Abstract

Efficient resource allocation in wireless networks is usually achieved by running iterative algorithms that incur a high computational cost, which makes real-time signal processing difficult. One of the previously proposed ways to solve this issue was that given a set of inputs and outputs of an iterative algorithm, a deep neural network (DNN) can effectively approximate the mapping [1]. Once the network is trained, it is more computationally efficient in allocating power to users than the approximated iterative algorithm. It was previously shown [1] that the WMMSE (Weighted Minimum Mean Squared Error) algorithm can be successfully approximated using a deep neural network. However, the approximation performance, determined by comparing the average sum-rates of the WMMSE algorithm and the DNN, can be further improved. In this report, I demonstrate the process of thorough hyperparameters tuning to determine an alternative DNN configuration that improves the previously achieved results.

Contents

List of Figures

List of Tables

Chapter 1: Introduction

Part 1: Introduction and Motivation

Optimisation of resource allocation in wireless networks is a dynamically developing field that considers different approaches to finding an optimal solution. Highly effective iterative algorithms have been developed for solving a variety of signal processing tasks. The current issue with the iterative algorithms used in resource allocation is their computational complexity, which makes it difficult to achieve real-time processing needed in wireless networks. The aim of this project is to investigate the effectiveness with which deep neural networks (DNN) can approximate the existing iterative algorithm building on the example of the weighted minimum mean squared error (WMMSE) algorithm in interfering multiple-access channel (IMAC) model. The research presented in [1] suggests that the DNN can perform multiple times faster than the WMMSE. In this report, I considered recent advances in training deep neural networks to improve the existing approximation accuracy shown in the previous work.

Deep neural networks have been previously used in solving other signal processing tasks. The research in [2] introduces a deep neural network architecture that is suitable for Multiple-Input-Multiple-Output (MIMO) detection. The results show that the DNN can be highly accurate with significantly lower complexity. The research in [3] presents a novel physical layer scheme for single-user MIMO communications based on unsupervised deep learning using autoencoder. The results suggest that deep learning-based schemes have high potential to approach and exceed the performance of the methods widely used in existing wireless MIMO systems. In the paper [4], the authors proposed a non-linear, feed-forward predictor to produce an approximation of the sparse code. There are many more applications of deep learning that can be explored, and the results of existing research suggest great potential in this developing area.

Part 2: Report structure

In Chapter 2, the background theory behind the IMAC model, the WMMSE algorithm and deep learning is introduced. Chapter 3 presents the design and implementation parts of the report. In Chapter 4, the full process of hyperparameters optimisation (tuning) is described with the use of brief descriptions, tables and graphs. Chapter 5 presents the final results with the comments. Chapter 6 evaluates and concludes the report.

Chapter 2: Background

Part 1: IMAC

An interference channel (IC) model considers the case when N independent transmitters try to communicate to N independent receivers, which forms N direct links and N(N-1) interference links [5]. This is the foundation model on which interfering multiple-access channel (IMAC) is built on, and it can be viewed as a special IC with co-located receivers [1].

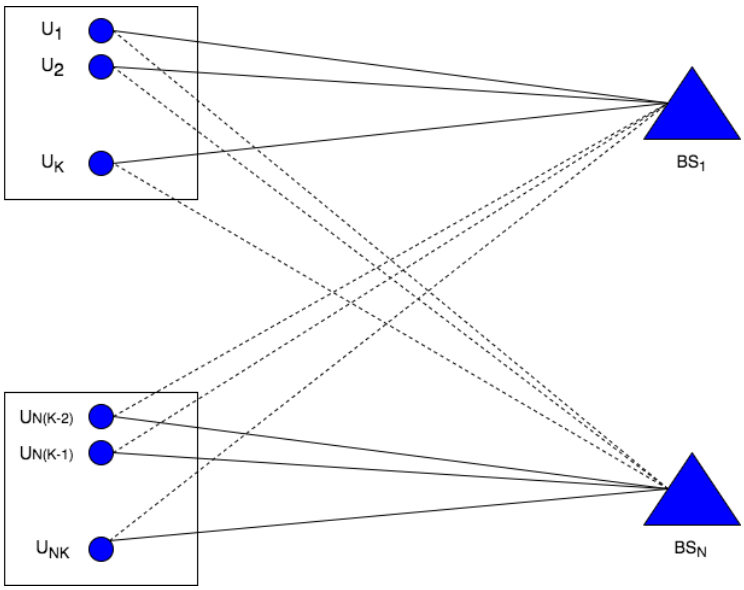

Figure 1: the schematic representation of IMAC, where K is the number of users in each cell and N is the number of cells and base stations.

There are N cells (and base stations) and K number of users per each cell. $\{h_{ij}\}$ is the set of channel realisations between transmitters (j) and receivers (i). For example, if the model has 3 cells (N) with 4 users in each cell (K), there are 12 (N*K) direct channel realisations and 24 interference channel realisations (N*K)(N-1). On Figure 1 solid lines represent direct links and dotted lines represent interference links. The model has been previously simulated in Matlab by Sun et al. [1] The full explanation of the data generation process will be further discussed in the later sections of the report.

Part 2: WMMSE Algorithm

The major task for any wireless network is to allocate the resources in the most efficient way, which can be measured by the throughput of the system. There are various approaches to addressing this issue, but the general method is to use some type of optimisation. Optimisation can be viewed as a set of mathematical techniques aimed at maximising or minimising a certain function under given constraints by altering the parameters of the stated problem.

Optimisation problem presented in the paper [1] can be formulated as follows:

$$\min \quad g(x;z), \text{ s.t. } x \in X, \qquad \textbf{(1)} \text{ [1, eq. (1)]}$$

where:
g: $\mathbb{R}^n \to \mathbb{R}$ – continuous objective function
x – problem parameter
$X \in \mathbb{R}^n$ – feasible region
z ∈ $\mathbb{R}^m$ – vector of problem parameters

The power allocation problem can be defined as in (1), and the paper by Sun et al [1] considers a sample interference channel (IC) model to formulate the problem further. If there is a network of K single-antenna transceivers pairs, we can denote $h_{kk} \in \mathbb{C}$ as the direct channel between transmitter k and receiver k, and $h_{kj} \in \mathbb{C}$ as the interference channel from transmitter j to receiver k. The signal to interference-plus-noise ratio (SINR) for each receiver k is given by:

$$sinr_k \triangleq \frac{|h_{kk}|^2 p_k}{\sum_{j \neq k} |h_{kj}|^2 p_j + \sigma_k^2}, \qquad \textbf{(2)} \text{ [1]}$$

where:
σ_k^2 – noise power at receiver k
p_k – transmission power of transmitter k
p_j – transmission power of transmitter j

The problem can be further formulated as the following nonconvex problem:

$$\max_{p := \{p_1, \ldots, p_K\}} \sum_{k=1}^{K} \alpha_k \log\left(1 + \frac{|h_{kk}|^2 p_k}{\sum_{j \neq k} |h_{kj}|^2 p_j + \sigma_k^2}\right)$$

$$\text{s.t. } 0 \leq p_k \leq P_{max}, \forall k = 1, 2, \ldots, K, \qquad \textbf{(3)} \text{ [1, eq. (2)]}$$

where:
P_{max} – the power budget of each transmitter
$\{\alpha_k > 0\}$ – weights

By casting (3) into (1), we obtain the following identification:

$$x \equiv p, z \equiv \left(\{|h_{ij}|\}_{ij}, \{\sigma_k\}_k\right), X \equiv \{p | 0 \leq p_k \leq P_{max}, \forall k\} \qquad \textbf{(4)} \text{ [1]}$$

The aim of an optimisation algorithm is to solve the problem defined at (1). The common approach in signal processing is to use iterative algorithms, which achieve optimum solution by carrying out calculations and altering the parameters after each step (iteration).

The weighted minimum mean squared error algorithm was introduced in [6]. The paper proposes a linear transceiver design algorithm for weighted sum-rate maximisation that is based on iterative minimisation of weighted MSE. The proposed algorithm only requires local channel knowledge and converges to a stationary point of the weighted sum-rate maximisation problem [6]. Sun et al. [1] introduce their version of the WMMSE algorithm applied to solve the problem in (3). It is worth noting that the proposed version of the algorithm [1] works in the real domain to simplify the latter implementation of DNN, although the original algorithm introduced by Shi et al. [6] works with complex entries. The channel realisation h_{kj} is converted to $|h_{kj}|$, which does not change the WMMSE algorithm [1].

The authors of [1] arrived at the following weighted MSE minimisation problem:

$$\min_{\{w_k,u_k,v_k\}_{k=1}^{K}} \sum_{k=1}^{K} \alpha_k (w_k e_k - \log(w_k))$$
$$\text{s.t. } 0 \le v_k \le \sqrt{P_k}, \forall k = 1, 2, \dots, K, \quad \textbf{(5)} \text{ [1, eq. (4)]}$$

where:
All optimisation variables are real numbers and
e_k – mean-square estimation error

$$e_k = (u_k |h_{kk}| v_k - 1)^2 + \sum_{j \neq k} (u_k |h_{kj}| v_j)^2 + \sigma_k^2 u_k^2 \quad \textbf{(6)} \text{ [1, eq. (5)]}$$

Algorithm 1: Pseudo Code of WMMSE for the Scalar IC [1]

1. Initialise v_k^0 such that $0 \le v_k^0 \le \sqrt{P_k}, \forall k$
2. Compute $u_k^0 = \frac{|h_{kk}| v_k^0}{\sum_{j=1}^{K} |h_{kj}|^2 (v_j^0)^2 + \sigma_k^2}, \forall k$
3. Compute $w_k^0 = \frac{1}{1 - u_k^0 |h_{kk}| v_k^0}, \forall k$
4. Set t = 0
5. repeat
6. t = t + 1 // iterations
7. Update v_k:
$$v_k^t = \left[\frac{\alpha_k w_k^{t-1} u_k^{t-1} |h_{kk}|}{\sum_{j=1}^{K} \alpha_j w_j^{t-1} \left(u_j^{t-1}\right)^2 |h_{jk}|^2} \right]_0^{\sqrt{P_{max}}}, \forall k$$
8. Update u_k: $u_k^t = \frac{|h_{kk}| v_k^t}{\sum_{j=1}^{K} |h_{kj}|^2 (v_j^t)^2 + \sigma_k^2}, \forall k$
9. Update w_k: $w_k^t = \frac{1}{1 - u_k^t |h_{kk}| v_k^t}, \forall k$
10. until some stopping criteria is met
11. output $p_k = (v_k)^2, \forall k$

where:
v_k – transmit beamformer
u_k – receive beamformer
w_k – weight variable
h_{kk} – direct channel coefficient
h_{kj} – interference channel coefficient

The WMMSE solves the problem in (5) using the block coordinate descent method, which means that at each time optimising one set of variables while keeping the rest fixed. It was previously shown in [6] that the WMMSE algorithm can reach the stationary solution for (3). Since the IMAC model can be viewed as a special IC with co-located receivers, the Algorithm 1 can be applied to solving the power allocation problem of IMAC as well [1].

Part 3: Deep Learning

3.1. Introduction to Machine Learning

Machine learning (ML) is a dynamically developing field with limitless potential applications. ML algorithm is able to learn from the data it is provided with. The formal definition of learning was introduced by Mitchell [7, p.12]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." The dataset provided to the algorithm can be treated as experience from which it learns. It is common to describe a dataset as a design matrix with every row of the matrix representing one example. The problem stated in the previous section be approached with supervised learning algorithms, which are a subset of ML algorithms. "In supervised learning the agent observes some example input-output pairs and learns a function that maps from input to output" [8, p.695].

The main aim of any machine learning algorithm is to minimise its generalisation error (test error). Generalisation can be formally defined as "the ability to perform well on previously unobserved inputs" [9, p.107]. The two major factors defining the performance of a machine learning algorithm are the size of the training error and the difference between the training and test errors. If the test error is high, it means that the model is underfitting, and if the gap between the training and test error is high, it signals that the model is overfitting. The model can be controlled by altering its capacity, which is the ability of the model to fit a variety of functions [9]. The model capacity can be adjusted by varying its hypothesis space, which is "the set of functions that the learning algorithm is allowed to select as being the solution" [9, p.109].

3.2. Motivation for Deep Learning

The development of deep learning is bolstered by a few challenges that traditional machine learning algorithms face. Deep learning algorithms achieve better generalisation on high-dimensional data than traditional machine learning algorithms, which tend to be insufficient when it comes to data with the high number of dimensions. This phenomenon can be defined as curse of dimensionality, which was first introduced by Bellman [10], and can be described as "the problem caused by the exponential increase in volume associated with adding extra dimensions to Euclidean space" [11].

3.3. Introduction to Deep Learning

Deep learning (DL) is a type of machine learning inspired by the way neurons in human brains percept information. Deep neural networks (DNN) can work particularly well with large volumes of data, and can be effective in finding the mapping between inputs and outputs of a given system.

For the aims of this project, the concentration will be on feedforward neural networks. The goal of a feedforward neural network is to approximate a given function. A feedforward neural network defines a mapping between inputs **x** and outputs **y**, by learning the value of the parameters θ that results in the best approximation [9].

The main difference between a feedforward neural network and a recurrent neural network is that the latter has feedback connections with the output of the model feeding back to itself. Feedforward neural networks can be viewed as a sequence of functions (network) connected one after another. Each function represents the layer of the network and the overall length of the network is its depth.

There are three major types of layers: input, hidden and output. Input layer consists of the features (variables) used to learn the network and the number of neurons is defined by the number of features. The hidden layers are the layers of neurons between the input and output layers. They are used to control the model's capacity. The output layer can consist of a varying number of neurons depending on the nature of the problem being addressed. The number of neurons in a layer is called its width.

Feedforward neural networks are also called multilayer perceptrons (MLPs). The fundamental idea behind the feedforward neural networks can be introduced through a perceptron algorithm, which is limited in its applications, but is vital for understanding the larger network structures.

Perceptron model is a classifier that takes a set of features as an input and gives out a binary output. At first step, the dot product between the input vector **x** and the vector of weights **w** is found, bias, which is an independent element from the input variables, is also added. Then an activation function, f, determines the state of the output (0 or 1). The major issue with the perceptron classifier is that it can only converge if the training data is linearly separable. The learning process for updating weights and activation functions will be introduced in the further sections of this part of the report.

$$\hat{y} = f(\sum_{i=1}^{n} w_i x_i + b), \quad \textbf{(7)}$$

where:
$\hat{y}$ – predicted output
f – activation function
w_i – i_{th} weight
x_i – i_{th} input
b – bias

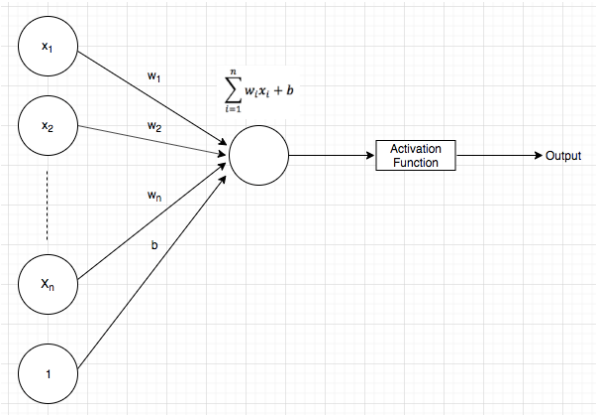

Figure 2: the schematic representation of the single-layer perceptron model.

3.4. Feedforward Neural Networks (Multilayer Perceptrons)

Perceptron model has a limited model capacity. Multilayer perceptron solves this problem by having one or more hidden layers that make it possible to approximate nonlinear functions. It is possible to extend linear models to represent nonlinear functions of **x** by applying the linear model to a transformed input g(**x**), where g is a nonlinear transformation [9]. Formally, in this approach:

$$\begin{gathered}\boldsymbol{h_1} = g_1(\boldsymbol{W_1^T x} + \boldsymbol{b_1}) - \text{first hidden layer} \\ \boldsymbol{h_2} = g_2(\boldsymbol{W_2^T h_1} + \boldsymbol{b_2}) - \text{second hidden layer} \\ \dots \\ \boldsymbol{h_i} = g_i(\boldsymbol{W_i^T h_{i-1}} + \boldsymbol{b_i}) - \text{i}_{\text{th}} \text{ hidden layer,}\end{gathered} \quad \textbf{(8)}$$

where:
x – inputs vector
g_i – activation function at layer i
$\boldsymbol{W_i}$ – weights matrix at layer i
$\boldsymbol{b_i}$ – bias vector at layer i
$\boldsymbol{h_i}$ – hidden features vector at layer i

Fig. 3 represents a sample feedforward neural network with a single hidden layer. Although, it was previously shown by Hornik [12] that a multilayer feedforward network with a single hidden layer that has a finite number of neurons can approximate any continuous functions on compact subsets, adding more hidden layers can result in a better approximation accuracy, but at the same time there is a higher chance of overfitting the model. The neural networks are informally called “deep” if there is more than a single hidden layer in the network.

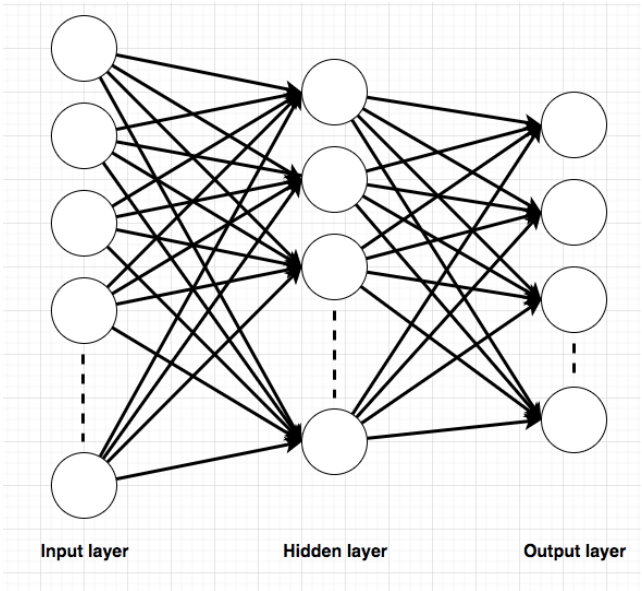

Figure 3: the schematic representation of a fully-connected feedforward neural network with a single hidden layer.

3.5. Learning Process

As mentioned before in 3.3. the goal of a feedforward neural network is to find the mapping between the input vector **x** and the output vector **y** by altering the parameters θ. The parameters that the network is learning are a set of weights for each connection between neurons and a set of biases that are independent from the input features. A fully-connected (dense) neural network is the one that has every neuron in one layer being connected to every possible neuron in the neighbouring layer. The learning is done by calculating the gradient of the cost function with respect to every network parameter, and applying the updates in the backward pass. When designing the network there are several model decisions to be made:

1. Number of layers (depth)
2. Number of neurons in each layer (width)
3. Activation functions
4. Weights initialisation
5. Cost function
6. Optimiser
7. Regularisation techniques

3.6. Activation functions

Activation functions decide which neurons get activated. When a neuron receives the combined input from the neurons from the previous layer, the input is passed through an activation function and the final state of the neuron is decided based on its output. The main purpose of activation functions is to allow for learning nonlinear functions. If the activation functions were not present in neural networks, the model would be simply a linear regression, and would not be able to account for any nonlinearity presented in data. (8) in 3.4. demonstrates how the outputs of neurons are calculated with g representing the nonlinear activation function.

The major requirement for an activation function is it must be differentiable, for the network to be able to learn. If the activation function is not differentiable, it prevents the network from completing the backpropagation step in which the gradients are calculated with respect to the parameters of the network. An extensive research was conducted by Nwankpa et al. [13] on the comparison between several activation functions. The authors presented the most widely used functions and some of their variants. In Chapter 4, three major groups of functions were tried when tuning the hyperparameters: ReLU and its variants, Tanh and its variants, Sigmoid.

3.6.1. ReLU (Rectified Linear Unit) Function and Its Variants

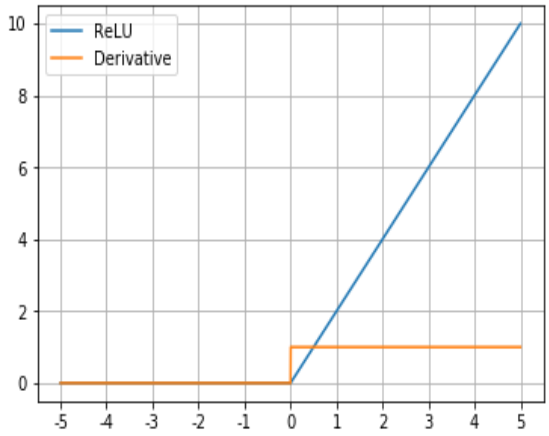

Figure 4: ReLU activation function and its derivative.

ReLUs are the most widely used functions in deep learning, and it is a simple threshold function that brings all negative values to zero, and all positive values are left as they were inputted. (9) formally presents the ReLU function.

$$f(x) = \max(0, x) = \begin{cases} x, if\ x \geq 0, \\ 0, otherwise. \end{cases} \quad \textbf{(9)}$$

ReLU usually guarantees faster computation in comparison to other activation functions, since there are no computationally-heavy multiplication or division operations involved in calculating

the output [13]. However, the major issue that arises when using ReLUs is the problem of "dead" neurons. ReLU brings all the negative values to 0, which sometimes can cause neurons not to activate at all, which then limits the training performance. The reason is that the gradient of the function at 0 is also 0, which prevents the weights from being further updated. To address the described issue, alternative ReLU functions were introduced. One of them is Leaky ReLU (10), which introduces a small negative slope for inputs less than 0. The λ parameter prevents gradients from being zero during training.

$$f(x) = \begin{cases} x, if\ x \geq 0, \\ \lambda x, otherwise. \end{cases} \quad \textbf{(10)}$$

Some of other common variants of ReLU activation function include RReLU (Randomised Leaky ReLU), ELU (Exponential Linear Units) and ReLU6. RReLU alters the constant, λ, each time by taking a value from uniform distribution, $\lambda_i \sim U(a, b)$, where a < b, and $a, b \in [0,1]$. It was previously demonstrated that Leaky ReLU and RReLU outperform ReLU in classification tasks [14]. The ELU function offers a similar approach, and can be viewed as a modification to Leaky ReLU. In the case of ELU, the negative values are the power of e multiplied by a constant λ, as can be seen in (11).

$$f(x) = \begin{cases} x, if\ x \geq 0, \\ \lambda(e^x - 1), otherwise. \end{cases} \quad \textbf{(11)}$$

Finally, ReLU6 (12) is the same as ReLU for negative inputs, but it adds a cut-off value of 6 when the input values are positive. In some cases, ReLU6 can learn sparse features quicker [15].

$$f(x) = \min(\max(0, x), 6) = \begin{cases} x, if\ x \geq 0\ and\ x \leq 6, \\ 6, if\ x > 6 \\ 0, otherwise. \end{cases} \quad \textbf{(12)}$$

3.6.2. Sigmoid

The Sigmoid activation function (13) is a non-linear function that is often used at the output layer of the network, especially in binary classification tasks, as its output can be treated as probability. It is not often used in other layers, as it is vulnerable to gradient saturations and slow convergence [13]. As can be seen on fig. 5, when the input value is small or large, the gradient is almost zero, which results in no learning, which is undesirable. Some of the drawbacks of the Sigmoid function have been addressed by using hyperbolic tangent function and other variants.

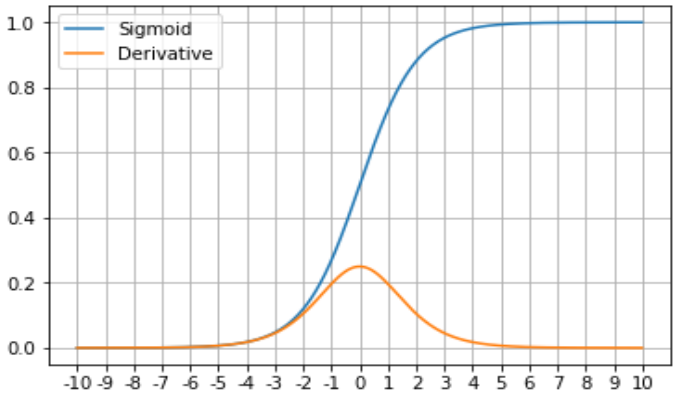

Figure 5: Sigmoid activation function and its derivative.

$$f(x) = \frac{1}{1+e^{-x}} \quad \textbf{(13)}$$

3.6.3. Tanh and Its Variants

The Tanh (14) is a hyperbolic function that lies in the range between -1 and 1 and centered at 0.

$$f(x) = \left(\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}\right) \quad \textbf{(14)}$$

The use of Tanh can result in a better training performance in comparison to Sigmoid function [13]. Its main advantage is that it is centered around zero, unlike Sigmoid, which is desirable, as it means that the average activation value is around zero. This fact can lead to faster convergence, and that is the reason why Tanh is preferred over Sigmoid. However, the Tanh function also does not solve the problem of vanishing gradients that is typical to Sigmoid, as can be seen on fig. 6.

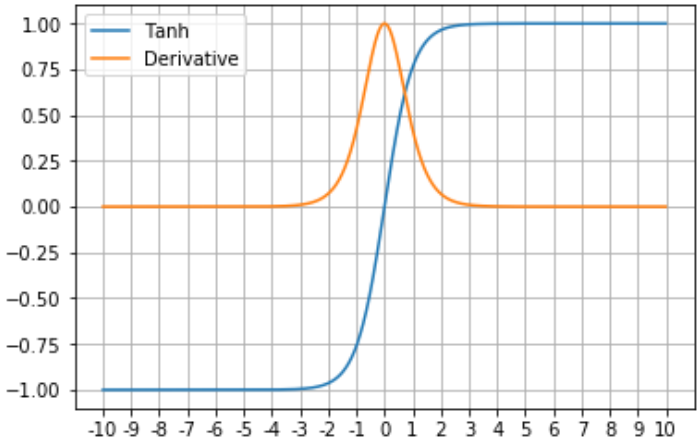

Figure 6: Tanh activation function and its derivative.

Hardtanh function (15) “straightens” out Tanh, which makes it more computationally efficient [13].

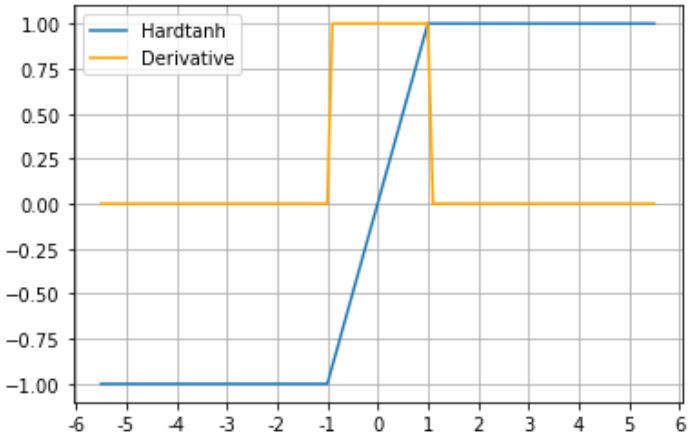

Figure 7: Hardtanh activation function and its derivative.

$$f(x) = \begin{cases} -1, if\ x < -1, \\ x,\ if -1 \leq x \leq 1, \\ 1,\ otherwise. \end{cases} \quad \textbf{(15)}$$

3.7. Weights Initialisations

Initial weights have a large impact on the training performance of a neural network, as they are the parameters (alongside biases) that are being optimised to reduce the cost function. If all weights are set to the same value at the beginning of the training, the model cannot learn as the same weight update is passed throughout the network during the backpropagation step. It is often denoted as the "Symmetry problem", and the aim of any weights initialisation strategy is to break the symmetry. The most common approach is to initialise weights randomly with small magnitudes. Xavier Glorot and Yoshua Bengio [16] proposed an approach that is now known as Xavier initialisation. Fig. 8 shows Uniform Xavier initialisation and fig. 9 shows Normal Xavier initialisation when there are 1 000 neurons in the previous layer and 1 000 neurons in the current layer. For the Uniform distribution, the absolute boundary is calculated by taking the square root of 6 divided by the size of the previous and current layers, where the size refers to the number of units (neurons). For the Normal distribution, the mean is set to 0, while the standard deviation is equal to the square root of 6 divided by the sizes of the previous and current layers. Xavier initialisation comes from the fact that it is desired to have the same variance for all activation vectors in each layer to keep the information flowing, and also to have the same variance for the partial derivatives of the cost function with respect to the argument vector of the activation function at each layer [16]. To satisfy both conditions, the variance of weights for each layer in a deep neural network should be as shown in (16), where i denotes the layer.

$$\forall i, Var[W^i] = \frac{2}{n_i + n_{i+1}}, \quad \textbf{(16)} \text{ [16]}$$

where:

W^i – weights matrix

n_i – number of neurons in layer i

The standard initialisation commonly used as a default in many deep learning libraries is shown in (17). This initialisation gives the rise to the property shown in (18). This is an undesired property as the variance of the gradients propagated backwards depends on the layer. The authors of [16] introduced the normalised initialisation (Uniform Xavier Initialisation) (19) to keep the variances of the activation functions and gradients the same throughout the neural network.

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n_i}}, \frac{1}{\sqrt{n_i}}\right] \quad \textbf{(17)} \text{ [16]}$$

$$n_i Var[W] = \frac{1}{3} \quad \textbf{(18)} \text{ [16]}$$

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right] \quad \textbf{(19)} \text{ [16]}$$

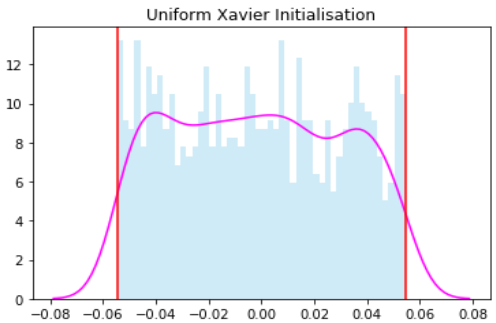

Figure 8: Uniform Xavier Initialisation: Uniform($-\sqrt{\frac{6}{2000}}, \sqrt{\frac{6}{2000}}$) ~ U(-0.0548,0.0548)

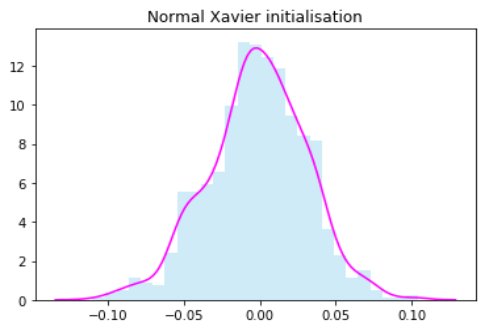

Figure 9: Normal Xavier Initialisation: Normal(0, $\sqrt{\frac{6}{2000}}$) ~ N(0,0.0548)

He at al. [17] studied the nonlinearity of ReLU (Rectified Linear Unit) and PReLU (Parametric Rectified Linear Unit) and derived the weights initialisation strategy that yields the better convergence for very deep models (the authors mentioned 30 layers as an example). The Xavier Initialisation relies on the assumption that the activation functions are linear, which is not the case for the class of ReLU functions. By conducting the analysis of variances in both forward and backward passes, the authors arrived to the weights initialisation strategy (20), which is known as Kaiming (or He) Initialisation. Uniform and Normal Kaiming Initialisations are shown on figures 10 and 11 respectively.

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_i}}, \frac{\sqrt{6}}{\sqrt{n_i}}\right], \quad \textbf{(20)}\ [17]$$

where:
n_i – number of neurons in layer i

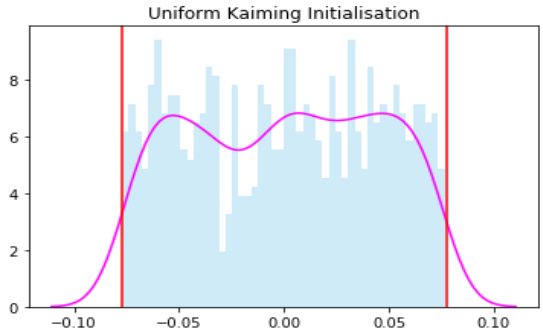

Figure 10: Uniform Kaiming Initialisation: Uniform($-\sqrt{\frac{6}{1000}}, \sqrt{\frac{6}{1000}}$) ~ U(-0.0775,0.0775)

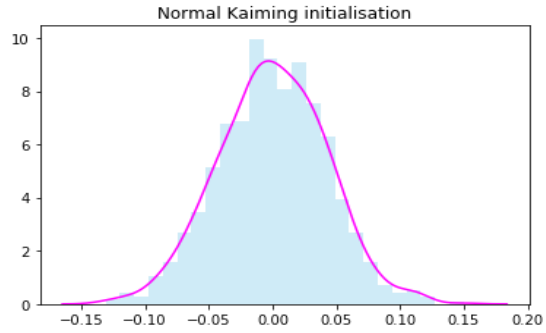

Figure 11: Uniform Normal Initialisation: Normal(0, $\sqrt{\frac{6}{1000}}$) ~ N(0,0.0775)

3.8. Cost functions

Cost function is an important part of the training process, as it is the function that determines how close are the predictions of the model relative to the true values. When it comes to choosing a cost function, it is important to understand the class of the problem that is being solved. The most common cost functions for numerical (continuous) predictions are Mean Absolute Error (21) and Mean Squared Error (22). In this project, the output of the model is a numerical (continuous), so the cost function can be chosen from the two mentioned functions. Mean Absolute Error (MAE) calculates the absolute difference between the predicted output and the true value, while Mean Squared Error (MSE) calculates the square of the difference between the prediction and the true value. The main difference between MAE and MSE is the effect of the outliers on the output of the cost function. Because of the squared term, MSE tends to "penalise" large errors harder than MAE, as it squares all the errors.

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}|y - \hat{y}|, \quad \textbf{(21)}$$

where:
J – cost function
$\boldsymbol{\theta}$ – network parameters
N – number of data points;
$\hat{y}$ – predicted outputs;
y – true outputs.

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}(y - \hat{y})^2 \quad \textbf{(22)}$$

3.9. Optimisation for Training DNN

Optimising neural networks is a non-trivial task. There is a difference between pure optimisation and learning in neural networks. In the case of learning, we select a cost function, and assume that by reducing the value of the cost function, the performance metric (e.g. accuracy) increases. However, in pure optimisation the minimisation of the cost function is the goal itself.

This part of the chapter introduces the gradient-based optimisation in the context of neural networks. The goal of this type of optimisation is to "improve" the output of the cost function. Usually, the improvement refers to the decrease in the difference between the predicted and true values, as discussed in previous section 3.8. In order to determine how to update the parameters (weights and biases) of the neural network, the gradients are found with respect to them. After the gradients were found, small "steps" are taken towards the minimum and parameters are updated. The algorithms that describe the process of updating weights based on calculating gradients are often called optimisers. Optimisers are responsible for the learning process, and selecting the right one is the key design decision to be made before starting training. Each optimiser can be characterised by the number of training samples it uses to update the parameters of the network. Batch or deterministic algorithms, use the whole training dataset simultaneously in both forward and backward passes. The other group of optimisers is called stochastic or online algorithms, and they take a single input-output pair at a time. Finally, the third and most commonly used type of algorithms is called minibatch algorithms, and they are in between batch and stochastic methods, as they use more than one, but less than all training examples in a single pass. One of the tests in Chapter 4 ("Hyperparameters Tuning") describes the process of picking the optimal batch size, but generally, algorithms that only calculate the gradient can perform well with a smaller batch size (e.g. 100), however optimisers that calculate the derivative of the gradient (i.e. Hessian matrix) suffer more from fluctuations, and require a larger batch size (e.g. 1000, 10 000).

Optimisation of neural networks involves many challenges. One of the most common challenges faced when training neural networks is avoiding a local minimum, which is not guaranteed to be a global minimum in the case of nonconvex functions. Deep neural networks are almost guaranteed to have a significant number of local minimums [9]. However, some researchers believe that with a sufficiently large neural network, it is not vital to find the global minimum, but instead it is important to find a local minimum with a low cost, which does not have to be the lowest [9]. Local minima are not the only flat regions (i.e. regions where gradient is zero) that can cause issues for training neural nets. Saddle points are common for high-dimensional nonconvex functions, and can be viewed as points where partial derivatives are zero, but they do not represent neither local minimum or local maximum.

3.9.1. Common Algorithms

3.9.1.1. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent is a very commonly used algorithm both in machine learning and deep learning. The standard SGD algorithm is presented in Algorithm 2.

Algorithm 2: Stochastic Gradient Descent [9]

Inputs: parameters (θ) and learning rate (ϵ).

1. **while** stopping criterion not met **do**
2. sample a minibatch of m examples from the training set {x_1,...,$\mathbf{x_m}$} with corresponding targets ($\mathbf{y_i}$)
3. compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x_i}; \theta), \boldsymbol{y_i})$
4. update parameters: $\theta \leftarrow \theta - \epsilon \boldsymbol{g}$
5. **end while**

(L – loss function)

The learning rate, ϵ, has a significant effect on the learning process of the neural network. Larger values can result in the algorithm oscillating too quickly and missing the optimal value, but smaller values slow down the process of finding the minimum value, as the steps taken are too small.

3.9.1.2. Stochastic Gradient Descent (SGD) with Momentum

Training using SGD algorithm can sometimes be too slow. By introducing the method of momentum [18], the learning process can be accelerated. The momentum method accumulates an exponentially decaying moving average of past gradients and continuously moves in the same direction [9]. In the implementation of SGD with Momentum, there are two new initial parameters that must be set. Initial velocity, $\boldsymbol{\nu}$, plays the role of momentum itself, as in the physics analogy momentum = mass x velocity, and for this algorithm, m, is assumed to be 1. Another hyperparameter, α, determines how quickly the impact of previous gradients decays. α should be in the range between 0 and 1.

Algorithm 3: Stochastic Gradient Descent with Momentum [9]

Inputs: parameters (θ), learning rate (ϵ), momentum parameter (α), initial velocity ($\boldsymbol{\nu}$).

1. **while** stopping criterion not met **do**
2. sample a minibatch of m examples from the training set {x_1,...,$\mathbf{x_m}$} with corresponding targets ($\mathbf{y_i}$)
3. compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x_i}; \theta), \boldsymbol{y_i})$
4. compute velocity: $\boldsymbol{\nu} \leftarrow \alpha\boldsymbol{\nu} - \epsilon \boldsymbol{g}$
5. update parameters: $\theta \leftarrow \theta + \boldsymbol{\nu}$
6. **end while**

Algorithms with momentum consider the sequence of gradients and how they are related to each other. The step size increases if several consecutive gradients have the same direction.

Another variant of SGD is SGD with Nesterov Momentum. The algorithm is similar to the standard momentum method with the only difference that in Algorithm 4, the gradient is calculated after applying current velocity.

Algorithm 4: Stochastic Gradient Descent with Nesterov Momentum [9]

Inputs: parameters (θ), learning rate (ϵ), momentum parameter (α), initial velocity ($\boldsymbol{v}$).

1. **while** stopping criterion not met **do**
2. sample a minibatch of m examples from the training set {x_1,...,$\mathbf{x_m}$} with corresponding targets ($\mathbf{y_i}$)
3. apply interim update: $\tilde{\theta} \leftarrow \theta + \alpha \boldsymbol{v}$
4. compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\boldsymbol{x_i}; \tilde{\theta}), \boldsymbol{y_i})$
5. compute velocity: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
6. update parameters: $\theta \leftarrow \theta + \boldsymbol{v}$
7. **end while**

3.9.1.3. Adaptive Learning Rates Algorithms

Learning rate is an important hyperparameter to set as it has a significant effect on the learning process. Adaptive learning rates algorithms are a class of optimisers that have a distinct learning rate for each parameter that is adapted during training. A few commonly used optimisers correspond to this class of algorithms, for example: AdaGrad [19], RMSProp [20] and Adam [21].

AdaGrad algorithm takes into account all gradients calculated during training. The learning rates of the parameters with the largest partial derivatives are decreased significantly in comparison to the parameters with the smaller partial derivatives. The major issue with AdaGrad is the fact that all historical gradients are accumulated, and the learning rates can become too small before reaching the minimum value. The RMSProp algorithm addresses this issue by introducing a decay value that reduces the impact of gradients from distant past. The more recent algorithm, Adam, is presented in algorithm 5.

Algorithm 5: Adam [9]

Inputs: parameters (θ), global learning rate (ϵ), decay rate 1 (α_1) for the first moment estimate, $\boldsymbol{m_1}$, decay rate 2 (α_2) for the second moment estimate, $\boldsymbol{m_2}$, constant for numerical stabilisation (δ).

1. Initialise m_1, m_2
2. Initialise time step t = 0
3. **while** stopping criterion not met **do**
4. sample a minibatch of m examples from the training set {x_1,...,$\mathbf{x_m}$} with corresponding targets ($\mathbf{y_i}$)
5. compute gradient $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x_i}; \theta), \boldsymbol{y_i})$
6. t = t+1

7. update $\boldsymbol{m_1}: \boldsymbol{m_1} \leftarrow \alpha_1 \boldsymbol{m_1} + (1 - \alpha_1)\boldsymbol{g}$
8. update $\boldsymbol{m_2}: \boldsymbol{m_2} \leftarrow \alpha_2 \boldsymbol{m_2} + (1 - \alpha_2)\boldsymbol{g} \odot \boldsymbol{g}$
9. correct bias in $\boldsymbol{m_1}: \widehat{\boldsymbol{m_1}} \leftarrow \frac{\boldsymbol{m_1}}{\boldsymbol{1-\alpha_1}^t}$
10. correct bias in $\boldsymbol{m_2}: \widehat{\boldsymbol{m_2}} \leftarrow \frac{\boldsymbol{m_2}}{\boldsymbol{1-\alpha_2}^t}$
11. update $\Delta\boldsymbol{\theta} = -\epsilon \frac{\widehat{\boldsymbol{m_1}}}{\sqrt{\widehat{\boldsymbol{m_2}}}+\delta}$
12. update parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
13. **end while**

Recently, the authors of [22] proposed another variant of Adam – AMSGrad. In the case of AMSGrad, the maximum of the past second moment estimate, $\boldsymbol{m_2}$ (alg. 5), is taken, which can allow rarely-occurring batches with large gradients to have a greater impact on the learning path of the algorithm. In their paper, the authors showed that there are cases when Adam does not necessarily converge even in convex scenarios. The authors claimed that AMSGrad can provide provide a guaranteed convergence, while preserving the benefits of Adam [22].

3.9.2. Fixed Learning Rate / Cyclical Learning Rates

Usually, the learning rate is set before the beginning of the training, as it is one of the most important hyperparameters with a significant impact on the performance of the neural network. After a certain number of iterations, the learning rate is slowly decayed as usually smaller steps near the minimum help reaching the desired point. However, a new approach, the "Cyclical Learning Rates", was introduced by Smith [23]. Instead of having a single fixed learning rate decaying over time, the method lets the learning rate vary between two boundaries. This policy comes from the author's observation that a short term negative effect while training is compensated by a longer term beneficial effect. The long term positive effect of implementing cyclical learning rates can be explained by the fact that saddle points are a bigger threat to the neural network performance than local minima. Because saddle points have very small gradients, they can slow down, or even stop the learning process. By increasing the learning rate, the saddle point regions can be escaped.

Firstly, the use of cyclical learning rates showed to outperform the use of the fixed learning rate [23] for a number of image classification datasets. Secondly, the use of cyclical learning rates excludes the need to tune the initial learning rate. However, instead of selecting the fixed learning rate, the boundaries between which the learning rates vary must be chosen, which can be easily done with "LR range test" [23]. This test simply consists of running the neural network starting from a very low learning to a large learning rate for a few epochs. When running the test, the region where there is a steep decrease in the validation loss curve shows the appropriate range of the learning rates. There are also a few alternatives to how the learning rates can vary within the boundaries. The author suggested that linear, parabolic and sinusoidal curves yielded the same results, which encourages the use of the linear curve (i.e. triangular window), as it is the easiest to implement. An example of a triangular window can be seen on fig. 12. The step size is the number of epochs times a constant (in the range between 2-10, 5 on fig. 12.). The number of iterations per epoch depends on the size of the batch. In the case of fig. 12, there are 200 iterations in each epoch and 1000 iterations in each step ((200 000 / 1 000) * 5).

```
Training size - 200000
Training batch size - 1000
Lower bound - 0.000010
Upper bound - 0.001000
Number of iterations in each epoch - 200
Number of iterations in each step - 1000
Total number of iterations - 10000
```

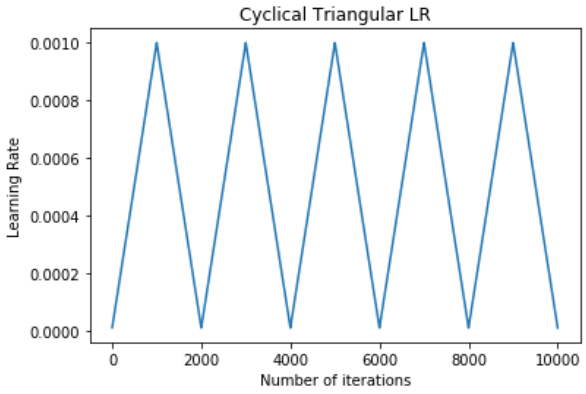

Figure 12: the cyclical learning rates (triangular oscillations)

3.10. Regularisation Techniques

Regularisation can be seen as a set of alterations that we make to an algorithm that are aimed at improving the generalisation performance of an algorithm by reducing testing error, rather than training error [9]. This section introduces the regularisation strategies that were used in the training of the neural networks in this project.

3.10.1. Dropout

Dropout is one of the most common regularisation techniques used for preventing overfitting. The main idea is that some neurons and their connections are removed from the network in the training stage [24]. By "dropping" neurons, the neural network is prevented from adapting too much to the training set, which can then reduce the error when testing on unseen data. The improvement in generalisation performance is achieved by combining simpler models together, which is a common machine learning approach. Dropout technique follows the same idea by efficiently combining a large number of neural networks together.

To implement Dropout, a hyperparameter, p, which is the probability of a single unit to be retained in the network independent of other neurons, must be set. In the paper by Srivastava et al. [24], the authors suggest setting probability to 0.5 for hidden layers and a larger probability for the input units. Any network consisting of n units can be viewed as a set of 2^n possible networks, and training a neural net with Dropout can be seen as training 2^n neural nets with each net being rarely trained, or even not trained at all [24]. Even though some weights are dropped out during training, when it is finished, all of them are present for testing the network. All the weights outgoing from a neuron, that was dropped with the probability p during training, are multiplied by p, so that the output at test time is the same as the expected output when training.

3.10.2. Weight Decay (L^2 Parameter Regularisation)

One of the most common regularisation approaches is adding a parameter norm penalty to the cost function, J, as shown in (23).

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \lambda Reg(\boldsymbol{\theta}), \tag{23}$$

where:
$\tilde{J}$ – cost function with L^2 regularisation
λ – weight decay value
Reg – regularisation term

The main idea is that lower weights improve generalisation, and depending on the norm penalty chosen, weights are "penalised", which drives them closer to 0. The two major approaches are L^1 and L^2 regularisations as shown in (24) and (25) respectively.

$$Reg(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 \tag{24}$$

$$Reg(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 \tag{25}$$

L^2 regularisation is the more commonly used, and it can be better understood by writing out the single gradient step that the optimisation algorithm does in backpropagation. If we assume that the set of parameters, $\boldsymbol{\theta}$, can be limited to weights only, we get (26), which is the cost function with weight decay term added.

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \frac{\lambda}{2}\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{w} \tag{26}$$

During the backpropagation part of the training, the derivative of the cost function is taken, and can be seen in (27).

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \lambda \boldsymbol{w} \tag{27}$$

The constant factor, λ, determines how quickly the weights shrink each step, and it is an important hyperparameter to set before starting training. The gradient step is shown in (28).

$$\boldsymbol{w} \leftarrow (1 - \epsilon\lambda)\boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})), \quad \textbf{(28)}$$

where:
ϵ – learning rate

In the case of Stochastic Gradient Descent algorithm and its variants, L^2 regularisation and weight decay are equivalent. However, the authors of [25] noticed that in the adaptive learning rates algorithms (e.g. AdaGrad, RMSProp, Adam), L^2 regularisation and weight decay are not identical, due to the fact that L^2 regularisation causes weights with large historic parameter and/or gradient amplitudes not being regularised as much as they would with weight decay, which makes the two methods different. The authors offered their variant of Adam algorithm – AdamW (alg. 6).

Algorithm 6: AdamW (*Adam with* L^2 regularisation, *Adam with weight decay (AdamW)*) [25]

1. Initialise m_1, m_2
2. Initialise time step t = 0
3. **while** stopping criterion not met **do**
4. sample a minibatch of m examples from the training set $\{x_1,...,\mathbf{x_m}\}$ with corresponding targets ($\mathbf{y_i}$)
5. compute gradient $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x_i}; \theta), \boldsymbol{y_i}) + \lambda\theta$
6. t = t+1
7. update $\boldsymbol{m_1}$: $\boldsymbol{m_1} \leftarrow \alpha_1 \boldsymbol{m_1} + (1 - \alpha_1)\boldsymbol{g}$
8. update $\boldsymbol{m_2}$: $\boldsymbol{m_2} \leftarrow \alpha_2 \boldsymbol{m_2} + (1 - \alpha_2)\boldsymbol{g} \odot \boldsymbol{g}$
9. correct bias in $\boldsymbol{m_1}$: $\widehat{\boldsymbol{m_1}} \leftarrow \frac{\boldsymbol{m_1}}{\mathbf{1}-{\alpha_1}^t}$
10. correct bias in $\boldsymbol{m_2}$: $\widehat{\boldsymbol{m_2}} \leftarrow \frac{\boldsymbol{m_2}}{\mathbf{1}-{\alpha_2}^t}$
11. update $\Delta\theta = -\epsilon \frac{\widehat{\boldsymbol{m_1}}}{\sqrt{\widehat{\boldsymbol{m_2}}}+\delta} + \lambda\theta$
12. update parameters: $\theta \leftarrow \theta + \Delta\theta$
13. **end while**

As can be seen in alg. 6, when the regularisation term is added to the gradient in step 5, its gradient gets adapted alongside the sum of gradients of the loss function. However, when the regularisation term is decoupled (i.e. added directly to the update value), only the sums of the gradient of the loss function is adapted. Effectively, all weights in AdamW are regularised with by the same factor, λ, which is larger than the standard L^2 regularisation in Adam does.

3.10.3. Early Stopping

When the neural network has enough capacity to overfit, it is common to see the validation loss steadily going down and then starting to rise again. At this point, it is important to stop the training process as further learning will reduce the model's ability to perform well on unseen data. The standard approach to implementing the early stopping is to maintain the model parameters for every iteration that improves the validation loss. If the validation loss is not improving over time, after a fixed number of epochs (i.e. patience), the training is stopped. After the termination of the training, the parameters that yielded the lowest validation loss are returned.

Chapter 3: Design and Implementation

Part 1: Design

1.1 IMAC (Interfering Multiple-Access Channel)

As previously mentioned in Chapter 2, the setting considered in the report is IMAC (interfering multiple-access channel) model. There are four IMAC scenarios considered in the project as presented in table 1. The scenario shown on fig. 13 is used for the hyperparameters tuning, as it is not too large for performing multiple trainings, and not too small to be "easy" for the neural network to learn the mapping between inputs and outputs quickly. Fig. 14 shows the structure of a single cell, where R is the cell radius, and r is the inner radius. Users are randomly distributed between r and R. In Chapter 5, the approximation results based on the varying r and R are presented to show that the model trained on a particular cell configuration can be reasonably accurate in other settings.

N – number of cells (= base stations)
K – number of users per cell
N*K – number of direct channel realisations (channel between a user and a corresponding base station)
N*(N-1)*K – number of interference channel realisations (channel between the user and the base station in a different cell)

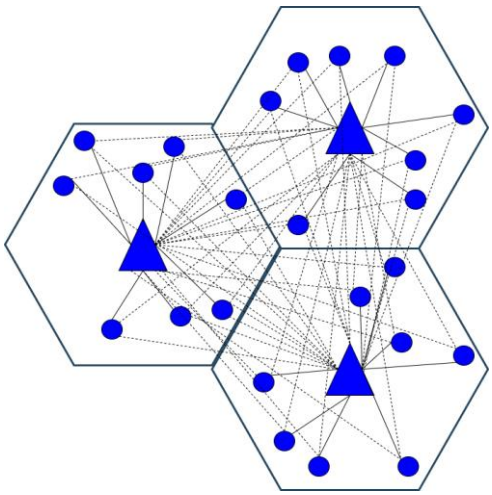

Figure 13: the IMAC scenario used for hyperparameters tuning (BS = 3, K = 8)

# of base stations and users per cell (N, K)	# of direct channel realisations (N*K) / # of interference channel realisations (N*(N-1)*K) / Total	The size of the channel matrix $(N*K)^2$
(3,4)	12 / 24 / 36	144
(3,6)	18 / 36 / 54	324
(3,8)	24 / 48 / 72	576
(7,4)	28 / 168 / 196	784

Table 1: the description of four IMAC scenarios considered in the project.

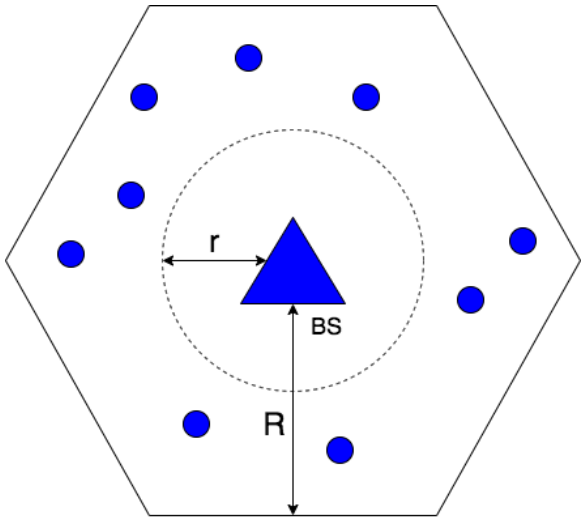

Figure 14: the structure of a single cell, where R is the cell radius and
r is the inner radius (i.e. all users are placed randomly between r and R).

The users are randomly and uniformly distributed between r and R, and the channel realisations are randomly generated according to a Rayleigh fading distribution with zero mean and variance $(200/d)^3$ L, where d denotes the distance between the BS and users, and 10log10(L) following a normal distribution with zero mean and variance 64, as described in [1]. For simplicity, each cell has the same number of users.

1.2. WMMSE

After the users within cells and channel realisations, $\{h_{kj}^i\}$, are generated, the WMMSE algorithm is run to generate the power allocations, $\{p_k^i\}$, which are used in the project as training labels. The maximum power, P_{max}, and noise, σ_k, are fixed to 1 throughout the project for simplicity. The WMMSE algorithm is initialised with $v_k^0 = \sqrt{P_{max}}$ for all k, and with the following termination criteria: $obj_{new} - obj_{old} < 10^{-5}$. An input-output pair, or a training sample, is denoted as $(\{|h_{kj}^i|\}, \{p_k^i\})$.

1.3. Neural Network

The neural network consists of one input layer with N^2*K neurons (i.e. the total number of direct and interference channel realisations), three hidden layers of 200 neurons (suggested in [1]), one output layer with N*K neurons, which corresponds to the number of direct links within the setting. Chapter 4 presents all the hyperparameters tuning steps with the explanation of why the particular optimisers (and their hyperparameters), activation functions and regularisation techniques were used. However, it is worth noting that the activation function for the output layer is shown in (29)., which enforces the power constraint (i.e. $P_{max} = 1$ throughout the project).

$$f(x) = \min(\max(x, 0), P_{max}) \quad \textbf{(29)}$$

Adam is chosen to be the initial optimiser, and not RMSProp as in [1]. The algorithms are very similar; however, Adam is a more advanced implementation of an adaptive learning rates algorithm with a few variants (e.g. AdamW, AMSGrad) published in recent years. In Chapter 4, in

the first few tests the weights initialisation is referred to as “Default”. The reason for that is the weights initialisation shown in fig. 15 is the default initialisation in PyTorch (deep learning library used for the project). On fig. 15, the weights distribution is shown for the case when the input layer has 1 000 neurons.

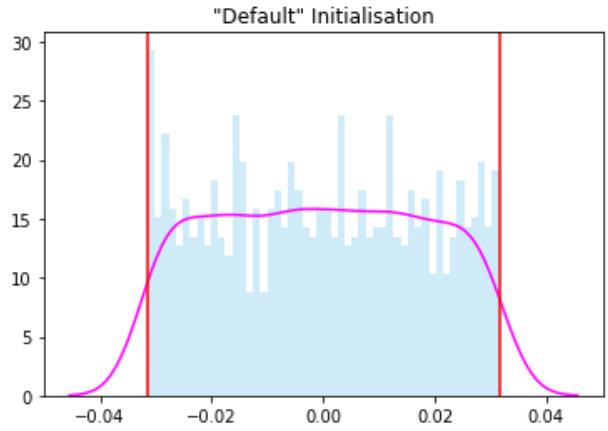

Figure 15: “Default” Weights Initialisation in PyTorch: Uniform($-\sqrt{\frac{1}{1000}}, \sqrt{\frac{1}{1000}}$) ~ U(-0.0316,0.0316)

Early stopping (described in Chapter 2, 3.10.3.) is used throughout the hyperparameters tuning and final testing stages. The implemented algorithm checks if the validation loss of the prediction in the current epoch is the lowest to this point, and saves the model parameters if it is. The model parameters that yielded the lowest validation error are returned at the end of the training. The training is stopped when the average validation loss for the past “patience” epochs is lower than the validation loss between the past 2 x “patience” epochs and “patience” epochs. For example, if the patience is 50, after 100 (50*2) epochs, the algorithm checks if the average validation loss between the first and fiftieth epochs is lower than the average validation loss between the fiftieth and the hundredth epochs. If it is, then the training is stopped.

1.4. Data Generation

The data generated in three sets: training, validation and testing. The training sets used in the project consist of 50 000, 100 000, 200 000 and 500 000 samples. For all training sets generated for each IMAC scenario, the inner radius, r, is fixed to 0 m, and the cell radius, R, is fixed to 100 m. The validation sets have 50 000 samples, and they are used to track the learning of the neural network. Finally, the testing sets consist of 10 000 samples All sets are generated independently with different random seeds.

1.5. Performance Metric

The performance metric used in the project is the sum-rate approximation accuracy. At first, the average sum-rate is calculated for the WMMSE algorithm, and then the average sum-rate is calculated for the DNN. The higher the approximation accuracy, the better the deep neural network managed to learn the mapping between the inputs and outputs. [1] compares WMMSE and DNN in terms of the sum-rate approximation accuracy and computational performance. However, the significant improvement in the computational speed of the trained deep neural network in comparison to the WMMSE algorithm was already sufficiently shown in the previous research [1].

Part 2: Implementation

2.1. Data Generation

All training, validation and testing datasets were generated using the Matlab code published on GitHub (https://github.com/Haoran-S/SPAWC2017) and then exported in CSV format. Fig. 16 shows an example of the channel realisations generated for the case of 3 base stations and 8 users in each cell. 72 columns correspond to the channel realisations and 10 000 rows correspond to the number of samples generated. Fig. 17 demonstrates the power allocations generated by the WMMSE algorithm. Most of the allocations are binary (i.e. 0 or 1), and later in the testing stage to compare the performance of the model against [1], the predictions of the model will be binarised.

	0	1	2	3	4	5	6	7	8	9	...	62	63	64	65
0	0.913303	0.635586	0.732245	10.082932	0.669806	0.178469	6.448977	1.065513	1.657700	2.046075	...	8.335969	0.112466	2.263363	19.885263
1	3.718990	0.680312	0.758762	0.846845	1.227745	1.543749	4.430808	0.228245	0.557598	43.413668	...	3.333358	0.162943	0.403044	9.735576
2	0.700215	0.073692	2.524394	1.168489	1.720738	0.822137	2.739949	1.475344	6.180362	1.663166	...	37.757684	0.410490	7.630992	3.519531
3	0.115209	0.764788	0.525525	0.433081	1.081123	0.926457	8.718458	0.185566	0.186748	18.169513	...	1.547626	0.199816	0.186023	2.287612
4	1.821653	3.812032	0.899769	15.507798	1.638921	0.109969	10.624140	0.546895	0.199382	2.285731	...	4.775660	0.181227	0.450742	2.773270
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9995	2.019121	1.535621	0.427826	0.326315	0.835661	0.299568	0.294960	0.465767	1.254073	1.863381	...	6.350542	0.480167	0.457176	5.589638
9996	2.441243	0.303933	1.954166	4.806436	0.599607	1.986924	2.228655	2.767179	5.181410	62.078344	...	1.429345	0.769411	0.197073	17.464079
9997	8.814737	1.012907	0.070853	2.773695	1.570005	0.844868	2.270152	4.566364	3.615607	8.243047	...	12.518383	0.687408	1.365030	5.512511
9998	176.877122	0.393318	2.366990	11.527846	0.085697	0.224054	1.122359	2.039782	1.170215	1.209784	...	2.136011	1.113411	0.244396	3.456825
9999	11.820139	7.806074	0.470321	9.049169	2.329179	0.122832	1.480498	1.075734	0.030578	0.397718	...	1.543658	0.980775	0.256844	3.672034

10000 rows × 72 columns

Figure 16: example of the input data with 72 channel realisations (columns) and 10 000 samples (rows).

	0	1	2	3	4	5	6	7	8	9	...	14	15	16	17	18	19	20	21	22	23
5545	0.0	0.0	0.0	0.0	0.0	1.0000	0.0	0.0	0.0	0.0	...	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
8015	0.0	1.0	0.0	0.0	0.0	0.0000	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
5027	1.0	0.0	0.0	0.0	0.0	0.0000	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
1051	0.0	0.0	0.0	0.0	0.0	0.9511	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
4932	0.0	0.0	0.0	0.0	0.0	1.0000	0.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0

Figure 17: example of the labels with 24 power allocations (columns) and a row number corresponding to a single row in the input data.

2.2. Development Environment

The deep neural network is implemented using Python 3.7.0 with PyTorch 1.4.0. Other Python libraries used in the project:

- Pandas 0.24.2 – allows opening CSV files;
- NumPy 1.16.4 – adds support for multidimensional arrays and matrices and high-level mathematical functions to operate on them;
- Matplotlib 3.0.3 – plotting library;
- Seaborn 0.8.1 – another plotting library based on Matplotlib, adds extra functionality.

The Python code itself is written in the Jupyter Notebook, which is a web-based interactive computational environment for creating Jupyter notebook documents. A notebooks itself is a

JSON document with input/output cells. It is commonly used in many data science and artificial intelligence projects, because of its simplicity and the opportunity to convert the notebook to one of the many output formats (e.g. HTML, presentation slides, PDF, LaTeX, Python script).

The results for each of the four IMAC settings are presented in the form of Jupyter Notebooks and HTML documents.

2.3. Amazon Web Services (AWS)

To speed up the training and testing processes, the Amazon Web Services EC2 instance with a Graphical Processing Unit (GPU) was used. Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the AWS cloud. p2.xlarge instance was used for the project, which is one of the P2 instances designed for general-purpose GPU applications suited machine learning, and deep learning in particular. p2.xlarge instance has 1 GPU (NVIDIA K80), 4 vCPUs (virtual central processing units) and 61 GiB RAM (random access memory).

Chapter 4: Hyperparameters tuning

This chapter introduces all steps involved in hyperparameters tuning. Throughout the whole chapter, the following scenario is used: BS = 3, K = 8, R = 100 m, r = 0 m. The size of the training set was 100 000 samples (unless otherwise stated), the size of the validation set was 50 000 samples and the size of the testing set was 10 000 samples, however in some parts 1 000 samples were used for testing to speed up the process, as it generally takes a significant amount of time to run 10 000 iterations. MSE (Mean Squared Error) was used as a loss function throughout the whole hyperparameters tuning and testing stages.

Test 1: Activation Function

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = **VARIABLE**
- Optimizer = Adam
- Learning rate = 0.001
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 100)

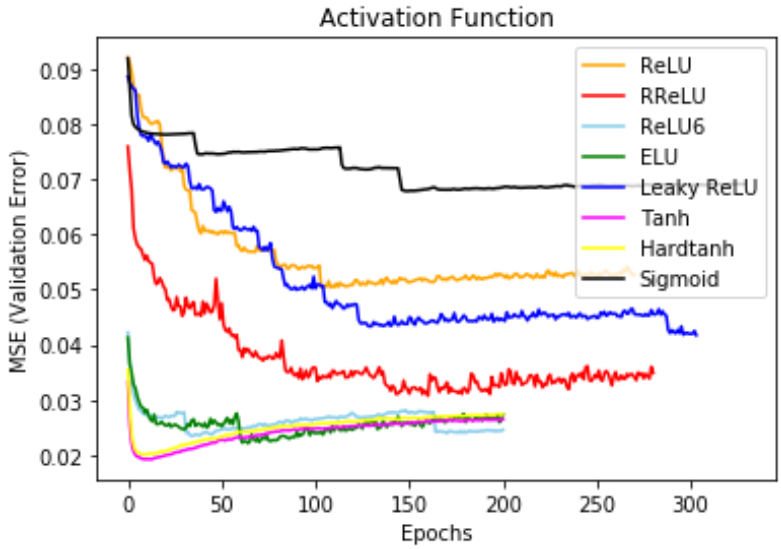

Figure 18: the validation losses recorded when training the neural network with different activation functions.

Analysis: Tanh and Hardtanh activation functions converged the quickest and showed the lowest validation errors, while different variations of ReLU showed inconsistent and not smooth learning curves, which required further testing a few of the functions with different Adam learning rates.

Test 2: Adam Learning Rate

Before continuing searching for the optimal activation function, the effect of altering learning rate used in Adam algorithm should be considered. Tanh was used at this stage, as the best performed activation function in Test 1.

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = Tanh
- Optimizer = Adam
- Learning rate = **VARIABLE**
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 100)

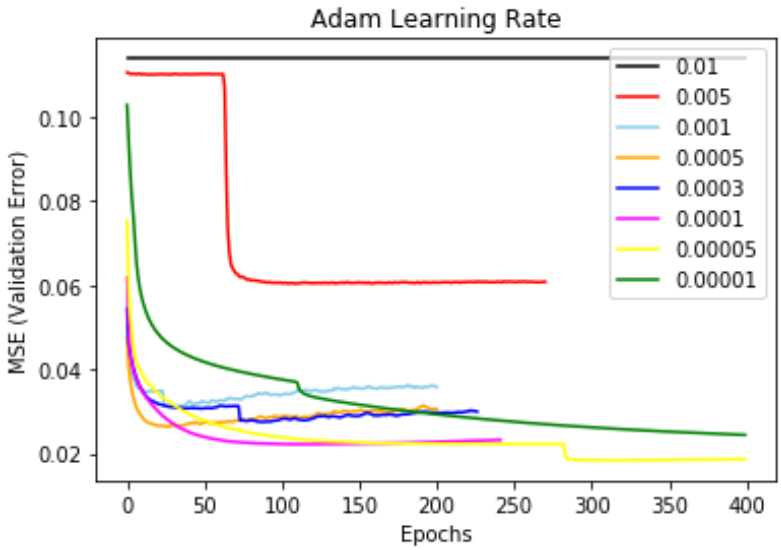

Figure 19: the validation losses recorded when training the neural network with different Adam learning rates.

Analysis: the results demonstrated above match the predicted behaviour of Adam when altering the learning rate. For a "large" learning rate, for example 0.01, the steps taken to adjust the weights were too "large" and not leading to convergence. While a "small" learning rate, for example 10^{-5}, took longer time to converge and the steps were too "small". The optimal fixed learning rate range was between 10^{-3} and 10^{-5}. For the next few tests, the learning rate used was $1x10^{-4}$ as it was an optimal trade-off between convergence time and generalisation performance.

Test 3: Adam Batch Size

Selecting the optimal training batch size is an important step in the overall training performance of deep neural networks. Larger batch sizes lead to slower convergence, but smaller batch sizes tend to result in a less stable convergence.

- Number of epochs = 400
- Training batch size = **VARIABLE**
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = Tanh
- Optimizer = Adam
- Learning rate = 0.0001
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 100)

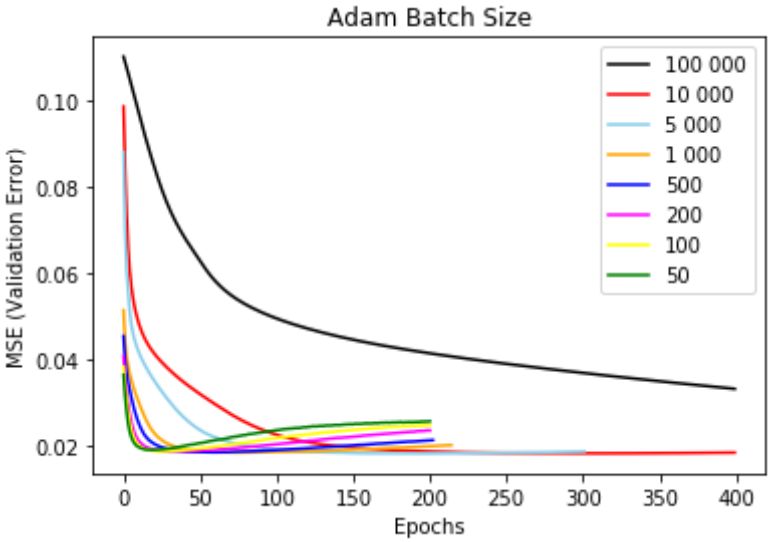

Figure 20: the validation losses recorded when training the neural network with different training batch sizes.

Batch Size	Training time, s
100 000	72
10 000	76
5 000	61
1 000	65
500	100
200	238
100	468
50	930

Table 2: the training times when using training batches of different sizes.

Analysis: the results show that the training batch sizes of 5000 or 1000 samples give the best trade-off between training time and performance. In the next experiments, the batch size of 1000 training samples was used.

Test 4: Grid search (Activation Function / Adam Learning Rate)

Grid search is a common technique used when selecting hyperparameters. In Test 4, two finite sets of activation functions and Adam learning rates were taken, and then the model was trained using every single combination of values from both sets. For example, if there are 3 learning rates and 3 activation functions, there will be 3x3 = 9 trainings completed. Grid search works well with relatively small sets for 2-3 hyperparameters, as the computational cost grows exponentially with the number of hyperparameters.

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = **VARIABLE**
- Optimizer = Adam
- Learning rate = **VARIABLE**
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 100)

LR / Activ. Func.	**0.001**	**0.0005**	**0.0003**	**0.0001**
Leaky ReLU	0.0413	0.0491	0.0633	0.0386
ReLU6	0.0280	0.0266	0.0221	0.0180
ELU	0.0284	0.0277	0.0235	0.0429
RReLU	0.0409	0.0328	0.0273	0.0310
Tanh	0.0192	0.0186	0.0183	0.0183
Hardtanh	0.0199	0.0196	0.0195	0.0192

Table 3: the lowest validation errors achieved during training for each combination of the activation function and the learning rate.

LR / Activ. Func.	**0.001**	**0.0005**	**0.0003**	**0.0001**
Leaky ReLU	84.43%	74.48%	57.95%	79.03%
ReLU6	90.15%	88.69%	90.31%	92.70%
ELU	89.22%	87.73%	91.16%	74.06%
RReLU	80.41%	87.11%	88.61%	85.03%
Tanh	89.47%	90.41%	90.39%	90.11%
Hardtanh	89.97%	89.82%	89.38%	89.29%

Table 4: the percentages of achieved sum-rates of the trained neural networks over that of the WMMSE for each combination of the activation function and the learning rate (averaged using 1 000 testing samples).

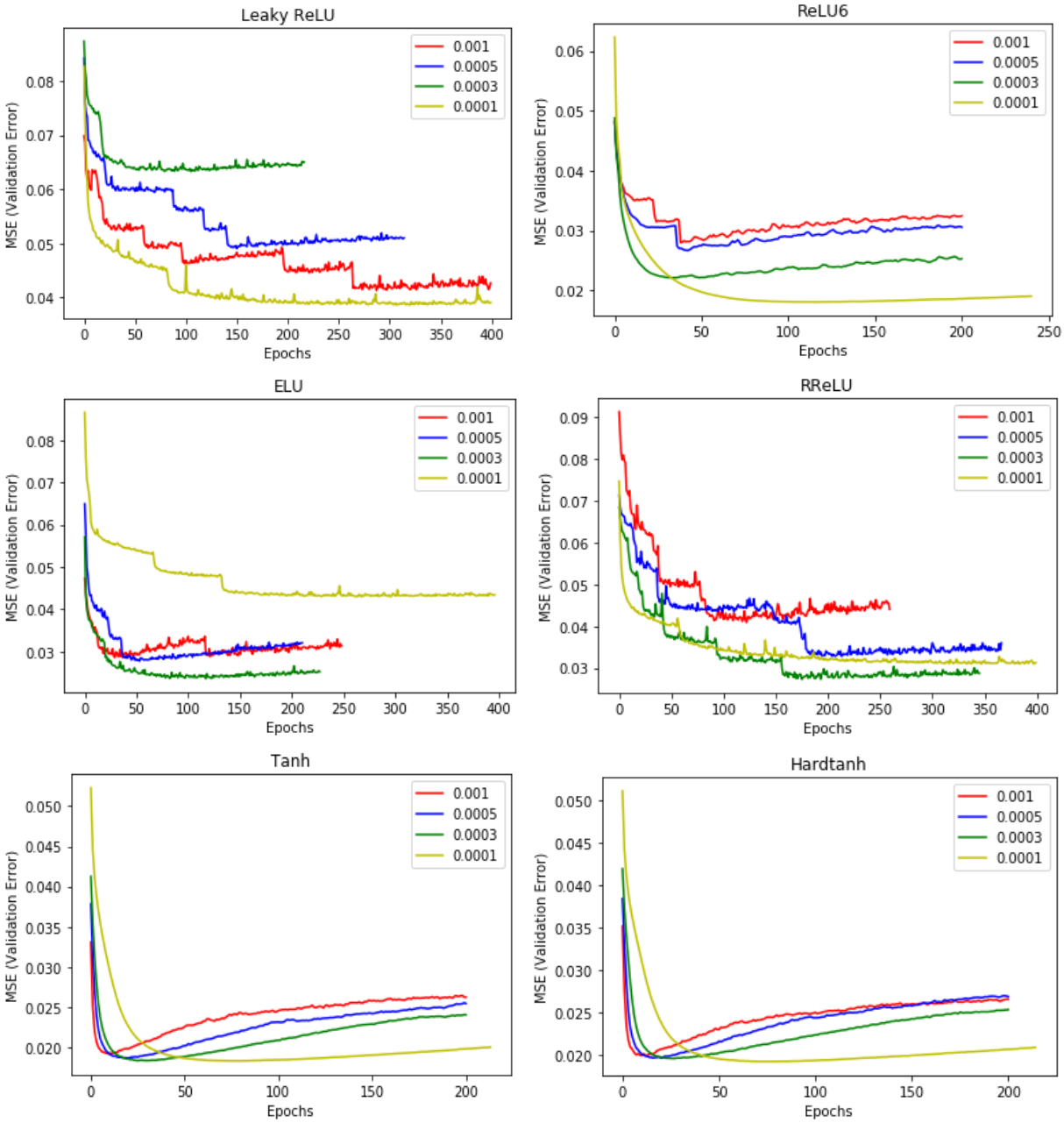

Figure 21: the validation losses recorded when each of the activation functions was used in combination with different Adam learning rates.

Analysis: following the results shown in Test 1, Tanh and HardTanh showed fastest convergence and low validation errors, whereas 4 different variants of ReLU function showed unstable behaviour and sudden drops of validation error. However, ReLU6 displayed the best approximation result and the lowest validation error with 1e-4 Adam learning rate. As the result, in the following steps of hyperparameters tuning ReLU6 was used as an activation function.

Test 5: Weights Initialisation

Weights initialisation can be done in different ways and it has a significant influence on the training performance of a deep neural network. In this test, the comparison of the most common strategies was completed.

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = Adam
- Learning rate = 0.0001
- Weights initialisation = **VARIABLE**
- Regularisation techniques = Early Stopping (patience = 100)

Weights Initialisation	Default (Uniform)	Xa - U	Xa - N	Kaiming - U	Kaiming - N
Lowest validation error	0.0180	0.0454	0.0369	0.0737	0.0612
Sum-rate approx.	92.70%	76.51%	83.36%	54.92%	70.56%

Table 5: the lowest validation errors achieved during training and the sum-rate approximation accuracies (averaged using 1 000 testing samples) achieved when using different weights initialisations.

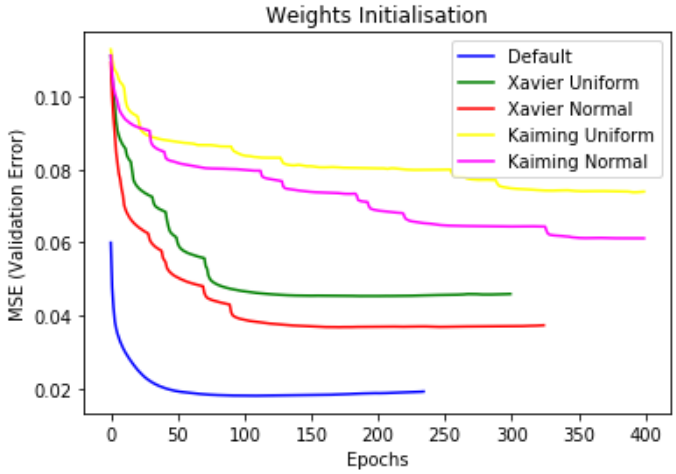

Figure 22: the validation losses recorded when using different strategies for weights initialisation.

Analysis: the model performance decreased as the magnitude of the initial weights increased. The magnitude of weights in Default initialisation, Xavier initialisation and Kaiming initialisation can be described as follows: Default < Xavier < Kaiming. The results suggested that the magnitude of initial weights is inversely proportional to generalisation performance, however there must be a lower threshold, closer to 0, where initial weights are too small for training. Default uniform distribution seems to be the best choice as the result of Test 5. In the next test, a few more custom initialisations are presented to compare their performance against the Default uniform distribution.

Test 6: Custom Weights Initialisations

As identified in the previous test, the quickest convergence was achieved when the initial weights were distributed uniformly between $-\sqrt{\frac{1}{input\ layer\ size}}$ and $\sqrt{\frac{1}{input\ layer\ size}}$ (Default, fig. 15), which resulted in a set of weights with smaller magnitudes than in Kaiming or Xavier distributions. Test 6 shows the performance of the neural network when the weights between the hidden layers were initialised uniformly between lower absolute values than in the Default uniform initialisation. In the case when there are 3 hidden layers, there are 5 layers in total, and 4 sets of weights. The shape of the weights matrices between the input layer and the first hidden layer changes with the altering number of base stations and users, and in this test, they were still initialised in the Default way. When the size of the hidden layer was 200, by default, the weights were initialised as follows:

$$U(-\sqrt{\frac{1}{200}}, \sqrt{\frac{1}{200}}) \sim U(-0.0707, 0.0707)$$

- Number of epochs = 400
- Training batch size = 1000 (1%)
- Validation batch size = 1000 (2%)
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = Adam
- Learning rate = 0.0001
- Weights initialisation = **VARIABLE**
- Regularisation techniques = Early Stopping (patience = 100)

Weights Initialisation	Default (Uniform)	U(-0.05,0.05)	U(-0.03,0.03)	U(-0.01,0.01)
Lowest validation error	0.0180	0.0257	0.0333	0.0580
Sum-rate approx.	92.70%	87.31%	82.20%	63.99%

Table 6: the lowest validation errors achieved during training and the sum-rate approximation accuracies (averaged using 1 000 testing samples) achieved when using different custom weights initialisations.

Analysis: although, when the weights were initialised by U(-0.05,0.05), the neural network improved its sum-rate approximation performance in comparison to Xavier and Kaiming initialisations, the Default initialisation remained the highest-performing. As the absolute magnitude of boundaries, between which the initial weights were uniformly initialised, decreased, the lowest validation error increased. Therefore, the Default uniform initialisation proved to be the most appropriate strategy to use.

Test 7: Dropout

The main aim of this test was to introduce a regularisation technique, dropout, to the training process and find the optimal dropout probability of neurons in the hidden layers. As discussed in Chapter 2, 3.10.1., the probability of input units to retain in the network should be higher than for the hidden layers. Throughout the test, the dropout probability (the probability of being dropped) for the input layer was set to 0.2.

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = Adam
- Learning rate = 0.0001
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 100), Dropout (probability = **VARIABLE**)

Dropout prob.	**0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**
Lowest validation error	0.0180	0.0222	0.0178	0.0220	0.0265	0.0385	0.0345	0.0834	0.0516	0.0592
Sum-rate approx.	92.70%	90.37%	92.94%	89.65%	86.60%	78.65%	81.69%	38.09%	70.66%	61.92%

Table 7: the lowest validation errors achieved during training and the sum-rate approximation accuracies (averaged using 1 000 testing samples) achieved when setting different probabilities for the hidden units to be dropped.

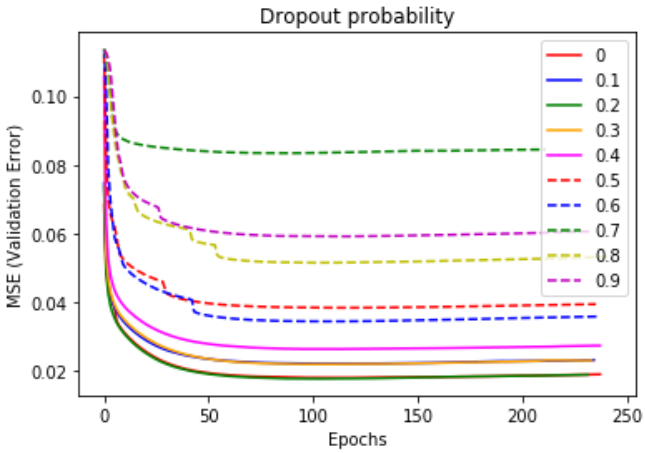

Figure 23: the validation losses recorded when setting different probabilities for the hidden units to be dropped.

Analysis: introduction of dropout resulted in a slight performance improvement (92.70% -> 92.94%) when the probability of units being removed from the network during training was 0.2. Also, from the results it can be seen that the performance severely degraded after the probability reached 0.3. Therefore, the appropriate range of dropout probability values was between 0 to 0.3. For the next tests 0.2 was used.

Test 8: AdamW, finding optimal weight decay value

As mentioned in Chapter 2, 3.10.2., AdamW is a variant of the standard Adam algorithm, but with a more efficient implementation of the weight decay. The motivation of Test 8 was to find the optimal weight decay value that could improve the generalisation performance of the model.

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = AdamW
- Learning rate = 0.0001
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 100), Dropout (probability = 0.2), Weight decay = **VARIABLE**

Weight decay value	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**
Lowest validation error	0.0182	0.0182	0.0176	0.0175	0.0177
Sum-rate approx.	92.59%	91.85%	92.92%	93.06%	92.71%

Table 8: the lowest validation errors achieved during training and the sum-rate approximation accuracies (averaged using 1 000 testing samples) achieved when applying different weight decay values to the cost function.

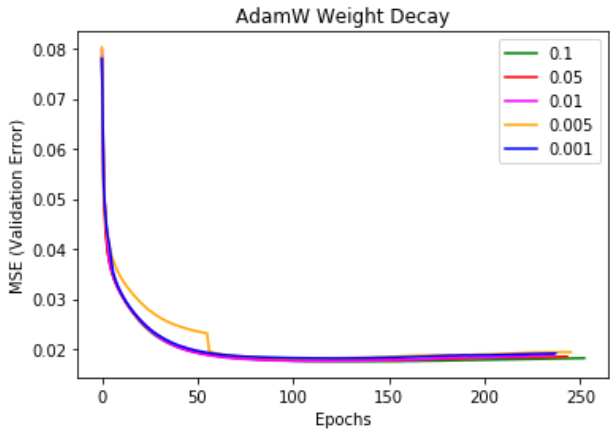

Figure 24: the validation losses recorded when applying different weight decay values to the cost function.

Analysis: the suggested default weight decay value in PyTorch is 0.01. The results of the test did not provide a definitive answer about the most appropriate weight decay value, however there was a slight improvement in the lowest validation error (0.0178 -> 0.0175) in comparison to the previous tests. Overall, both introduction of dropout and weight decay had a positive impact on the model performance, although it was not significant. The weight decay value of 0.05 was used in the following tests.

Test 9: SGD and Its Variants

The choice of the right optimiser for a deep neural network often lies between Adam and its variants (e.g. AdamW, AMSGrad) and Stochastic Gradient Descent and its variants (e.g. SGD with momentum, SGD with Nesterov momentum). Test 9 was split into four parts to find the optimal set of hyperparemeters (i.e. learning rate, batch size, momentum parameter, Nesterov momentum parameter) for SGD and compare its performance against the adaptive learning rates algorithms (Adam, AdamW and AMSGrad).

Part 1: SGD, configuring learning rate (no momentum)

- Number of epochs = 800
- Training batch size = 200
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = SGD
- Learning rate = **VARIABLE**
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 50), Dropout (probability = 0.2)

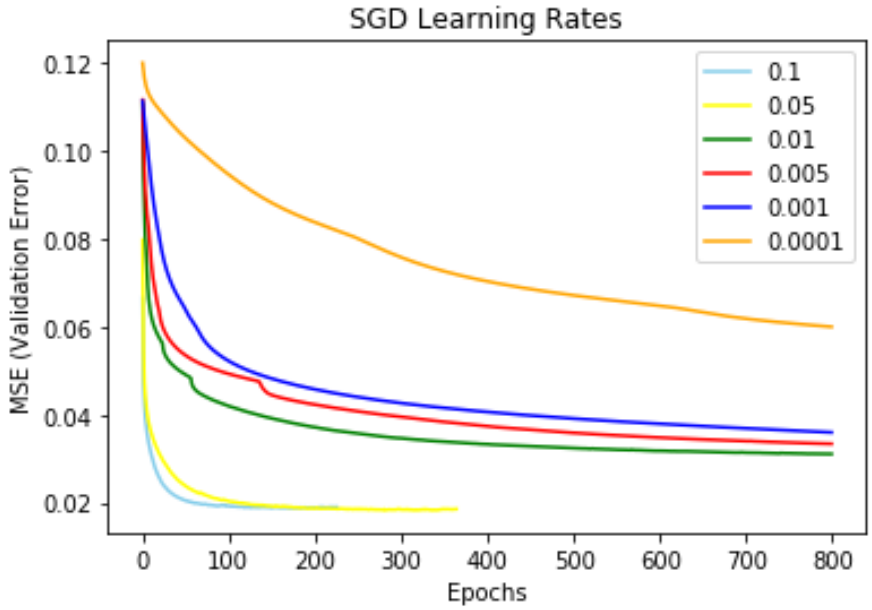

Figure 25: the validation losses recorded when training the neural network with different SGD learning rates.

Analysis: the learning rate has a significant impact on the training performance of SGD, and 0.1 and 0.05 were the best performing learning rates. Also, it can be seen on fig. 25 that as the learning rate decreased, the time of convergence increased.

Part 2: SGD, configuring batch size (no momentum)

- Number of epochs = 800
- Training batch size = **VARIABLE**
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = SGD
- Learning rate = 0.1
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 50), Dropout (probability = 0.2)

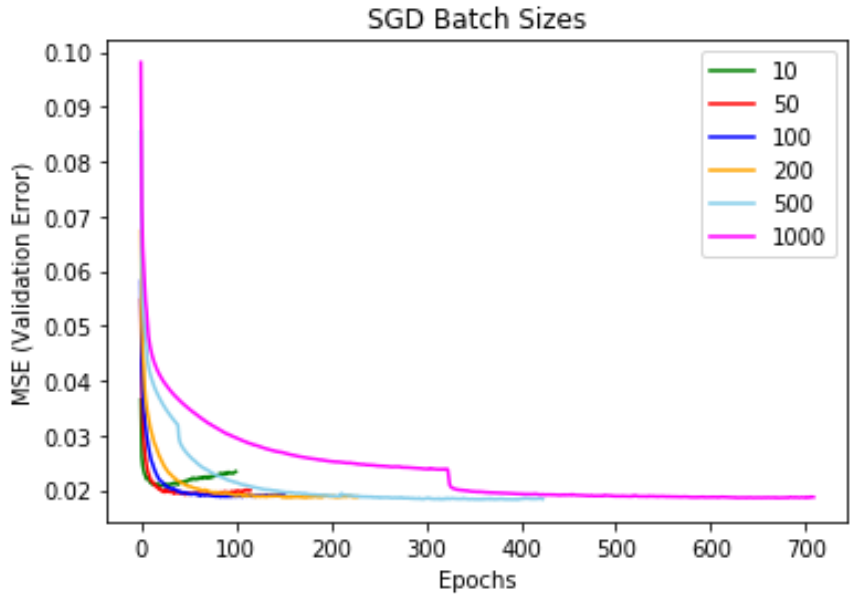

Figure 26: the validation losses recorded when training the neural network with different training batch sizes.

Batch Size	Training time, s
10	1955
50	418
100	332
200	268
500	325
1 000	362

Table 9: the training times when using training batches of different sizes.

Analysis: the training batch sizes of 200 and 500 showed to be the most appropriate choices, as they are the best trade-off between the speed of convergence and the training time. The batch size of 200 training samples was used in the next tests.

Part 3: SGD, Grid Search (Learning Rate / Momentum)

This part of Test 9 presents the grid search to find the optimal combination of the learning rate and the momentum value.

- Number of epochs = 800
- Training batch size = 200
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = SGD (momentum = **VARIABLE**)
- Learning rate = **VARIABLE**
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 50), Dropout (probability = 0.2)

Momentum \ LR	**0.1**	**0.05**	**0.01**	**0.005**
0.99	0.0627	0.0423	0.0558	0.0553
0.9	0.0277	0.0243	0.0186	0.0184
0.5	0.0190	0.0187	0.0225	0.0190

Table 10: the lowest validation errors achieved during training for each combination of the learning rate and the momentum value.

Momentum \ LR	**0.1**	**0.05**	**0.01**	**0.005**
0.99	62.18%	76.84%	65.99%	65.20%
0.9	86.91%	90.51%	92.33%	92.32%
0.5	92.74%	92.76%	89.22%	89.24%

Table 11: the percentages of achieved sum-rates of the trained neural networks over that of the WMMSE for each combination of the learning rate and the momentum value (averaged using 1 000 testing samples).

Analysis: the test did not provide a definitive answer for which combination of the learning rate and the momentum value is the most optimal, however when the momentum value was 0.99, the performance of the model was significantly worse than in the case of 0.9 and 0.5.

Part 4: SGD, Grid Search (Learning Rate / Nesterov Momentum)

As mentioned in Chapter 2, 3.9.1.2., there are two major ways of adding the momentum term to SGD. In Part 3 of Test 9, the grid search was performed for the standard momentum method, and in this part, the same grid search was done for the Nesterov momentum method.

- Number of epochs = 800
- Training batch size = 200
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = SGD (Nesterov momentum = **VARIABLE**)
- Learning rate = **VARIABLE**
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 50), Dropout (probability = 0.2)

Momentum \ LR	**0.1**	**0.05**	**0.01**	**0.005**
0.99	0.0512	0.0418	0.0365	0.0636
0.9	0.0199	0.0234	0.0305	0.0224
0.5	0.0187	0.0187	0.0191	0.0269

Table 12: the lowest validation errors achieved during training for each combination of the learning rate and the Nesterov momentum value.

Momentum \ LR	**0.1**	**0.05**	**0.01**	**0.005**
0.99	69.84%	76.09%	82.19%	59.18%
0.9	91.28%	89.94%	84.07%	90.43%
0.5	92.59%	93.20%	91.89%	84.27%

Table 13: the percentages of achieved sum-rates of the trained neural networks over that of the WMMSE for each combination of the learning rate and the momentum value (averaged using 1 000 testing samples).

Analysis: as in Part 3, the combinations of 0.1 and 0.5 (learning and momentum resp.), and 0.05 and 0.5 showed the best results. When the learning rate was 0.05 and the momentum value 0.5, the model showed the highest sum-rate approximation across two grid searches. In the final comparison between all optimisers considered in the Chapter 4, the learning rate was set to 0.05 and the momentum value was set to 0.5.

Test 10: Adam / AdamW / AMSGrad / SGD with Momentum / SGD with Nesterov

In this test, the results of comparing Adam, AdamW, AMSGrad, SGD with momentum and SGD with Nesterov momentum are presented. 200 000 samples were used during training and 10 000 samples during testing to perform the final check of the selected hyperparameters.

- Number of epochs = 800
- Training batch size = 200
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- **Optimizer = Adam (lr = 0.0001), AdamW (lr = 0.0001, weight decay = 0.05), AMSGrad (lr = 0.0001, weight decay = 0.05), SGD with Nesterov (lr = 0.05, momentum =0.5), SGD with Momentum (lr = 0.05, momentum =0.5)**
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 50), Dropout (probability = 0.2)

Weight decay value	Adam	AdamW	AMSGrad	SGD with momentum	SGD with Nesterov
Lowest validation error	0.0164	0.0160	0.0156	0.0164	0.0166
Sum-rate approx.	94.46%	94.73%	94.87%	94.11%	93.85%

Table 14: the lowest validation errors achieved during training and the sum-rate approximation accuracies (averaged using 10 000 testing samples) achieved when training the neural network with five different optimisers.

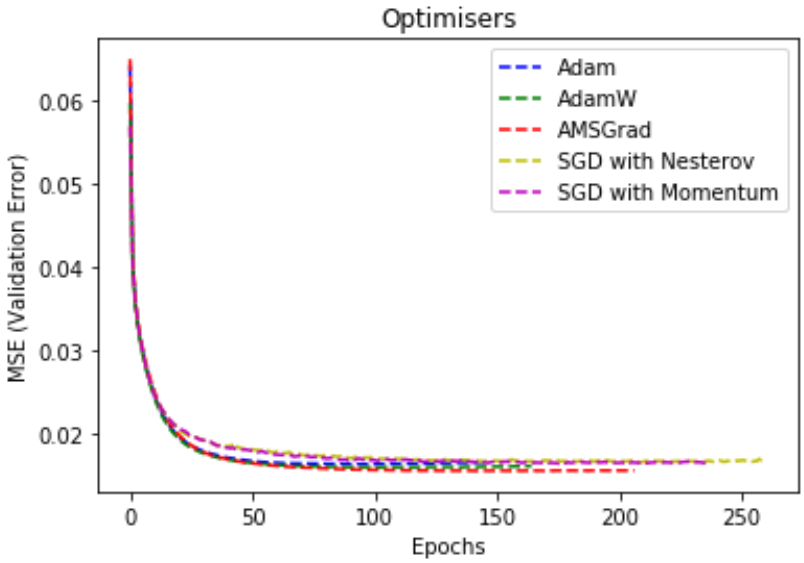

Figure 27: the validation losses recorded when training the neural network with five different optimisers.

Analysis: all three adaptive learning algorithms (Adam, AdamW, AMSGrad) showed superior performance against SGD with momentum and SGD with Nesterov momentum, and AMSGrad proved to be the best performing optimiser.

Test 11: Cyclical Learning Rates

The final test of the Chapter 4 presents the effect of implementing the cyclical learning rates (described in Chapter 2, 3.9.2.) on the sum-rate approximation performance of the neural network. Fig. 28 shows the "LR Range Test" [23], which is an easy way to find the appropriate boundaries for the oscillating learning rate. The validation loss dropped significantly starting at around 5e-6 and continued to go down until it reached approximately the value of 5e-3. AMSGrad was selected for the performance comparison as the use of the optimiser resulted in the highest sum-rate approximation accuracy in Test 10. The table 15 shows the comparison of the four ranges of AMSGrad learning rate. The learning rate oscillated linearly between the boundaries (i.e. triangular window).

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = AMSGrad (weight decay = 0.05)
- Learning rate = **CYCLICAL (VARIABLE)**
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 50), Dropout (probability = 0.2)

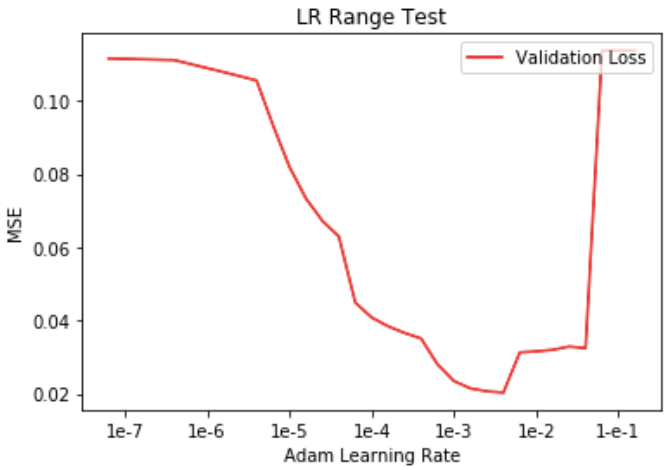

Figure 28: the validation loss recorded when steadily increasing the Adam learning rate starting at a very small value.

Weight decay value	1e-5 – 1e-3	1e-6 – 1e-3	1e-4 – 1e-3	1e-5 – 1e-4
Lowest validation error	0.0151	0.0193	0.0153	0.0156
Sum-rate approx.	95.60%	92.28%	95.22%	95.01%

Table 15: the lowest validation errors achieved during training and the sum-rate approximation accuracies (averaged using 10 000 testing samples) achieved when training the neural network with different cyclical learning rates ranges.

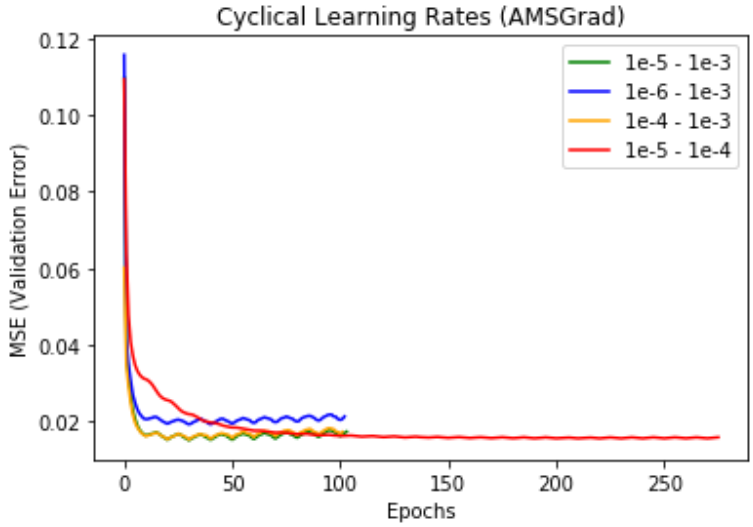

Figure 29: the validation losses recorded when training the neural network with different cyclical learning rates ranges.

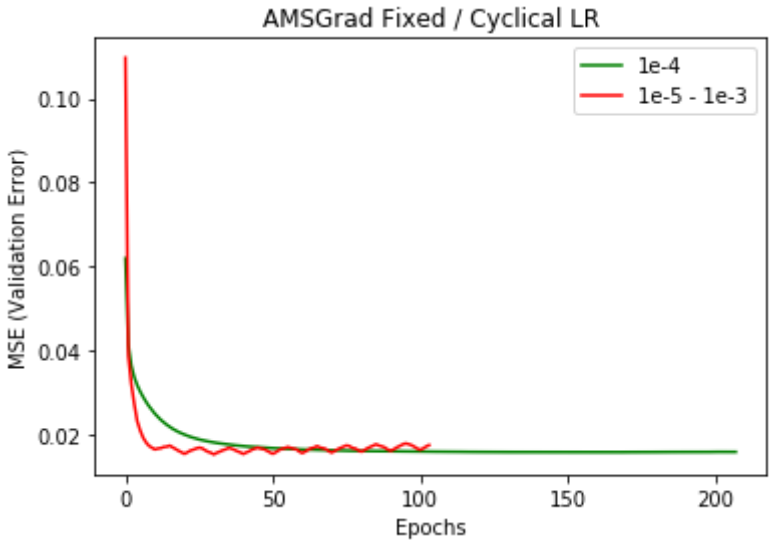

Figure 30: the validation losses when training the neural network with the fixed AMSGrad learning rate and the cyclical learning rates.

Analysis: fig. 29 demonstrates the triangular validation curves for the four ranges of AMSGrad cyclical learning rates. Fig. 30 shows the comparison of the AMSGrad algorithm with the fixed learning rate (1e-4) and cyclical learning rates (1e5 – 1e-3). By introducing the cyclical learning rates, an improvement in the lowest validation loss (0.0155 -> 0.0151) and the sum-rate approximation accuracy was achieved (94.87% -> 95.60%).

Chapter 5: Results and Evaluation

As the result of Chapter 4, the optimal structure and hyperparameters of the deep neural network for solving the outlined task (i.e. approximating WMMSE algorithm) were found. This chapter tests the proposed model. Part 1 presents the testing of the sum-rate approximation performance in the standard R = 100 m, r = 0 m setting, in which the model is trained. Part 2 shows the testing of the sum-rate approximation performance for varying r (see fig. 14) and constant R (100 m). Part 3 presents the testing of the sum-rate approximation performance for varying R (see fig. 14) and constant r (0 m). For parts 2 and 3, the same model, which was trained on R = 100 m, r = 0 m, was used to test the robustness of the trained model in varying scenarios.

The comparison between using the fixed learning rate and cyclical learning rates strategies is provided to show the general beneficial impact of using the latter approach. To better show the impact of using more samples in training, the results are provided for 50 000, 100 000, 200 0000 and 500 000 training samples. It is worth mentioning that table 20 shows the effect of "binarisation" (i.e. rounding up predictions higher than 0.5 to 1, and lower than 0.5 to 0). The previous results in [1] are presented having the predictions binarised, so in order to compare the results of the project and the results from [1], the binarisation is done in tables 21, 22 and 23.

Note: all results presented in Chapter 5 and the code used to generate them can be accessed in additional materials. The output files, stored as HTML files, are provided and can be opened using a web-browser.

- Number of epochs = 400
- Training batch size = 1000
- Validation batch size = 1000
- Number of hidden layers = 3
- Number of hidden units in each layer = 200
- Activation function = ReLU6
- Optimizer = AMSGrad (weight decay = 0.05)
- Learning rate = Cyclical (1e-5 to 1e-3) and Fixed (1e-4)
- Weights initialisation = Default
- Regularisation techniques = Early Stopping (patience = 50), Dropout (probability = 0.2)

Part 1: R = 100, r = 0

	Fixed LR	**Cyclical LR**
(3,4)	94.61% (16.79/17.74)	95.61% (16.96/17.74)
(3,6)	92.27% (18.83/20.41)	89.85% (18.34/20.41)
(3,8)	87.27% (19.47/22.31)	92.09% (20.55/22.31)
(7,4)	89.53% (30.66/34.24)	91.41% (31.30/34.24)

Table 16: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 50 000 samples over that of the WMMSE (averaged using 10 000 testing samples).

	Fixed LR	Cyclical LR
(3,4)	96.64% (17.15/17.74)	97.11% (17.23/17.74)
(3,6)	94.78% (19.34/20.41)	95.73% (19.54/20.41)
(3,8)	90.30% (20.15/22.31)	93.46% (20.86/22.31)
(7,4)	92.54% (31.69/34.24)	94.27% (32.28/34.24)

Table 17: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 100 000 samples over that of the WMMSE (averaged using 10 000 testing samples).

	Fixed LR	Cyclical LR
(3,4)	97.36% (17.27/17.74)	97.72% (17.34/17.74)
(3,6)	96.70% (19.73/20.41)	96.82% (19.76/20.41)
(3,8)	94.67% (21.13/22.31)	96.45% (21.52/22.31)
(7,4)	93.98% (32.18/34.24)	94.46% (32.34/34.24)

Table 18: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 200 000 samples over that of the WMMSE (averaged using 10 000 testing samples).

	Fixed LR	Cyclical LR
(3,4)	97.61% (17.32/17.74)	97.95% (17.38/17.74)
(3,6)	96.60% (19.71/20.41)	97.16% (19.83/20.41)
(3,8)	95.82% (21.38/22.31)	96.45% (21.52/22.31)
(7,4)	94.43% (32.34/34.24)	96.49% (33.04/34.24)

Table 19: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 500 000 samples over that of the WMMSE (averaged using 10 000 testing samples).

	Cyclical without binarisation	Cyclical with binarisation
(3,4)	97.95% (17.38/17.74)	98.99% (17.56/17.74)
(3,6)	97.16% (19.83/20.41)	98.34% (20.07/20.41)
(3,8)	96.45% (21.52/22.31)	98.26% (21.93/22.31)
(7,4)	96.49% (33.04/34.24)	96.95% (33.20/34.24)

Table 20: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 500 000 samples without and with binarised predictions over that of the WMMSE (averaged using 10 000 testing samples).

	My results (with binarisation)	Sun et al. [1] results (with binarisation)
(3,4)	98.99% (17.56/17.74)	98.30% (17.72/18.03)
(3,6)	98.34% (20.07/20.41)	97.45% (20.08/20.61)
(3,8)	98.26% (21.93/22.31)	96.82% (21.93/22.65)
(7,4)	96.95% (33.20/34.24)	94.53% (33.51/35.45)

Table 21: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 500 000 samples with binarised predictions over that of the WMMSE (averaged using 10 000 testing samples) compared with the results achieved in [1].

Analysis: tables 16 – 19 show continuous improvement in the sum-rate approximation accuracy with the increasing number of training samples used. As the size of the problem increases, the sum-rate approximation accuracy starts going down as can be seen in the mentioned tables. The authors of [1] achieved 97.25% (BS = 3, K = 4) and 95.30% (BS = 3, K = 8) without binarisation using 1 000 000 training samples. The model proposed in the project shows superior performance starting with 200 000 training samples, and even better performance with 500 000 training samples. In almost all tests (15/16), the use of cyclical learning rates yielded the better approximation accuracy, which encourages the use of the strategy in further research and practical applications. Table 20 shows clearly that using the binarisation is advantageous, and provides a better sum-rate approximation. Finally, table 21 provides the comparison of the sum-rate approximations for each of the four IMAC scenarios between the model proposed in the project and the results achieved in [1]. The proposed model helped moving the sum-rate approximation of the DNN closer to 100%.

Part 2: R = 100, r = {0,20,50,99}

	My results (20)	Sun et al. [1] results (20)	My results (50)	Sun et al. [1] results (50)	My results (99)	Sun et al. [1] results (99)
(3,4)	98.87% (14.38/14.54)	98.29%	98.91% (12.11/12.25)	98.38%	98.95% (10.03/10.14)	98.46%
(3,6)	98.45% (16.37/16.63)	96.99%	98.37% (13.81/14.03)	96.87%	98.23% (11.52/11.73)	96.64%
(3,8)	97.94% (17.71/18.08)	95.85%	97.92% (14.97/15.29)	95.41%	97.79% (12.59/12.87)	94.65%
(7,4)	96.91% (26.73/27.59)	93.80%	96.84% (22.08/22.81)	93.47%	95.94% (17.89/18.64)	92.55%

Table 22: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 500 000 samples with binarised predictions over that of the WMMSE (averaged using 10 000 testing samples) compared with the results achieved in [1] for different values of the inner radius, r.

Analysis: the proposed model showed the better approximation performance in all 12 tests across the 4 IMAC scenarios. As previously shown in [1], the sum-rate approximation performance slightly degrades with the increasing r, which is intuitively explained by the radically different scenarios used for testing and training. However, the results are still encouraging, and even when the inner radius, r, is 99 (i.e. all users are placed on the cell boundary), the sum-rate approximation accuracy is still high (i.e. > 95% for all scenarios).

Part 3: R = {200,300,500}, r = 0

	My results (200)	Sun et al. [1] results (200)	My results (300)	Sun et al. [1] results (300)	My results (500)	Sun et al. [1] results (500)
(3,4)	100.03% (12.91/12.90)	99.51%	100.58% (9.59/9.54)	99.62%	100.32% (5.67/5.65)	98.91%
(3,6)	99.51% (15.39/15.47)	98.27%	99.59% (11.83/11.88)	97.80%	98.85% (7.22/7.30)	95.58%
(3,8)	98.50% (17.12/17.38)	96.77%	96.85% (13.09/13.52)	93.83%	90.74% (7.83/8.62)	87.32%
(7,4)	98.47% (26.96/27.38)	97.19%	95.46% (20.16/21.12)	95.41%	86.50% (11.21/12.97)	90.22%

Table 23: the average achieved sum-rate performances (in brackets) and the percentages of achieved sum-rates of the neural networks trained with 500 000 samples with binarised predictions over that of the WMMSE (averaged using 10 000 testing samples) compared with the results achieved in [1] for different values of the cell radius, R.

Analysis: the proposed model showed the better approximation performance in 11 out of 12 tests across the 4 IMAC scenarios. Overall, there is a visible trend suggesting that as the radius of the cell, R, increases, the sum-rate approximation degrades (e.g. dropping from 98.47% when R = 200 m to 86.50% when R = 500 m for BS = 7, K = 4).

Chapter 6: Conclusion

The interference management in wireless networks is a developing topic that attracts attention, because of its applications in real-world communications. The traditional iterative algorithms, such as the WMMSE, that aim to come to the optimal solution by performing continuous computations until some stopping criteria is met, often make it hard to achieve real-time implementation. With the rise of the deep neural networks and the computational capacity to accommodate the learning of the large multilayer networks, the deep neural networks can offer a robust alternative to solving many problems. Their capability of learning the nonlinear mappings between the inputs and outputs can be leveraged in many ways. Once the deep neural network "learned" the parameters, the model can provide fast predictions by passing the new set of inputs forward through the network, which provides the often much-needed computational performance speed up.

The project demonstrates the steps involved in finding the optimal set of hyperparameters and the structure of the deep neural network. The proposed model offers the better sum-rate approximation performance than previously offered solution in [1], and encourages further research about how the deep neural network's performance can be improved. The project shows how the recently offered innovative methods such as AdamW [25], AMSGrad [22], Cyclical Learning Rates [23] can be used to improve the performance of the deep neural networks.

It was previously shown in [1] as well as in this report, that the multilayer perceptron model can be successfully used to approximate the WMMSE algorithm. However, there are a number of issues that still should be addressed. Firstly, it is almost impossible to train a deep neural network model for each possible interfering multiple-access channel scenario. If the number of users constantly varies within cells, there is a large number of potential configurations, and it is not viable to train a neural network for each scenario. However, the model proposed in the report, shows that a relatively high approximation accuracy can be achieved even with varying inner radius, r, and cell radius, R. The model performance only starts to significantly degrade when the cell radius, R, is very different from the scenario that the model was trained on (e.g. R = 500 m). Secondly, the proposed model relies on being trained on the labels provided by running the WMMSE algorithm, which complicates the whole process. Finally, the proposed deep neural network model works in the real domain, which is a simplification, as the channel realisations are complex numbers. This gives the rise to some potential improvements:

1. Is it possible to come up with a feasible unsupervised learning scheme that does not require training labels to allocate power efficiently?
2. What is the practical and efficient way of designing a deep neural network that can take complex values as inputs?

Bibliography

[1] H. Sun, X. Chen, Q. Shi, M. Hong, X. Fu & N. D. Sidiropoulos (2018), “Learning to Optimize: Training Deep Neural Networks for Interference Management”, *IEEE Transactions Signal Processing*, vol. 66, no. 20, pp. 5438–5453, 2018.

[2] N. Samuel, T. Diskin and A. Wiesel, "Deep MIMO detection," *2017 IEEE 18th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC),* Sapporo, 2017, pp. 1-5.

[3] T. J. O’Shea, T. Erpek, and T. C. Clancy, “Deep learning based MIMO communications,” 2017, arXiv: 1707.07980.

[4] Karol Gregor and Yann LeCun, “Learning fast approximations of sparse coding”, *In Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML’10)*, 2010, pp. 399–406.

[5] S. Venkatraman, “Interference Channels: A Review”, Accessed on: April 14, 2020 [Online]. Available: https://pdfs.semanticscholar.org/4a65/d5f48c995aaca959090dfbba6c9ae7ecaa04.pdf

[6] Q. Shi, M. Razaviyayn, Z. Luo and C. He, "An Iteratively Weighted MMSE Approach to Distributed Sum-Utility Maximization for a MIMO Interfering Broadcast Channel," *IEEE Transactions on Signal Processing*, vol. 59, no. 9, pp. 4331-4340, Sept. 2011.

[7] T.M. Mitchell, “Introduction” in *Machine Learning*, McGraw-Hill Science/Engineering/Math, p.12, 1997.

[8] S.J. Russell and Peter Norvig, “Supervised Learning” in *Artificial Intelligence: A Modern Approach*, Third Edition, Prentice Hall, p.695, 2010.

[9] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, Cambridge, MA : MIT Press, 2017.

[10] R.E. Bellman, “Dynamic programming”, Princeton, NJ, Princeton University Press, 1957.

[11] E. Keogh, A. Mueen, “Curse of Dimensionality” In: Sammut C., Webb G.I. (eds) “Encyclopedia of Machine Learning and Data Mining”, Boston, MA, Springer, 2017.

[12] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[13] C.E. Nwankpa, W. Ijomah, A. Gachagan and S. Marshall, “Activation Functions: Comparison of Trends in Practice and Research for Deep Learning”, 2018, arXiv: 1811.03378.

[14] B. Xu, N. Wang, T. Chen and M. Li, "Empirical Evaluation of Rectified Activations in Convolutional Network", 2015, arXiv: 1505.00853

[15] A. Krizhevsky, “Convolutional Deep Belief Networks on CIFAR-10”, 2010 [Online]. Available: https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf

[16] Xavier Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks”, *Journal of Machine Learning Research – Proceedings,* Track.9, pp. 249-256, 2010.

[17] K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *2015 IEEE International Conference on Computer Vision (ICCV),* pp. 1026-1034, 2015.

[18] B.T. Polyak, “Some methods of speeding up the convergence of iteration methods”, *USSR Computational Mathematics and Mathematical Physics,* vol.4, pp. 1-17, 1964.

[19] J. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, *Journal of Machine Learning Research*, 12, pp. 2121-2159, 2011.

[20] G. Hinton, N. Srivastava, and K. Swersky, “Lecture 6a overview of mini–batch gradient descent,” University of Toronto Slides, 2012. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[21] D.P. Kingma and J.L. Ba, “Adam: A Method For Stochastic Optimization”, published at *International Conference on Learning Representations (ICLR),* 2015.

[22] S.J. Reddi, S. Kale and S. Kumar, “On The Convergence of Adam And Beyond”, published at *International Conference on Learning Representations (ICLR),* 2018

[23] L. N. Smith, “Cyclical Learning Rates for Training Neural Networks”, *IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464-472, 2017.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, editor: Y. Bengio, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *Journal of Machine Learning Research 15,* pp. 1929-1958, 2014.

[25] I. Loshchilov and F. Hutter, “Decoupled Weight Decay Regularisation”, published at *International Conference on Learning Representations (ICLR)*, 2019.

Additional Materials

The source code used to generate the results presented in the report is available in two formats. The four HTML files present in the zip file can be opened using any web-browser, and they will contain all the code with the comments and graphs.

1. 3_4.html – 3 base stations, 4 users in each cell
2. 3_6.html – 3 base stations, 6 users in each cell
3. 3_8.html – 3 base stations, 8 users in each cell
4. 7_4.html – 7 base stations, 4 users in each cell

The other four files (.ipynb format) are the Jupyter notebooks, which can be opened using the widely-used distribution of Python – Anaconda, which includes the web-based environment where the files can be opened.

1. 3_4.ipynb – 3 base stations, 4 users in each cell
2. 3_6.ipynb – 3 base stations, 6 users in each cell
3. 3_8.ipynb – 3 base stations, 8 users in each cell
4. 7_4.ipynb – 7 base stations, 4 users in each cell

The sample of the data used in training the neural network is also provided. However, due to the large size of the CSV files, only a few examples of the data used for training can be included. All datasets were generated from Generate_IMAC_function.m and Layout.m files available at https://github.com/HaoranS/SPAWC2017.

1. train_channels_50.csv – 50 000 channel realisations used for training
2. train_labels_50.csv – 50 000 power allocations used for training
3. valid_channels.csv – 50 000 channel realisations used for validation
4. valid_labels.csv – 50 000 power allocations used for validation
5. test_channels.csv – 10 000 channel realisations used for testing
6. test_labels.csv – 10 000 power allocations used for testing
7. test_H.csv – 10 000 channels used for testing