997), 1735–80.

[25] J. D. Hunter, *Matplotlib: A 2d graphics environment*, Computing in Science & Engineering **9** (2007), no. 3, 90–95.

[26] M. Ibukahla, J. Sombria, F. Castanie, and N. J. Bershad, *Neural networks for modeling nonlinear memoryless communication channels*, IEEE Transactions on Communications **45** (1997), no. 7, 768–771.

[27] Jeremy Jordan, *Setting the learning rate of your neural network*, https://www.jeremyjordan.me/nn-learning-rate/, https://www.jeremyjordan.me/nn-learning-rate/.

[28] Diederik P. Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*.

[29] Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J. Inman, *1d convolutional neural networks and applications: A survey*.

[30] Serkan Kiranyaz, Turker Ince, and Moncef Gabbouj, *Personalized monitoring and advance warning system for cardiac arrhythmias*, Scientific Reports **7** (2017), no. 1.

[31] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing, *Jupyter notebooks – a publishing format for reproducible computational workflows*, Positioning and Power in Academic Publishing: Players, Agents and Agendas (F. Loizides and B. Schmidt, eds.), IOS Press, 2016, pp. 87 – 90.

[32] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE **86** (1998), no. 11, 2278–2324.

[33] Yann LeCun, *Mnist dataset*, http://yann.lecun.com/exdb/mnist/.

[34] Yann Lecun and Y. Bengio, *Convolutional networks for images, speech, and time-series*, 01 1995.

[35] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio, *Object recognition with gradient-based learning*, Shape, Contour and Grouping in Computer Vision, Springer Berlin Heidelberg, 1999, pp. 319–345.

[36] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar, *A system for massively parallel hyperparameter tuning*.

[37] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar, *Hyperband: A novel bandit-based approach to hyperparameter optimization*.

[38] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis, *Dying relu and initialization: Theory and numerical examples*.

[39] David López Bueno, Thi Quynh Anh Pham, Gabriel Montoro López, and Pere Lluís Gilabert Pinal, *Linealización digital de transmisores mediante redes neuronales no lineales*, Simposium Nacional de la Unión Científica Internacional de Radio. "Proceedings URSI", 2019.

[40] Warren S. McCulloch and Walter Pitts, *A logical calculus of the ideas immanent in nervous activity*, The Bulletin of Mathematical Biophysics **5** (1943), no. 4, 115–133.

[41] Farouk Mkadem, *Behavioural modeling and linearization of rf power amplifier using artificial neural networks*, 2010.

[42] D. R. Morgan, Z. Ma, J. Kim, M. G. Zierdt, and J. Pastalan, *A generalized memory polynomial model for digital predistortion of rf power amplifiers*, IEEE Transactions on Signal Processing **54** (2006), no. 10, 3852–3860.

[43] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek, *Cuda, release: 10.2.89*, 2020.

[44] M. Ohta and K. Yamashita, *A chaotic neural network for reducing the peak-to-average power ratio of multicarrier modulation*, Proceedings of the International Joint Conference on Neural Networks, 2003., IEEE.

[45] The pandas development team, *pandas-dev/pandas: Pandas*, February 2020.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, *Pytorch: An imperative style, high-performance deep learning library*, Advances in Neural Information Processing Systems 32 (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), Curran Associates, Inc., 2019, pp. 8024–8035.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine learning in Python*, Journal of Machine Learning Research **12** (2011), 2825–2830.

[48] J. Pedro and N. B. Carvalho, *Intermodulation distortion in microwave and wireless circuits*, 2003.

[49] M. Rawat and F. M. Ghannouchi, *A mutual distortion and impairment compensator for wideband direct-conversion transmitters using neural networks*, IEEE Transactions on Broadcasting **58** (2012), no. 2, 168–177.

[50] M. Rawat, K. Rawat, and F. M. Ghannouchi, *Adaptive digital predistortion of wireless power amplifiers/transmitters using dynamic real-valued focused time-delay line neural networks*, IEEE Transactions on Microwave Theory and Techniques **58** (2010), no. 1, 95–104.

[51] Sebastian Ruder, *An overview of gradient descent optimization algorithms*.

[52] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, *Learning representations by back-propagating errors*, Nature **323** (1986), no. 6088, 533–536.

[53] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu, *Benchmarking state-of-the-art deep learning software tools*.

[54] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, *Going deeper with convolutions*, Computer Vision and Pattern Recognition (CVPR), 2015.

[55] Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio, and Christopher J Pal, *Deep complex networks*.

[56] D. Wang, M. Aziz, M. Helaoui, and F. M. Ghannouchi, *Augmented real-valued time-delay neural network for compensation of distortions and impairments in wireless transmitters*, IEEE Transactions on Neural Networks and Learning Systems **30** (2019), no. 1, 242–254.

[57] Michael Waskom, Olga Botvinnik, Drew O'Kane, Paul Hobson, Saulius Lukauskas, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi Warmenhoven, Julian de Ruiter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba, Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Alistair Miles, Yoav Ram, Tal Yarkoni, Mike Lee Williams, Constantine Evans, Clark Fitzgerald, Brian, Chris Fonnesbeck, Antony Lee, and Adel Qalieh, *mwaskom/seaborn: v0.8.1 (september 2017)*, September 2017.

[58] J. Wood, *Behavioral modeling and linearization of rf power amplifiers*, 05 2014.

[59] Matthew D Zeiler and Rob Fergus, *Visualizing and understanding convolutional networks*.

[60] A. Zhu, J. C. Pedro, and T. J. Brazil, *Dynamic deviation reduction-based volterra behavioral modeling of rf power amplifiers*, IEEE Transactions on Microwave Theory and Techniques **54** (2006), no. 12, 4323–4332.

[61] A. Zhu, M. Wren, and T. J. Brazil, *An efficient volterra-based behavioral model for wideband rf power amplifiers*, IEEE MTT-S International Microwave Symposium Digest, 2003, vol. 2, 2003, pp. 787–790 vol.2.