**ISTANBUL TECHNICAL UNIVERSITY**
**FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING**

# QUEUEING RECURRENT NEURAL NETWORK: A COMBINATION OF RECURRENT NEURAL NETWORKS AND QUEUING THEORY

**Bachelor's Thesis**

**Bilgehan KÖSEM**

**Control and Automation Engineering**

**Thesis Supervisor: Ass. Prof. Gülay ÖKE GÜNEL**

**JULY 2020**

**ISTANBUL TECHNICAL UNIVERSITY**
**FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING**

# QUEUEING RECURRENT NEURAL NETWORK: A COMBINATION OF RECURRENT NEURAL NETWORKS AND QUEUING THEORY

**Bachelor's Thesis**

**Bilgehan KÖSEM**
**040150404**

**Control and Automation Engineering**

**Thesis Supervisor: Ass. Prof. Gülay ÖKE GÜNEL**

**JULY 2020**

## PREFACE

I would like to thank my family who supported me throughout my education life, my friends who made my university life worth to remember and my teacher Ass. Prof. Gülay ÖKE GÜNEL who guided me in this thesis.

July 2020 Bilgehan KÖSEM

**TABLE OF CONTENTS**

**Page**

**ABBREVIATIONS**

**QRNN** **:** Queueing Recurrent Neural Network
**RANN** **:** Random Neural Network
**RERANN** **:** Recurrent Random Neural Network
**ESQN** **:** Echo State Queuing Network
**LSTM** **:** Long-Short Term Memory
**CEC** **:** Constant Error Carousel
**RNN** **:** Recurrent Neural Network
**GRU** **:** Gated Recurrent Unit
**RMS** **:** Root Mean Square
**ESN** **:** Echo State Network
**SGD** **:** Stocastich Gradient Descent
**NAG** **:** Nesterov Accelerated Gradient
**BPTT** **:** Backpropagation Through Time

**TABLE OF FIGURES**

**Page**

**TABLE OF TABLES**

**Page**

## QUEUEING RECURRENT NEURAL NETWORK: A COMBINATION OF RECURRENT NEURAL NETWORKS AND QUEUING THEORY

## SUMMARY

Time series data analysis is widely used operation in numerous real-world applications. Tools such as statistical methods and Machine Learning algorihtms aims to acquire the characteristics of the data to make predictions about future with high accuracy. Among the Machine Learning algorithms a certain type of Artificial Neural Networks are speciliazed in time series regression problem and commonly preferred over the others due to their success on comprehending and revealing the charachteristics of sequential data. It is called Recurrent Neural Networks due to their recurrent connections expanding thorugh time. The purpose of this thesis is to propose a new kind of Recurrent Neural Network algorithm to be used in the are of sequential data analysis, especially to work with time series. It is called Queueing Recurrent Neural Network which is a compose of Random Neural Networks and Recurrent Neural Networks. The new model uses the the dynamic structure of the first one and preserves the theoretical backgorund of the second. This composite nature of Queuing Recurrent Network enables it to use the Queueing Theory while working with sequential datasets.

This thesis begins with the required background informations regarding Recurrent Neural Networks chronologically by explaining the mathematical models and the working principles. Then, investigates Random Neural Networks by presenting historical backgorund, mathematical models and application areas. Afterwards, using the previously given informations, the thesis introduces Queueing Recurrent Neural Network by giving mathematical model with detailled explanations for each equation and term. Finally, in order to measure its performance and clarify its position in the Machine Learning area especially sequential data analysis, different experiments have been runned. In the experiment phase, the performance of 5 different neural networks, have been evaluated by using 4 dataset and 10 different optimization algorithms. As a result, overall performance of Queueing Recurrent Neural Network have been evaluated with detailed comparisons. According to results, it was concluded that Queuing Recurrent Neural Neural Network performs as good as the other Recurrent Neural Networks which are LSTM, Simple RNN, GRU and even outperforms them in the majority of test cases.

## QUEUEING RECURRENT NEURAL NETWORK: RECURRENT NEURAL NETWORK VE KUYRUK TEORİSİNİN BİR KOMBİNASYONU

## ÖZET

Zaman serisi veri analizi, çok sayıda uygulamada yaygın olarak kullanılan bir analiz türüdür. İstatistiksel yöntemler ve Makine Öğrenimi algoritmaları gibi araçlar, geleceğe ilişkin tahminleri yüksek doğrulukla yapmak için verilerin karakteristik özelliklerini elde etmeyi amaçlamaktadır. Makine Öğrenimi algoritmaları arasında belirli bir Yapay Sinir Ağları türü, zaman serisi regresyon problemi için özelleşmiştir ve ardışıl veri setlerinin karakteristik özelliklerini anlama ve ortaya koymadaki başarıları nedeniyle diğerlerine göre yaygın olarak daha fazla tercih edilmektedir. Bu Yapay Sinir Ağları, zaman ekseninde genişleyen tekrarlayan bağlantıları nedeniyle Recurrent Neural Networks olarak adlandırılırlar. Bu tezin amacı, ardışıl veri setlerinin analizinde, özellikle zaman serileriyle çalışmak için kullanılacak yeni bir Recurrent Neural Network algoritması ortaya koymaktır. Random Neural Networks ve Recurrent Neural Networks'ün bir bileşimi olan bu yeni yapı Queueing Recurrent Neural Network olarak adlandırılır. Yeni model, birincisinin dinamik yapısını kullanırken ve ikincisinin teorik altyapısını korur. Queueing Recurrent Neural Network'ün bu bileşik yapısı, ardışıl veriler ile çalışırken Kuyruk Teorisini kullanmasını sağlar.

Bu tez, matematiksel modelleri ve çalışma prensiplerini açıklayarak Recurrent Neural Network yapıları ile ilgili gerekli temel bilgileri kronolojik olarak sunarak başlar. Ardından, Random Neural Network yapısını, tarihçesine, matematiksel modelline ve uygulama alanlarınına değinerek inceler. Daha sonra, öncesinde verilen bilgileri kullanarak, her denklem ve denklemlerdeki her terim için ayrıntılı açıklamalar içeren matematiksel modeli vererek Queueing Recurrent Neural Network yapısını sunar. Bu yapının performansını ölçmek ve Makine Öğrenimi alanındaki konumunu, (özellikle ardışıl veri analizini alanındaki) netleştirmek için yapılan farklı testler ve sonuçları da tez içinde mevcuttur. Deney aşamasında 5 farklı Artificial Neural Network'ün performansı 4 ayrı veri seti ve 10 farklı optimizasyon algoritmasından yaralanılarak incelenmiştir. İnceleme ile birlikte, Queueing Recurrent Neural Network'ün yapılan testlerdeki genel performansı, yorumlar ile birlikte diğer test edilen yapılar ile karşılaştırılmıştır. Bu değerlendirmeden yola çıkarak, Queueing Recurrent Neural Network'ün en az LSTM, Simple RNN, GRU ve Random Neural Network yapıları kadar iyi performans gösterdiği ve hatta testlerin çoğunda daha başarılı sonuçlar elde ettiği sonucuna ulaşılmıştır.

## 1. INTRODUCTION

Time series data means a set of chronologically sorted sequential observation points that all have dependency with the previous and the next ones. Analysis of time series data is a technique which is widely used in modelling time variant systems. In the applications of the sequential data analysis, machine learning algorithms are commonly used, powerful tools to obtain successful results especially in the modelling of highly nonlinear systems. The use of this tool, plays a key role in many real-world applications such as: mathematical finance, speech recognition, signal processing, econometrics, weather forecasting, resource allocation, business planning, astronomy, and music composition.

Recurrent Neural Networks are a certain type of machine learning algorithms which produce relatively successful results within neural networks, at investigating the characteristics of the time series data. The reason behind this superior performance of Recurrent Neural Networks is their mathematical models that contains loops with recurrences. These loops enable networks to grab certain features of the data by taking the previous samples into account. In a certain group of Recurrent Neural Networks these loops have static connections that works as a memory in the system. ELMAN-JORDAN [1] neural networks can be given as an example to this kind of architecture. Besides, another group of RNNs have dynamic connections that have a ability to enlarge or shrink through the time. As an example, LSTM, Vanilla RNN or GRU can be presented as a member of this group. It is shown that training Recurrent Neural Networks have some difficulties that is detailly covered in [2] and [3] regarding optimization of the network and the stability. One of the major problems of RNNs is a phenomenon called vanishing/exploding gradient. As gradient calculations are performed, error signal shrink or grows exponentially in a way that limits learning capacity of long-term dependencies.This phenomenon occurs in the process of backpropagation through time (BPTT) which is detailly explained in [4].

All of the previously mentioned networks are consisting of a biologically inspired perceptron model which is called as McCulloch Pitts perceptron [5]. Beside this conventional structure, in 1989 a new artificial neuron structure has been presented by Erol Gelenbe. This new architecture which is called as "Random Neural Network" was an implementation of Queueing Theory to Artifical Neural Networks. Random neural network is a model inspired by the spiking behavior of biological neurons. According to this model, each neuron has a positive potential which enables itself to emit two kind of signals: positive and negative. These signals with unit amplitude, circulate in the networks and affects the potential of the target neurons by increasing or decreasing. Unless the potential value of the neuron which is defined as integer is zero, the negative signals creates inhibition effect and decrease the potential by 1 since it is amplitude is 1. Otherwise potential does not get affected by the inhibiting signals and have a potential of zero until a positive signal is received [6]. Network can also establish signal transmission with the outside world. In 1990, Gelenbe declared the stability condition of the Random Neural Network [7] and in 1993 a recurrent structure regarding Random Neural Network is presented with detailed explanations of feedforward and backpropagation phases [8]. This algorithm and the Random Neural Network are used successfully in various applications areas such as optimization, image processing and compression, associative memories and recently time series analysis by using a structure that is combination of reservoir computing and Random Neural Network [9]. This extension model is called Echo State Queueing Network (ESQN) which has recurrences in the reservoir. In the paper that ESQN has been proposed, results regarding to the test with various univariate benchmark dataset can be found.

Beside the abilities of Random Neural Networs and its extensions (RERANN and ESQN) none of this model approaches time series data as conventional RNN model. The reason why RERANN and ESQN are declared as recurrent structures, is the static recurrent connections in the network. For example, because of these static connections, RERANN can be rendered into an autoencoder or can be used in an associative memory application. But the same recurrence architecture also makes these networks unavailable to operate with an input structure that contains multivariate time series data with multiple time steps. This situation keeps them away from the power of analysis of time series data with multiple time steps input.

The purpose of this thesis is to propose a new neural networks structure that copies queuing features Random Neural Networks and combines them with the network architecture of conventional Recurrent Neural Networks. It is called Queueing Recurrent Neural Network (QRNN). In the following sections, firstly some of the conventional RNN structures that is inspired in the QRNN architecture will be covered detailly. Then, models and features of the Random Neural Networks will be investigated with the feedforward and recurrent structure. After the presentation of these basic knowledge regarding both sides, in the next section mathematical model and general network structure of QRNN will be explained. A section about test results and performance comparison with the it's equivalents in the same network category will be presented and the position of QRNN in the machine learning ecosystem will be tried to be determined. Finally, evaluation of the test results and a summary of the essence regarding this thesis will be given in the conclusion section.

## 2. RECURRENT NEURAL NETWORKS

In this chapter mathematical models regarding some of the Recurrent Neural Networks will be presented in a chronological order to demonstrate a brief overview about evolution of RNNs through the time. In addition to that causality relations between problems and the solutions that RNN models offers will be discussed with comparisons.

### 2.1 General Working Principle

Before getting into the architecture of RNNs, general working principle regarding the networks that will be presented in this section, is given with the figure 2.1 below.

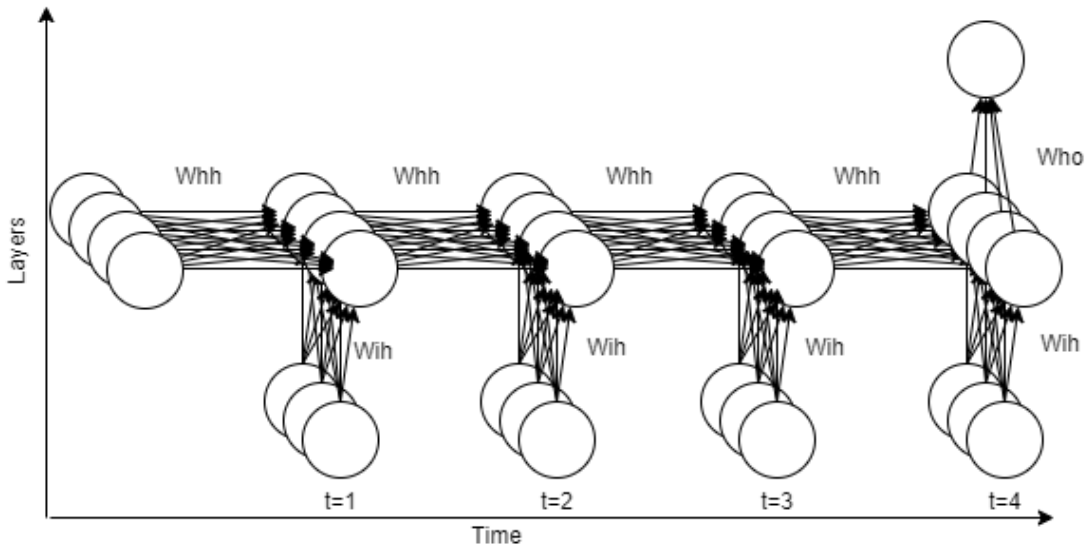


**Figure 2.1 :** General look of a Recurrent Neural Network unfolded through time using many-to-one architecture

The depicted network contains three layers as it can be seen by looking at the “Layers” axis. It has 3 neurons in input layer, 4 neurons in hidden layer and 1 neuron in output layer. The other axis “Time” shows the amount of time steps in the input data. As a solid example, assume that there exists a data set which contains 3 different variable. About the data set, it is also known that the value of 3rd variable has a complex dependency with other two variables. As a multivariate time series forecasting

application, it is wanted to predict next value of the 3rd variable by taking the last 4 value of all three variables into account. So, each node in the input layer corresponds to a variable category and each tick in time axis corresponds to a time step.

In the forward pass, for each time step t, hidden layer values of that time step are calculated as a function of input layer at time step t and hidden layer at time step t-1. Another key point about this structure is the weights. As it is written in the figure 2.1, all the occurrence of the weights Wih, Whh and Who is identical. Thus, for this particular example there only exists 3 kind of weight matrices which are shared through the time. As an initial condition, hidden layer at the time step 0 is filled with zeros and input vectors start to be given into network from time step 1.

The example in the paragraph above can be seen as a typical "many-to-one" application. It means that it takes an input with more than one time steps and gives only one output at the end of calculations through time. There also exist different applications such as "one-to-many" or "many-to-many" as it can be seen in the figure 2.2.

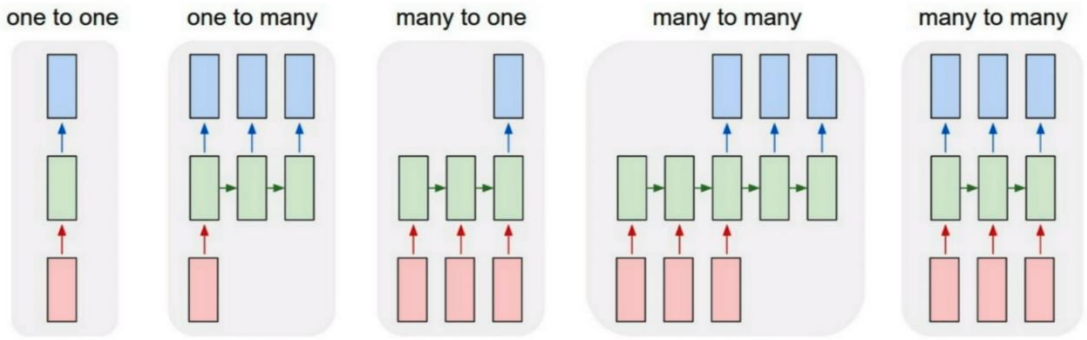


**Figure 2.2 :** Different working schemes of Recurrent Neural Networks [10].

## 2.2 Simple (Vanilla) RNN

This section contains informations regarding the mathematical structure of Simple RNN.

### 2.2.1 Mathematical Model

In the formal definition of a standard Recurrent Neural Networks for a sequence of input vectors $x_1, x_1, x_1, \dots, x_T$, the networks creates a sequence of hidden layer states $h_1, h_1, h_1, \dots, h_T$ as a function of previous hidden state vector and present input vector,

Finally a sequence of output vectors $y_1, y_1, y_1, \ldots, y_T$ are produced. In the feedforward calculation, the equations 2.1 and 2.2, are performed for each time step $t$, where the expressions $W_{xh}$, $W_{hh}$, and $W_{hy}$ denote weight matrices, $b_h$ and $b_y$ denote biases and $\sigma$ denotes activation function. It is also a useful practice to define an initial bias for the place of the term $W_{hh}h_0$ which returns zero.

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{2.1}$$

$$y_t = W_{hy}h_t + b_y \tag{2.2}$$

The illustrated version of the of the equations 2.1 and 2.2 are given with the figure 2.3.

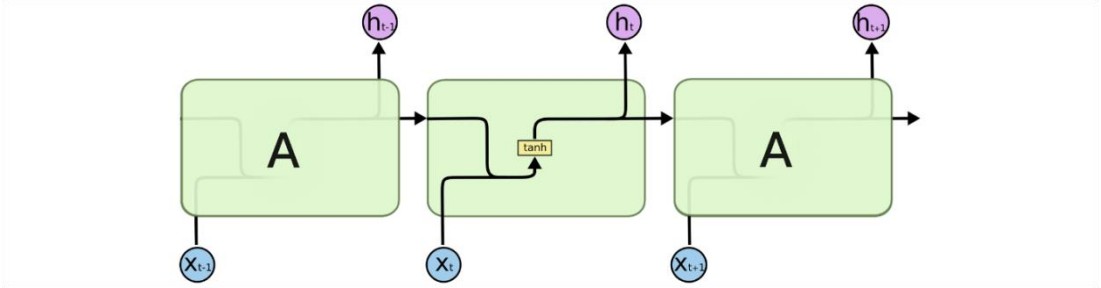


**Figure 2.3 :** Illustration of Simple RNN [11].

### 2.2.2 Backpropagation

After the forward calculation phase, total error regarding the output vectors is calculated according to the equation:

$$\mathcal{E} = \sum_{1 \le t \le T} \mathcal{E}_y \tag{2.3}$$

where each error value for each output is calculated according to loss function $\mathcal{E}_y = \mathcal{L}(y_t)$. In order to optimize weights according to calculated error, backpropagation through time (BPTT) algorithm [4] is used with the optimizer stochastic gradient descent. For the sake of simplicity all of the weight matrices and biases will be denoted with symbol $\theta$ for the following mathematical formulas as in the notation in [12]:

$$\mathcal{E} = \sum_{1 \le t \le T} \mathcal{E}_y \tag{2.4}$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial y_t} \frac{\partial y_t}{\partial h_k} \frac{\partial h_k}{\partial \theta}\right) \tag{2.5}$$

$$\frac{\partial y_t}{\partial h_k} = \prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} = W_{hh}^T diag(\sigma'(h_{i-1})) \tag{2.6}$$

As it is shown, the term $\frac{\partial \mathcal{E}_t}{\partial \theta}$ is actually a sum of all the matrices $\frac{\partial \mathcal{E}_t}{\partial \theta}$ at each time step $k$ for each time step $t$, where $T > t > k$. In other words, as the network opened up through time for an output vector at time step $t$, it is visible that all the equations of the previous time steps which contain the term $\theta$, have an effect on that particular output vector. In order to optimize this effect, derivatives of these terms with respect to that particular output vector have to be taken into account with a sum operation.

### 2.2.3 Appearance of vanishing gradient

During the backpropagation phase, in the equation 2.5 the term $\frac{\partial y_t}{\partial h_k}$ enables the calculated error to be carried along the time steps (from $t$ to $k$). In 1994 it is revealed that as the distance between $t$ and $k$ grows, the magnitude of the term $\frac{\partial y_t}{\partial h_k}$ decay exponentially [2]. This situation which is called "Vanishing Gradient" decrease the efficiency of learning the long-term dependencies. In [2], it is concluded that under the assumption of the use of hyperbolic activation functions, either network would not be robust to input noise or would not be efficiently trainable by gradient descent with required long-term dependencies. As an indirect outcome it is also noted that, deep neural networks without recurrences may also be a subject to this problem, due to the fact that an unfolded recurrent neural network is not so different than a deep feedforward neural network with shared weights.

As a solution to this problem different optimization techniques have been offered such as "Simulated Annealing" in [13], "Time-Weighted Pseudo-Newton Optimization" in [2] and [14] or "Discrete Error Propagation" in [2]. Beside these solutions, today an

alternative practice for hidden layers is to use ReLu activation function which is an abbreviation for “Rectified Linear Unit”, instead of using sigmoid or tangent hyperbolic activation functions that map the magnitude of the output between certain intervals. The problem with sigmoid and tangent hyperbolic functions is that they squeeze the output to an interval. Thus, they are only sensitive for the values in this interval. Otherwise output gets saturated if it is out of the range which is $(-1,1)$ for tangent hyperbolic and $(0,1)$ for sigmoid. In [15], it is stated that due to its linear features no gradient vanishing effect depending on the non-linear natures of tanh and sigmoid. Furthermore, as a computational cost of training a network with ReLu is much lower than sigmoid or tanh functions. In contrast to these advantages on preventing vanishing gradient, ReLu brings its own problems, such as not having an upper limit which causes exploding gradient problem or another a phenomenon called “Dying ReLu” which happens frequently when a neuron with ReLU become inactive and returns constantly zero no matter what the inputs are [16]. This insensitivity to the inputs leads erroneous results and insufficient convergence. Many alternative versions of ReLu such as ELU, LReLu or SReLu have been proposed to overcome “Dying ReLu” or other possible problems that may occur [17].

### 2.3 Long-Short Term Memory (LSTM)

As a solution to the problems of Vanilla RNN structure, first version of LSTM has been proposed by Sepp Hochreiter and Jürgen Schmidhuber with the research papers in published in 1995 [18] and 1997 [19]. By the contributions of further research through time, several variants of LSTM proposed. Among the presented variants, the idea that controlling the cell state with gates has been kept for each version. This feature was also a solution for LSTMs to handle the vanishing gradient problem which is described in the previous section.

In the initial version of a LSTM cell [18-19], only input and output gates were controlling the neuron state. Compare to most common versions today, features such as “forget gate”, “unit biases” and “input activation function” were missing. Furthermore, except the gradients of the cells, other recurrent connections were not backpropagated through time. Finally, this initial structure had one specific feature that is not copied at the following versions, which was the way that the gates connected

[19]. As it is explained in [20] all gates were receiving inputs from all the gates from previous time step.

Another significant modification of the LSTMs was the forget gate [21]. In [21] it is stated that continual input streams generally have a requirement to reset its state occasionally, in the tasks which are a compose of multiples subtasks. The mind behind this explanation was to forget some of the carried-out information as it is used and become unnecessary to be hold, which happens frequently in NLP.

One of the other following major contribution which is called "Peepholes Connections" has been added to the architecture of LSTM in 2000 [22]. Peepholes are defined as the connections between an internal state of a neuron at time step $t-1$ and the gates of the neuron $t$. As it is denoted in [22], since there is no direct connection from CEC which is supposed to control, all can be directly observed is the output regarding that LSTM cell. In the case of a closed output gate, it is not possible to carry any essential information belongs to the data. This is the main reason for the use peepholes.

In this thesis we will be using the LSTM structure with the forget gates, not with the peepholes, since it is used as default LSTM structure in the widely used machine learning frameworks such as PyTorch and TensorFlow.

The forward calculation of LSTM is performed as follows, with the given number of LSTM unit N, the number of input unit M, where $x_t$ and $h_t$ denote the variables input and hidden state of the neuron respectively at the time step t:

$$f_t = \sigma\big(W_{xf}x_t + W_{hf}h_{t-1} + b_f\big) \tag{2.7}$$

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{2.8}$$

$$\hat{C}_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.9}$$

$$C_t = f_t \odot c_{t-1} + i_t \odot \hat{C}_t \tag{2.10}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{2.11}$$

$$h_t = o_t \odot \tanh(c_{t-1}) \tag{2.12}$$

The symbols $W_{xf}, W_{xi}, W_{xc}, W_{xo} \in \mathbb{R}^{NxM}$ and $W_{hf}, W_{hi}, W_{hc}, W_{ho} \in \mathbb{R}^{NxN}$ in the equations 2.7-2.11 denote the weights between signals $x, h$ and $f, i, c, o$ which are an abbreviation for forget gate output, input gate output, cell state and output of output gate respectively where the symbols $b_f, b_i, b_c, b_o$ denotes biases. Even if the biases are not used in the first proposal [19], it is also a widely used practice in the following researches and applications. The symbol $\odot$ shows pointwise multiplication. The visualized version of the equations 2.7-2.12 is given in the figure 2.4.

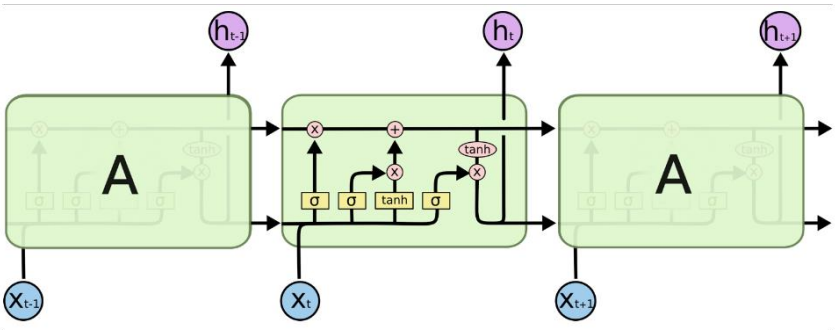


**Figure 2.4 :** Illustration of LSTM [11].

## 2.4 Gated Recurrent Unit (GRU)

Gated Recurrent Unit is another RNN structure which is designed to capture long and short-term dependencies just like LSTM. GRU is proposed by Cho in 2014 [23]. Similarly, to LSTM, GRU cells have internal gates which enables it to regulate the information flow through the network. However, GRU mainly has 2 gates whereas LSTM has 3. Because of its less complex nature, it requires less computational power than LSTM. This gating structure also proposed as a remedy to vanishing gradient problem.

GRU architecture consist of 2 gates: update and reset gates. These gates decide which information to keep and which information to forget. In the mathematical model, the gate calculations are performed as follows:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \tag{2.13}$$

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{2.14}$$

As it can be seen in the equations 2.13 and 2.14, output of the reset and update gates which are denoted with the terms $r_t$ and $z_t$, are calculated using the same formulation with different weight matrices for the time step $t$. Even if the equations are given with this structure in the proposal, it became an alternative practice to add biases to the mathematical model, as it can be seen in [24].

After the evaluation of gate values, these parameters are used in the calculation of next hidden state.

$$\tilde{h}_t = \phi(Wx_t + (r_t \odot Uh_{t-1}) \tag{2.15}$$

$$h_t = z_t h_{t-1} + (1 - z_t)\tilde{h}_t \tag{2.16}$$

Visual representation regarding the mathematical model of GRU is given in the figure 2.5

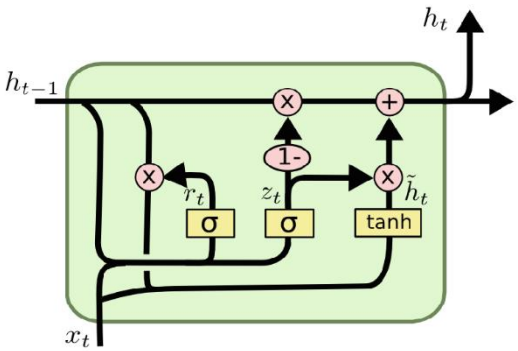


**Figure 2.1 :** Illustration of GRU [11].

The term $\tilde{h}_t$ is used as a candidate output in the equation 2.15 where the symbol $\phi$ denotes the activation function, which is generally taken as $tanh$. The candidate output is calculated by summing the weighted input vector and weighted output vector which is regulated by the reset gate. Afterwards, update gate decides how much of the previous information should be kept and how much of it should be updated with the candidate output $\tilde{h}_t$. This decision-making process is run by the value of the update gate $z_t$ as it can be seen in the equation 2.16.

# 3. RANDOM NEURAL NETWORK

Random Neural Network (RANN) is a mathematical model that has been proposed by Erol Gelenbe in 1989 [6]. It has been declared as a composition of queueing theory and artificial neural networks. It's working principle relies on the circulation of positive and negative spikes with unit amplitude throughout the network.

## 3.1 Mathematical Model

In order to express the model clearly, a supermarket example is widely used in literature. In this example, assume that there is a single queue where customers arrive and leave. The arriving process of customers are regulated by a Poisson process with a rate defined as $\lambda > 0$. In order to express leaving process, a parameter called $r$ has been defined to represent service times which is exponentially distribute and greater than zero. There exist two kind of customers: positive and negative ones. As a positive customer arrives the queue, the customers in the network which is defined as potential of the queue increases by 1. In contrast, arrival of a negative customer creates an inhibition effect on the queue and decreases the potential by 1, if only the potential is already greater than zero. Otherwise, negative customers disappear without any impact on the queue. Beside the arrival of negative customers, departing customers that has been served by the service rate $r$ also have an inhibition effect on the potential if only there is at least one customer in the queue. In other words, departure process is run only with a positive potential. When a customer has been served in the queue $i$, it can either leave the network with a probability of $d_i$ or it can move to another queue $j$ with a probability $p_{ij}$. This probability is a compose of two sub-probabilities $p_{ji}^{+}$ and $p_{ji}^{-}$ where each denotes a possibility of moving as a positive or a negative customer respectively. In a Random Neural Network with $N$ interconnected neuron the relationship between these probabilities can be expressed as it has been shown in the equation 3.1.

$$d_i + \sum_{j=1}^{N} (p_{ij}^{+} + p_{ij}^{-}) = 1 \tag{3.1}$$

Considering the neural network structure, it can be said that the term $d_i$ is equal to 1 only for the output neurons, otherwise it is equal to 0, since neurons in the input and hidden layers do not emit signals to the outside environment. In other words, the term $d_i$ enable output neurons to transmit signals only outside by making the probabilities $p_{ij}^{+}$ and $p_{ij}^{-}$ zero. As a contrast, in the case of input layer, signals can only be received by the outside environment. In addition to the explanations about arrival processed at the paragraph above, the Poisson process rates that regulates these arrivals can be defined with the symbols $\lambda_i^{+}$ and $\lambda_i^{-}$. A common practice about the RANN in the literature is to use the notations $W_{ij}^{+} = r_j p_{ij}^{+}$ and $W_{ij}^{-} = r_j p_{ij}^{-}$ that will help to establish an analogy with the conventional neural networks. In these representations the terms $W_{ij}^{+}$ and $W_{ij}^{-}$ denotes the positive and negative weights between the neurons $i$ and $j$. The difference between the conventional weights and these ones, can be observed from their definition. As it can be seen, $W_{ij}^{+}$ and $W_{ij}^{-}$ are a product of two probabilities. Since both of the components of the product operation are defined as rates which are at least 0, the weights cannot take negative values.

In [7], the stability conditions for a Random Neural Network has been presented. According to theorem in [7], for a neuron $i$, the potential of the neuron which is denoted as $q_i$ can be found with the following expression:

$$Q_i = \frac{T_i^{+}}{r_i + T_i^{-}} \tag{3.2}$$

where the terms $T_i^{+}$ and $T_i^{-}$ denote total positive and negative contribution to neuron $i$ from inside and outside environment respectively and the term $r_i$ denotes the service rate of the particular neuron $i$. In the calculation of this formula, the total positive and negative arrival rates can be evaluated as follows:

$$T_i^{-} = \lambda_i^{-} + \sum_{j=1}^{N} Q_j W_{ij}^{-} \tag{3.3}$$

$$T_i^- = \lambda_i^- + \sum_{j=1}^{N} Q_j W_{ij}^- \tag{3.4}$$

In the special case of input layer, $T_i^+$ and $T_i^-$ are directly equal to $\lambda_i^+$ and $\lambda_i^-$ respectively, since there is no input to this layer from the inside of the network.

The final term, needed in equation 3.2 is $r_i$. Using the manipulations $W_{ij}^+ = r_j p_{ij}^+$ and $W_{ij}^- = r_j p_{ij}^-$ , in the equation 3.2, $r_i$ can be evaluated with the formula:

$$r_i = \frac{1}{1 - d_i} + \sum_{j=1}^{N} (W_{ij}^+ + W_{ij}^-) \tag{3.5}$$

### 3.2 Gradient Based Optimization

Training a neural network requires a forward and a backward pass. Above, generalized forward pass formulas are given with extensive descriptions regarding the logic behind the Random Neural Network. In this part of the chapter 3, gradient based backpropagation algorithm will be covered detailly.

In 1993, first attempt to implement optimization algorithm to Random Neural Network, has been made by Erol Gelenbe. Beside of that, as it is compiled in [25], there also exist, implemented second order optimizations algorithms such as; Gauss-Newton (GN) methods, the Broyden-Fletcher-Goldfarb-Shanno (BFGS), the Davidon, Fletcher and Powell (DFP), the Levenberg-Marquardt (LM) and the LM with Adaptative Momentum (LM-AM). In this thesis, classical gradient descent-based backpropagation algorithm [26] will be covered.

Assume a data set $S = \{(x^k, y^k), k = 1, \dots, K\}$ exists where each vector $x^k$ and $y^k$ represent a subset of $X$ and $Y$. The duty of neural networks in supervised learning, is to create a mapping function, that imitates the relationship between $X$ and $Y$. In order to establish this relationship with the interior parameters of a neural network, all the trainable parameters get updated after each forward pass. The magnitude and the direction of the gradient of a trainable parameter with respect to a loss function, determines the effect of update over that individual trainable parameter. As the network iterate over the data set $S$, it is expected from network to optimize trainable parameters to reach an optimal point that approximation results are close enough to

mimic original relationship between $X$ and $Y$. This general working principle of supervised learning algorithms, is also valid for Random Neural Networks as well.

Suppose a forward pass has been performed for a Random Neural Network with one hidden layer, using $I$ units in input layer, $H$ units in hidden layer and $O$ units in output layer. As an output, a vector $Q$ has been created for each neuron $i$ where $i \in N = I + H + O$. Using the mean squared error loss function, loss gets calculate as follows:

$$L = \frac{1}{2}\sum_{i=1}^{N} d_i \, (q_i - y_i)^2 \tag{3.6}$$

The reason behind the use of the factor $\frac{1}{2}$ is to eliminate the factor of 2 when taking the derivative of the loss function with respect to each output $q_n$. Beside the extra element $\frac{1}{2}$, another extra term is $d_n$ which can be explained as a designator that distinguish output neurons from the rest. As it is noted at section 3.1, $d_n$ is taken as 1 for the output neuron and 0 for the rest, in general practice.

In the case of Random Neural Networks, the trainable parameters are positive and negative weights. As it is explained in [26], the update procedure of trainable parameters runs with the calculation:

$$W_{uv}^{*(k)} = W_{uv}^{*(k-1)} + \delta_{uv}^{*(k)} \tag{3.7}$$

In order to not to write equations twice with the same structure, the term $W_{uv}^{*}$ is used as a representation of the both positive and negative weights at the between neuron $u$ and $v$, where $k$ denotes the iteration number. Another point the notice is the size of the matrix $W_{uv}^{*(k)}$. In [8], it has been defined as $NxN$ matrix that contains all the connections between all the neurons. It means there is no layer-wise discrimination in this most commonly used representation. The term $\delta_{uv}^{*(k)}$ that performs the update is calculated as follows:

$$\delta_{uv}^{*(k)} = -\eta \sum_{i=1}^{N} d_i \frac{\partial L}{\partial q_i^{(k)}} \frac{\partial q_i^{(k)}}{\partial W_{uv}^{*}} \tag{3.8}$$

In the equation 3.8, the term $\eta$ is called the learning rate which takes value in a range [0,1]. Another term $\frac{\partial L}{\partial q_i^{(k)}}$ can be expressed as $\left(q_i^{(k)} - y_i^{(k)}\right)$, when the derivative of the equation 3.6 is taken with respect to $q_i^{(k)}$. Finally, the calculation of the remaining gradient term $\frac{\partial q_i^{(k)}}{\partial W_{uv}^*}$ is actually the main focus of backpropagation phase in the Random Neural Networks.

In order to formulate backpropagation phase, Erol Gelenbe has proposed a formula structure to update positive and negative weight matrices in [8]. As it can be observed in the equations 3.9 and 3.10, there exist two new parameters that have not been explained in the previous chapters.

$$\frac{\partial q_i^{(k)}}{\partial W_{uv}^+} = \gamma_{uv}^+ q_u [I - \Omega]^{-1} \tag{3.9}$$

$$\frac{\partial q_i^{(k)}}{\partial W_{uv}^-} = \gamma_{uv}^- q_u [I - \Omega]^{-1} \tag{3.10}$$

In equations 3.9 and 3.10, the terms $\gamma_{uv}^+$ and $\gamma_{uv}^-$ represent results of the two multiple conditional equalities, as it can be seen in the equations 3.11 and 3.12. The values that these terms can take are actually the results of basic derivatives of neurons $q_u$ and $q_v$ with respect to the nominators and denominators. But as a simplified rule, it is formulated as follows.

$$\gamma_{uv;i}^+ = \begin{cases} -\dfrac{1}{r_i + T_i^-}, & if\ u = i, v \neq i \\ \dfrac{1}{r_i + T_i^-}, & if\ u \neq i, v = i \\ 0, & otherwise \end{cases} \tag{3.11}$$

$$\gamma_{uv;i}^- = \begin{cases} -\dfrac{1 + q_i}{r_i + T_i^-}, & if\ u = i, v = i \\ -\dfrac{1}{r_i + T_i^-}, & if\ u = i, v \neq i \\ -\dfrac{q_i}{r_i + T_i^-}, & if\ u \neq i, v = i \\ 0, & otherwise \end{cases} \tag{3.12}$$

As a beneficial feature of this representation in the equation 3.12, the first condition is $i = u = v$. This equality implies a recurrent connection that a neuron may have with itself, which leads to structure that has potential to be used with a "Recurrent Topology" which will be investigated in further chapters.

Another unexplained term is a matrix which is a subject to a process of taking inverse. In the formal notation it is symbolized with "$W$", but in order to prevent confusion it is shown with $\Omega$, using the notation in [25].

$$\Omega = \frac{W_{ij}^{+} - W_{ij}^{-} q_j}{r_j + T_j^{-}} \tag{3.13}$$

Just like the terms $\gamma_{uv;i}^{+}$ and $\gamma_{uv;i}^{-}$, the idea behind the equation 3.13 also based on the derivatives of neuron $i$ with respect to neuron $j$. As an interpretation of the matrix $\Omega$, its shape is $NxN$. This representation of connections enables the Random Neural Network to perform as a recurrent structure as well as the feedforward one. As useful tool, there is a MATLAB implementation of the calculations regarding forward and backward passes, which is written by Hossam Abdelbaki in 1999 [27].

### 3.3 Feedforward Topology

In the feedforward topology, connections can only exist between the neighbor layers such as, input and hidden, hidden and output, or hidden and hidden in deep learning applications. In such cases, as the neurons are numerated incrementally starting from the first neuron of input layer to the last neuron of output layer, the positive and negative weight matrices would have a shape of an $NxN$ upper triangular matrix. Because of the shape of the weight matrices, the term $\Omega$ also have this upper triangular look, which facilitates the inverse operation in the backpropagation phase. These situations can be clearly observed in the implementation of Hossam Abdelbaki. The effect of numeration process, the weight matrices and the parameters in the backpropagation phase can be examined, to have a better understanding about the interior dynamics of the Random Neural Network. Unfortunately, this toolbox has no quick option to increase the number of hidden layers, which makes it not suitable for deep structures.

In the backpropagation there also exist another method to calculate derivatives, which is a simple implementation of classical derivative rules. Consider a Random Neural Network with $N$ hidden layer. Thus, there exist $N-1$ weight matrices between the layers. As the network yield an output from the layer $N$, the loss function creates a loss value to update the values of the weights. Since the function is trivial in terms of explaining the process, it is just denoted as a generic function $L$ which takes the output of the network and real output value as inputs. Notice that $q_N$ and $y$ are vectors with the same size.

$$L^k = L(q_N, y) \tag{3.14}$$

For each weight matrices $n$ in the $k^{th}$ iteration, the delta update rule [26] is applied as follows:

$$W_n^{*(k+1)} = W_n^{*(k)} + \delta_n^{*(k)} \tag{3.15}$$

The term $\delta^{*(k)}$ in equation 3.15 is calculated with the equation 3.16.

$$\delta^{*(k)} = -\eta \frac{\partial L^k}{\partial q_N} \frac{\partial q_N}{\partial W_n^*} \tag{3.16}$$

Normally in the feedforward neural networks, a weight can only affect the next layers output. In the case of RANN, considering the equation 3.2, a weight matrix affects the both of the layers that it connects. This situation can be seen in the equation 3.17. As a written expression, if the layers and weights have been numbered starting from the same number, the gradient of a weight matrix with the index $n$, will be depending on the layers $q_{n+1}$ and $q_n$.

$$\frac{\partial q_N}{\partial W_n^*} = \frac{\partial q_N}{\partial q_{n+1}} \left( \frac{\partial q_{n+1}}{\partial W_n^*} + \frac{\partial q_{n+1}}{\partial q_n} \frac{\partial q_n}{\partial W_n^*} \right) \tag{3.17}$$

The last unexplained term in order to perform the calculation 3.15 is the term $\frac{\partial q_N}{\partial q_{n+1}}$. As it can be seen in the equation 3.18, this term can be evaluated using the chain rule:

$$\frac{\partial q_N}{\partial q_n} = \prod_{N > j \geq n} \frac{\partial q_{j+1}}{\partial q_j} \tag{3.18}$$

### 3.4 Recurrent Topology

The recurrent topology of Random Neural Network is proposed with the backpropagation algorithm at chapter 3.2. The name of the article was “Leaning in The Recurrent Random Neural Network”. What is meant by the word “Recurrent” was actually the capability of establishing connections among the neurons without any restriction of layer. This definition of “Recurrent” is completely different than the meaning of “Recurrent” for the Simple RNN, LSTM or GRU. The reason for these 3 models to be called as “Recurrent Neural Networks” is actually about their ability of having dynamical connections that can be unfold through time. The mathematical models and working principles regarding these 3 networks have been detailly covered in the section 2. Considering this difference in the approach of the word “Recurrent”, Recurrent Random Neural Network (RERANN) is definitely not a part of this “Recurrent Neural Networks Family” in the sense of recurrence connections expanding through time. Furthermore, considering the nature of RERANN, it is also not a similar model to ELMAN-JORDAN type networks neither unless some special modifications are performed. In contrast to examples above, it can be used in the associative memory applications, due to its ability to be used as a Hopfield Network.

# 4. QUEUEING RECURRENT NEURAL NETWORK

The aim of previous sections was to provide necessary background information for a new RNN structure which is named Queueing Recurrent Neural Network (QRNN). It is simply a compose of conventional Recurrent Neural Networks and Random Neural Networks. This nature of QRNN enables it to be used in every kind of applications that conventional RNNs are used.

## 4.1 Mathematical Model

Working principle of QRNN is the same as conventional RNNs. Everything that has been written in the beginning of section 2 is also valid for QRNN as well, which is the part that distinguishes it from Recurrent Random Neural Network. Furthermore, the forward calculations are run by using the formulas of RANN including some extra adaptions without changing the queuing nature of the neurons.

The architecture of QRNN with one hidden layer, consist of an input layer with $I$ units, a hidden layer with $H$ neuron and an output layer with $O$ neuron. Normally it is a common practice for all neural network applications to normalize or standardize the data in the preprocessing phase. Since rate of a Poisson process can only take values in the interval $[0,1]$, after the preprocessing phase it makes sense the scale the input data to this interval and give it to the term $\lambda_{i_t}^{+}$ which is the input data of the neurons in the input layer. In this scenario, the term $\lambda_{i_t}^{-}$ actually become useless.

The output formula of the neurons in the input layer is quite similar to the output formula of any arbitrary neuron in RANN. A small difference is the index $t$, which denotes the change through the time. For the rest of the thesis, each term with an index $t$ means that this variable changes through the time and each term without the index means, this variable does not change through the time.

$$q_{i_t} = \frac{\lambda_{i_t}^{+}}{\sum_{j=1}^{H}(W_{ih_j}^{+} + W_{ih_j}^{-}) + \lambda_{i_t}^{-}} \tag{4.1}$$

In the equation 4.1 total positive and negative inputs are represented directly with terms $\lambda_{i_t}^{+}$ and $\lambda_{i_t}^{-}$, since there is no connection for the input neurons with any others within the network. As a useful practice, the rate of the neuron $r_i$ is written explicitly in the equation 4.1 and the following ones. The reason for that, is mainly facilitating the comprehension of the terms in the backpropagation phase. Notice that the term $r_i$ do not have the index $t$. Since all the weights are the same through the time, in other words not time variant, the input layers from different time steps shares the same service rate $r_i$ where $W_{ih}^{+}$ and $W_{ih}^{-}$ denotes the positive and negative weight matrices respectively between input and hidden layers. Considering the fact that the shape of the matrix $W_{ih}^{+}$ is $IxH$, the term $W_{ih_j}^{+}$ actually shows a row of this two dimensional matrix which corresponds to a vector that contains all the connections that an input neuron have with the hidden neurons at the same time step. Additionally, notice that actually every term in the equation 4.1 is a vector. Length of the all the vectors $\lambda_{i_t}^{+}$, $\lambda_{i_t}^{-}$, $q_{i_t}$, $W_{ih_j}^{+}$ and $W_{ih_j}^{-}$ are the same which is $I$.

In the case hidden layers, there exits multiple slightly different equations. This situation is caused by the dynamic nature of RNNs. For each architecture in the figure 2.2, the connections of hidden layers change depending on not only the application, but also the positions of this layer in time. Since the rate parameter of each neuron is calculated based on the outputs of that particular neuron, dynamic structure of RNNs changes the fully explicit version of the formula:

$$q_{h_t} = \frac{T_{h_t}^{+}}{r_h + T_{h_t}^{-}} \tag{4.2}$$

$$T_{h_t}^{+} = \sum_{j=1}^{I}(q_{i_t} W_{ih_j}^{+}) + \sum_{j=1}^{H}(q_{h_{t-1}} W_{hh_j}^{+}) \tag{4.3}$$

$$T_{h_t}^{-} = \sum_{j=1}^{I}(q_{i_t} W_{ih_j}^{-}) + \sum_{j=1}^{H}(q_{h_{t-1}} W_{hh_j}^{-}) \tag{4.4}$$

$$r_h = \sum_{j=1}^{H} (W_{hh_j}^{+} + W_{hh_j}^{-}) + \sum_{j=1}^{O} (W_{ho_j}^{+} + W_{ho_j}^{-}) \tag{4.5}$$

In the equation 4.2 the most general form of hidden layer formulation is presented. Depending the architecture of the model (among the architectures in figure 2.2), some of the terms may be directly 0. Remember the example in section 2.1 and consider the connections of the hidden layers from the aspect of this specific example. In this architecture, a hidden layer $t$ takes input from the input layer $t$ and hidden layer $t-1$. This statement is valid for each hidden layer in time. Thus, total positive and negative inputs of hidden layers $T_{h_t}^{+}$ and $T_{h_t}^{-}$ are the same end equal to equations 4.3 and 4.4 for all the time steps. On the other hand, a hidden layer t gives output only to the hidden layer $t+1$ unless $t$ is the last time step. Because at the last time step, the output of the hidden layer is given as an input to the output layer. It can be seen in the figure 2.1. Consequently, rate of last hidden layer is equals to $\sum_{j=1}^{O}(W_{ho_j}^{+} + W_{ho_j}^{-})$ while the rest of them equal to $\sum_{j=1}^{H}(W_{hh_j}^{+} + W_{hh_j}^{-})$. This dynamic formulation increases the complexity of not only the forward phase, but also the backward one.

In the case of output layer, since there is no connection between output layers through the time and only input comes from the hidden layers of the same time step, the equation of output layer is relatively simpler compare to hidden layers.

$$q_{o_t} = \frac{\sum_{j=1}^{H}(q_{h_t} W_{ho_j}^{+})}{r_o + \sum_{j=1}^{H}(q_{h_t} W_{ho_j}^{-})} \tag{4.6}$$

Notice that the rate parameter of output layer, $r_o$, is not given explicitly. Considering the fact that the term $r_o$ is about the output connections of a neuron, for the output layer there is an ambiguous situation. It is actually a free parameter. Thus, it can be given arbitrarily. But the experiments show that in some cases, the static value of $r_o$ actually affect the convergence which will be discussed in the section 5. This effect is not spectacular, but could be vital for an application where a small change matter.

### 4.2 Optimization

This section contains information about backpropagation through time algorithm and how it is used to train QRNN. In the section 3.2 backpropagation algorithm of

RERANN which is also valid for RANN, has been presented by using the formulation in [8]. This backpropagation algorithm was actually an interpretation of simple gradient descent algorithm which is not suitable for training RNNs. Besides, in the section 3.4 the difference between RNNs and RERANN has been discussed from the aspect of internal structure and connections. Considering the fact that, QRNN is an algorithm which is based on general working principle of RNNs, its backpropagation mechanism is quite similar to RNNs and also different from RERANN or RANN. The implementation of BPTT algorithm to QRNN is quite similar to Simple RNN. However, the generic formulations of BPTT for QRNN is a bit more complex due to the number of trainable parameters of QRNN model.

The following expressions are derived for a simple QRNN with one hidden layer. For the sake of simplicity, in the following equations, some abbreviations and new notations are used. The new of denotations are as follows:

- All the positive and negative weight matrices: $W^*$
- Positive and negative weight matrices together: $W_{ih}^*$, $W_{hh}^*$, $W_{ho}^*$
- Nominator of the output formula of a neuron (which is $(T_{*_t}^+)$): $N_{i_t}$, $N_{h_t}$, $N_{o_t}$
- Denominator of the output formula of a neuron (which is $(r_*+T_{*_t}^-)$): $D_{i_t}$, $D_{h_t}$, $D_{o_t}$
- Input-Output pair at iteration k: $(x_k, y_k)$

In order to update weights, "The Generalized Delta Rule" [26] will be used as it has been used in the previous backpropagation calculations before this section.

$$W^{*(k+1)} = W^{*(k)} + \delta^{*(k)} \tag{4.7}$$

First step is to calculate loss by comparing the network output with the real one. Loss function can be chosen arbitrarily according to the application. Since the method is trivial for the rest of the BPTT process it will be denoted with the vector $L_t$ as a function of vectors $q_{o_t}$ and $y_t$ where the length of the each vector is the same and equal to the length of the output layer which is $O$.

$$\frac{\partial q_{o_t}}{\partial W_{ho}^*} = \frac{\partial q_{o_t}}{\partial W_{ho}^*} + \frac{\partial q_{o_t}}{\partial q_{h_t}} \frac{\partial q_{h_t}}{\partial W_{ho}^*} \tag{4.8}$$

As it can be seen in the forward calculation equations, the weights are a variable for both of the layers that they are located, unlike the rest of the RNNs. In the other RNN structures the weights are only the variable for the target layer that receives input from the other, not the layers that gives their output. From this point of view, QRNN has a unique position in RNNs because every change in the weight matrices, affects both of the layer that shares that weight matrix. Considering the general formulation in equation 3.2 negative weight matrices create inhibition effect for both of the layers that shares this matrix. In contrast, positive weight matrices create excitation effect on the layer that receive signal and inhibition effect for the layer that sends the signal. This characteristic feature enables QRNN, RERANN and RANN to learn highly non-linear correlations between inputs and outputs.

The update rule of $W_{hh}^*$ which is given in the equations 4.9 and 4.10, is relatively more complex compare to the update rule of $W_{ho}^*$

$$\frac{\partial q_{o_t}}{\partial W_{hh}^*} = \frac{\partial q_{o_t}}{\partial q_{h_t}} \sum_{m=1}^{t} \frac{\partial q_{h_t}}{\partial q_{h_m}} \frac{\partial q_{h_m}}{\partial W_{hh}^*} \tag{4.9}$$

$$\frac{\partial q_{h_t}}{\partial q_{h_m}} = \prod_{t \geq j \geq m} \frac{\partial q_{h_j}}{\partial q_{h_{j-1}}} \tag{4.10}$$

The weight matrices $W_{hh}^*$ and $W_{ih}^*$ are get updated by the sum of the gradient for each time step output, which is calculated by iterating backward from the time step $t$ to the time step 1 using the counter $m$. In other words, there are two recursive "For Loops" where the range of inner one is determined by the value of the outer one. In this case, counter of the outer one is $t$ with the upper limit $T$ and the counter of the inner one is $m$ with the upper limit $t$. This recursive structure is valid for all the RNNs since they all use BPTT as well.

In the case of $W_{ih}^*$, in terms of number of parameters, the BPTT becomes more complex than $W_{hh}^*$, since it affects the outputs of both input and hidden layers.

$$\frac{\partial q_{o_t}}{\partial W_{ih}^*} = \frac{\partial q_{o_t}}{\partial q_{h_t}} \sum_{m=1}^{t} \frac{\partial q_{h_t}}{\partial q_{h_m}} \left( \frac{\partial q_{h_m}}{\partial W_{ih}^*} + \frac{\partial q_{h_m}}{\partial q_{i_m}} \frac{\partial q_{i_m}}{\partial W_{ih}^*} \right) \tag{4.11}$$

In the equation 2.6, it is stated that the term $\frac{\partial h_i}{\partial h_{i-1}}$ is evaluated by the calculation of $W_{hh}^T diag(\sigma'(h_{i-1}))$. Since in Simple RNN, the outputs of neurons are calculated as a sum of product form, the derivative of a neuron with respect to other one can be represented by using only the product operator. In the case of QRNN, the weights and the other terms can be located in the nominator or the denominator of the equation 3.2. For some cases they appear in both. This situation allows the same weights to change the output of a neuron positively and negatively at the same time. Because of this highly non-linear formulation, the gradients between layers or neurons are relatively complex and long expressions compare to Simple RNN. If the equation 2.6, is wanted to be constructed for QRNN, the formulation would be shaped as follows.

$$\begin{aligned} \frac{\partial q_{h_j}}{\partial q_{h_{j-1}}} &= \frac{\partial q_{h_j}}{\partial N_{h_j}} \frac{\partial N_{h_j}}{\partial q_{h_{j-1}}} + \frac{\partial q_{h_j}}{\partial D_{h_j}} \frac{\partial D_{h_j}}{\partial q_{h_{j-1}}} \\ &= \frac{1}{D_{h_j}} W_{hh}^+ - \frac{N_{h_j}}{(D_{h_j})^2} W_{hh}^- \\ &= \frac{W_{hh}^+ - q_{h_j} W_{hh}^-}{D_{h_j}} \end{aligned} \tag{4.12}$$

$$\begin{aligned} \frac{\partial q_{\ell_j}}{\partial q_{\ell-1_j}} &= \frac{\partial q_{\ell_j}}{\partial N_{\ell_j}} \frac{\partial N_{\ell_j}}{\partial q_{\ell-1_j}} + \frac{\partial q_{\ell_j}}{\partial D_{\ell_j}} \frac{\partial D_{\ell_j}}{\partial q_{\ell-1_j}} \\ &= \frac{1}{D_{\ell_j}} W_{\ell,\ell-1}^+ - \frac{N_{\ell_j}}{(D_{\ell_j})^2} W_{\ell,\ell-1}^- \\ &= \frac{W_{\ell,\ell-1}^+ - q_{\ell_j} W_{\ell,\ell-1}^-}{D_{\ell_j}} \end{aligned} \tag{4.13}$$

Even if the form and evaluation method of formulations are quite similar, the equations that evaluate the gradients between layers are shown with two different but also similar representation. The reason behind this notation is to emphasize that there exists two kind of direction in the process of gradient calculation. Considering the figure 2.1, one of direction is the "$time$" axis. The formulation of gradient in the axis of "$time$" is given in the equation 4.12. For the other axis which is "$layer$", the formulation of

gradient is given in the equation 4.13. Since the calculations between time steps are only performed for hidden layers in time, for the term $\partial q_{h_j}$ the index $h$ is preserved during the calculation while the index $j$ is decremented by 1 which denotes the hidden layer $j-1$. On the other hand, since the equation 4.13 shows the gradient calculation for the axis "$layer$" the time step $j$ preserved and layer index $\ell$ is decremented by 1 in order to obtain the gradient.

# 5. EMPIRICAL EVALUATION

This chapter contains empirical evaluations regarding the performance comparison of QRNN with other previously mentioned Recurrent Neural Networks and Random Neural Network. In order to measure the performance, multiple time series datasets with varying characteristic features are chosen to obtain a generalized performance evaluation within the competitor structures. Besides, another parameter for machine learning techniques is the optimization algorithms. Thus, the comparison results will also be given with the different optimization algorithms.

## 5.1 Datasets

Regression for time series is a problem that is highly dependent to the characteristics of the datasets. Especially for a time series dataset obtained from the real world may have different characteristics. While the datasets such as stock market prices can be considered as highly non-linear, the datasets such as temperature, traffic volume, human occupancy for a time period may contain a repeating pattern like a sinusoidal wave form or a monotonic behavior with an increasing or decreasing trend. The latter one may also contain seasonality effects.

In this comparison 4 datasets which are collected from the real world, with different characteristics are used to evaluate performance of the networks. Two of them are used as a univariate and 2 of them are used as multivariate to obtain more accurate and generalized results.

### 5.1.1 Preprocessing phase of google stok price dataset

As it is seemed in the figure 5.1, Google stock price data can be characterized as highly non-linear and non-periodic dataset. Google stock price data is taken from a public Kaggle dataset [28]. It contains 6 columns:

- Date
- Open
- High
- Low
- Close
- Volume

Each sample is given with a date with one day period of time. In Kaggle two separate CSV files have been presented to be used in training and testing. Training file contains data starting from the January 3rd of 2012 until the December 30th of 2016. Test file contains data starting from the January 3rd of 2017 until the January 23rd of 2017. Totally there exist 1278 sample with 1258 in the training set and 20 in the test set, without any missing data point.

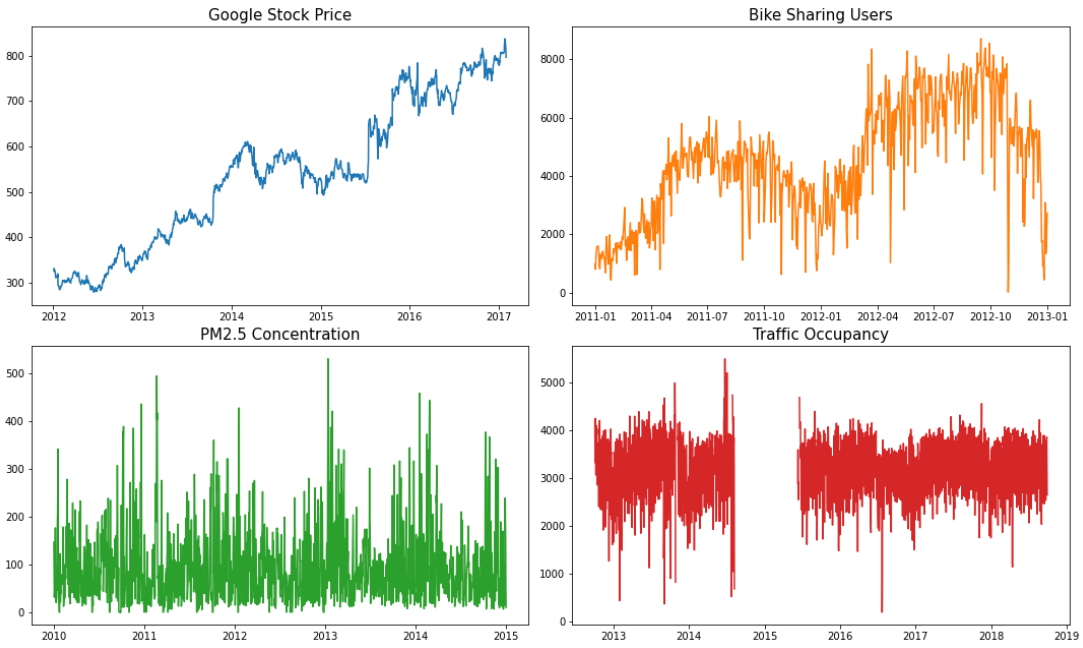


**Figure 5.1 :** The look of raw datasets

In the experiments, dataset is used as it has been given, so it is not resampled in another period of time. Thus %98 of the univariate data is used in training phase and the %2 of the data is used in testing phase. Additionally, only the column “Open” which denotes the opening stock prices are used in the evaluation.

In terms of regression problem, the goal of this application is to forecast the next outputs based on the previous inputs. Thus, the aim is determined as forecasting the opening stock price of the next day by considering the last opening stock prices of last

60 days. It implies that the Recurrent Neural Networks will takes opening stock prices of the roughly last 2 months as an input and yields the stock price of the next day as an output. Since it is considered as a univariate dataset, each Recurrent Neural Network will contain 1 hidden neuron in input layer and the unfolded length of the network through time is 60 time steps. The output of the network will be produced by the output layer of the network with 1 unit at $60^{th}$ time step.

After the dataset is converted into a proper shape, it is also normalized before it has been to be given as an input to network.

### 5.1.2 Preprocessing phase of bike sharing users dataset

As a similarity, Bike sharing dataset has a highly nonlinear characteristic just like Google stock prices. However characteristic of the Bike Sharing dataset cannot be categorized as an increasing monotonic series unlike the stock price of the Google Bike. Besides, the seasonality effect can be seen clearly since the number of users increase towards to the summer and decrease towards to the winter. Despite of the seasonality, general look of the data consists of not expected steep changes which creates non-linearities during the given time period.

Bike sharing dataset is taken from UCI Machine Learning Repository [29]. It provides number of users regarding the “Capital Bikeshare System”. Beside of the number of users, different environmental parameters such as

- Date
- Season
- Year
- Month
- Holiday
- Weekday
- Working day
- Weather Situation
- Normalized temperature
- Normalized feeling temperature

- Normalized humidity
- Normalized windspeed
- Casual: Number of casual customers
- Registered: Number of registered customers
- Total Customer: Number of total customers

Two different version of the raw data is provided as a daily resampled dataset and an hourly resampled dataset.

In the performance evaluation the daily sampled dataset is used with total 731 sample. In the daily sampled dataset, there exist no missing value. Despite of the fact that it can be used as multivariate, in this application only the "cnt" column which denotes total number of users has been used and a univariate regression application has been performed. The %85 of the dataset starting from the first sample, has been used as a training set and the rest of the data which is %15 of the total has been used as a test set.

In the forecasting application, similar to the regression that has been performed for Google stock price, previous 60 time step has been used to predict the number of users of the next day. The input layer has only one unit and the unfolded length of the network through time is 60 time step. The output of the network will be produced by the output layer of the network with 1 unit at $60^{th}$ time step given just like the it has been done for Google stock price dataset. After the data has been reshaped into a proper form to be given to the networks, it has been normalized and preprocessing phase has been completed.

### 5.1.3 Preprocessing phase of PM2.5 concentration dataset

PM2.5 concentration dataset is taken from UCI Machine Learning Repository [30]. The data has been collected hourly from the US Embassy in Beijing. The meteorological data has been obtained from Beijing Capital International Airport. It contains the columns:

- PM2.5 Concentration: in $\mu g/m^3$
- Dew Point
- Temperature

- Pressure: Air pressure in hPa
- CBWD: Combined wind direction
- LWS: Cumulated wind speed in m/s
- LS: Cumulated hours of snow
- LR: Cumulated hours of rain

In the dataset, there exist missing values in some columns.

In the PM2.5 concentration dataset, there exist 43824 sample. The time span of the dataset is between the dates 01.01.2010 and 31.12.2014. In the performance evaluation all the samples are not used because of the computation time concerns since the dataset will has been used as multivariate. The samples regarding the first 365 days which means 365x24 = 8760 samples have been taken as training set and the hourly sampled PM 2.5 concentrations regarding the next of 30 days which correspond to the January 2011, has been chosen as test set. According to this distribution of the data to training and test set %92 of the chosen data has been used in training and %8 of it has been used in test.

In the forecasting application, unlike the use of previous datasets, PM 2.5 dataset has been used as multivariate. All the columns except wind direction is numerical values so one-hot encoding algorithm is performed to render this text values in to categorical data. In total there exist 4 kind of direction. After the one-hot encoding operation total 11 columns have been reshaped into the required form to be trained. The aim is chosen as to forecast the PM 2.5 concentration of the next hour by considering the concentration data of previous 24 hour. Considering this configuration, in the application the input layer of the network will contain 11 unit and the unfolded length of the network through time is 24 time step. At the end of the forward calculation, the size of the output layer at the $24^{th}$ time step is 1, since only the PM 2.5 concentration has been tried to be predicted. As a final step of preprocessing the data has been normalized.

### 5.1.4 Preprocessing phase of traffic volume dataset

Traffic volume dataset is taken from UCI Machine Learning Repository [31]. The data has been collected hourly from Interstate 94 Westbound in the Minneapolis-St Paul,

MN. The weather features and holidays also exist as a column in the dataset. The columns are as follows:

- Holiday
- Temperature
- Rain: Numeric amount of rain in mm that occurred in an hour),
- Snow: Numeric amount of snow in mm that occurred in an hour),
- Clouds: Numeric percentage of cloud cover
- Weather Main: Categorical text value
- Weather Description: Detailed categorical text value
- Date Time: Date time in according to local time zone
- Traffic Volume: Numeric hourly reported I-94 westbound traffic volume

In the Traffic volume dataset, there exist 48204 sample starting from the date 02.10.2012 until the date 30.09.2018. But as it is seemed in the Figure 5.1, there exist a huge gap between the August 2014 and the June 2015. Beside of that, there also exist data losses for smaller intervals which have been forward filled to the preprocessing phase.

The traffic volume data is a multivariate dataset, but not all the columns are numerical. “Weather Main” and “Weather Description” columns contain 11 and 38 different textural values. In the case of a one-hot encoding operation the size of the input sequence will increase significantly. One the other hand, word embedding which is a widely used method in natural language processing, can be used to decrease the number of input node, but also the complexity of the application would be increased inevitably. In order to prevent this situation and decrease the computational time, these two columns are not used. In total, rest of the 6 different numerical columns are used in the calculations.

In the preprocessing phase, hourly given traffic volume data resampled daily and first 720 days beginning from the date 30.06.2015 have been chosen as training data. The rest of the data which corresponds to 346 days are chosen as test data. The goal is determined as to predict the traffic volume of the next day by considering the previous 60 days. Thus, the unfolded length of the Recurrent Neural Network has been used is

60 time step and the length of the input layer is 6. The output of the network is yielded from the output layer with 1 unit at the 60$^{th}$ time step.

## 5.2 Optimization Algorithms

In this section 10 different optimization algorithms that have been used to train the network, are briefly introduced. In the notations of this section, the notation in [32] is used.

### 5.2.1 Gradient descent

Gradient descent is one of the most common optimization algorithms in machine learning literature. It is also the simplest one compare to the others that will be covered in this section. It has 3 main variants as Stochastic Gradient Descent, Batch Gradient Descent and Mini-Batch Gradient Descent. The main difference between them can be expressed as follows:

- **Stochastic Gradient Descent:** After each training sample, backpropagation is performed.
- **Batch Gradient Descent:** After each epoch, backpropagation is performed.
- **Mini-Batch Gradient Descent:** After a number of training sample, backpropagation is performed.

In the empirical evaluation Mini Batch Gradient Descent has been used. The mathematical formula for $n$ batch is given as follows:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i:\, i+n)}; y^{(i:\, i+n)}) \tag{5.1}$$

$\theta$, $\eta$, and $\nabla_\theta J(\theta)$ denote a general term regarding all the weights, learning rate and the objective function respectively.

### 5.2.2 Momentum

Momentum is an augmented version of gradient descent with another term called momentum. The mathematical formula can be expressed as follows:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{5.2}$$

$$\theta = \theta - v_t \tag{5.3}$$

The logic behind it is to increase the speed of the gradient in steep surfaces of loss function and obtain a faster and better convergence. The term $\gamma$ is a factor that balance the previous $v_{t-1}$ value and the current gradient. It is generally chosen as 0.9.

### 5.2.3 Nesterov Accelerated Gradient

Nesterov accelerated gradient [33] is introduced by Yurii Nesterov in 1983.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{5.4}$$

$$\theta = \theta - v_t \tag{5.5}$$

As it can be seen in the equation 5.4 the forward propagation is performed with predicted weights $\theta - \gamma v_{t-1}$. Rest of the algorithm follows the formulation of momentum algorithm. Using predicted weights aims to slow down the gradient and prevent oscillations around local or global minimum points.

### 5.2.4 Adagrad

The previous methods are used a static learning rate and adds extra elements into calculations to increase convergence. Adagrad algorithm [34] offers a chance to adapt the learning rate for each time the backpropagation algorithm is performed. Mathematical equations are given as follows by using the notation in [32].

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{5.6}$$

$$\theta = \theta - v_t \tag{5.7}$$

As a difference of Adagrad from the previously mentioned algorithms, Adagrad uses different learning rates for each weight $i$ at iteration $t$. The term $g_{t,i}$ and $G_{t,ii}$ corresponds to gradient vector at iteration $t$ and the sum of square of each gradient respectively. In order to prevent zero division by zero error, the term $\epsilon$ which is generally chosen a relatively in significant and small value is used in the denominator.

The problem with the Adagrad algorithm is the decreasing magnitude of the gradient due to the sum operator in the term $G_{t,ii}$. As the number of iterations increases, the

term $G_{t,ii}$ increase and the learning rate becomes insignificant. In other words, networks lose its ability to learn from data because of the insufficient magnitude of the learning rate.

### 5.2.5 Adadelta

Adadelta [35] aims to solve the problem that Adagrad faces. In order to do that it does not sums all the previous gradients but uses a memory efficient version of moving average algorithm.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{5.8}$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma) \Delta\theta_t^2 \tag{5.9}$$

$$\Delta\theta_t = \frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{5.10}$$

$$\theta_{t+1} = \theta_t - \Delta\theta_t \tag{5.11}$$

The term $\gamma$ which is generally taken as 0.9 enables to use moving average algorithm without keeping each previous value or counting the iteration number. $E[g^2]_t$ and $E[\Delta\theta^2]_t$ denote the moving average value or gradient and the update value of the weights for iteration t. As it can be seen at equation 5.10 there is no need to assign a learning rate for Adadelta algorithm.

### 5.2.6 RMSprop

RMSprop algorithm [36] is introduced as another alternative solution to the drawbacks of Adagrad.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{5.12}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{5.13}$$

RMSprop can be declared as a version of Adadelta with a starting learning rate or a version of Adagrad with moving average operation instead of sum of the gradients.

The $\gamma$ value is generally taken as 0.9 as it is stated in the previously mentioned algorithms.

### 5.2.7 Adam

Adam [37] provides another alternative adaptive algorithm to optimize weights.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{5.14}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{5.15}$$

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{5.16}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{5.17}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \widehat{m}_t \tag{5.18}$$

Adam algorithm not only uses the square of the gradient but also the gradient itself with moving average. In the first iterations the terms $m_t$ and $v_t$ take values close to zero. In order to prevent this inefficiency in the beginning, the operations in the equations 5.16 and 5.17 are performed. As the number of iterations increases the term $\beta_1^t$ and $\beta_2^t$ will approach to zero. Thus, the terms $\widehat{m}_t$ and $\hat{v}_t$ will be almost equal to $m_t$ and $v_t$ respectively.

### 5.2.8 Adamax

Adamax algorithm uses the $\ell_\infty$ norm of $g_t$ instead of $\ell_2$ as it has been used in Adam algorithm. Formulation of Adamax can be evaluated by replacing the term $\sqrt{\hat{v}_t} + \epsilon$ by $v_t$ as it is shown at equation 5.22. In cases where the first gradient is somehow 0, keeping the term $\epsilon$ might be a good practice in implementation.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)|g_t| \tag{5.19}$$

$$v_t = \max(\beta_2 v_{t-1}, |g_t|) \tag{5.20}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{5.21}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{v_t}\hat{m}_t \tag{5.22}$$

### 5.2.9 Nadam

Nadam algorithm is a compose of Adam and Nesterov Accelerated Gradient algorithms. The simplified version of the equation can be given as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\left(\beta_1\hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right) \tag{5.23}$$

It updates the weight by using the gradient that is calculated according to predicted weights. The equation 5.23 is presented as a simplified version of Nadam. The detailed explanation regarding the formula 5.23 can be found in [32].

### 5.2.10 AMSGrad

AMSGrad algorithm prevents the learning rate to vanish by using the "max" operator which is declared as the main drawback of Adagrad.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \tag{5.24}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \tag{5.25}$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t) \tag{5.26}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t \tag{5.27}$$

Furthermore, it also provides an alternative approach to adaptive algorithms that use moving average. Since the memory of the algorithms with moving average is limited, there exists a possibility for the gradient to forget some required informations for convergence. As it can be seen in the equation 5.26. AMSGrad algorithm takes the max value of the $\hat{v}_{t-1}$ and $v_t$. As a similarity with the other adaptive methods, learning rates does not increase in AMSGrad. But it does not approach to zero quickly like

Adagrad or it does not use moving average operation which acts like a short-term memory.

### 5.3 Results

This section covers the empirical evaluation results for QRNN by presenting comparisons with the other Recurrent Neural Networks using the previously mentioned optimization algorithms.

#### 5.3.1 QRNN with different optimization algorithms

In the testing process QRNN is tested with 4 datasets with different characteristics. For each dataset 10 different optimization algorithm which have been introduced in the section above have been used. Unlike the rest of the Recurrent Neural Networks, QRNN 1 extra hyper parameter that needs to be tuned. This parameter is called rate. Normally the rate of each neuron is calculated according to the equation 4.5 except the output neurons. Since there is no connection from output neurons to the inside of the network, the firing rate of the output neurons are simply a free parameter. No matter what the value of the firing rate regarding the output neurons might be, QRNN has an ability to adapt itself and converge. But in some case, it has experienced that the value of firing rate in the output neurons may accelerate or slow down the convergence process. For the experiments in this section, the firing rate of output neurons are tuned to a value that works efficiently.

##### 5.3.1.1 Google stock price

In the figure 5.2 raw data regarding to the change of the training loss has been given. The hyperparameters for the training section is given in the table 5.1.

**Table 5.1 :** Hyperparameters of QRNN for Google stock price dataset.

| Number of Hidden Layers | Number of Hidden Neurons | Batch Number | Number of Epoch | Learning Rate | Firing Rate of Output Neurons | Weight Scaler |
|---|---|---|---|---|---|---|
| 1 | 50 | 32 | 50 | 0.01 | 0.25 | 0.05 |

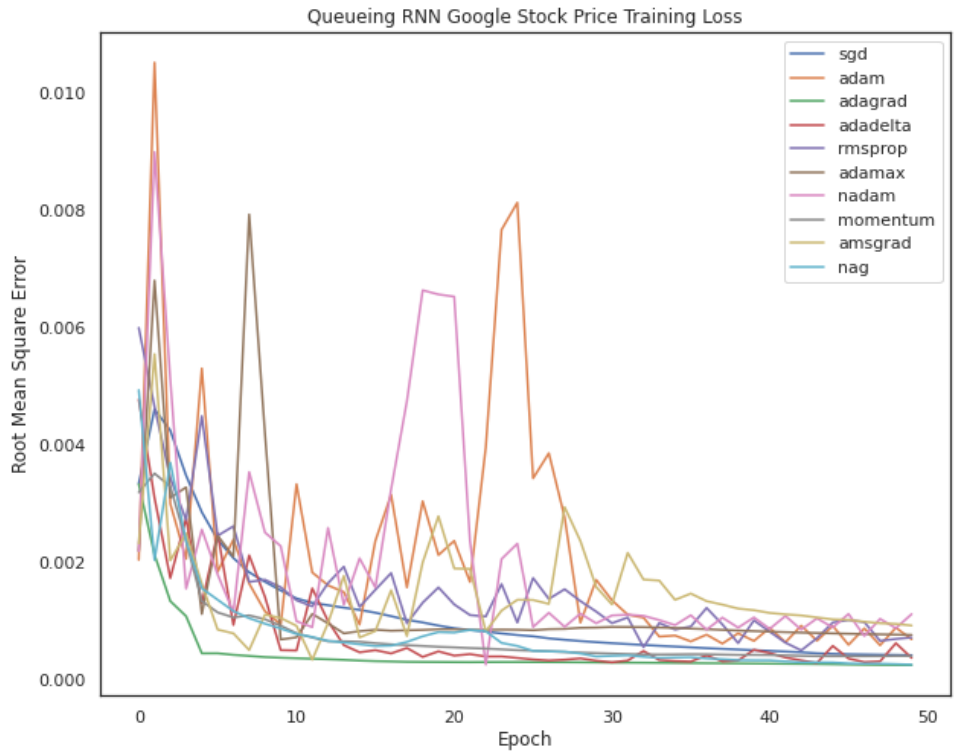


**Figure 5.2 :** QRNN Google stock price training loss.

As the figure 5.2 is examined, for most of the optimization algorithms the oscillating behavior can be seen. Especially for Nadam, Adam and Adamax, there exist a steep increase and decrease between $10^{th}$ and $30^{th}$ epochs. Considering the highly non-linear nature of Google stock price dataset, this fluctuating loss lines can be explained with the characteristic of the dataset. Another point to be look at is the line of Adagrad. As it is seemed in the figure 5.2, the loss line of Adagrad approaches zero very quickly and remain its position after around the $5^{th}$ epoch, due to the vanishing learning rate.

#### 5.3.1.2 Bike sharing users

n Number of bikes sharing users is another univariate dataset. The hyperparameters are given in the table 5.2.

**Table 5.2 :** Hyperparameters of QRNN for bike sharing users dataset.

| Number of Hidden Layers | Number of Hidden Neurons | Batch Number | Number of Epoch | Learning Rate | Firing Rate of Output Neurons | Weight Scaler |
|---|---|---|---|---|---|---|
| 1 | 50 | 32 | 50 | 0.01 | 0.1 | 0.05 |

In the figure 5.3 change of training loss for each optimization algorithm is given. The spiking behavior that the figure 5.2, is not exist for the figure 5.3. However, for the Nesterov accelerated gradient, RMSprop and Nadam there exists a relatively big spike in the training loos at the first epochs. Considering the fact that Nadam and NAG, uses predictive weights in the training, it can be said that the weight predictive optimization algorithms did not perform well for Bike sharing dataset.


**Figure 5.3 :** QRNN bike sharing users training loss.

#### 5.3.1.3 PM 2.5 concentration

The hyperparameters regarding the training phase of PM 2.5 Concentration dataset is given in the table 5.3.

**Table 5.3 :** Hyperparameters of QRNN for PM 2.5 concentration dataset.

| Number of Hidden Layers | Number of Hidden Neurons | Batch Number | Number of Epoch | Learning Rate | Firing Rate of Output Neurons | Weight Scaler |
|---|---|---|---|---|---|---|
| 1 | 50 | 16 | 25 | 0.01 | 0.25 | 0.01 |

Using the parameters in the table 3, training loss of optimization algorithms are shown in the figure 5.4. It is obvious that, for this dataset in particular, all the optimization algorithms converge approximately to the same RMS error value at the end of training phase, even if the starting points varies. Furthermore, notice that the paths that optimizers took is smoother than the the Bike sharing and Google stock price datasets. The reason behind it could be the use of dataset. Since the previous ones were univariate, it might be harder for QRNN to understand the complex non-linear relationship between the points. Thus, the multivariate use of PM 2.5 Concentration dataset might lead the optimizers to a relatively less fluctuating convergence.

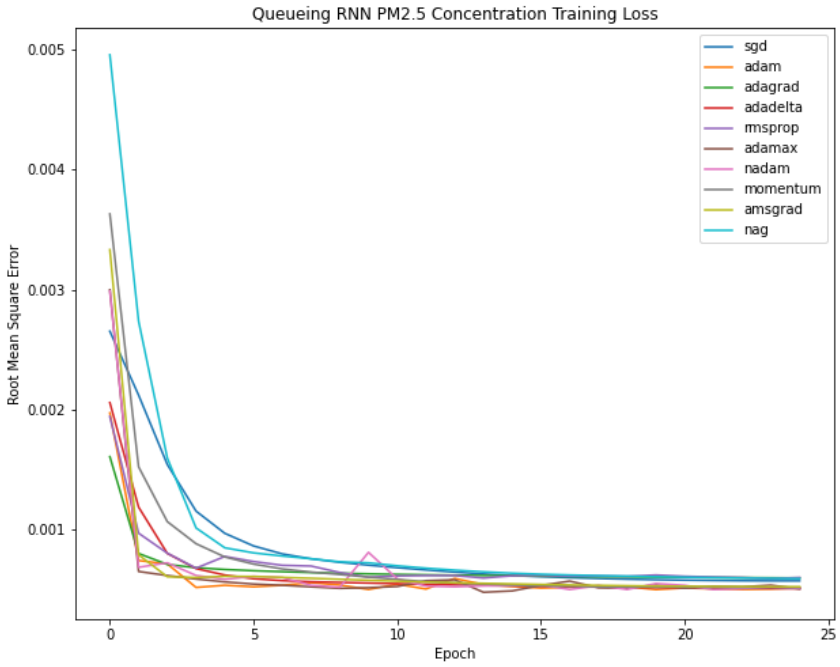


**Figure 5.4 :** QRNN PM 2.5 concentration training loss.

### 5.3.1.4 Traffic volume

The hyper parameters that has been used in the training phase of traffic volume dataset is given in the table 5.4.

**Table 5.4 :** Hyperparameters of QRNN for PM 2.5 concentration dataset.

| Number of Hidden Layers | Number of Hidden Neurons | Batch Number | Number of Epoch | Learning Rate | Firing Rate of Output Neurons | Weight Scaler |
|---|---|---|---|---|---|---|
| 1 | 50 | 4 | 50 | 0.01 | 0.25 | 0.05 |

As it can be seen in the figure 5.5, training loss of the multivariate Traffic volume dataset is much smoother than the previous datasets. The reason behind it can be simply expressed with the fact of seasonality. In the figure 5.7, the graph at the right left corner shows the performance of QRNN on the test set where the line with the red denotes the real value and the line with the blue denotes the predicted value. As it can be seen, red line oscillates with a certain interval and uncertain magnitude.

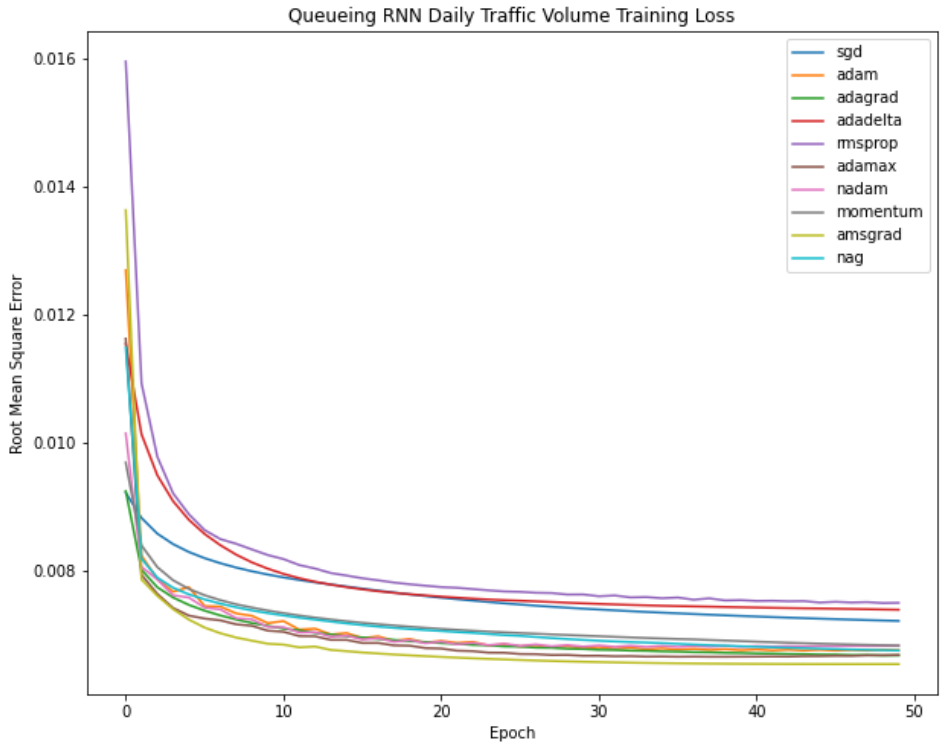


**Figure 5.5 :** QRNN daily traffic volume training loss.

The interval for this dataset is 7 days since it has been resampled daily. In other words, the shape of the raw data repeats itself with a certain period of 7 days. The traffic volume difference between the workdays and the weekend creates this fluctuation. If the data would not be sampled hourly than a similar pattern could have been observed during the hours of day and the period might have seen as 24 hours. As a result of this repeating pattern training loss graph is smoother than the previous ones.

The figure 5.5 reveals that most of the optimization algorithms except “SGD”, “RMSprop” and “Adadelta” converges approximately to the same value by following similar paths. However, these 3 algorithms perform slightly worse than the others.

#### 5.3.1.5 General performance evaluations of the optimization algorithms on QRNN

In the figure 5.6 the RMS error regarding the prediction that QRNN made over test sets are shown with a heat map format. The color scale of each row has been normalized within itself. Thus, the lower RMS error values are shown with yellow and the higher RMS error values are shown with green.

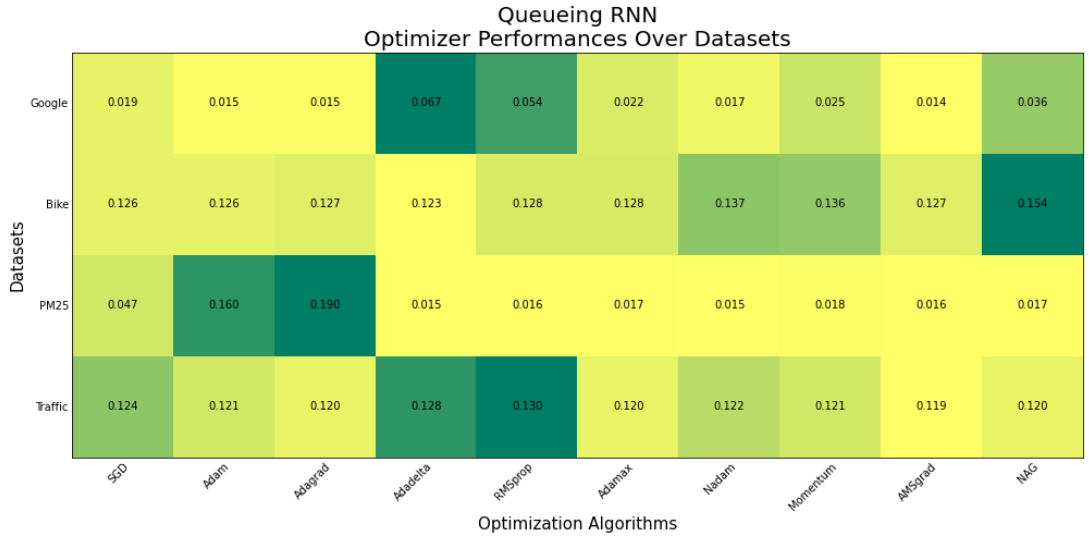


**Figure 5.6 :** QRNN optimizer performance heat map.

The figure 5.6 shows that for each dataset the behavior of the optimizers might change. While Adadelta algorithm outperforms the rest of the optimizers on the PM 2.5 concentration dataset, it performs the worst on the test set of Google stock price.

In order to give a glance of an idea that how the predictions and the real data of test set looks the figure 5.7 is given above. In this figure the results of AMSGrad algorithm has been chosen, due to its relatively good performance in overall evaluation.

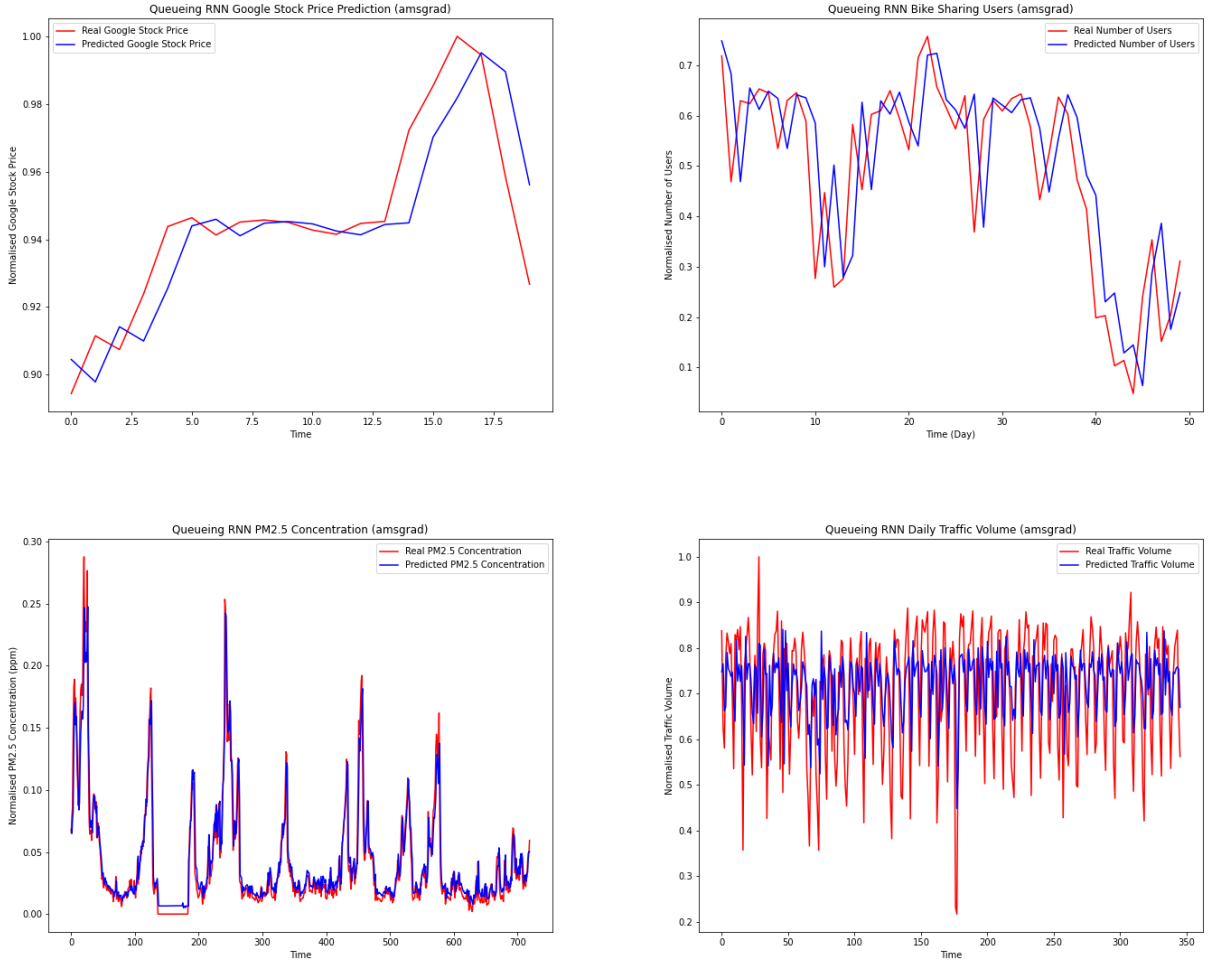


**Figure 5.7 :** Comparison of the predictions of QRNN using AMSGrad and real results

### 5.3.2 Comparison of QRNN with Other Recurrent Neural Networks

In the 3rd part of the section 5, the performance comparison of QRNN with LSTM, Vanilla RNN and GRU will be investigated using 4 different dataset and 10 different optimization algorithms. In order to visualize the results, 4 heat map that shows the performance of networks with different optimizers on a dataset by using colors and numerical values have been provided. Furthermore, a boxplot that demonstrate the distribution of the results is also given.

In the training phase of Vanilla RNN, LSTM and GRU, a Python library called Keras with TersorFlow backend is used. Since Keras does not support Momentum, AMSGrad and Nesterov Accelerated Gradient algorithms, the missing values in the heatmaps have been colorized with gray. The color scale of each heat map is given at the right-hand side of the figure and the it has been normalized through the surface unlike the figure 5.6 which has a color scale that has been normalized through the rows. In order to emphasize the comparison results, the lowest RMS error value for each column is framed with blue color and the lowest RMS error value of the figure is designated with the green frame.

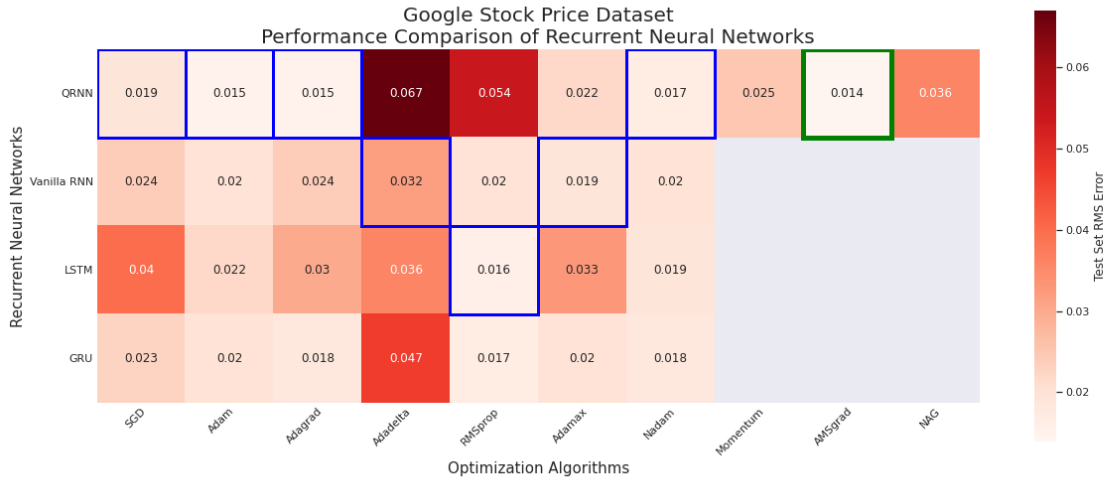


**Figure 5.8 :** RMS error heat map regarding the performance of Recurrent Neural Networks on the test set of Google stock price data

Figure 5.8 demonstrates the performance of Recurrent Neural Networks on Google stock price dataset. As it is shown with blue frames, overall performance of QRNN outperforms the others by having the 4 out of 7 lowest RMS error value among the common optimizers in the test set. Besides, the lowest achieved RMS error value also

belongs to QRNN by using the optimizer AMSGrad. However, it can also be said that the largest error value which is obtained by Adadelta, belongs to QRNN as well. As the column of Adadelta examined, it is the column that also has the largest values for Vanilla RNN and GRU.

As a result, it can be stated that the performance of QRNN has similarities with other Recurrent Neural Networks while operating with different optimizers and also outperforms the others by considering the fact that it has 4 out 7 lowest RMS error value on test set, beside the lowest RMS error value that has been achieved by itself. The figure 5.9 shows the best results that have been achieved by each Recurrent Neural Network, considering the lowest RMS error value in each row of figure 5.8.

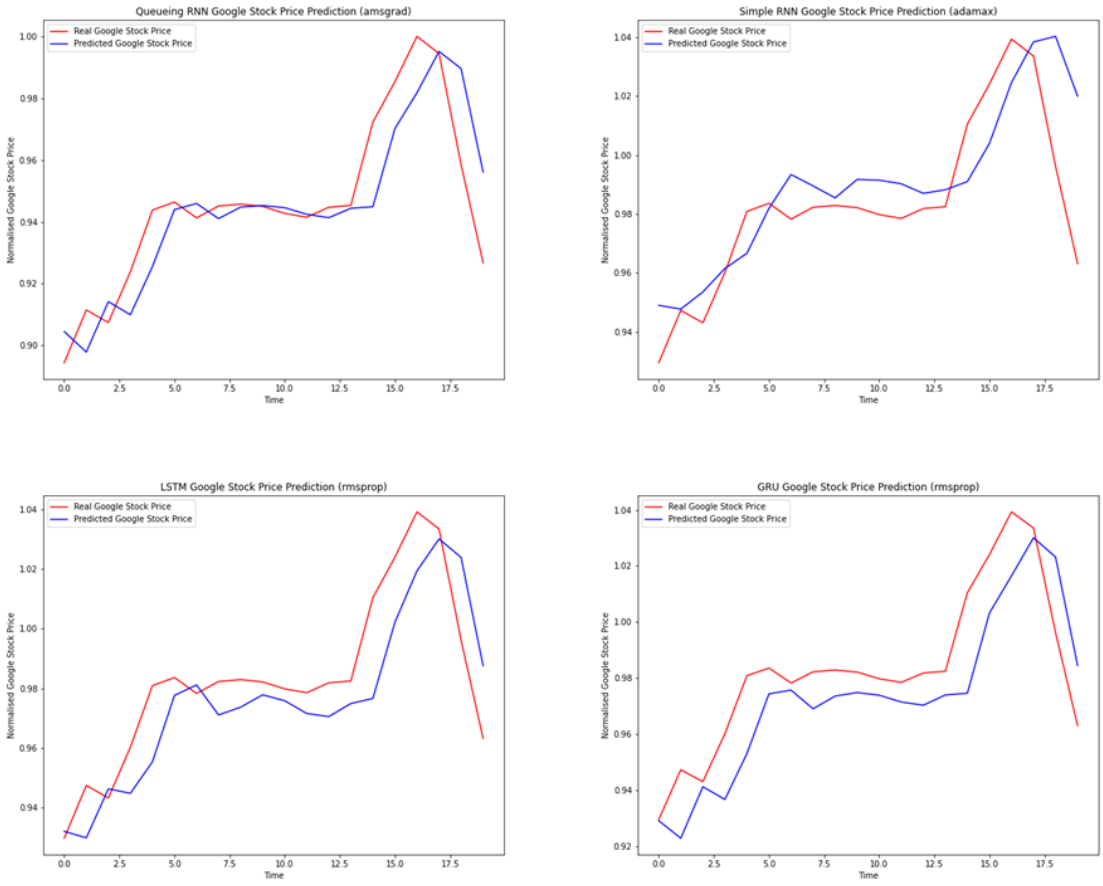


**Figure 5.9 :** Comparison of Recurrent Neural Networks on the test set of Google stock price data

In the figure 5.10 the results of Recurrent Neural Networks over the test set of Bike sharing dataset have been given. The results show that for all of the 7 common optimizer columns, QRNN outperforms the other Recurrent Neural Networks. Besides

the lowest RMS error value has been achieved by Adadelta which has been performed the worst for the Google stock price dataset.

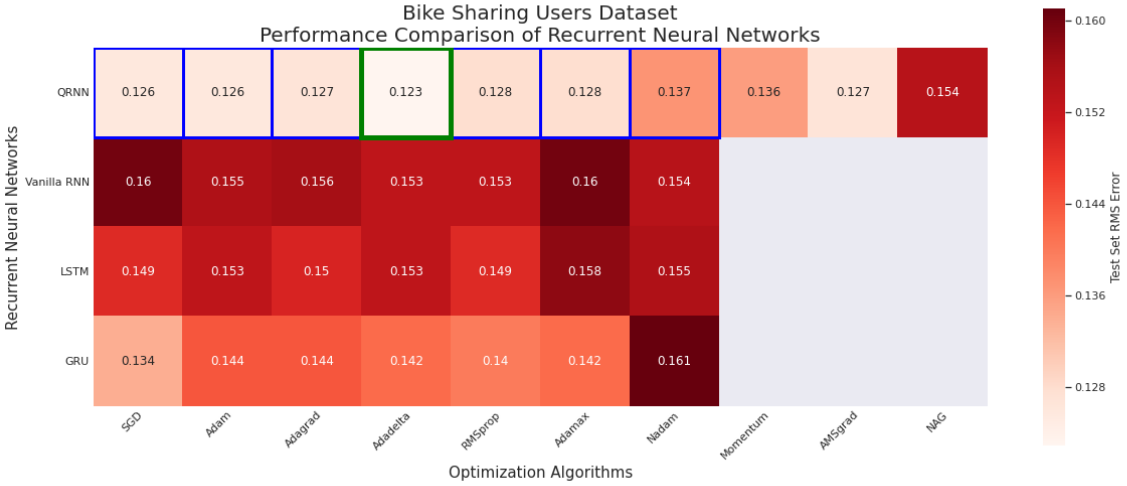


**Figure 5.10 :** RMS error heat map regarding the performance of Recurrent Neural Networks on the test set of bike sharing users data

As the figure 5.11 examined, the performance results from figure 5.10 seems obvious. Comparing the best results of each row from the heat map above, it can be stated that QRNN comprehends the characteristic of the Bike sharing data, way much better than the other Recurrent Neural Networks. While LSTM, Vanilla RNN and GRU deal with the problem of acquiring the non-linear dependencies through time, QRNN manages to comprehend the data and yield more successful predictions.

In the figure 5.12 the results of Recurrent Neural Networks over the test set of PM 2.5 Concentration data have been demonstrated. It shows that the lowest number of RMS error on test set has been achieved multiple times by multiple Recurrent Neural Networks. Considering the green and the blue frames, QRNN have been achieved to reach the lowest error by using 5 optimizers out of 7 where it shares the green frame with LSTM in the column of RMSprop.

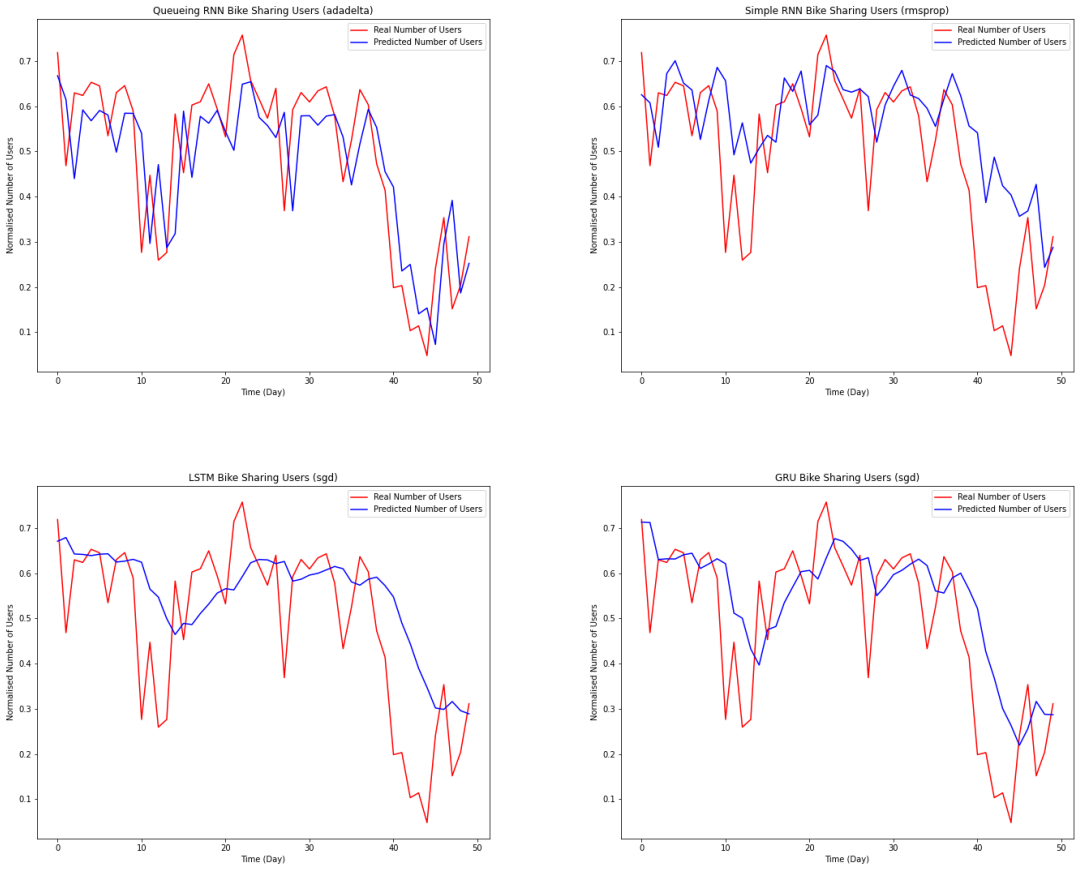


**Figure 5.11 :** Comparison of Recurrent Neural Networks on the test set of bike sharing users data

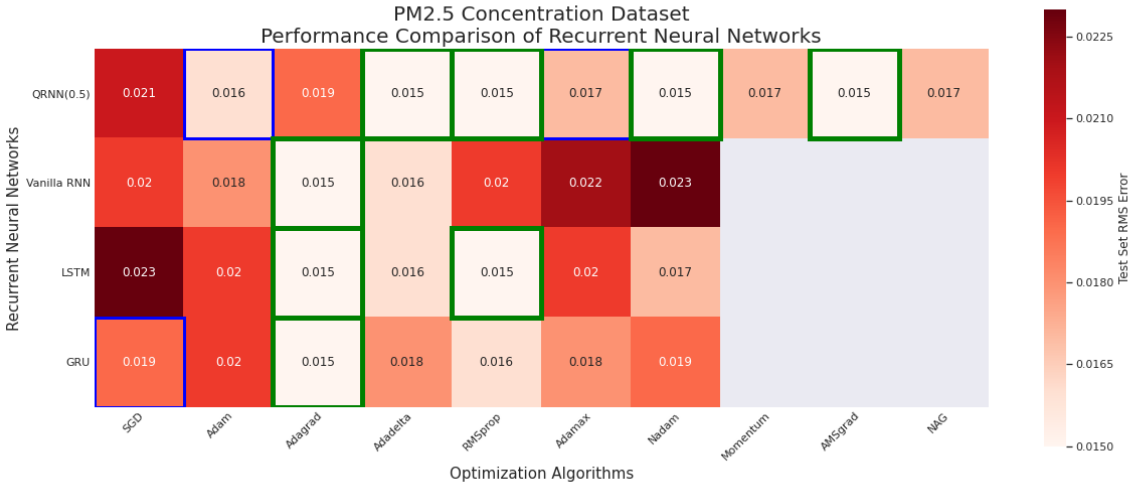


**Figure 5.12 :** RMS error heat map regarding the performance of Recurrent Neural Networks on the test set of PM 2.5 concentration data

The figures 5.12 and 5.13 both shows that all of the networks are capable of converging the same minimum point successfully by using different optimizers. The general distribution of the RMS error values is similar among the networks. However, the results of QRNN is slightly better than the Recurrent Neural Networks as it can be observed from the figure 5.12.

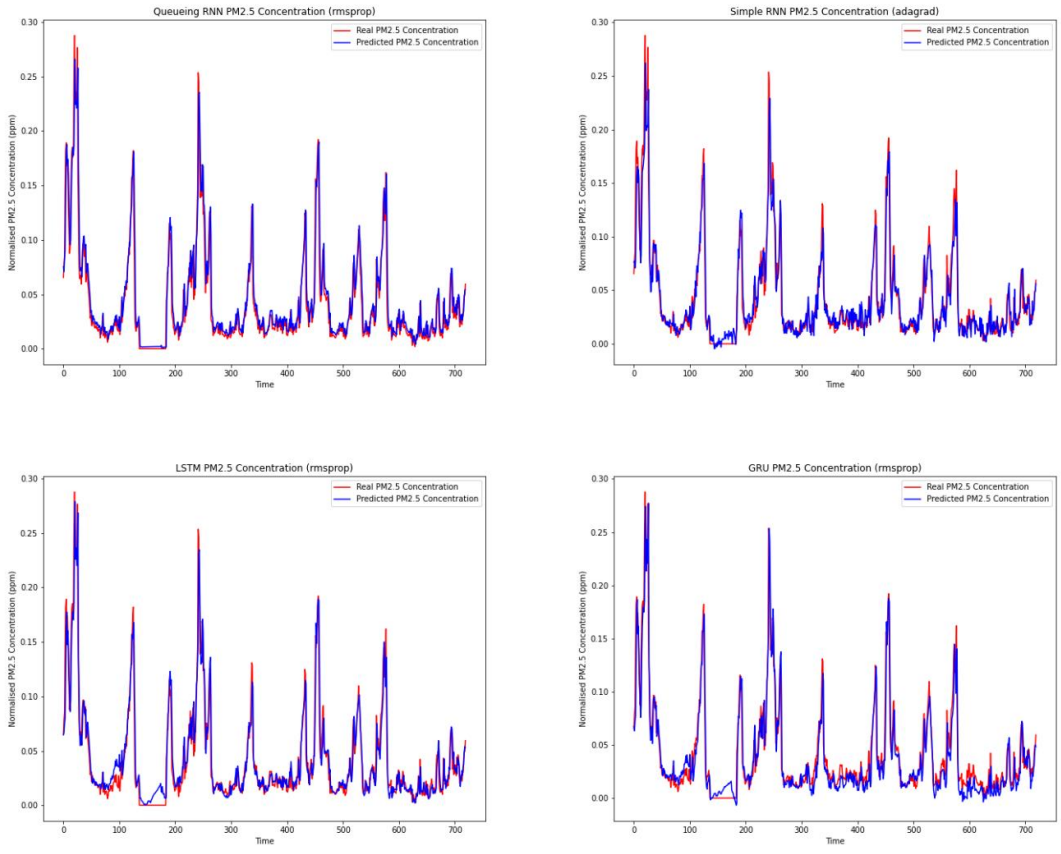


**Figure 5.13 :** Comparison of Recurrent Neural Networks on the test set of PM 2.5 concentration data

The test results regarding to another multivariate dataset which is the Traffic volume data is given in the figure 5.14. As it has been mentioned in the section 5.3.1.4 Traffic volume dataset has a periodic character with a certain period of time which is 7 days. In other word, general pattern repeats itself after each 7 days. Considering the figure 5.14, it can be said that Vanilla RNN outperforms the rest of the Recurrent Neural Networks on this dataset. In the comparison of columns, QRNN takes 2$^{nd}$, 3$^{rd}$, 4$^{th}$, 4$^{th}$, 3$^{rd}$, 2$^{nd}$, 4$^{th}$ order respectively within the common optimization algorithms. By considering the figures 5.14 and 5.15, it can be said that QRNN performed poorly compare to other in this dataset.

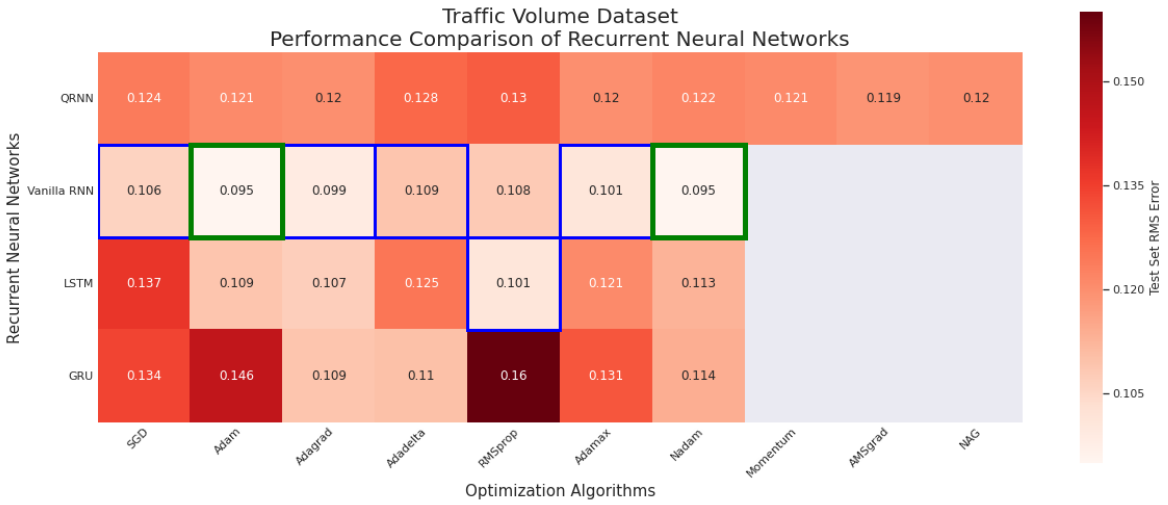


**Figure 5.14 :** RMS error heat map regarding the performance of Recurrent Neural Networks on the test set of traffic volume data

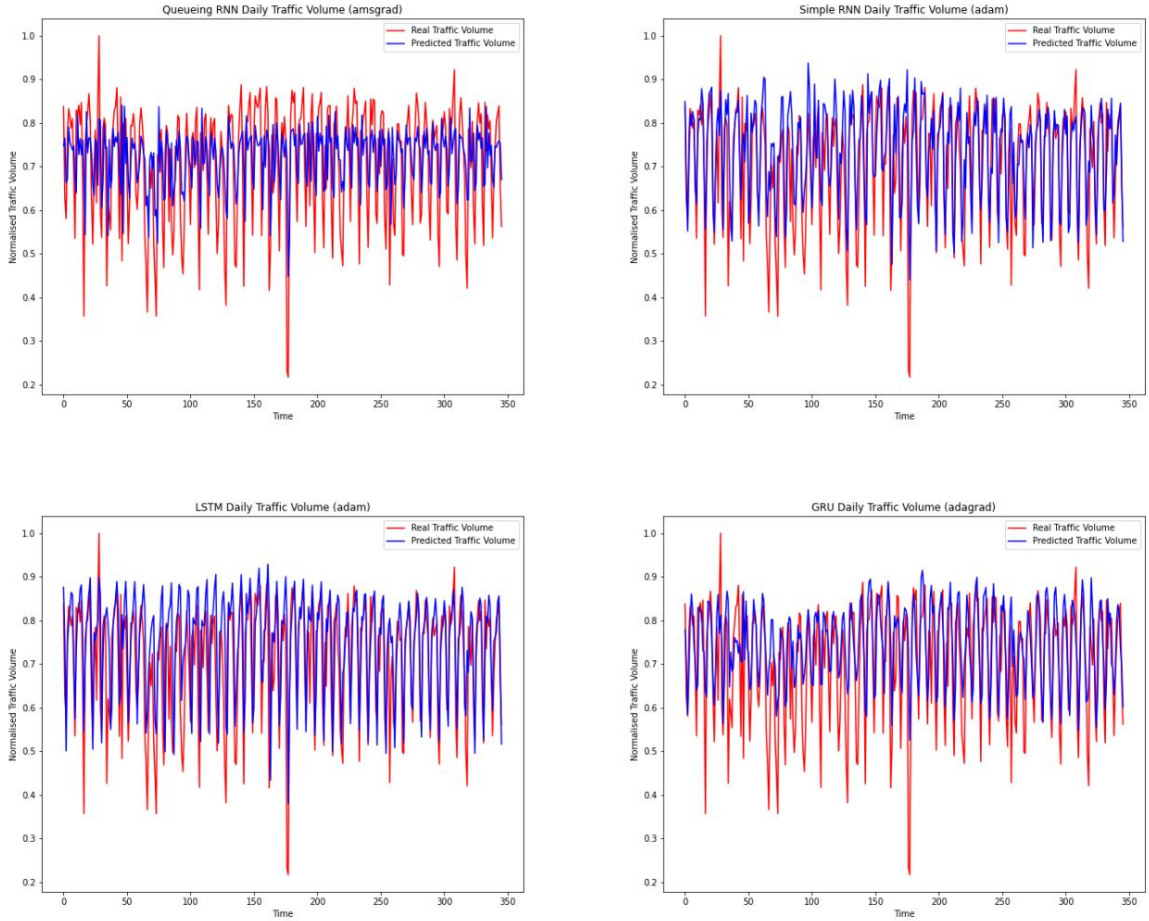


**Figure 5.15 :** Comparison of Recurrent Neural Networks on the test set of PM 2.5 concentration data

In order to evaluate and visualize the overall performance of QRNN compare to the other Recurrent Neural Networks the distributions regarding the each row of the figures 5.8, 5.10, 5.12 and 5.14 are combined and given with a box plot in the figure 5.16. Each box corresponds to a row in the previously mentioned figures. Considering the position of the QRNN in each plot, it can be stated that QRNN has been generally managed to reach one of the lowest RMS errors of each dataset. Particularly, for Google stock price dataset, the distribution of the scores regarding the optimization algorithms is cumulated (considering the position of median) on a lower point than the others. For the bike sharing users data, the lowest score and the median values regarding each Recurrent Neural Network reveals success of QRNN. Similarly, in the PM 2.5 concentration dataset, QRNN has the lowest median value, even if it shares the lowest score with other networks. Finally, for the traffic volume dataset, the span of the values for QRNN is relatively compressed to a relatively mediary RMS error value considering the general scale of the traffic volume plot.

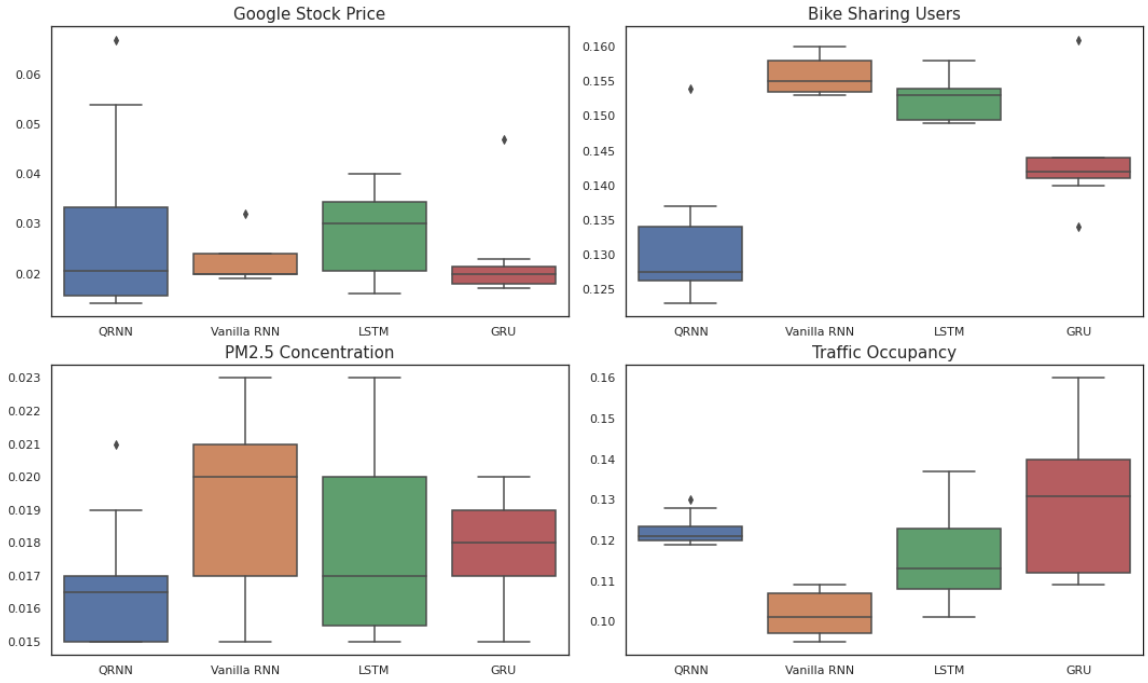


**Figure 5.16 :** Overall performance of QRNN with box plot

### 5.3.3 Comparison of QRNN with Random Neural Network

This section contains the comparison results of Random Neural Network and QRNN. It is known that the Recurrent Neural Networks perform better in the applications with time series or most kind of sequential data. Since the performance evaluation of RANN with time series data is not a topic that is commonly handled except few researches

such as [38], this section presents a brief examination regarding the performance of RANN as well.

Considering the fact that the input shape that has been used for the experiments which has been performed for the section 5.3.2 can not be used for Random Neural Network because of its feedforward structure, only the univariate datasets which are Google stock prices and Bike sharing users are used in this evaluation. The preprocessing phase which has been covered in the sections 5.3.1.1 and 5.3.1.2, is used to train RANN.

Besides, hyperparameters of QRNN are used directly for the RANN as well. However, since the capacity of QRNN is greater than RANN with the same number of hidden neurons to acquire a dataset (considering the number of weights and neurons when the network is unfolded through time), RANN has been tested for 2 different number of hidden neurons which are 50 (the same as QRNN) and 100.

In the figure 5.17 and 5.18, the performance of optimization algorithms with Random Neural Network is given (The data is obtained from the training phase of RANN with 50 hidden neurons). As the figures examined, for both datasets the optimizers converge relatively different RMS errors. Considering the figure 5.17 and 5.18, basic gradient descent algorithm yields the one of the worst convergences among the optimization algorithms. On the other hand, RMSprop performs the best convergence considering only the paths of training loss graphs and the reached minimum RMS error.

The figure 5.19 reveals the comparison between QRNN and RANN in terms of test set RMS errors. As it has been done in the previous heat maps the lowest RMS error for each column is framed with blue and the lowest RMS error score of the heat map is designated with green frame. As the results are considered, it is obvious that Queueing RNN outperforms the Random Neural Network by reaching the lowest RMS errors on the test set. However results of the RANN on the Google stock price data can be considered as quite close to the QRNN for some of the optimizer columns.

Finally, the figure 5.20 demonstrate the predicted and real values regarding two datasets. Since RANN performs the best with Nadam and RMSprop for Google stock price and bike sharing data respectively, at the left-hand side of the figure 5.20, the graphs regarding the two of them are given.

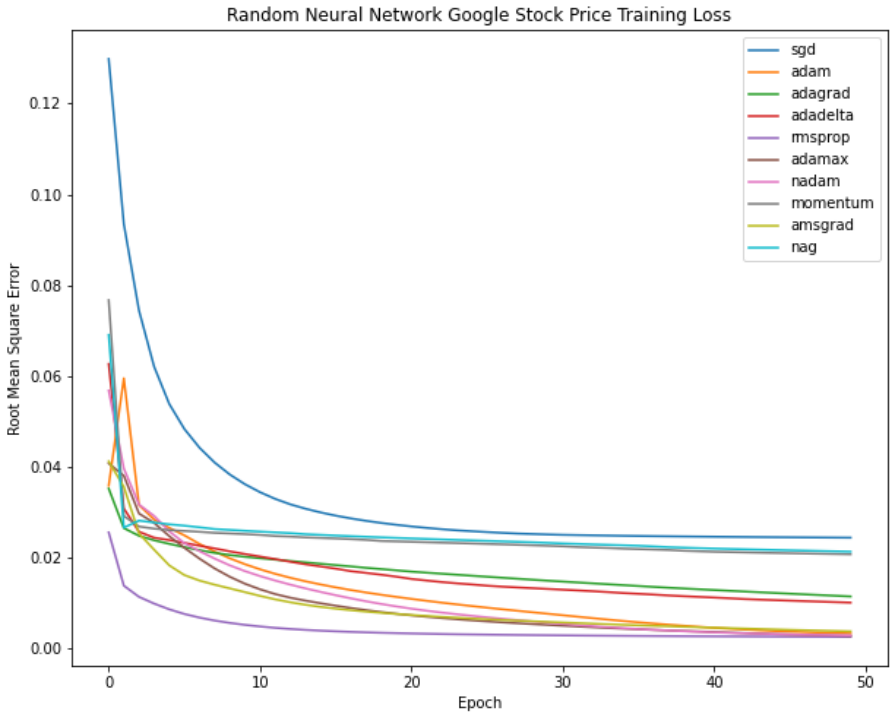


**Figure 5.17 :** RANN Google stock price training loss.

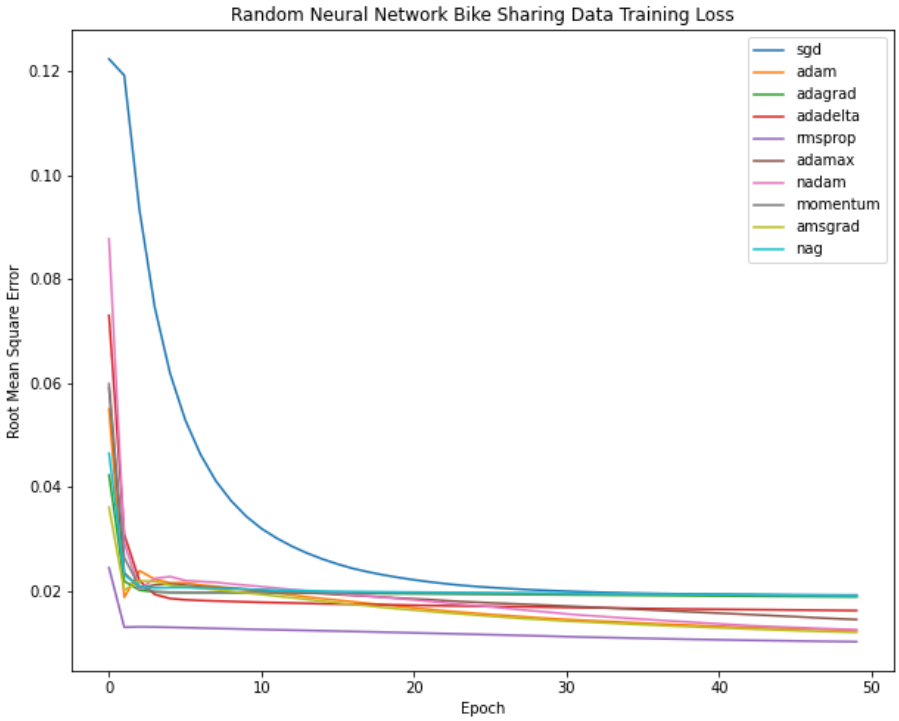


**Figure 5.18 :** RANN bike sharing users training loss.

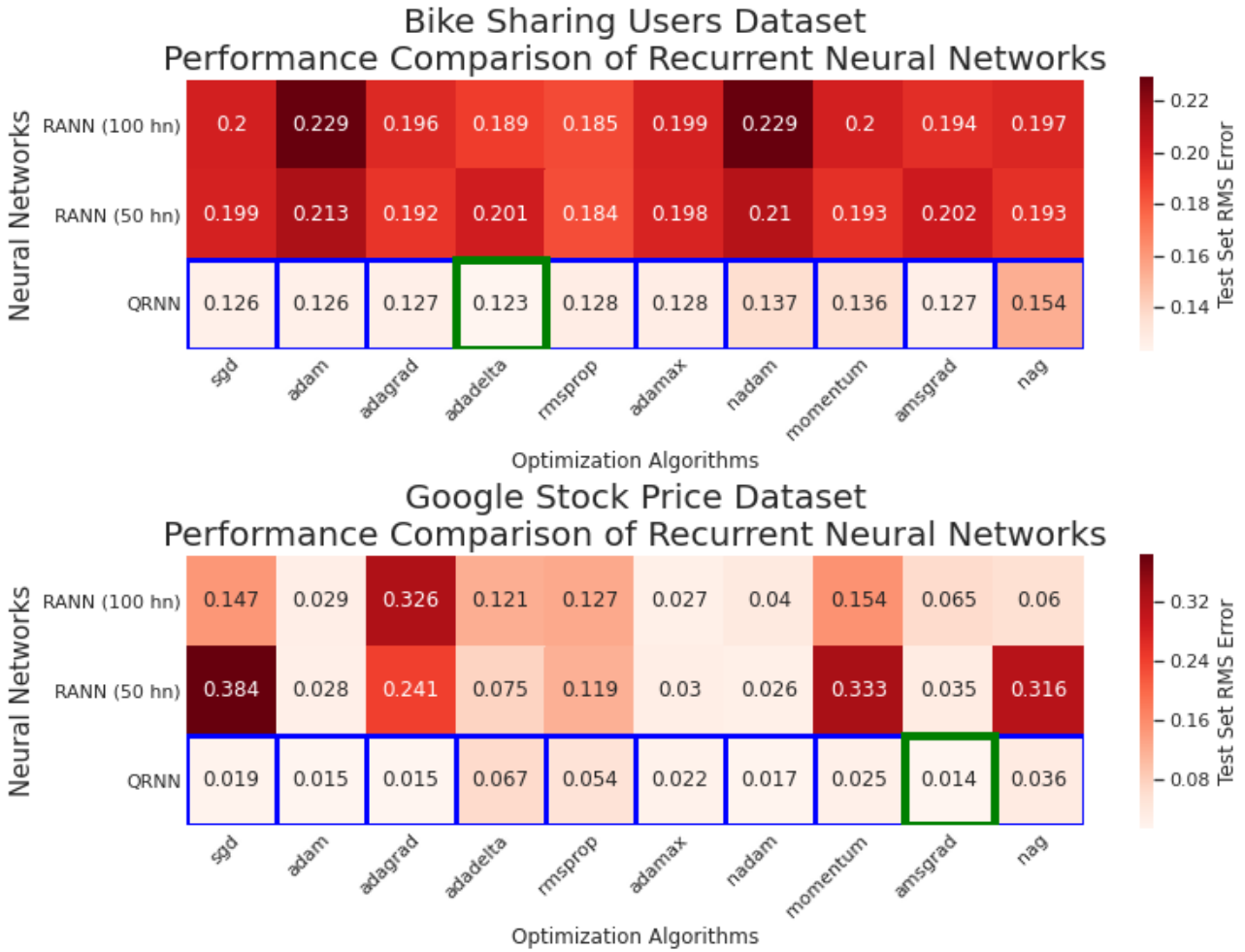


**Figure 5.19 :** RMS error heat map regarding the performance of Recurrent Neural Networks on the test set of bike sharing users and Google stock price data.

Similarly, at the right-hand side performance of QRNN with the same optimizers are shown. Considering the figure 5.20, it is obvious that RANN have problems of dealing with the prediction of fluctuating real value, where QRNN adapts itself to changing conditions.

To conclude, it can be stated that performance of QRNN is superior over RANN, in the applications of time series forecasting according to the results given in the figures 5.19 and 5.20. This supremacy can also be considered as a natural result regarding the difference of recurrent and feedforward neural networks. Considering the evaluation results, QRNN offers a better solution to the sequential regression problem than RANN by preserving the spiking behavior of the Random Neural Networks.

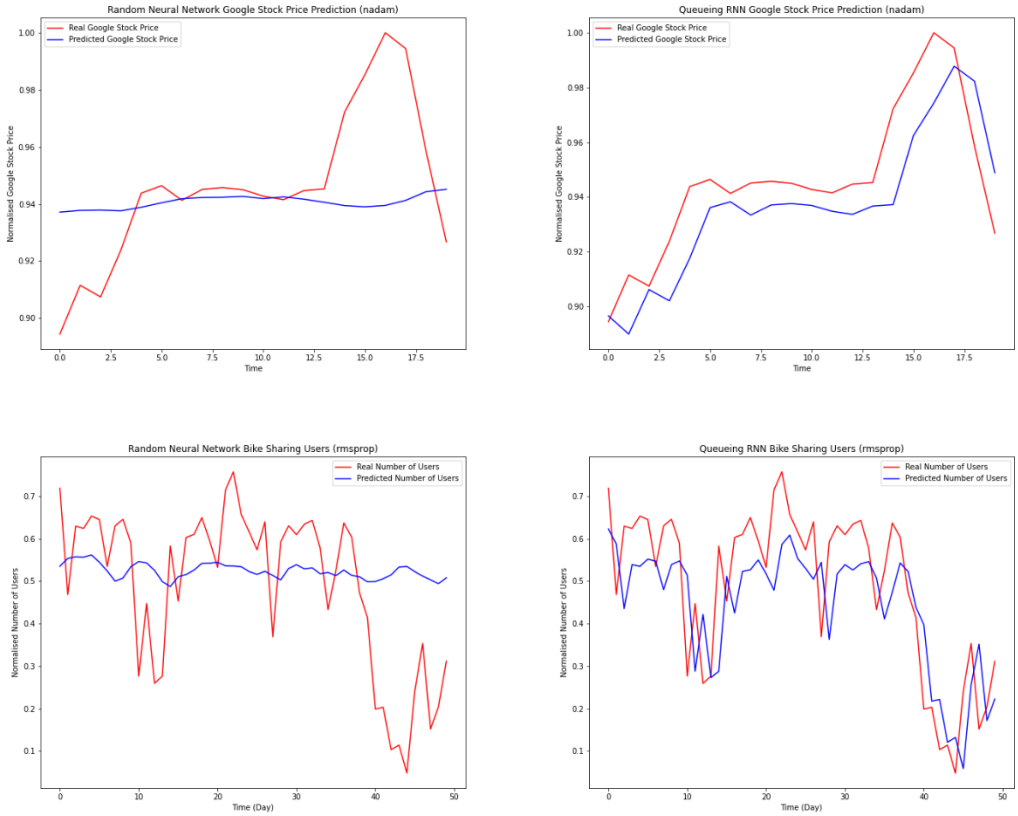


**Figure 5.20 :** Comparison of the predictions of RANN and QRNN for the datasets bike sharing users and Google stock price.

# 6. CONCLUSIONS

Time series data analysis is widely used in numerous real world applications. In the analysis process machine learning algorithms, especially Recurrent Neural Networks, plays a key role in obtaining meaningful and coherent results. This thesis proposes a new Machine Learning technique that is capable of performing in every application areas where Recurrent Neural Networks are used. In the first chapters, the evoluation of Recurrent Neural Networks, their mathematical models and working principles have been presented in chronological order. Afterwards, the RANN structure has been detailly explained. As a combination of these different machine learning techniques, Queueing Recurrent Neural Network has been proposed. Beside the mathematical background, performance evaluation of QRNN with detailled comparison results from different aspects have been provided.

Considering the outcomes of previously perfomed experiments, it can be said that QRNN has reached better scores and convergence results for most of the test case compare to its alternatives. This performance, reveals the potential of QRNN to be easily used not only in the time series data analysis but also in various applications with sequential data, such as natural language processing, anomaly dedection, sentiment analysis or seasonality and trend analysis.

## 6.1 Adventages and Disadventages

Advantages:

- Better convergence results especially in highly non-linear datasets (According to test results from chapter 5)
- No need for an activation function to be choose and deal with the problems that it brings.
- Non-linear mathematical model
- Capability of modelling sequential data by preserving spiking behavior

Disadvantages:

- Greater computational time and effort, due to the complexity of the model
- Limitations in the input scaling
- 2 times larger number of parameters to be trained, due to the mathematical model

### 6.2 Further Researces

This thesis mainly focuses on the time series regression problems. Thus other areas that Recurrent Neural Networks commonly used might be investigated for QRNN. Beside the applications, the architectures such as one-to-many and many-to-many can be implemented and tested in different cases. A detailed analysis regarding the effect of vanishing gradient on QRNN might be searched. The performance evaluation of QRNN for the other loss fucntions such as Cross-Entropy can be investigated. The performance of a hybrid structure that contains QRNN with convuloutional neural networks can be tested in applications such as image processing.

## REFERENCES


**[**1] **Abrahart, R. J. & See, L.** (1998). Neural Network vs. ARMA Modelling: Constructing Benchmark Case Studies of River Flow Prediction.In J.Blenc, (Ed.), *GeoComputation '98. Proceedings of the Third International Conference on GeoComputation*, (pp.145-154). United Kingdom : University of Bristol, September 17-19.

[2] **Elman, J.L.** (1990). Finding Structure in Time. *Cogn. Sci., 14*, 179-211.

[3] **Bengio, Y., Simard, P., & Frasconi, P.** (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157–166. https://doi.org/10.1109/72.279181

[4] **Pascanu, R., Mikolov, T., & Bengio, Y.** (2013). On the difficulty of training recurrent neural networks. *ICML*.

[5] **Werbos, P.J.** (1990). Backpropagation Through Time: What It Does and How to Do It.

[6] **McCulloch, W.S., & Pitts, W.F.** (1990). A logical calculus of the ideas immanent in nervous activity.

[7] **Gelenbe, Erol.** (1989). Random Neural Networks with Negative and Positive Signals and Product Form Solution. Neural Computation - NECO. 1. 502-510. 10.1162/neco.1989.1.4.502.

[8] **Gelenbe, Erol.** (1993). Learning in the Recurrent Random Neural Network. Neural Computation. 5. 154-164. 10.1162/neco.1993.5.1.154.

[9] **Basterrech, Sebastián & Rubino, Gerardo.** (2017). Echo State Queueing Networks: a combination of Reservoir Computing and Random Neural Networks. Probability in the Engineering and Informational Sciences. 1-20. 10.1017/S0269964817000110.

[10] **Url-1** <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> date retrieved 21.05.2015.

[11] **Url-2** <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> date retrieved 27.08.2015.

[12] **Pascanu, Razvan & Mikolov, Tomas & Bengio, Y..** (2012). On the difficulty of training Recurrent Neural Networks. 30th International Conference on Machine Learning, ICML 2013.

[13] **Corana, Angelo & Marchesi, Michele & Martini, Claudio & Ridella, Sandro.** (1987). Minimizing Multimodal Functions Of Continuous-Variables with Simulated Annealing Algorithm. ACM Transactions on Mathematical Software. 13. 262-280. 10.1145/29380.29864.

[14] **Becker, Suzanna & Lecun, Yann.** (1989). Improving the Convergence of Back-Propagation Learning with Second-Order Methods.

[15] **Glorot, X., Bordes, A. & Bengio, Y..** *(2011). Deep Sparse Rectifier Neural Networks. Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, in PMLR 15:315-323*

[16] **Lu, L., Shin, Y., Su, Y., & Karniadakis, G.E.** (2019). Dying ReLU and Initialization: Theory and Numerical Examples. *ArXiv, abs/1903.06733*.

[17] **Clevert, D., Unterthiner, T., & Hochreiter, S.** (2016). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *CoRR, abs/1511.07289*.

[18] **Hochreiter, Sepp & Schmidhuber, Jürgen.** (1995). Long Short-term Memory. Technical Report FKI-207-95, Technische Universitat München, München.

[19] **Hochreiter, Sepp & Schmidhuber, Jürgen.** (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.

[20] **Greff, Klaus & Srivastava, Rupesh & Koutník, Jan & Steunebrink, Bas & Schmidhuber, Jürgen.** (2015). LSTM: A search space odyssey. IEEE transactions on neural networks and learning systems. 28. 10.1109/TNNLS.2016.2582924.

[21] **Gers, Felix & Schmidhuber, Jürgen & Cummins, Fred.** (2000). Learning to Forget: Continual Prediction with LSTM. Neural computation. 12. 2451-71. 10.1162/089976600300015015.

[22] **Gers, F.A., & Schmidhuber, J.** (2000). Recurrent nets that time and count. Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, 3, 189-194 vol.3.

[23] **Cho, Kyunghyun & van Merriënboer, Bart & Bahdanau, Dzmitry & Bengio, Y..** (2014). On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. 10.3115/v1/W14-4012.

[24] **Dey, R., & Salem, F.M.** (2017). Gate-variants of Gated Recurrent Unit (GRU) neural networks. *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 1597-1600.

[25] **Basterrech, S., & Rubino, G.** (2015). Random Neural Network Model for Supervised Learning Problems. *Neural Network World, 25*, 457-499.

[26] **Rumelhart. D.,Hinton, G. & Williams, R.** (1986). Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (Vol. 1, pp.318-362). Cambridge, MA: MIT Press.

[27] **Abdelbaki H.** (1999). rnnsimv2.zip. Retrieved September 9, 1999. Available from https://www.mathworks.com/matlabcentral/fileexchange/91-rnnsimv2-zip

[28] **Rawat, M** (2018). Google Stock Price. Retrieved April 09, 2018 from kaggle.com/medharawat/google-stock-price.

[29] **Fanaee-T, Hadi, and Gama, Joao.** (2013). Event labeling combining ensemble detectors and background knowledge, Progress in Artificial Intelligence (pp. 1-15), Springer Berlin Heidelberg [https://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset]

[30] **Liang, X., Zou, T., Guo, B., Li, S., Zhang, H., Zhang, S., Huang, H. and Chen, S. X.** (2015). Assessing Beijing's PM2.5 pollution: severity, weather impact, APEC and winter heating. Proceedings of the Royal Society A, 471, 20150257.

[31] **Hogue, J.** (2019). Metro Interstate Traffic Volume Data Set. Retrieved May 07, 2019 from https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume.

[32] **Ruder, Sebastian.** (2016). An overview of gradient descent optimization algorithms.

[33] **Nesterov, Y.** (1983). A method for unconstrained convex minimization problem with the rate of convergence. Doklady AN SSSR. 269. 543-547.

[34] **Duchi, John & Hazan, Elad & Singer, Yoram.** (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research. 12. 2121-2159.

[35] **Zeiler, Matthew.** (2012). ADADELTA: An adaptive learning rate method. 1212.

[36] **Toronto University,** (n.d.). Neural Networks for Machine Learning [PowerPoint slides]. Retrieved from https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[37] **Kingma, D.P., & Ba, J.** (2015). Adam: A Method for Stochastic Optimization. *CoRR, abs/1412.6980*.

[38] **Serrano W.** (2020) The Random Neural Network in Price Predictions. In: Maglogiannis I., Iliadis L., Pimenidis E. (eds) Artificial Intelligence Applications and Innovations. AIAI 2020. IFIP Advances in Information and Communication Technology, vol 583. Springer, Cham.

## APPENDIXES

**APPENDIX A:** Python Implementation of Random Neural Network.
**APPENDIX B:** Python Implementation of Simple Recurrent Neural Network.
**APPENDIX C:** Python Implementation of Queueing Recurrent Neural Network.

**APPENDIX A:** Python Implementation of Random Neural Network.

```
# -*- coding: utf-8 -*-
import numpy as np
import pandas as pd
from numpy import concatenate
from math import sqrt
import copy

class RANN:
    def __init__(self,
                 layer_list,
                 weight_scaler = 0.05,
                 firing_rate_scaler = 0.1,
                 learning_rate = 0.01,
                 loss_function = 'mse',
                 optimizer = 'sgd'):

        self.optimizer = optimizer
        self.learning_rate = learning_rate
        self.shape = layer_list
        self.loss_func = loss_function
        n = len(layer_list)

        self.layers = []
        for i in range(n):
            self.layers.append(np.zeros(self.shape[i]))

        self.rate = []
        for i in range(n):
            self.rate.append(np.zeros((self.shape[i],1)))
        self.rate[-1] = firing_rate_scaler * np.ones((self.rate[-1].shape))

        self.wplus = []
        self.wminus = []
        for i in range(n-1):
            self.wplus.append(np.zeros((self.layers[i].size,
                                        self.layers[i+1].size)))
            self.wminus.append(np.zeros((self.layers[i].size,
                                        self.layers[i+1].size)))

        self.Q = []
        for i in range(n):
            self.Q.append(np.zeros((self.shape[i],1)))

        self.lambda_plus = []
        self.lambda_minus = []

        self.D = []
        self.global_var2 = []
        self.global_var = []

        self.pre_grad_p = copy.deepcopy(self.wplus)
        self.pre_grad_m = copy.deepcopy(self.wminus)
        self.pre_grad_pm =
np.array([copy.deepcopy(self.wplus),copy.deepcopy(self.wminus)])
        self.pre_grad_pm_2 = copy.deepcopy(self.pre_grad_pm)
        self.pre_grad_pm_3 = copy.deepcopy(self.pre_grad_pm)
        self.init_weights(weight_scaler)

        self.bp_counter=0

    def init_weights(self,weight_scaler):

        for i in range(len(self.wplus)):
            self.wplus[i] =
weight_scaler*np.random.rand(self.wplus[i].shape[0],self.wplus[i].shape[1])
        for i in range(len(self.wminus)):
            self.wminus[i] =
weight_scaler*np.random.rand(self.wminus[i].shape[0],self.wminus[i].shape[1])
        return 1

    def calculate_rate(self):
        for layer_num in range(len(self.layers)-1): #there is no rate for output
layer
            for neuron in range(len(self.layers[layer_num])):
```

```
                self.rate[layer_num][neuron] = (self.wplus[layer_num][neuron] +
self.wminus[layer_num][neuron]).sum()

        return 1


    def feedforward(self,input_list):
        self.D.clear() #clear list of D matrixes for the next iteration
        self.lambda_plus = np.where(input_list > 0, input_list, 0).reshape(-1,1)
        self.lambda_minus = np.where(input_list < 0, -input_list, 0).reshape(-1,1)

        # Input Layer
        self.D.append(self.rate[0]+self.lambda_minus)

        self.Q[0]=self.lambda_plus/self.D[0]
        np.clip(self.Q[0],0,1,out=self.Q[0])

        for i in range(1,len(self.shape)):
            # Hidden Layer
            T_plus = np.dot(self.wplus[i-1].transpose(),self.Q[i-1])
            T_minus =  np.dot(self.wminus[i-1].transpose(),self.Q[i-1])
            self.D.append(self.rate[i] + T_minus)
            self.Q[i] = T_plus / self.D[i]
            np.clip(self.Q[i],0,1,out=self.Q[i])

        return copy.deepcopy(self.Q[-1])

    def backpropagation(self,real_output,tmp1=np.zeros((2,1))):
        self.bp_counter=self.bp_counter+1
        if self.optimizer=='nag':
            weights_pm = np.array([self.wplus,self.wminus])
            x_ahead = weights_pm-self.pre_grad_pm*0.9

self.wplus,self.wminus=copy.deepcopy(list(x_ahead[0])),copy.deepcopy(list(x_ahead[1])
)

            for i in range(len(self.wminus)):
                np.clip(self.wminus[i],a_min=0.001,a_max=None)
                np.clip(self.wplus[i],a_min=0.001,a_max=None)

        loss = 0
        grad_p = []
        grad_m = []
        for i in reversed(range(len(self.shape)-1)):
            ver_grad = ((self.wplus[i].transpose()-
(self.wminus[i].transpose()*self.Q[i+1]))/self.D[i+1]).transpose()
            do_dwp1 = ((self.Q[i] @ (1/self.D[i+1]).transpose() + (-
self.Q[i]/self.D[i])*ver_grad).transpose()*tmp1).transpose()
            do_dwm1 = ((-self.Q[i] @ (self.Q[i+1]/self.D[i+1]).transpose() + (-
self.Q[i]/self.D[i])*ver_grad).transpose()*tmp1).transpose()
            tmp1 = ver_grad @ tmp1
            grad_p.append(do_dwp1)
            grad_m.append(do_dwm1)

        grad_p.reverse()
        grad_m.reverse()

        grad_p = np.asarray(grad_p)
        grad_m = np.asarray(grad_m)
        grad_pm = np.array([grad_p,grad_m])

        if self.optimizer == 'sgd':
            self.update_weigths(grad_pm*self.learning_rate)

        elif self.optimizer == 'momentum':
            moment_pm = self.pre_grad_pm*0.9;
            self.pre_grad_pm = moment_pm + grad_pm*self.learning_rate
            self.update_weigths(self.pre_grad_pm)

        elif self.optimizer == 'nag':
            self.pre_grad_pm = self.pre_grad_pm*0.9 + grad_pm*self.learning_rate
            self.update_weigths(self.pre_grad_pm)

        elif self.optimizer == 'adagrad':
            grad_pm_2 = np.square(grad_pm)
            self.pre_grad_pm+=copy.deepcopy(grad_pm_2)
            for i in range(grad_pm.shape[1]):
```

```
                grad_pm[0,i] =
(self.learning_rate/(np.sqrt(self.pre_grad_pm[0,i]+0.000001)))*grad_pm[0,i]
                grad_pm[1,i] =
(self.learning_rate/(np.sqrt(self.pre_grad_pm[1,i]+0.000001)))*grad_pm[1,i]
            self.update_weigths(grad_pm)

        elif self.optimizer == 'adadelta':
            eps=0.000001;beta=0.90;
            grad_pm_2 = np.square(grad_pm)

            self.pre_grad_pm_2 = beta*self.pre_grad_pm_2 + (1-beta)*grad_pm_2

            delta_teta = copy.deepcopy(self.pre_grad_pm)
            for i in range(grad_pm.shape[1]):
                            delta_teta[0,i] =
(np.sqrt(self.pre_grad_pm[0,i]+0.000001)/(np.sqrt(self.pre_grad_pm_2[0,i]+0.000001)))
*grad_pm[0,i]
                delta_teta[1,i] =
(np.sqrt(self.pre_grad_pm[0,i]+0.000001)/(np.sqrt(self.pre_grad_pm_2[1,i]+0.000001)))
*grad_pm[1,i]

            self.pre_grad_pm = beta*self.pre_grad_pm + (1-beta)*np.square(delta_teta)

            for i in range(grad_pm.shape[1]):
                grad_pm[0,i] =
(np.sqrt(self.pre_grad_pm[0,i]+0.000001)/(np.sqrt(self.pre_grad_pm_2[0,i]+0.000001)))
*grad_pm[0,i]
                grad_pm[1,i] =
(np.sqrt(self.pre_grad_pm[1,i]+0.000001)/(np.sqrt(self.pre_grad_pm_2[1,i]+0.000001)))
*grad_pm[1,i]

            self.update_weigths(grad_pm)

        elif self.optimizer == 'rmsprop':
            eps=0.00000001
            grad_pm_2 = np.square(grad_pm)
            self.pre_grad_pm_2 = 0.9*self.pre_grad_pm_2 + 0.1*grad_pm_2

            for i in range(grad_pm.shape[1]):
                grad_pm[0,i] =
(self.learning_rate/(np.sqrt(self.pre_grad_pm_2[0,i]+0.000001)))*grad_pm[0,i]
                grad_pm[1,i] =
(self.learning_rate/(np.sqrt(self.pre_grad_pm_2[1,i]+0.000001)))*grad_pm[1,i]

            self.update_weigths(grad_pm)

        elif self.optimizer == 'adam':
            eps=0.000001;beta1=0.9;beta2=0.999;
            grad_pm_2 = np.square(grad_pm)
            self.pre_grad_pm = beta1*self.pre_grad_pm + (1-beta1)*grad_pm
            self.pre_grad_pm_2 = beta2*self.pre_grad_pm_2 + (1-beta2)*grad_pm_2
            gt = self.pre_grad_pm/(1-beta1**self.bp_counter)
            gt2 = self.pre_grad_pm_2/(1-beta2**self.bp_counter)

            for i in range(grad_pm.shape[1]):
                grad_pm[0,i] = (self.learning_rate/(np.sqrt(gt2[0,i])+eps))*gt[0,i]
                grad_pm[1,i] = (self.learning_rate/(np.sqrt(gt2[1,i])+eps))*gt[1,i]

            self.update_weigths(grad_pm)

        elif self.optimizer == 'adamax':
            eps=0.000001;beta1=0.9;beta2=0.999;

            grad_pm_2 = np.square(grad_pm)
            self.pre_grad_pm = beta1*self.pre_grad_pm + (1-beta1)*grad_pm
            gt = self.pre_grad_pm/(1-beta1**self.bp_counter)

            for i in range(self.pre_grad_pm_2.shape[1]):
                if self.bp_counter==1:
                    self.pre_grad_pm_2[0,i].fill(eps)
                    self.pre_grad_pm_2[1,i].fill(eps)
                self.pre_grad_pm_2[0,i]
=np.maximum(beta2*self.pre_grad_pm_2[0,i],abs(grad_pm)[0,i])
                self.pre_grad_pm_2[1,i]
=np.maximum(beta2*self.pre_grad_pm_2[1,i],abs(grad_pm)[1,i])

            #learning rate is 0.002 as an adviced value
```

```
            grad_pm = (self.learning_rate / self.pre_grad_pm_2) * gt
            self.update_weigths(grad_pm)

        elif self.optimizer == 'nadam':
            eps=0.000001;beta1=0.9;beta2=0.999;

            grad_pm_2 = np.square(grad_pm)
            self.pre_grad_pm = beta1*self.pre_grad_pm + (1-beta1)*grad_pm
            self.pre_grad_pm_2 = beta2*self.pre_grad_pm_2 + (1-beta2)*grad_pm_2
            gt = self.pre_grad_pm/(1-beta1**self.bp_counter)
            gt2 = self.pre_grad_pm_2/(1-beta2**self.bp_counter)
            gt = (beta1*gt-((1-beta1)/(1-beta1**self.bp_counter))*gt)
            for i in range(grad_pm.shape[1]):
                grad_pm[0,i] = (self.learning_rate/(np.sqrt(gt2[0,i])+eps))*gt[0,i]
                grad_pm[1,i] = (self.learning_rate/(np.sqrt(gt2[1,i])+eps))*gt[1,i]
            self.update_weigths(grad_pm)

        elif self.optimizer == 'amsgrad':
            eps=0.000001;beta1=0.9;beta2=0.999;

            grad_pm_2 = np.square(grad_pm)
            self.pre_grad_pm = beta1*self.pre_grad_pm + (1-beta1)*grad_pm
            self.pre_grad_pm_2 = beta2*self.pre_grad_pm_2 + (1-beta2)*grad_pm_2
            gt = self.pre_grad_pm/(1-beta1**self.bp_counter)
            gt2 = self.pre_grad_pm_2/(1-beta2**self.bp_counter)

            for i in range(self.pre_grad_pm_3.shape[1]):
                self.pre_grad_pm_3[0,i] =np.maximum(self.pre_grad_pm_3[0,i],gt2[0,i])
                self.pre_grad_pm_3[1,i] =np.maximum(self.pre_grad_pm_3[1,i],gt2[1,i])

            for i in range(grad_pm.shape[1]):
                grad_pm[0,i] =
(self.learning_rate/(np.sqrt(self.pre_grad_pm_3[0,i])+eps))*gt[0,i]
                grad_pm[1,i] =
(self.learning_rate/(np.sqrt(self.pre_grad_pm_3[1,i])+eps))*gt[1,i]
            self.update_weigths(grad_pm)


        else:
            raise Exception('Unknown optimizer : \'{}\''.format(self.optimizer))

        return loss

    def update_weigths(self,grad_list):

        for i in range(len(self.shape)-1):
            self.wplus[i] = copy.deepcopy(np.clip(self.wplus[i] -
grad_list[0][i],a_min=0.001,a_max=None))
            self.wminus[i] = copy.deepcopy(np.clip(self.wminus[i] -
grad_list[1][i],a_min=0.001,a_max=None))

    def softmax(self,xs):
      return np.exp(xs) / sum(np.exp(xs))
```

**APPENDIX B:** Python Implementation of Simple Recurrent Neural Network.

```
# -*- coding: utf-8 -*-
import numpy as np
import copy
from statistics import mean
from math import sqrt

class SimpleRNN:
    def __init__(self,
                 layer_list,
                 time_steps,
                 weight_scaler = 1,
                 firing_rate_scaler = 0,
                 learning_rate = 0.1,
                 loss_function = 'mse',
                 optimizer = 'sgd'):

        #Assign basic parameters
        self.optimizer = optimizer
        self.learning_rate = learning_rate
        self.shape = layer_list
        self.time_steps = time_steps
        self.loss_func = loss_function
        n = len(layer_list)
        self.dh_list = []
        self.gradient_list=[]

        #Initialize layers
        self.layers = []
        for i in range(n):
            self.layers.append(np.zeros(self.shape[i]))

        #Q and D are initiliazed with zeros
        self.Q_intime = []
        for t in range(self.time_steps+1):
            Q = []
            for i in range(n):
                Q.append(np.zeros((self.shape[i],1)))
            self.Q_intime.append(copy.deepcopy(Q))

        # Initialize vertical weights: W_vertical
        self.W_ver = []
        for i in range(n-1):
            self.W_ver.append(np.zeros((self.shape[i],self.shape[i+1])))
            self.gradient_list.append(np.zeros((self.shape[i],self.shape[i+1])))


        #Initialize horizontal wieghts: W_hor
        self.W_hor = np.zeros((self.shape[1],self.shape[1]))
        self.gradient_list.append(np.zeros((self.shape[1],self.shape[1])))

        self.bh = np.zeros((self.shape[1], 1))
        self.gradient_list.append(np.zeros((self.shape[1], 1)))

        self.by = np.zeros((self.shape[2], 1))
        self.gradient_list.append(np.zeros((self.shape[2], 1)))

        self.init_weights(weight_scaler)
        self.bp_counter = 0
        self.gradient_list_2=copy.deepcopy(self.gradient_list)
        self.gradient_list_3=copy.deepcopy(self.gradient_list)

    def init_weights(self,weight_scaler):

        for i in range(len(self.W_ver)):
            self.W_ver[i] =
weight_scaler*np.random.rand(self.W_ver[i].shape[0],self.W_ver[i].shape[1])
        self.W_hor =
weight_scaler*np.random.rand(self.W_hor.shape[0],self.W_hor.shape[1])

        return 1

    def feedforward(self,input_list):

        t=0
```

```
        for t,input_t in enumerate(input_list,start = 1):

            self.Q_intime[t][0]=np.array(input_t).reshape(-1,1)
            self.Q_intime[t][1]=np.tanh(self.W_ver[0].transpose() @
self.Q_intime[t][0] +
                    self.W_hor.transpose() @ self.Q_intime[t-1][1] +
                    self.bh)

        self.Q_intime[t][2] = self.W_ver[1].transpose()@self.Q_intime[t][1] + self.by

        return copy.deepcopy(self.Q_intime[t][2])

    def dh_dfi(self,t):
        return 1-self.Q_intime[t][1]**2

    def backpropagation(self,real_output,tmp1):
        self.bp_counter=self.bp_counter+1

        if self.optimizer=='nag':

weights=np.asarray([self.W_ver[0],self.W_ver[1],self.W_hor,self.bh,self.by])

            x_ahead = weights-np.asarray(self.gradient_list)*0.9

            self.W_ver[0] = copy.deepcopy(x_ahead[0])
            self.W_ver[1] = copy.deepcopy(x_ahead[1])
            self.W_hor = copy.deepcopy(x_ahead[2])
            self.bh = copy.deepcopy(x_ahead[3])
            self.by = copy.deepcopy(x_ahead[4])

        loss = 0

        d_Wih = []
        d_Whh = []
        d_Who= []
        d_bh= []
        d_by = []

        #tmp1 = 1

        d_Who.append(tmp1*self.Q_intime[self.time_steps][1])
        d_by.append(tmp1)

        d_hidden_layer = copy.deepcopy(tmp1*self.W_ver[1])
        der_chain = d_hidden_layer * self.dh_dfi(self.time_steps)

        for t in reversed(range(1,self.time_steps+1)):#2 1 0

            d_Wih.append((der_chain @ self.Q_intime[t][0].transpose()).transpose())
            d_Whh.append((der_chain @ self.Q_intime[t][1].transpose()).transpose())
            d_bh.append(der_chain)
            self.dh_list.append(der_chain)
            der_chain = self.W_hor @ (self.dh_dfi(t-1) * der_chain)

        #dh_list.clear()
        #Create final gradients
        gradient_Wih = sum(d_Wih)
        gradient_Who = sum(d_Who)
        gradient_Whh = sum(d_Whh)
        gradient_bh = sum(d_bh)
        gradient_by = sum(d_by)

        for d in [gradient_Wih, gradient_Whh, gradient_Who, gradient_bh,
gradient_by]:
          np.clip(d, -1, 1, out=d)


grads=np.asarray([gradient_Wih,gradient_Who,gradient_Whh,gradient_bh,gradient_by])
        if self.optimizer == 'sgd':
            self.update_weigths(grads*self.learning_rate)

        elif self.optimizer == 'momentum':
            moment = np.asarray(self.gradient_list)*0.9
            grads = moment + grads*self.learning_rate
            self.update_weigths(grads)
            self.gradient_list=copy.deepcopy(grads)
```

```
        elif self.optimizer == 'nag':
            self.gradient_list = np.asarray(self.gradient_list)*0.9 +
grads*self.learning_rate
            self.update_weigths(self.gradient_list)

        elif self.optimizer == 'adagrad':
            grads_2=np.square(grads)
            self.gradient_list+=copy.deepcopy(grads_2)
            for i in range(len(grads)):
                grads[i] =
(self.learning_rate/(np.sqrt(self.gradient_list[i]+0.000001)))*grads[i]
            self.update_weigths(grads)

        elif self.optimizer == 'adadelta':

            eps=0.000001;beta=0.90;
            grads_2 = np.square(grads)
            self.gradient_list_2 = beta*np.asarray(self.gradient_list_2) + (1-
beta)*grads_2

            delta_teta = copy.deepcopy(self.gradient_list)
            for i in range(len(grads)):
                delta_teta[i] =
(np.sqrt(self.gradient_list[i]+0.000001)/(np.sqrt(self.gradient_list_2[i]+0.000001)))
*grads[i]

            self.gradient_list = beta*np.asarray(self.gradient_list) + (1-
beta)*np.square(delta_teta)

            for i in range(len(grads)):
                grads[i] =
(np.sqrt(self.gradient_list[i]+0.000001)/(np.sqrt(self.gradient_list_2[i]+0.000001)))
*grads[i]

            self.update_weigths(grads)

        elif self.optimizer == 'rmsprop':
            eps=0.00000001
            grads_2 = np.square(grads)
            self.gradient_list = 0.9*np.asarray(self.gradient_list) + 0.1*grads_2

            for i in range(len(grads)):
                grads[i] = (self.learning_rate / np.sqrt(self.gradient_list[i]+eps))
* grads[i]
            self.update_weigths(grads)
        else:
            raise Exception('Unknown optimizer : \'{}\''.format(self.optimizer))

        return loss
    def update_weigths(self,grads):
        self.W_ver[0] = copy.deepcopy(self.W_ver[0] - grads[0])
        self.W_ver[1] = copy.deepcopy(self.W_ver[1] - grads[1])
        self.W_hor = copy.deepcopy(self.W_hor - grads[2])
        self.bh = copy.deepcopy(self.bh - grads[3])
        self.by = copy.deepcopy(self.by - grads[4])
    def softmax(self,xs):
      return np.exp(xs) / sum(np.exp(xs))
```

**APPENDIX C:** Python Implementation of Queueing Recurrent Neural Network.

```
# -*- coding: utf-8 -*-
import numpy as np
import copy
from statistics import mean
from math import sqrt

class QRNN:
    def __init__(self,
                 layer_list,
                 time_steps = 0,
                 weight_scaler = 1,
                 firing_rate_scaler = 0,
                 learning_rate = 0.1,
                 loss_function = 'mse',
                 optimizer = 'sgd'):

        #Assign basic parameters
        self.dh_list = []
        self.learning_rate = learning_rate
        self.shape = layer_list
        self.time_steps = time_steps
        self.loss_func = loss_function
        self.optimizer = optimizer
        n = len(layer_list)
        self.gradient_list=[]
        #Initialize layers
        self.layers = []
        for i in range(n):
            self.layers.append(np.zeros(self.shape[i]))

        #Initialize rates for horizontal and vertical
        self.rate = [] #Horizontal rates : sum of horizontal weights
        for i in range(n):
            self.rate.append(np.zeros((self.shape[i],1)))
        self.rate[-1] = firing_rate_scaler * np.ones(self.rate[-1].shape)

        self.rate_h = np.zeros((self.shape[1],1)) #Vertical rates : sum of Vertical
weights
        # self.rate.shape = (H,)

        #Q and D are initiliazed with zeros
        self.Q_intime = []
        self.D_intime = []
        if self.time_steps != 0:
            for t in range(self.time_steps+1):
                Q = []
                for i in range(n):
                    Q.append(np.zeros((self.shape[i],1)))
                self.Q_intime.append(copy.deepcopy(Q))
            self.D_intime = copy.deepcopy(self.Q_intime)

        # Initialize vertical wegihts: wplus, wminus
        self.wplus = []
        self.wminus = []
        for i in range(n-1):
            self.wplus.append(np.zeros((self.layers[i].size,
                                        self.layers[i+1].size)))
            self.gradient_list.append(np.zeros((self.layers[i].size,
                                        self.layers[i+1].size)))
            self.wminus.append(np.zeros((self.layers[i].size,
                                        self.layers[i+1].size)))
            self.gradient_list.append(np.zeros((self.layers[i].size,
                                        self.layers[i+1].size)))

        #Initialize horizontal weights: wplus_h, wminus_h
        self.wplus_h = np.zeros((self.layers[1].size,
                                 self.layers[1].size))
        self.gradient_list.append(np.zeros((self.layers[i].size,
                                        self.layers[i].size)))
        self.wminus_h = np.zeros((self.layers[1].size,
                                  self.layers[1].size))
        self.gradient_list.append(np.zeros((self.layers[i].size,
                                        self.layers[i].size)))
```

```
        #Initialize lambdas : lambda_plus, lambda_minus
        self.lambda_plus = []
        self.lambda_minus = []

        self.global_var2 = []
        self.global_var = []
        self.init_weights(weight_scaler)
        self.gradient_list_2=copy.deepcopy(self.gradient_list)
        self.gradient_list_3=copy.deepcopy(self.gradient_list)

        self.bp_counter=0

    def recreate_Q_intime(self,step):
        #Q and D are initiliazed with zeros
        self.Q_intime = []
        self.D_intime = []
        for t in range(step+1):
            Q = []
            for i in range(len(self.shape)):
                Q.append(np.zeros((self.shape[i],1)))
            self.Q_intime.append(copy.deepcopy(Q))
        self.D_intime = copy.deepcopy(self.Q_intime)
        return 1

    def init_weights(self,weight_scaler):

        for i in range(len(self.wplus)):
            self.wplus[i] =
weight_scaler*np.random.rand(self.wplus[i].shape[0],self.wplus[i].shape[1])
            self.wminus[i] =
weight_scaler*np.random.rand(self.wminus[i].shape[0],self.wminus[i].shape[1])

        self.wplus_h =
weight_scaler*np.random.rand(self.wplus_h.shape[0],self.wplus_h.shape[1])
        self.wminus_h =
weight_scaler*np.random.rand(self.wminus_h.shape[0],self.wminus_h.shape[1])

        return 1

    def calculate_rate(self):
        for layer_num in range(len(self.layers)-1): #there is no rate for output
layer
            for neuron in range(len(self.layers[layer_num])):
                self.rate[layer_num][neuron] = (self.wplus[layer_num][neuron] +
self.wminus[layer_num][neuron]).sum()

        for neuron in range(len(self.layers[1])):
            self.rate_h[neuron] = (self.wplus_h[neuron] +
self.wminus_h[neuron]).sum()

        return 1

    def feedforward(self,input_list):
        #self.recreate_Q_intime()
        t=0
        if self.time_steps==0:
            self.recreate_Q_intime(input_list.shape[0])
        for t,input_t in enumerate(input_list,start = 1):

            #self.D.clear() #clear list of D matrixes for the next iteration
            self.lambda_plus = np.where(input_t > 0, input_t, 0).reshape(-1,1)
            self.lambda_minus = np.where(input_t < 0, -input_t, 0).reshape(-1,1)

            # Input Layer
            self.D_intime[t][0] = self.rate[0]+self.lambda_minus
            self.Q_intime[t][0] = self.lambda_plus/self.D_intime[t][0]
            np.clip(self.Q_intime[t][0],0,1,out=self.Q_intime[t][0])

            # Hidden Layer
            T_plus_i = self.wplus[0].transpose() @ self.Q_intime[t][0]
            T_minus_i =  self.wminus[0].transpose() @ self.Q_intime[t][0]
            T_plus_h = self.wplus_h.transpose() @ self.Q_intime[t-1][1]
            T_minus_h =  self.wminus_h.transpose() @ self.Q_intime[t-1][1]
            if t == input_list.shape[0]:
                self.D_intime[t][1] = self.rate[1] + T_minus_i + T_minus_h
            else:
                self.D_intime[t][1] = self.rate_h + T_minus_i + T_minus_h
```

```
            self.Q_intime[t][1] = (T_plus_i + T_plus_h)/(self.D_intime[t][1])
            np.clip(self.Q_intime[t][1],0,1,out=self.Q_intime[t][1])

        # Output Layer
        T_plus = self.wplus[1].transpose() @ self.Q_intime[t][1]
        T_minus = self.wminus[1].transpose() @ self.Q_intime[t][1]
        self.D_intime[t][2] = self.rate[2] + T_minus
        self.Q_intime[t][2] = T_plus / self.D_intime[t][2]
        np.clip(self.Q_intime[t][2] ,0,1,out=self.Q_intime[t][2])

        return copy.deepcopy(self.Q_intime[t][2])

    def vertical_gradient(self,t,l): #t:timestep, l=layer

        vergrad2 = self.wplus[l].transpose()-self.Q_intime[t][l+1].reshape(-
1,1)*self.wminus[l].transpose()
        vergrad3 = vergrad2/self.D_intime[t][l+1].reshape(-1,1)

        return vergrad3.transpose()

    def horizontal_gradient(self,t):
        hor_grad2 = self.wplus_h.transpose()-self.Q_intime[t][1].reshape(-
1,1)*self.wminus_h.transpose()
        hor_grad3 = hor_grad2/self.D_intime[t][1].reshape(-1,1)

        return hor_grad3.transpose()

    def backpropagation(self,real_output,tmp1=np.zeros((2,1))):
        self.bp_counter=self.bp_counter+1

        if self.optimizer=='nag':

weights=np.asarray([self.wplus[0],self.wminus[0],self.wplus[1],self.wminus[1],self.wp
lus_h,self.wminus_h])

            x_ahead = weights-np.asarray(self.gradient_list)*0.9

            self.wplus[0] = copy.deepcopy(np.clip(x_ahead[0],a_min=0.001,a_max=None))
            self.wminus[0] =
copy.deepcopy(np.clip(x_ahead[1],a_min=0.001,a_max=None))
            self.wplus[1] = copy.deepcopy(np.clip(x_ahead[2],a_min=0.001,a_max=None))
            self.wminus[1] =
copy.deepcopy(np.clip(x_ahead[3],a_min=0.001,a_max=None))
            self.wplus_h = copy.deepcopy(np.clip(x_ahead[4],a_min=0.001,a_max=None))
            self.wminus_h = copy.deepcopy(np.clip(x_ahead[5],a_min=0.001,a_max=None))

        d_Who_p = []
        d_Who_m = []
        d_Whh_p = []
        d_Whh_m = []
        d_Wih_p = []
        d_Wih_m = []

        d_Who_p.append(((self.Q_intime[-1][1] @ (1/self.D_intime[-
1][2]).transpose()).transpose()*tmp1).transpose())
        d_Who_m.append((-(self.Q_intime[-1][1] @ (self.Q_intime[-
1][2]/self.D_intime[-1][2]).transpose()).transpose()*tmp1).transpose())

        steps = len(self.Q_intime)-1
        o3_h3 = self.vertical_gradient(steps,1)

        d_hidden_layer = o3_h3

        for t in reversed(range(1,steps+1)):#2 1 0
            if t==steps:
                d_whop = d_hidden_layer*(self.Q_intime[t][1]/self.D_intime[t][1])
                d_whom = copy.deepcopy(d_whop)
                d_Who_p.append((d_whop.transpose()*tmp1).transpose())
                d_Who_m.append((d_whom.transpose()*tmp1).transpose())
                d_hidden_layer = d_hidden_layer@tmp1
            ################## Gradients from hidden layers #################
            #dqh/dNi
            d_hidden_layer_N = d_hidden_layer/self.D_intime[t][1]
            #dqh/dDi
            d_hidden_layer_D = d_hidden_layer*(-
self.Q_intime[t][1]/self.D_intime[t][1])
```

```
            wihm = self.Q_intime[t][0] @ d_hidden_layer_D.transpose()
            wihp = self.Q_intime[t][0] @ d_hidden_layer_N.transpose()
            if t==steps:
                whhm = self.Q_intime[t-1][1] @ d_hidden_layer_D.transpose()
                whhp = self.Q_intime[t-1][1] @ d_hidden_layer_N.transpose()
            else :
                whhm = (1+self.Q_intime[t-1][1]) @ d_hidden_layer_D.transpose()
                whhp_n = self.Q_intime[t-1][1] @ d_hidden_layer_N.transpose()
                whhp_d = d_hidden_layer_D
                whhp = whhp_n + whhp_d

            d_Whh_p.append(whhp)
            d_Whh_m.append(whhm)
            d_Wih_p.append(wihp)
            d_Wih_m.append(wihm)

            ################### Gradients from input layers ###################
            #dqh/dqi
            d_input_layer = self.vertical_gradient(t,0)
            #dqi/dDi
            d_input_layer_D = d_input_layer*(-
(self.Q_intime[t][0]/self.D_intime[t][0]))

d_Wih_p.append((d_input_layer_D.transpose()*d_hidden_layer_D).transpose())

d_Wih_m.append((d_input_layer_D.transpose()*d_hidden_layer_D).transpose())

            ######################## New Hidden Layer ########################
            self.dh_list.append(d_hidden_layer)
            d_hidden_layer = self.horizontal_gradient(t) @ d_hidden_layer

        #dh_list.clear()

#weights=np.asarray([self.wplus[0],self.wminus[0],self.wplus[1],self.wminus[1],self.w
plus_h,self.wminus_h])

        if self.optimizer == 'sgd':

grads=np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),su
m(d_Whh_m)])*self.learning_rate
            self.update_weigths(grads)

        elif self.optimizer == 'momentum':
            moment = np.asarray(self.gradient_list)*0.9
            grads = moment +
np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),sum(d_Wh
h_m)])*self.learning_rate
            self.update_weigths(grads)
            self.gradient_list=copy.deepcopy(grads)

        elif self.optimizer == 'nag':
            grads =
np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),sum(d_Wh
h_m)])
            self.gradient_list = np.asarray(self.gradient_list)*0.9 +
grads*self.learning_rate
            self.update_weigths(self.gradient_list)

        elif self.optimizer == 'adagrad':
            grads =
np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),sum(d_Wh
h_m)])
            grads_2=np.square(grads)
            self.gradient_list+=copy.deepcopy(grads_2)
            for i in range(len(grads)):
                grads[i] =
(self.learning_rate/(np.sqrt(self.gradient_list[i]+0.000001)))*grads[i]
            self.update_weigths(grads)
            #self.gradient_list+=copy.deepcopy(grads_2)

        elif self.optimizer == 'adadelta':
            eps=0.000001;beta=0.90;

grads=np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),su
m(d_Whh_m)])

            grads_2 = np.square(grads)
```

```
            self.gradient_list_2 = beta*np.asarray(self.gradient_list_2) + (1-
beta)*grads_2

            delta_teta = copy.deepcopy(self.gradient_list)
            for i in range(len(grads)):
                #delta_teta[i] =
(self.learning_rate/(np.sqrt(self.gradient_list_2[i]+0.000001)))*grads[i]
                delta_teta[i] =
(np.sqrt(self.gradient_list[i]+0.000001)/(np.sqrt(self.gradient_list_2[i]+0.000001)))
*grads[i]

            self.gradient_list = beta*np.asarray(self.gradient_list) + (1-
beta)*np.square(delta_teta)

            for i in range(len(grads)):
                grads[i] =
(np.sqrt(self.gradient_list[i]+0.000001)/(np.sqrt(self.gradient_list_2[i]+0.000001)))
*grads[i]

            self.update_weigths(grads)
            #self.gradient_list = beta*np.asarray(self.gradient_list) + (1-
beta)*np.square(grads)

        elif self.optimizer == 'rmsprop':
            eps=0.00000001

grads=np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),su
m(d_Whh_m)])
            grads_2 = np.square(grads)
            self.gradient_list = 0.9*np.asarray(self.gradient_list) + 0.1*grads_2

            for i in range(len(grads)):
                grads[i] = (self.learning_rate / np.sqrt(self.gradient_list[i]+eps))
* grads[i]
            self.update_weigths(grads)

        elif self.optimizer == 'adam':
            eps=0.000001;beta1=0.9;beta2=0.999;


grads=np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),su
m(d_Whh_m)])
            grads_2 = np.square(grads)
            self.gradient_list = beta1*np.asarray(self.gradient_list) + (1-
beta1)*grads
            self.gradient_list_2 = beta2*np.asarray(self.gradient_list_2) + (1-
beta2)*grads_2
            gt = self.gradient_list/(1-beta1**self.bp_counter)
            gt2 = self.gradient_list_2/(1-beta2**self.bp_counter)

            for i in range(len(grads)):
                grads[i] = (self.learning_rate / (np.sqrt(gt2[i])+eps)) * gt[i]
            self.update_weigths(grads)

        elif self.optimizer == 'adamax':
            eps=0.000001;beta1=0.9;beta2=0.999;


grads=np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),su
m(d_Whh_m)])
            grads_2 = np.square(grads)
            self.gradient_list = beta1*np.asarray(self.gradient_list) + (1-
beta1)*grads
            gt = self.gradient_list/(1-beta1**self.bp_counter)

            for i in range(len(self.gradient_list_2)):
                self.gradient_list_2[i]
=np.maximum(beta2*np.asarray(self.gradient_list_2)[i],abs(grads)[i])
            #learning rate is 0.002 as an adviced value
            grads = (self.learning_rate / (np.asarray(self.gradient_list_2)+eps)) *
gt
            self.update_weigths(grads)

        elif self.optimizer == 'nadam':
            eps=0.000001;beta1=0.9;beta2=0.999;
```

```
grads=np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),su
m(d_Whh_m)])
            grads_2 = np.square(grads)
            self.gradient_list = beta1*np.asarray(self.gradient_list) + (1-
beta1)*grads
            self.gradient_list_2 = beta2*np.asarray(self.gradient_list_2) + (1-
beta2)*grads_2
            gt = self.gradient_list/(1-beta1**self.bp_counter)
            gt2 = self.gradient_list_2/(1-beta2**self.bp_counter)
            gt = (beta1*gt-((1-beta1)/(1-beta1**self.bp_counter))*gt)
            for i in range(len(grads)):
                grads[i] = (self.learning_rate / (np.sqrt(gt2[i])+eps)) * gt[i]
            self.update_weigths(grads)

        elif self.optimizer == 'amsgrad':
            eps=0.000001;beta1=0.9;beta2=0.999;


grads=np.asarray([sum(d_Wih_p),sum(d_Wih_m),sum(d_Who_p),sum(d_Who_m),sum(d_Whh_p),su
m(d_Whh_m)])
            grads_2 = np.square(grads)
            self.gradient_list = beta1*np.asarray(self.gradient_list) + (1-
beta1)*grads
            self.gradient_list_2 = beta2*np.asarray(self.gradient_list_2) + (1-
beta2)*grads_2
            gt = self.gradient_list/(1-beta1**self.bp_counter)
            gt2 = self.gradient_list_2/(1-beta2**self.bp_counter)

            for i in range(len(self.gradient_list_2)):
                self.gradient_list_3[i]
=np.maximum(np.asarray(self.gradient_list_3)[i],gt2[i])

            for i in range(len(grads)):
                grads[i] = (self.learning_rate /
(np.sqrt(self.gradient_list_3[i])+eps)) * gt[i]
            self.update_weigths(grads)

        else:
            raise Exception('Unknown optimizer : \'{}\''.format(self.optimizer))

        return 1

    def update_weigths(self,grads):
        self.wplus[0] = copy.deepcopy(np.clip(self.wplus[0] -
grads[0],a_min=0.001,a_max=None))
        self.wminus[0] = copy.deepcopy(np.clip(self.wminus[0] -
grads[1],a_min=0.001,a_max=None))
        self.wplus[1] = copy.deepcopy(np.clip(self.wplus[1] -
grads[2],a_min=0.001,a_max=None))
        self.wminus[1] = copy.deepcopy(np.clip(self.wminus[1] -
grads[3],a_min=0.001,a_max=None))
        self.wplus_h = copy.deepcopy(np.clip(self.wplus_h -
grads[4],a_min=0.001,a_max=None))
        self.wminus_h = copy.deepcopy(np.clip(self.wminus_h -
grads[5],a_min=0.001,a_max=None))

    def softmax(self,xs):
      return np.exp(xs) / sum(np.exp(xs))
```