Univerzitet u Beogradu

Elektrotehnički fakultet

Predviđanje vremenskih serija upotrebom rekurentnih neuralnih mreža

Master rad

Mentor:

Doc.dr Predrag Tadić

Kandidat:

Ksenija Stepanović 3072/2018

Beograd, Septembar 2019

University of Belgrade

School of Electrical Engineering

Time Series Prediction Using Recurrent Neural Networks

Master thesis

Supervisor:

Doc.dr Predrag Tadić

Candidate:

Ksenija Stepanović 3072/2018

Belgrade, September 2019

Contents

Abstract

The application of deep learning approaches to important issues in daily life has always attracted great curiosity and attention. This thesis represents a deep learning framework where recurrent neural network architecture with long short term memory units has been utilized to develop predictive models for stock price prediction and anomaly detection in an electrocardiogram signal. A lot of work has already been done on these two tasks because of the benefits they bring to society. The forecasting of the opening and adjusted closing stock price features one day, two days and three days in advance, based on the previous days information, has been performed. The four different size companies' data has been downloaded from the *Yahoo Finance* website. The second application of deep learning techniques is to indicate the normal or abnormal behavior in a heartbeat electrical signal obtained from the MIT-BIH arrhythmia database. It contains anomaly binary classification and anomaly multiclass classification, where four different types of arrhythmia should be detected and classified. Results are promising and through experiments on stock price prediction and anomaly detection in electrocardiogram signals, they confirm that long short term memory networks are viable for time-series prediction and have a lot to contribute to the solving problems from meaningful fields.

Rezime

Aplikacije metoda dubokog učenja na bitne probleme iz svakodnevnog života su oduvek privlačile veliku radoznalost i pažnju. Ova teza predstavlja okvir dubokog učenja gde se rekurentne neuralne mreže sa dugoročnim memorijskim jedinicama koriste za razvoj prediktivnih modela koji se tiču predikcije cena akcija na tržištu i detekciju anomalija u okviru signala elektrokardiograma. Mnogo rada je već uloženo u rešavanje ova dva problema zbog koristi koju oni donose društvu. Predviđanje cene otvaranja i prilagođene cene zatvaranja sutradan, dva dana i tri dana unapred na osnovu informacija iz prethodnih dana je pokušano u ovom radu. Podaci četiri kompanije različite veličine su preuzeti sa *Yahoo Finance* internet stranice. Druga primena dubokog učenja je ukazivanje na normalno ili abnormalno ponašanje u električnom signal otkucaja srca, koji su dobijeni iz MIT-BIH baze podataka. Ova primena sadrži binarnu klasifikaciju anomalija i klasifikaciju anomalija u više klasa, gde je trebalo detektovati i klasifikovati četiri različite vrste aritmije. Rezultati su obećavajući i eksperimentima predviđanja cena akcija na tržištu i detekcije anomalija u signalu elektrokardiograma je potvrđeno da su dugoročne memorijske mreže pogodne za izvlačenje smislenih zaključaka iz vremenskih serija i da mogu značajno da doprinesu rešavanju bitnih problema.

1. Introduction

In many real-world applications, the data is recorded at particular periods or intervals over the course of time, forming a time-series. Examples of time-series data include yearly sales figures, hourly readings of air humidity and temperature, daily recordings of the company's closing stock price feature and continuous monitoring of a person's heart rate. Every object in the world produces the data all the time. Recently, that data has been started to be recorded and explored. Long short term memory networks (LSTMs) have been proven to be particularly useful in capturing meaningful information from time-series data by discovering temporal dependencies between the data indexed by time.

The application of LSTMs in finance and health industries has received a great deal of attention because of their importance in everyday life. The motivation of this work is drawing useful predictions from two time-series datasets related to the stock market and electrocardiogram (ECG) time signal. Stock price prediction is considered as a challenging task among time-series forecasting due to the noisy data nature [1]. The daily stock price record consists of opening price, the highest price, the lowest price, closing price, adjusted closing price and volume features. The dataset has been downloaded from the *Yahoo Finance* website [2] for four different companies and it consists of records for every day since the company's day of establishment. The first goal of this thesis is successful forecasting of opening and adjusted closing stock price features one day, two days and three days in advance for each of four companies. Electrocardiography is the process of recording an electrical signal of the heart over a period of time by placing electrodes on the patient's body [3]. The MIT-BIH arrhythmia dataset [4, 5] has been used for the heartbeat's anomaly detection and anomaly type detection. The dataset contains recordings of two electrical signals measured by two electrodes and annotations for each heartbeat's recording.

Python programming language (version 3.6) was used to write the code for the project. The main software libraries used and their versions are TensorFlow (version 1.14.0), Keras (version 2.2.4), sklearn (version 0.0) and numpy (version 1.17.0).

During past decades, machine learning algorithms, such as artificial neural networks (ANNs) [6] and support vector regression (SVR) [7] have been widely applied on stock price forecasting problems. In [8], backward propagation and recurrent neural networks (RNNs) have been used for prediction of the stock index for five different stock markets. In [9], PSO and LS-SVM machine learning algorithms have been used for the prediction of S&P 500 stock market. Genetic algorithms have also been challenged on financial data analysis [10]. However, like ANNs they have performed poorly due to the incapability of remembering long-term dependencies. With the introduction of LSTMs, financial time-

series data analysis has become more efficient. LSTM's efficiency in stock price forecasting has been demonstrated in [11, 12, 13, 14] on various datasets.

Extensive research has also been conducted in the area of arrhythmia detection, firstly by using machine learning algorithms and later on by employing deep learning. Data from a single lead was used for an anomaly detection achieving an accuracy of 94.74% [15]. Multiscale eigenspace analysis was carried out on 12 lead ECG data to achieve the same objective with an accuracy of 96% [16]. Deep learning has been proven to obtain the highest prediction accuracy. In [17], convolutional neural networks (CNNs) have reached an accuracy of 95.22% in automated anomaly detection. In [18], four types of arrhythmia from MIT-BIH arrhythmia database have been classified with an accuracy of 99.38%. The accuracy of 83.7% has been achieved in [19] by combining three-layer CNN with the LSTM network.

The thesis has been divided into five parts. Methodology section provides background material on the history of neural networks development, multilayer perceptron, description of recurrent neural networks structure and mathematical representation of forward and backward propagation. The need for LSTM networks development has also been explained. In the Methodology section, optimization and generalization methods for efficient neural networks training and faster convergence, evaluation metrics and ways for weights initialization have been covered. Section Experiment setup contains *Yahoo Finance* and MIT – BIH arrhythmia dataset descriptions. The neural network's architecture and values of network's parameters have also been described in this section. Section Results investigates the use of LSTMs on stock price prediction and anomaly classification problems in a heartbeat electrical signal. It offers visualization of achieved results. In the Discussion section improvements for proposed methods have been offered. Moreover, achieved results have been compared to the previous works. The conclusion is given in the last section.

2. Methodology

2.1 Data preprocessing

Data preprocessing may be a crucial step for successful neural network performance. In this case, it implies scaling the features to a range which is centered around zero. Data preprocessing prevents that the feature with a greater variance dominates machine learning algorithm and that algorithm drives conclusions mainly based on that feature. Two common applied feature scaling techniques used for preprocessing the data are:

1) Standardization is the transformation that centers the data by subtracting the mean value of each feature and then dividing each feature with their standard deviation:
   $$x_i\_scaled = \frac{x_i - mean(x)}{std(x)} \quad (2.1.1)$$
   After standardizing the data mean value is zero and standard deviation is one.
2) Normalization is the transformation that scales data in the range between zero and one or if there are negative values in the range between -1 and +1:
   $$x_i\_scaled = \frac{x_i - min(x)}{max(x) - \min(x)} \quad (2.1.2)$$
   Normalization is applied if the data doesn't have Gaussian distribution or if the data standard deviation is small. However, this method is sensitive to outliers.

Preprocessing will enable neural networks to make more accurate predictions.

2.2 Neural networks

2.2.1 History of neural networks

This section provides an overview of history of neural networks (NNs) during the last 80 years. It covers the birth of NNs with the perceptron in 1958, the artificial intelligence winter of the 70s, NNs return to popularity with the backpropagation in 1986, foundation of deep learning in the late 90s and progress it has been made since.

The first NN has been presented as a simple electrical circuit by neurophysiologist Warren McCulloch and mathematician Walter Pitts [20]. The first Hebbian network was successfully created at

Massachusetts Institute of Technology in 1954, as scientists tried to transform operations in biological systems into computational theory. Frank Rosenblatt, a psychologist at Cornell, proposed the idea of a perceptron in 1958 while analyzing the decision system presented in an eye of a fly, and named it Mark I Perceptron [21]. In Figure 2.2.1 Mark I Perceptron is shown [60].

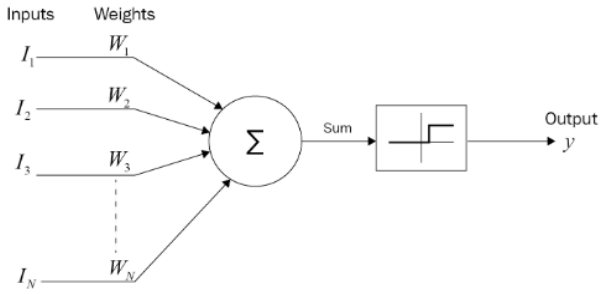

Figure2.2.1.A Mark I Perceptron. Image adapted from [45].

The idea of Mark I Perceptron is that its weights would be 'learned' through successively passed inputs while minimizing the difference between desired and actual output. However, this perceptron could only learn to separate linearly separable classes [60].

The main problems of NNs in this time were impractically long runtimes and inability to learn simple boolean exclusive-or circuits. Publication of the book *Perceptrons* by Marvin Minsky and Seymour Papert [22] made significant influence on the development of NNs in the next years. The book argued that the Rosenblatt's single perceptron approach to NNs could not be translated effectively into multi-layered NN. The book effectively led the scientific community and the funding establishments to the conclusion that there is no sense for continuing research on NNs. The effect of this text was powerful and dried up funding for the next 10–12 years, so it was impractical to conduct a research that has NNs as its premise. This age is known as artificial intelligence winter [60].

Backpropagation, a method devised by researchers since the 60's and continuously developed during artificial intelligence winter was an algorithm that led to the winter's end. Paul Werbos was the first person to see potential in backpropagation on NNs training. He wrote a PhD thesis expounding backpropagation importance and later on published a report on his work in 1985 [23]. It was only after being re-re-discovered by Geoffrey Hinton that the technique reawakened the interest of the scientific community in NNs [60].

The NNs, between its multiple applications, started taking place in image processing and speech recognition. Convolutional neural networks (CNNs), developed by Yann LeCun, became essential for image processing as recurrent neural networks (RNNs) became essential for speech recognition. RNNs represent the solution to the problem of giving the network memory by connecting the output of the last layer as an input to the first layer or just connecting the output of a neuron to itself. In Figure 2.2.2 simple RNN with one hidden layer is represented [46].

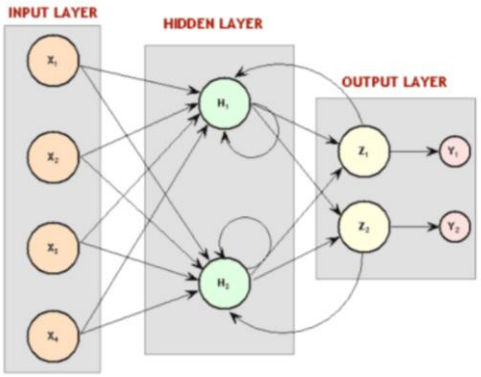

Figure 2.2.2.Diagram of recurrent neural network. Image adapted from [46].

As backpropagation relies on propagation the error from the output layer backward, the error in this case would go ahead and propagate from the first layer back to the output layer, and would just keep looping through the network, infinitely. The solution was backpropagation through time. The idea was to unroll the RNN by treating each loop through the NN as an input to another NN, and looping only a limited number of times as shown in Figure 2.2.3 [46].

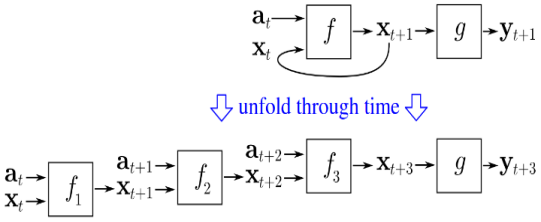

Figure 2.2.3.Concept of backpropagation through time. Image adapted from [46].

Although it was possible to train RNN on this way, it still didn't perform successfully. In 1993 Yoshua Bengio wrote the paper "A Connectionist Approach to Speech Recognition" and in the following sentence summarized the cause of failure of RNN: "Our experiments tended to indicate that their parameters settle in a suboptimal solution which takes into account short term dependencies but not long term dependencies*"* [24]. Mentioned problem is known as the problem of vanishing and exploding gradient where gradient propagating back through a network with a lot of layers tends to vanish or drastically increases as going further back [46].

In 1997 Jürgen Schmidhuber and Sepp Hochreiter introduced long short term memory concept [25] that seemed to solve the problem of vanishing and exploding gradient. Instead of every node representing only one RNN cell, in LSTM every node contains four cells that perform as gates, deciding on which information should be passed through them and candidate as the final output in that time step. However, twenty years ago computers have been too slow and RNN training required too much time and memory. Other emerging algorithms such as support vector machines (SVM) and random forests have been considered more suitable for solving problems [46].

The early 2000s represented the second winter for NNs. A small community led by Geoffrey Hinton, Yoshua Bengio and Yann LeCun continued research even though the papers on NNs have been constantly rejected and funding was modest and unstable. The name deep learning for NNs based approaches originates from this period. In 2006 Geoffrey Hinton, Simon Osindero and Yee-Whye Teh published the paper "A Fast Learning Algorithm for Deep Belief Nets" [26] that rekindled interest in NNs and marked a beginning of deep learning era. In this manner, weights have been initialized through unsupervised training of each layer instead of random initialization. They achieved performance of 1.25% test error on MNIST dataset using the above mentioned method, which was slightly better than 1.4% test error achieved using SVM. What is more important, the paper demonstrated that NNs can be trained efficiently [61].

As NNs have the potential of being trained on much more demanding datasets than MNIST, Geoffrey Hinton and two of his graduate students, Abdel-rahman Mohamed and George Dahl, tried them on more challenging task: Speech Recognition, breaking through previously achieved records. Hardware development backed up faster training, enabled usage of more training examples and building bigger NNs. In 2010 Jürgen Schmidhuber co-wrote the paper "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition" [27] where 0.35% error rate on MNIST test dataset has been achieved by using bigger NN, more training examples and GPU for efficient training. These accomplishments were enough to evoke interest of big companies, such as *Microsoft* and *Google*, in NNs and its application on their data [61].

Accordingly, Hinton's students started collaborating with *Microsoft*, *Google* and *IBM* and demonstrating power of NNs in Big Data and Speech Processing problems. Three research groups at these companies and one at the Hinton's laboratory published together a paper "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups" in 2012 presenting results to the larger community [28]. In 2011, Andrew Ng and Jeff Dean formed *Google Brain*, an experiment where NN with 1 billion weights was unsupervised trained on 16 000 CPU cores learning to recognize objects on *YouTube* videos without any labels [61].

The success of NNs arose a question: Why simple backpropagation didn't work? Unsupervised pre-training wasn't the only solution for preventing vanishing and exploding gradient problem. Careful weights initialization concerning the layer and right activation function showed to be efficient method also. Maintaining the sparse representation was important as Hinton demonstrated in the paper "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors" [29] where he introduced *Dropout* – a method that randomly sets outputs of certain neurons on zero. The final step in demonstrating power of deep learning was ILSVRC-2012 computer vision competition where CNN combined with recently emerged improvements (GPU training, *Dropout*, *ReLu* activation function) triumphed over machine learning algorithms with 12% difference in error rate [61].

Since then, constant improvements have been made. Deep learning has appealed to more and more companies. Some of the results achieved by using deep learning are shown in Figure 2.2.4 and Figure 2.2.5.

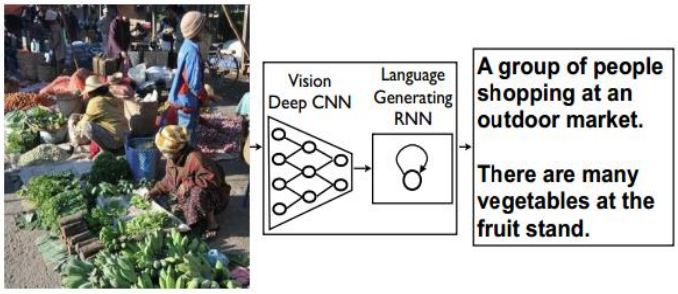

Figure2.2.4. Image into text conversion using convolutional neural networks and recurrent neural networks. Image adapted from [47].

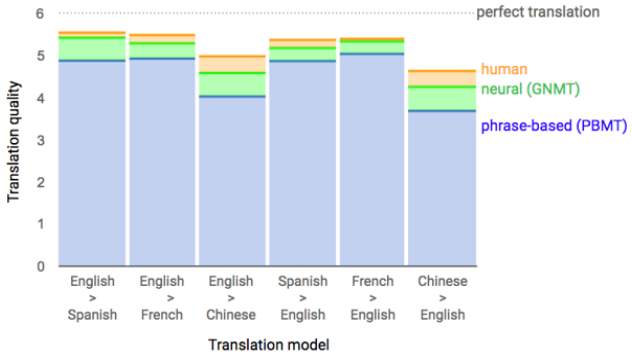

Figure 2.2.5. Google Translate results using recurrent neural networks. Image adapted from [48].

2.2.2 Multilayer perceptron

In this section Multilayer perceptron (MLP), the most widely used form of feedforward neural network (FNN), its structure, forward propagation and backward propagation have been introduced. This form of FNN is basis for NN understanding and remains in use till today.

MLP contains one input layer with number of nodes equivalent to number of data features, arbitrary number of hidden layers and one output layer. The MLP is illustrated in Figure 2.2.6. Firstly, weighted and summed data is propagated forward through the network. This is known as forward propagation. After that, an error between actual and desired output is calculated and weights are adjusted using gradient descent algorithm in a manner that actual output is as close as possible to the desired output.

This is known as backward propagation. Forward and backward propagation are repeated through many epochs on an entire training dataset until some stopping criteria has been reached.

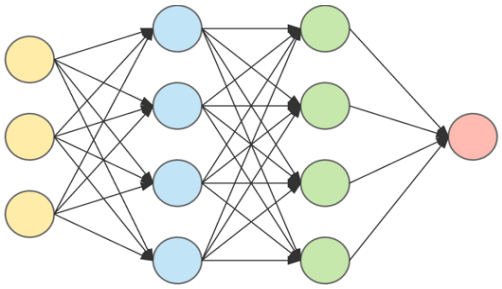

Figure 2.2.6. A multilayer perceptron with one input layer, two hidden layers and one output layer. Image adapted from [49].

MLP are universal function approximators. It has been proven by Kurt Hornik in 1989 [30] that any function can be approximated using MLP with one hidden layer containing sufficient number of non-linear units.

2.2.2.1 Forward propagation

In a forward propagation input data multiplied with weights, summed and passed through activation function is propagated from input to output layer. Letter h denotes one node in a hidden layer, $i = \overline{1, I}$ all nodes in an input layer connected with node h, θ_h an activation function of the node h and w_{ij} weight from node i to node j. The output of node h is derived as follows:

$$a_h = \sum_{i=1}^{I} w_{ih} x_i \tag{2.2.1}$$

$$y_h = \theta_h(a_h) \tag{2.2.2}$$

Commonly used activation functions in MLP are shown in Figure 2.2.7.

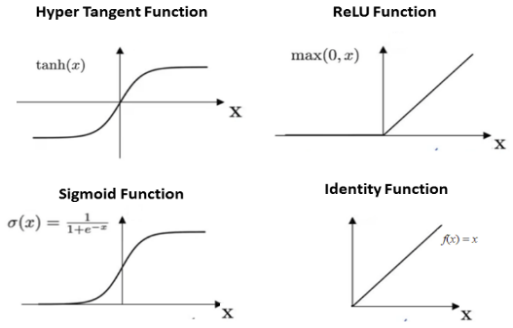

Figure 2.2.7. Activation functions in neural networks. Image adapted from [50].

Hyperbolic tangent is defined by:

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \tag{2.2.3}$$

Logistic sigmoid is defined by:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.2.4}$$

The output of hyperbolic tangent ranges from -1 to +1, while the output of logistic sigmoid ranges from 0 to +1. Accordingly, if output should be interpreted as probability, logistic sigmoid would be applied.

Two main advantages of these activation functions are nonlinearity and differentiability. Nonlinearity enables them to discover nonlinear boundaries between classes and approximate nonlinear equations. Differentiability of activation functions allows NNs to be trained using gradient descent algorithm. First derivates of hyperbolic tangent and logistic sigmoid are:

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh(x)^2 \tag{2.2.5}$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) * (1 - \sigma(x)) \tag{2.2.6}$$

Equations 2.2.1 and 2.2.2 show how to calculate output of the input layer. They can be generalized for any hidden layer. Output of the unit *h* in the l^{th} hidden layer H_l can be written by:

$$a_h = \sum_{h' \epsilon H_{l-1}} w_{h'h} y_{h'} \tag{2.2.7}$$

$$y_h = \theta_h(a_h) \tag{2.2.8}$$

Same can be applied for output layer. Input to the output unit *k* is derived as weighted sum of all units in the last layer *L* connected to that unit:

$$a_k = \sum_{h \in H_L} w_{hk} y_h \tag{2.2.9}$$

In NN, output of the network is often interpreted as probability that input vector belongs to the class *K*. If the problem is binary classification, output layer should have only one unit with logistic sigmoid activation function. Its output can be interpreted as probability that input vector belongs to the first class. Probability that input vector belongs to the first class and probability that input vector belongs to the second class are given, respectively:

$$p(C_1|x) = y = \sigma(x) \tag{2.2.10}$$

$$p(C_2|x) = 1 - y \tag{2.2.11}$$

In a coding scheme, if *z=1* marks first class C_1 and *z=0* second class C_2 equations 2.2.10 and 2.2.11 can be combined in one equation:

$$p(z|x) = y^z(1-y)^{1-z} \tag{2.2.12}$$

If number of classes *K>2*, number of units in output layer should be also *K*. Softmax function, presented as follows, is often used for obtaining probabilities:

$$p(C_k|x) = y_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}} \tag{2.2.13}$$

In a coding scheme, z is a vector of size K where all elements are equal to zero, except one on a position corresponding to the target class that is equal to one. For example, if *K=4* and target class is C_2 then vector z is $z = \{0,1,0,0\}$. Therefore, equation 2.2.13 can be summarized for all classes:

$$p(z|x) = \prod_{k=1}^{K} {y_k}^{z_k} \tag{2.2.14}$$

For evaluating accuracy of predictor and training NN different loss functions can be used. One of the most common is negative logarithmic likelihood. It is applied when the model outputs a probability for each class, rather than just the most likely class. Negative logarithmic likelihood is computed by:

$$L = -\frac{1}{N}\sum_{i=1}^{N} ln\, \hat{y}^{(i)} \tag{2.2.15}$$

For coding scheme in equations 2.2.13 and 2.2.14 negative logarithmic likelihood is calculated by:

$$L = -\ln p(z|x) \tag{2.2.16}$$

For logit model loss function is written by:

$$L = (z-1)\ln(1-y) - z\ln y \tag{2.2.17}$$

For multinomial logit model loss function is written by:

$$L = -\sum_{k=1}^{K} z_k \ln y_k \tag{2.2.18}$$

The second commonly used loss function is mean squared error (MSE). It is widely used in linear regression as the performance measure. The method of minimizing MSE is called Ordinary Least Squares. Its basic principle is that the optimized fitting line should be a line which minimizes the sum of distance of each point to the regression line. MSE loss function is defined by:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2 \tag{2.2.19}$$

N is the number of input vectors and $\hat{y}^{(i)}$, i=$\overline{1,N}$ are predictions for each input vector. However, if using logistic sigmoid as activation function, the quadratic loss function would suffer the problem of

slow *learning rate*. For example, by using sigmoid function $\hat{y}^{(i)} = \sigma\left(a_L^{(i)}\right) = \sigma(w^T b_{L-1}^{(i)})$, $(y - \sigma(a_L))^2$ derivate is computed by:

$$\frac{\partial L}{\partial w} = -\big(y - \sigma(a_L)\big)\sigma'(a_L)b_{L-1} \tag{2.2.20}$$

When $\sigma(a_L)$ is close to zero or one then $\sigma'(a_L)$ is close to zero and when $\sigma(a_L)$ is 0.5, $\sigma'(a_L)$ reaches its maximum in 0.25. In this case, when the difference $y - \sigma(a_L)$ is large, $\sigma'(a_L)$ will be small. It is improper, since learning speed should be fast when the error is large.

Often applied performance measure in linear regression problems is root mean squared error (RMSE), the variant of the mean squared error:

$$L = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2} \tag{2.2.21}$$

Mean Absolute Error (MAE) is also used as a loss function, however not so often. It is computed by:

$$L = \frac{1}{N}\sum_{i=1}^{N}\left|y^{(i)} - \hat{y}^{(i)}\right|^2 \tag{2.2.22}$$

where $|.|$ denotes the absolute value. Both MSE and MAE are used in predictive modeling. MSE is differentiable which makes it easier to compute gradient while MAE requires linear programming to compute gradient. Because of the square, large errors have greater influence on MSE than do smaller errors. Therefore, MAE is more robust on outliers since it doesn't make use of squares.

2.2.2.2 Backward propagation

Backward propagation or backpropagation is a practice of fine-tuning the weights of NN based on the loss function. Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization.

Two assumptions of loss function are needed for backpropagation. First assumption is that loss function can be written as an average $L = \frac{1}{N}\sum_x L_x$ over loss functions L_x for individual training example *x*. Second assumption is that loss function can be written as function of the outputs of the NN. It is about understanding how changing the weights and biases in a network changes the loss function, which implies computing the partial derivates $\frac{\partial L}{\partial w_{ij}^l}$, where w_{ij}^l stands for weight from node *i* in the *(l-1)*th layer to node *j* in the l^{th} hidden layer. The error of neuron *j* in the layer *l* is defined by:

$$\delta_j^l = \frac{\partial L}{\partial a_j^l} \tag{2.2.23}$$

Backpropagation gives a way of computing δ_j for every layer, and then relating those errors to the quantities of real interest $\frac{\partial L}{\partial w_{ij}^l}$. Backpropagation is based on three equations:

1) An equation for the error in the output layer δ^L is given by:
   $$\delta_j^L = \frac{\partial L}{\partial y_j^L}\sigma'(a_j^L) \tag{2.2.24}$$
   Matrix-based form of the above equation is defined by:
   $$\delta^L = \nabla_y L \odot \sigma'(a^L) \tag{2.2.25}$$
   Where $\odot$ stands for elementwise product, $\nabla_y L$ is a vector whose components are partial derivates $\frac{\partial L}{\partial y_j^L}$ and $\sigma'(a^L)$ is a vector whose components are derivates of activation functions in the output layer.
2) An equation for the error δ^l in terms of the error in the next layer δ^{l+1}:
   $$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(a^l) \tag{2.2.26}$$
   This equation enables propagation error backward and in combination with equation 2.2.25 makes it possible to compute the error for any layer in the network. Firstly, an error in the output layer is calculated using 2.2.25, then an error in last hidden layer δ^{L-1} is calculated using 2.2.26 and so on, all the way back through the network.
3) An equation for the rate of change of the cost with respect to any weight in the network:
   $$\frac{\partial L}{\partial w_{ij}^l} = y_i^{l-1}\delta_j^l \tag{2.2.27}$$
   Simplified notation of 2.2.27 is:
   $$\frac{\partial L}{\partial w} = y_{in}\delta_{out} \tag{2.2.28}$$
   For calculating gradient of weights in layer *l* , output of neurons in the previous layer *l-1* and error of neurons in layer *l* is needed.

Using these three equations one training epoch is given by:

1. Input vector *x*
2. Feedforward propagation: For each $l = \overline{2,L}$, $a^l = (w^l)^T y^{l-1}$ and $y^l = \theta(a^l)$
3. Loss function *L* is computed
4. Output error δ^L: δ^L=$\nabla_b L \odot \theta'(a^L)$
5. Backpropagate the error: For each $l = \overline{L,2}$, $\delta^l = ((w^{l+1})^T * \delta^{l+1}) \odot \sigma'(a^l)$
6. Output: The gradient of the cost function is given by: $\frac{\partial L}{\partial w_{ij}^l} = y_i^{l-1}\delta_j^l$
7. Weights adjusting: Adjusting the weights in a direction of a negative slope of loss function

$$w_{ij} = w_{ij} - \frac{\partial L}{\partial w_{ij}^{l}}$$

These steps are repeated until some criteria, such as the maximum number of epochs or error on validation dataset has been reached. Depending on the size of input data, entire dataset or its *batches* can be given to the network.

2.2.3 Recurrent neural networks

Recurrent neural networks are a family of neural networks designed specifically for sequential data processing (speech, time-series, video, sensor data, text) [31]. Traditional feedforward neural networks take a fixed amount of input data all at the same time and produce fixed amount of output data each time. In the traditional neural network, it is assumed that all inputs and outputs are independent of each other. On the other hand, RNN takes input data in a time sequence, where length of the sequence depends on how far in the past information should be taken into account when making a decision. RNNs are called recurrent because they perform the same task for every element of the sequence, with output being depended on the previous computations. RNNs have a "memory" which captures information about what has been calculated so far. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few time steps. The key point is that the recurrent connections allow memory of previous inputs to persist in the network's state and thereby influence the network output.

This section provides an overview of the base idea behind RNN's construction and its types. In Section 2.2.3.1 forward propagation algorithm has been described. In Section 2.2.3.2 backward propagation algorithm has been described. Section 2.2.3.3 discusses problems RNNs suffer from in real-world applications and offers possible solutions for them.

In Figure 2.2.8 an unfolded RNN has been shown. Unrolling means that the network is written for the complete time sequence. For example, if the sequence consists of five words, the network should be unrolled into a five layer network or if the sequence consists of 50 days with weather parameters obtained for each day the network should be unrolled into a 50 layer network. Each node represents a layer of network's units at a single time step. The weighted connections from input layer to hidden layer are labeled W_{xh}, those from hidden layer to itself (recurrent weights) are labeled W_{hh} and the hidden to output weights are labeled W_{hy}. Same weights are reused at each time step, which enables these networks to be flexible with input and output vector size, but causing vanishing and exploding gradient problems. It greatly reduces the number of parameters to be learned.

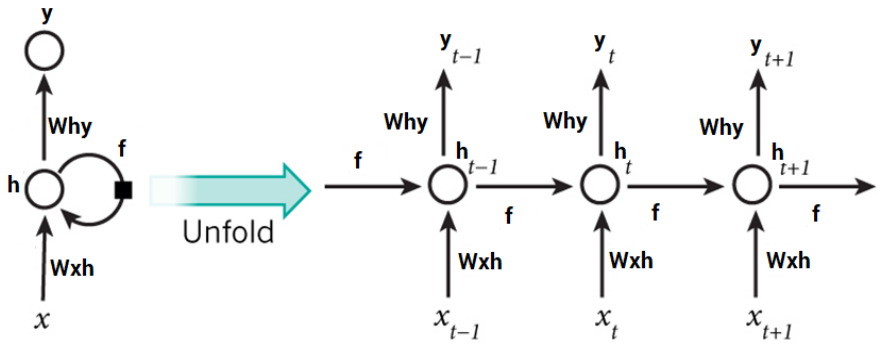

Figure 2.2.8. Unfolded recurrent neural network. Image adapted from [51].

The formulas that govern the computation happening in a RNNs are as follows:

- x^t is the input at time step *t*. For example, x^1could be a one-hot encoding vector corresponding to the second word of a sentence or the vector of features corresponding to today's weather data.
- h^t is the hidden state at time step *t*. It is the "memory" of the network and it captures information about what happened in all the previous time steps. h^t is calculated based on the previous hidden state and the input at the current time step: $h^t = f(w_{ih}^T * x^t + w_{h'h}^T * h^{t-1})$. The function *f* is usually nonlinearity, such as tanh or *ReLu*.
- y^t is the output at time step *t*. It is calculated based on the memory at time *t*. For example, if the task is predicting next word in a sentence the output can be a vector of probabilities across the used vocabulary: $y^t = softmax(w_{hk}^T * h^t)$ or if the task is predicting the weather parameter the output is a predicted value of that parameter.

In Figure 2.2.8 RNN has output at each time step, but depending on the task this may not be necessary. In Figure 2.2.9 five separate NNs are shown. The red rectangle represents input vector, green network state, blue output vector and arrow represents matrix multiplication. From left to right: (1) Feedforward neural network is shown with a fixed number of inputs and outputs. It can be used for image classification, to determine whether cat or dog is in the image. (2) Sequence output network is shown. It can be applied on image captioning where network takes an image and outputs its description. (3) Sequence input network is shown. It can be applied to sentiment analysis where the given sentence is classified as expressing positive or negative sentiment. (4) Sequence input and sequence output network is shown. It can be used for language translation. (5) Synced sequence input and output network is shown. It can be applied to labeling problems where every frame of video should be labeled.

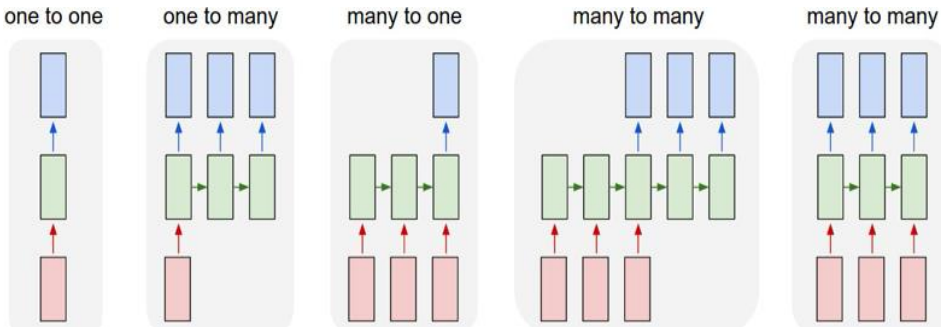

Figure 2.2.9. Types of neural networks. Image adapted from [52].

2.2.3.1 Forward propagation

The forward propagation of RNN with a single hidden layer is the same as forward propagation of multilayer perceptron, except that hidden layer consists of hidden cells (the number corresponds to the number of time steps), if the unfolded RNN has been imagined. Activation for each cell is not anymore only input signal, but current input signal and hidden state activations from the previous timestep. *T* denotes the length of input sequence *x* (number of timesteps), *H* number of hidden units in each hidden cell and *K* number of output units. x_i^t is the value of input *i* at time *t* , a_j^t is the network's input to unit *j* at time *t* and h_j^t is the activation of unit *j* at time *t*. Input to the hidden cell *h* at time step *t*:

$$a_h^t = \sum_{i=1}^{I} w_{ih}^T x_i^t + \sum_{h'=1}^{H} w_{h'h}^T h_{h'}^{t-1} \tag{2.2.29}$$

As mentioned in Section 2.2.1 RNNs with LSTMs have gained back on the popularity in 2006-2008. It made people ask why does backpropagation in RNNs didn't work well. One of the meaningful findings was that the particular non-linear activation function chosen for neurons in a NN makes a big impact on performance. More precisely, it can cause vanishing or exploding gradient problem. Three different groups explored the question of activation function: a group with Yann LeCun, with "What is the best multi-stage architecture for object recognition?" [32], a group with Geoffrey Hinton, in "Rectified linear units improve restricted Boltzmann machines" [33], a group with Yoshua Bengio in "Deep Sparse Rectifier Neural Networks" [34]. They all have found the same answer: the very simple function $f(x) = \max(0, x)$ tends to be the best. However, this function is not strictly differentiable (not differentiable in zero). Possible reasons for its success in preventing vanishing and exploding gradient problem are:

- Sparse representation: a small number of neurons will output value different than zero. It leads to more robust information representation and increases computational efficiency.

- Simplicity of the function makes it faster to work with than exponential sigmoid or the trigonometric tangent.

Andrew Ng co-wrote the analysis titled "Rectifier Nonlinearities Improve Neural Network Acoustic Models" [35] that showed that *ReLu* activation function performs very successfully. In Figure 2.2.10 *ReLu* activation function is shown.

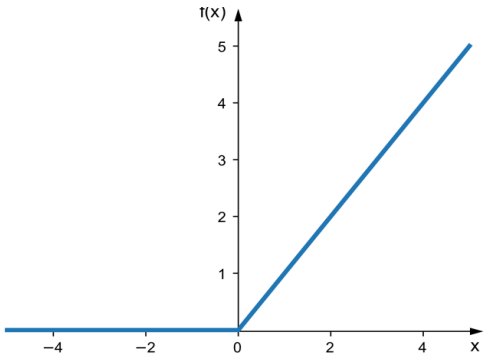

Figure 2.2.10. ReLu activation function

Activation function is applied on a_h^t:

$$h^t = \theta_h(a_h^t) \tag{2.2.30}$$

This operations are repeated for every timestep $t = \overline{1,T}$. In the first timestep, it is required to choose initial value h_i^0 that corresponds to the network's state before it has received any information. RNN's performance and stability can be improved by carefully choosing this initial values [44].

Network input to the output cell is calculated as follows:

$$a_k^t = \sum_{h=1}^{H} w_{hk}^T h_h^t \tag{2.2.31}$$

And output is written by:

$$y^t = \theta_k(a_k^t) \tag{2.2.32}$$

However, the output doesn't have to exist in every timestep. For example, for sentiment analysis task output is needed just at the end of time sequence, when *t=T*.

Dimensions of above vectors are as follows:

$x^t \in R^I x R^1$, where *I* is the number of features

$w_{ih} \in R^I x R^H$

$w_{h'h} \in R^H x R^H$

$h^t \in R^H x R^1$

$w_{hk} \in R^H x R^K$

$y^t \in R^K x R^1$

2.2.3.2 Backward propagation

Two well-known algorithms have been devised for calculating weight derivates: real time recurrent learning (RTRL) and backpropagation through time (BPTT). BPTT has been used in this work as it is simpler and more efficient in computation time.

Loss function *L* can be calculated as described in Section 2.2.2.1. The sum should be made over all time sequences in the training dataset, for each time step. Additionally, if there is defined number of *batches* sum should be repeated for that number of time sequences until weights update is applied.

The goal is to calculate the gradients of the error concerning parameters w_{ih}, $w_{h'h}$, w_{hk} and then learn optimal parameters using gradient descent algorithm. It is similar to standard backpropagation algorithm and consists of repeated application of chain rule. Error in time step *t* and for cell *j* is written by:

$$\delta_j^t = \frac{\partial L}{\partial a_j^t} = \frac{\partial y_j^t}{\partial a_j^t} * \frac{\partial L}{\partial y_j^t} = \theta'\left(a_j^t\right) * \frac{\partial L}{\partial y_j^t} \tag{2.2.33}$$

The difference compared to the standard backpropagation is that error in hidden cell *h* in time step *t* depends on the influence from the output layer and on the influence from hidden cell in the next time step.

$$\delta_h^t = \theta'(a_h^t) * (\sum_{k=1}^{K} \delta_k^t w_{hk} + \sum_{h'=1}^{H} \delta_{h'}^{t+1} w_{hh'}) \tag{2.2.34}$$

Calculation of the error for $t=0$ has to start from the error in $t=T$ and then propagate backward. As same weights are used in every timestep exploding or vanishing gradient problem occurs. Derivates with respect to the network weights are obtained summing over entire time sequence:

$$\frac{\partial L}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial L}{\partial a_j^t} * \frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_j^t y_i^{t-1} \tag{2.2.35}$$

2.2.3.3 Vanishing and exploding gradient problem

The RNN is a simple and powerful model, however it is hard to train properly in practice. Among the main reasons for that are exploding gradient and vanishing gradient problems. The network weights are updated using gradient descent algorithm that determines right direction and right amount for weights update. Due to the long term dependencies in RNNs, gradient can accumulate during an update and result in very large or very small gradients. Consequently, it causes large or minor updates to network weights. The exploding gradient problem leads to the unstable network as its weights can become so large as to overflow and result in *NaN* values. The vanishing gradient problem results in minor weights update, so weights remain unchanged during the training process and incapable of learning dependencies between the data. In vanishing gradient problem the gradients of the network's output concerning the parameters in the early layers become extremely small. Therefore, even a large change in the value of parameters in early layers doesn't have a big impact on the output. Both problems become more obvious as the number of layers in RNN's architecture increases.

As described in Formula 2.2.30 the hidden state value in time step t depends on hidden state value in the previous time step *t-1* and input value in time step t:

$$h^t = \theta_h(w_{h'h}^T * h^{t-1} + w_{ih}^T * x^t) = \theta_h(a_h^t) \tag{2.2.36}$$

$$y^t = \theta_k(w_{hk}^T * h^t) = \theta_k(a_k^t) \tag{2.2.37}$$

The total loss derivate is given as sum of derivates over all time steps:

$$\frac{\partial L}{\partial w_{h'h}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial w_{h'h}} \tag{2.2.38}$$

The loss derivate in certain time step is written by formula:

$$\frac{\partial L_t}{\partial w_{h'h}} = \sum_{s=1}^{t-1} \frac{\partial L_t}{\partial y^t} * \frac{\partial y^t}{\partial h^t} * \frac{\partial h^t}{\partial h^s} * \frac{\partial h^s}{\partial w_{h'h}} = \frac{\partial L_t}{\partial y^t} * \theta'(a_k^t) * w_{hk}^T * \sum_{s=1}^{t-1} \frac{\partial h^t}{\partial h^s} * \frac{\partial h^s}{\partial w_{h'h}} \tag{2.2.39}$$

The hidden state value in time step *t* is depended on hidden state values in all previous time steps:

$$\sum_{s=1}^{t-1} \frac{\partial h^t}{\partial h^s} * \frac{\partial h^s}{\partial w_{h'h}} = \frac{\partial h^t}{\partial h^{t-1}} * \frac{\partial h^{t-1}}{\partial w_{h'h}} + \frac{\partial h^t}{\partial h^{t-2}} * \frac{\partial h^{t-2}}{\partial w_{h'h}} + \ldots + \frac{\partial h^t}{\partial h^1} * \frac{\partial h^1}{\partial w_{h'h}} = \frac{\partial h^t}{\partial h^{t-1}} * \frac{\partial h^{t-1}}{\partial h^{t-2}} * \ldots * \frac{\partial h^1}{\partial w_{h'h}} \tag{2.2.40}$$

Therefore if the input sequence consists of only four time steps *(T=4)*, the formula (2.2.40) for *t=4* would look like:

$$\frac{\partial L_4}{\partial w_{h'h}} = \frac{\partial L_4}{\partial y^4} * \theta_k'(a_k^4) * w_{hk}^T * \sum_{s=1}^{3} \frac{\partial h^t}{\partial h^s} * \frac{\partial h^s}{\partial w_{h'h}} = \frac{\partial L_4}{\partial y^4} * \theta_k'(a_k^4) * w_{hk}^T * \frac{\partial h^4}{\partial h^3} * \frac{\partial h^3}{\partial h^2} * \frac{\partial h^2}{\partial h^1} * \frac{\partial h^1}{\partial w_{h'h}}$$

$$= \frac{\partial L_4}{\partial y^4} * \theta'_k(a_k^4) * w_{hk}^T * \theta'_h(a_h^3) * w_{h'h}^T * \theta'_h(a_h^2) * w_{h'h}^T * \theta'_h(a_h^1) * w_{h'h}^T * \theta'_h(a_h^1) * h^0 \quad (2.2.41)$$

One of the causes of vanishing gradient problem or exploding gradient problem is the usage of the same matrices in every time step. In [36] it is appropriately explained: "The same way the product of *t-s* real numbers can shrink to zero or explode to infinity, so does this product of matrices". The second cause of vanishing gradient problem may be the choice of activation function. The most commonly used activation functions are hyperbolic tangent and logistic sigmoid with property to 'squash' its input into a very small output range. This becomes obvious in networks with multiple layers because the first layer will map large input into smaller output, which will be mapped to an even smaller region by the second layer and so on. Moreover, their derivates, used in the gradient calculation, are even smaller than their output range (i.e maximum of logistic sigmoid function is 1, while maximum of its derivate is 0.25). It will additionally 'lower' the gradient and make it smaller.

The signs that the model may be suffering from exploding gradient problem are:

- The model weights quickly become large during the training.
- The model is unable to get traction on training data (poor loss).
- The model is unstable, resulting in large loss changes from update to update.

The signs that the model may be suffering from vanishing gradient problem are:

- The model weights remain unchanged during the training.
- The model is unable to get traction on training data (poor loss).
- The error gradient values are consistently very small for each node and layer during training.

Possible solutions for fixing exploding and vanishing gradient problems are:

- Carefully chosen activation function. As described in Section 2.2.3.1 multiple efforts have been made to determine which activation function performs the best. It has been concluded that *ReLu* activation function not only provides sparse data representation and calculation simplicity, but also prevents the vanishing gradient problem.
- Weights initialization. Instead of initializing matrices by sampling from normal distribution, it may be better to initialize them as identity matrices or on some other way described in Section 2.2.5.3.
- Using Long Short-Term Memory Network; a variant of the regular RNN which was designed to capture long-term dependencies in sequence data.
- Gradient clipping. This method is used as solution to exploding gradient problem. It places a predefined threshold on the gradient to prevent it from getting to large, and by doing so it doesn't change the direction of the gradient, it only changes its length. If the gradient norm is

greater than threshold, the gradient is replaced with the $\frac{threshold*g}{\|g\|}$, where g stands for the gradient and $\|g\|$ for its norm.

2.2.4 Long short-term memory networks

The Long Short-Term Memory Networks were designed to overcome the problem of vanishing and exploding gradient. The LSTM has been introduced in paper "Long Short-Term Memory" written by Sepp Hochreiter and Jürgen Schmidhuber in 1997 [25]. The idea was to enable constant error flow during backpropagation by replacing RNN cell with LSTM cell. Structurally more complex, LSTM cell consists of internal state and three gate units. The multiplicative gate units allow LSTM's internal state to store and access information over long periods of time by enforcing selectivity in writing in, forgetting and reading from each LSTM cell. The internal state should be considered as memory cell, separate part of the LSTM cell that enables constant error flow during backpropagation.

As it can be seen in Figure 2.2.8 information is written in each RNN cell (previous state h_{t-1} and input vector x_t) and read from RNN cell (current state h_t). However, the problem with the base RNN is that it doesn't pose any selectivity in writing in information in the cell. Beginning of the training is started with random initializations and RNN makes random writes. It will result in the bad performance at the beginning and chaotic state from which may be very hard to recover afterward. Hochreiter and Schmidhuber were aware of this problem and split it into three subproblems: "input weight conflict" that addresses selective writing into the cell, "output weight conflict" that addresses selective reading from the cell and "internal state drift" that addresses gates saturation problem. LSTM has been designed in order to overcome these problems.

This section describes the LSTM networks, its importance and influence on deep learning. In Section 2.2.4.1 intuition behind LSTM architecture has been described. In Section 2.2.4.2 structure of the LSTM cell has been discussed. Section 2.2.4.3 provides forward propagation equations, while Section 2.2.4.4 offers backward propagation equations. In Section 2.2.4.5 an overview of the LSTM cell variations has been given.

2.2.4.1 Intuitive explanation of the LSTM cell architecture

Three types of selectivity for overcoming "input weight conflict", "output weight conflict" and "overload conflict" realized through three multiplicative gate units have been introduced with LSTM networks. These three multiplicative gate units are:

- Input gate unit i_t has been proposed as solution for reducing "input weight conflict" by reinforcing selectivity in writing. The cell must have mechanism to cancel some of the unnecessary incoming information. Otherwise, the new information will be written in every element of the state, resulting in overcrowded and chaotic state not capable of remembering long-term dependencies.
- Output gate unit o_t has been proposed as solution for reducing "output weight conflict" by reinforcing selectivity in reading. The cell in time sample *t* outputs hidden state h_t that will be used not only for deriving output in time step *t*, but as cell input in the next timestep *t+1*. The cell should have a mechanism to reduce irrelevant information stored in the hidden state. Otherwise, the irrelevant information will be passed as input to the cell in the next time step.
- Forget gate unit f_t has been proposed as solution for reducing "overload conflict" by reinforcing selectivity in remembering. This gate unit hasn't been part of the first proposed LSTM cell structure. It is one of the reasons LSTMs didn't perform so successfully at the beginning. Rather, it was introduced by Felix Gers in 2000 [37]. The forget gate unit should exclude some of the elements from the internal state based on the previous internal state elements. It decides on which information is valuable of keeping in the internal state based on the information from the internal state in the previous time step. Without the forgetting the information would overload and state would grow indefinitely.

In Figure 2.2.11 vanishing gradient problem for RNN has been illustrated. Base RNN doesn't have any selectivity in writing in and reading from the cell. Therefore, it writes new information to every element of the state making it „overcrowded" and impossible to remember long-term dependencies. In Figure 2.2.11 it can be seen that input signal gradually vanishes as RNN has been forgetting information.

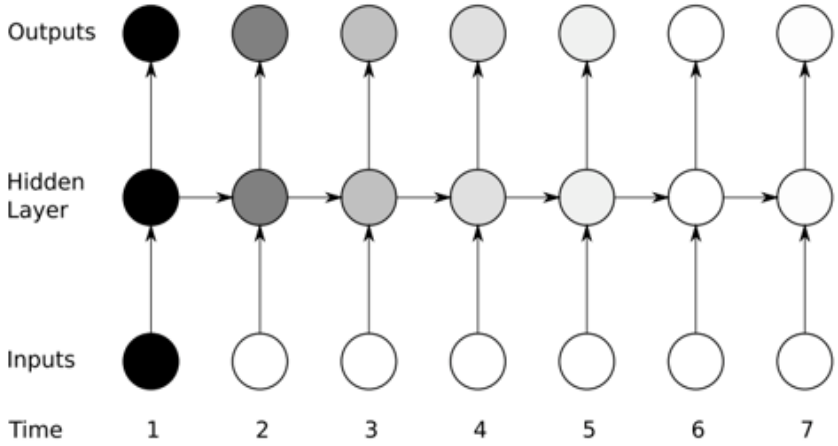

Figure 2.2.11. The vanishing gradient problem for RNN. . Image adapted from [53].

In Figure 2.2.12 LSTM network has been presented. The first input information perseveres in the next time steps because input gate remains close and forget gate open. The output gate got opened in time steps four and six without affecting the cell state.

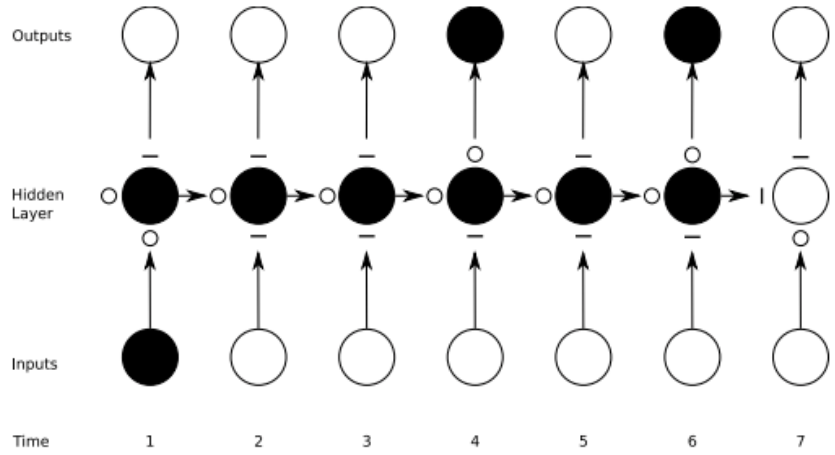

Figure 2.2.12. Gates as mechanism for selective reading, writing and forgetting. Image adapted from [54].

The difference between writing in the cell and reading from the cell is considerable. The writes affect the single element of the state in time step *t* and reads can't affect any element of the state in time step *t*. However, read decision impacts the entire state in the next time step *t+1*. It doesn't mean that the impact of reads is greater than impact of writes. In each time step read information is passed through nonlinear function that 'squashes' its values. It is well explained in [38]: "The impact of read decision is broad, but shallow while impact of write decision is narrow, but deep".

Input gate unit, output gate unit and forget gate unit are implemented using single-layer. Neural network's weights are learned during backpropagation using gradient descent algorithm. Therefore, even if it would be easier to consider input, output and remember decisions as binary decisions, they need to be performed using differentiable functions. The mostly used activation function is logistic sigmoid as its output ranges from zero to one and it is differentiable. The input, output and forget gate units output vector with elements range from zero to one, specifying the percentage of writing, reading and forgetting for each element of the state.

2.2.4.2 Structure of the LSTM cell

In this work the following notation is used: x_t is the input signal, h_t denotes the cell state at time step *t*, c_t denotes the internal cell state, $\tilde{c}_t$ is the internal cell state candidate, i_t is the input gate unit, f_t is the forget gate unit, o_t is the output gate unit. Input gate unit (i_t), output gate unit (o_t), forget gate unit (f_t) and internal cell state candidate ($\tilde{c}_t$) at time step *t* are calculated using input signal at time step *t* (x_t) and cell state at time step *t-1*(h_{t-1}).

$$i_t = \sigma(W_i * h_{t-1} + U_i * x_t) \tag{2.2.42}$$

$$\tilde{c}_t = tanh(W_c * h_{t-1} + U_c * x_t) \tag{2.2.43}$$

$$o_t = \sigma(W_o * h_{t-1} + U_o * x_t) \tag{2.2.44}$$

$$f_t = \sigma(W_f * h_{t-1} + U_f * x_t) \tag{2.2.45}$$

The internal cell state at time step *t* (c_t) is calculated by adding candidate cell state ($\tilde{c}_t$) filtered through input gate (i_t) to internal cell state at the previous time step (c_{t-1}) filtered through forget gate (f_t). This sum enables information cancellation from the candidate cell state based on the information stored in the previous cell state.

$$c_t = i_t \odot \tilde{c}_t + f_t \odot c_{t-1} \tag{2.2.46}$$

Where $\odot$ represents elementwise-multiplication.

The LSTM cell encounters the internal state drift problem when input values are mostly positive or mostly negative. It can cause internal cell state to drift away over time. Therefore, the internal state activation function should be carefully chosen. The disadvantage of linear function is unrestricted memory range. Hyperbolic tangent has been mostly applied activation function. However, its disadvantage is small derivate in case of internal state's big values. It can cause vanishing gradient problem.

The cell state in time step *t* (h_t) is derived as 'squashed' internal cell state ($tanh(c_t)$) filtered through output gate unit (o_t).

$$h_t = o_t \odot tanh(c_t) \tag{2.2.47}$$

In Figure 2.2.13 LSTM cell has been illustrated.

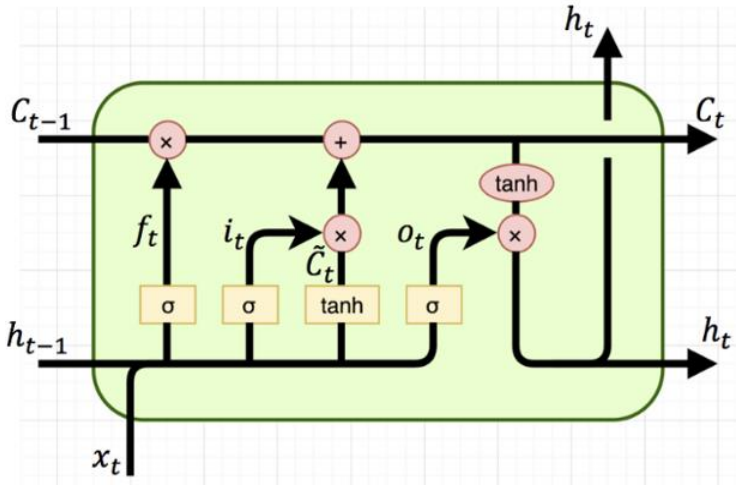

Figure 2.2.13. The LSTM cell. Image adapted from [55].

2.2.4.3 Forward propagation

Forward propagation has already been described during LSTM structure explanation. Equations for input gate, internal state candidate, output gate and forget gate unit are written by:

$$i_t = \sigma(W_i * h_{t-1} + U_i * x_t) = \sigma(\hat{i}_t) \tag{2.2.48}$$

$$\tilde{c}_t = tanh(W_c * h_{t-1} + U_c * x_t) = tanh(\widehat{\tilde{c}_t}) \tag{2.2.49}$$

$$o_t = \sigma(W_o * h_{t-1} + U_o * x_t) = \sigma(\hat{o}_t) \tag{2.2.50}$$

$$f_t = \sigma\big(W_f * h_{t-1} + U_f * x_t\big) = \sigma(\hat{f}_t) \tag{2.2.51}$$

The matrix form of above written equations ignoring non-linearities is:

$$z_t = \begin{bmatrix} \hat{i}_t \\ \hat{\tilde{c}}_t \\ \hat{o}_t \\ \hat{f}_t \end{bmatrix} = \begin{bmatrix} W_i & U_i \\ W_C & U_C \\ W_O & U_O \\ W_f & U_f \end{bmatrix} x \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} = W\, x\, I_t \tag{2.2.52}$$

The output of the network ignoring non-linearity in time step t is:

$$y_t = W_y * h_t \tag{2.2.53}$$

If input vector *x* size is *Ix1* (it has *I* features), output of the network *y* size is *Kx1* and there are *H* memory units inside of each LSTM cell state h_t (dimension of h_t is *Hx1*), dimensions of above matrices are as follows:

$W_* \epsilon\, R^H x\, R^H$

$U_* \epsilon\, R^H x\, R^I$

$W_y \epsilon\, R^K x\, R^H$

$W \epsilon\, R^{4*H} x\, R^{H+I}$

2.2.4.4 Backward propagation

In Figure 2.2.14 backward propagation for unfolded LSTM network during three time steps has been shown. The arrows indicate gradient flow during the backpropagation. The internal cell state in time step *T*, c^T receives the gradient from h^T, as well as the gradient from the next cell state c^{T+1}.

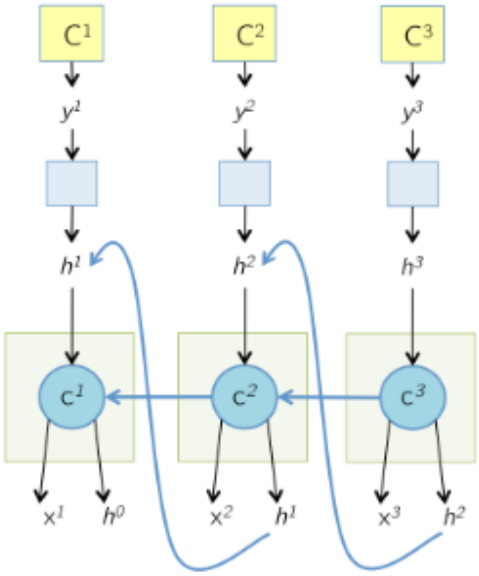

Figure 2.2.14. Backward propagation in LSTM network. Image adapted from [56].

The loss function L (MSE loss function) is calculated summing the errors in every time step:

$$L=\frac{1}{T}*\sum_{t=1}^{T} L_t = \frac{1}{T}*\sum_{t=1}^{T}(y_t-\hat{y}_t)^2$$

The following steps are applied for deriving weight gradient:

1) Forward pass: $\hat{y}_t = W_y * h_t$ (2.2.54)

$$\delta h_t = \frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial \hat{y}_t} * \frac{\partial \hat{y}_t}{\partial h_t} = (\delta \hat{y}_t^T * W_y)^T$$

2) Forward pass: $h_t = o_t \odot tanh(c_t)$ (2.2.55)

$$\delta o_i^t = \frac{\partial L}{\partial o_i^t} = \frac{\partial L}{\partial h_i^t} * \frac{\partial h_i^t}{\partial o_i^t} = \delta h_i^t \odot tanh(c_i^t)$$

$$\delta o_t = \delta h_t \odot tanh(c_t)$$

$$\delta c_i^t = \frac{\partial L}{\partial c_i^t} = \frac{\partial L}{\partial h_i^t} * \frac{\partial h_i^t}{\partial c_i^t} = \delta h_i^t * o_i^t * (1 - tan\, h(c_i^t)^2)$$

$$\delta c_t = \delta h_t \odot \delta o_t \odot (1 - tan\, h(c_t)^2)$$

During backpropagation internal state's gradient in time step *t* is influenced by cell state's gradient in time step *t*, but also by internal state's gradient in time step *t+1*.

3) Forward pass: $c_t = i_t \odot \tilde{c}_t + f_t \odot c_{t-1}$ (2.2.56)

$$\delta i_i^t = \frac{\partial L}{\partial i_i^t} = \frac{\partial L}{\partial c_i^t} * \frac{\partial c_i^t}{\partial i_i^t} = \delta c_i^t * \tilde{c}_i^t$$

$$\delta i_t = \delta c_t \odot \tilde{c}_t$$

$$\delta f_i^t = \frac{\partial L}{\partial f_i^t} = \frac{\partial L}{\partial c_i^t} * \frac{\partial c_i^t}{\partial f_i^t} = \delta c_i^t * c_i^{t-1}$$

$$\delta f_t = \delta c_t \odot c_{t-1}$$

$$\delta \tilde{c}_i^t = \frac{\partial L}{\partial \tilde{c}_i^t} = \frac{\partial L}{\partial c_i^t} * \frac{\partial c_i^t}{\partial \tilde{c}_i^t} = \delta c_i^t * i_i^t$$

$$\delta \tilde{c}_t = \delta c_t \odot i_t$$

$$\delta c_i^{t-1} = \frac{\partial L}{\partial c_i^{t-1}} = \frac{\partial L}{\partial c_i^t} * \frac{\partial c_i^t}{\partial c_i^{t-1}} = \delta c_i^t * f_i^t$$

$$\delta c_{t-1} = \delta c_t \odot f_{t-1}$$

4) Forward pass: $z_t = \begin{bmatrix} \hat{i}_t \\ \hat{\tilde{c}}_t \\ \hat{o}_t \\ \hat{f}_t \end{bmatrix} = \begin{bmatrix} W_i & U_i \\ W_C & U_C \\ W_O & U_O \\ W_f & U_f \end{bmatrix} x \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix}$ (2.2.57)

$$\delta \hat{\tilde{c}}_t = \delta \tilde{c}_t \odot (1 - tanh^2(\hat{\tilde{c}}_t))$$

$$\delta \hat{i}_t = \delta i_t \odot \hat{i}_t \odot (1 - \hat{i}_t)$$

$$\delta \hat{f}_t = \delta f_t \odot \hat{f}_t \odot (1 - \hat{f}_t)$$

$$\delta \hat{o}_t = \delta o_t \odot \hat{o}_t \odot (1 - \hat{o}_t)$$

$$\delta z^t = \left[\delta \hat{i}_t \ \ \delta \hat{\tilde{c}}_t \ \ \delta \hat{o}_t \ \ \delta \hat{f}_t\right]^T$$

5) Forward pass: $z_t = W\ x\ I_t$ (2.2.58)

$$\delta I_t = W^T \odot z_t$$

$$\delta W_t = \delta z_t \odot {I_t}^T$$

6) Parameter update (2.2.59)

If input vector *x* consists of *T* time steps $[x_1\ \ x_2 \ldots\ x_T]$ weight gradient is sum of gradients in all time steps: $\delta W = \sum_{t=1}^{T} \delta W_t$

The weights are updated using gradient descent algorithm: $W \leftarrow W - \alpha * \delta W$

2.2.4.5 LSTM variations

The popular LSTM cell variants are Gated Recurrent Unit (GRU) and LSTM with "peephole connections". They can perform slightly better than standard LSTM cell in some cases, depending on the nature of the input data.

The GRU cell has been introduced by Cho in 2014 [39]. The GRU doesn't have to use an internal cell state to control the flow of information like the LSTM unit. It can directly make use of all hidden states without any control. GRUs have fewer parameters and thus may train a bit faster or need fewer data to generalize. But, with large data, the LSTMs may lead to better results. The GRU combines the input and forget gate unit into a single update gate. It is described with the following equations:

$$z_t = \sigma(W_z * h_{t-1} + U_z * x_t) \tag{2.2.60}$$

$$r_t = \sigma(W_r * h_{t-1} + U_r * x_t) \tag{2.2.61}$$

$$\tilde{h}_t = \sigma(W_h * r_t * h_{t-1} + U_h * x_t) \tag{2.2.62}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{2.2.63}$$

Figure 2.2.15 shows GRU cell, while Figure 2.2.16 shows LSTM cell with "peephole connections".

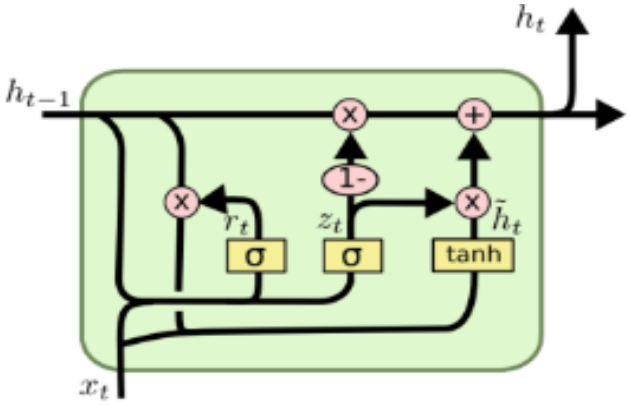

Figure 2.2.15. The GRU unit. Image adapted from [57].

The LSTM cell with "peephole connections" has been introduced by Gers & Schmidhuber in 2000 [40]. The idea is that gates can look at the internal cell state. This type of LSTM cell is described using equations:

$$i_t = \sigma(W_i * h_{t-1} + S_i * c_{t-1} + U_i * x_t) \tag{2.2.64}$$

$$o_t = \sigma(W_o * h_{t-1} + S_o * c_{t-1} + U_o * x_t) \tag{2.2.65}$$

$$f_t = \sigma\big(W_f * h_{t-1} + S_f * c_{t-1} + U_f * x_t\big) \tag{2.2.66}$$

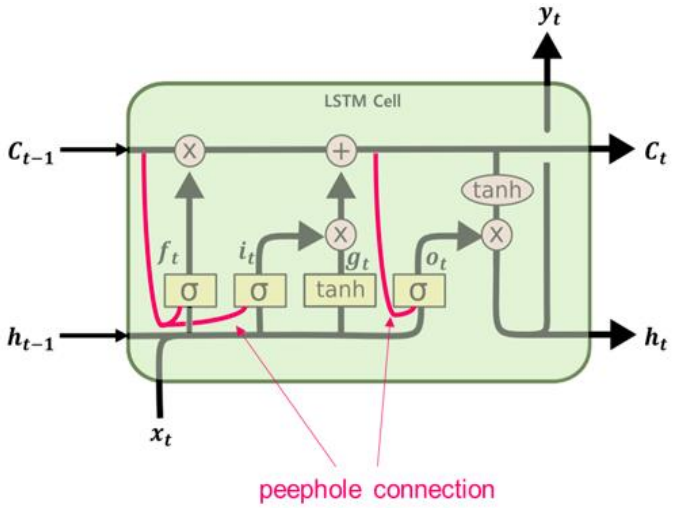

Figure 2.2.16. The LSTM cell with "peephole connections" . Image adapted from [58].

2.2.5 Network training

Good results on the testing dataset can be achieved only with the properly built and trained NN. During the training process, weights of the model are changing in order to minimize the loss function and make predictions as correct as possible. The "quality" of the model is evaluated by the specified metrics. Optimizers shape model into its most accurate possible form by adjusting the weights. On the other hand, generalization methods prevent model from overfitting. Overfitting is a phenomenon when model performs well on the training dataset, but fails on the testing dataset. It occurs when NN's structure is too complex (too many layers and units per layer) compared to training dataset size or when the NN has been trained for too many epochs. In Section 2.2.5.1 applied evaluation metrics are listed. In Section 2.2.5.2 different optimizers are described. In Section 2.2.5.3 common methods for preventing overfitting are provided. In Section 2.2.5.4 most efficient ways for weights initialization concerning activation function in a layer are described.

2.2.5.1 Evaluation metrics

Logarithmic likelihood (Formula 2.2.18), mean squared error (Formula 2.2.19), root mean squared error (Formula 2.2.21) and mean absolute error (Formula 2.2.22) metrics have already been described in Section 2.2.2.1. Metrics for evaluating classification tasks are listed in this section.

Classification accuracy is the ratio of number of correct predictions to the total number of inputs:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (2.2.67)$$

Where *TP (True Positives)* is the number of cases when the first class has been predicted and the actual output is the first class, *TN (True Negatives)* is the number of cases when the second class has been predicted and the actual output is the second class, *FP (False Positives)* is the number of cases when the first class has been predicted, but the actual output is the second class, *FN (False Negatives)* is the number of cases when the second class has been predicted, but the actual output is the first class. However, accuracy can give false sense of good model performance. If the 98% of the data belongs to the first class and 2% of the data belongs to the second class, accuracy of 98% can be simply achieved by classifying all the data into the first class. The problem arises when the cost of misclassification of minor class samples is high. For example, a group of patients should be classified into diabetes positive or diabetes negative class. A cost of failing to diagnose the disease of a sick person (False Negative) is much higher than the cost of classifying healthy person into diabetes positive class (False Positive). Therefore, other metrics have been introduced.

Confusion matrix gives a matrix as output and describes the complete performance of the model:

$$Confusion\ matrix = \begin{bmatrix} TP & FN \\ FP & TN \end{bmatrix} \quad (2.2.68)$$

Recall (sensitivity) and precision have been derived from the confusion matrix. Recall is the proportion of correctly classified positive samples, with respect to all positive samples:

$$Recall = \frac{TP}{TP+FN} \quad (2.2.69)$$

Recall increases the cost of misclassified positives samples. It will be low if significant number of positive class instances is misclassified. Recall gives information how robust the classifier is.

Precision is the proportion of correctly classified positive samples, with respect to all samples classified as positive:

$$Precision = \frac{TP}{TP+FP} \quad (2.2.70)$$

It gives information how precise the classifier is (how many instances have been correctly classified).

F1-score encourages both high precision and high recall:

$$F1 - score = 2 * \frac{1}{\frac{1}{precision}+\frac{1}{recall}} \quad (2.2.71)$$

Therefore, it is often used as evaluation metric beside accuracy.

2.2.5.2 Optimization

Gradient descent is the base optimization technique. The weights are iteratively updated by subtracting its derivate in order to reach cost function minimum. The derivates give information on how to change model's parameters in order to lower the output of the loss function and thereby make model more accurate:

$$\Delta w_{ij} = \alpha * \frac{\partial L}{\partial w_{ij}} \tag{2.2.72}$$

Where α is the *learning rate*.

The gradient descent common problems are getting stuck on local minima or plateau. Carefully chosen *learning rate* and *momentum* are ways of dealing with these problems.

Learning rate is an amount that the weights are updated during the training process. Usually it is a very small number the gradient is scaled by. Too large *learning rate* can cause the model to converge quickly to suboptimal solution (local minimum). Too small *learning rate* can cause the process to get stuck on plateau. The recipe for *learning rate* change is:

- if the error continues getting worse or oscillating decrease *learning rate*
- if the error continues decreasing slowly, but consistently increase *learning rate*

If derivates of the loss function are computed on small *batches* they are "noisy". It means that the gradient doesn't always change weights in an optimal direction. *Momentum* helps by averaging gradients. It accelerates gradients in the right direction:

$$\Delta w_{ij} = \alpha * \frac{\partial L}{\partial w_{ij}} + m * \Delta w_{ij}^{t-1} \tag{2.2.73}$$

Where $m \epsilon [0,1]$ is the *momentum* parameter.

If the *momentum* term is large then the *learning rate* should be kept smaller. A large value of *momentum* also means that the convergence will happen fast. But if both the *momentum* and *learning rate* are kept at large values, then the minimum might be skipped with a huge step.

Three types of gradient descent algorithm are used depending on the size of the training dataset and desirable computational efficiency:

1) Batch gradient descent calculates the error for each example within the training dataset. Parameters are updated after all training examples have been evaluated. It is called a *training epoch*. Advantage of batch gradient descent is that it produces stable error gradient and stable convergence. The disadvantage is that entire training dataset is required for gradient

computation. Therefore, in case of big training dataset this type of gradient calculation can't be applied.

2) Stochastic or online gradient descent updates the model's parameters based on the error calculated on a single training example. The frequent updates can cause the gradient to fluctuate instead of decreasing. Moreover, as the update is performed for each training example, stochastic gradient descent can be computational more expensive than batch gradient descent.
3) Mini batch gradient descent is a combination of stochastic and batch gradient descent algorithm. It separates the training dataset into small *batches* and calculates the error on those *batches*. The problem may be determining the best value for the *batch* size.

The process of weight update is repeated until some stopping criteria i.e failure to reduce the loss for a given number of steps or number of epochs has been met.

Trying to increase the accuracy of neural networks, other types of optimizers based on the gradient descent algorithm have been introduced over time:

- Adagrad optimizer adapts the *learning rate* based on the individual training examples. Therefore, some weights will have the different *learning rates* than others. The main disadvantage is that *learning rate* tends to get very small over time.
- RMSprop optimizer has been introduced by Professor Geoffrey Hinton in his class on neural networks [43]. The central idea of RMSprop is to keep the moving average of the squared gradients for each weight. And then the gradient is divided with the squared root of mean square. It prevents loss function oscillations and accelerates gradients in the right direction.
- Adam optimizer is shortcut for adaptive moment estimation and it is the most widespread optimizer for training the neural networks. Adam utilizes the concept of *momentum* by adding fractions of previous gradients to the current one.

2.2.5.3 Generalization

Early stopping is the common form of regularization used to avoid overfitting. Overfitting is a phenomenon when model performs well on the training dataset, but fails on the testing dataset. The overfitted model is inaccurate because the trend doesn't reflect the reality of data [41]. The clear sign that overfitting has occurred is when the loss on validation dataset starts increasing, while the loss on training dataset continues decreasing as illustrated in Figure 2.2.17. The training should be stopped when loss on validation dataset has been increasing for a number of epochs greater than network's *patience* parameter. The best weights (weights for which the loss on validation dataset has been the lowest, weights before the loss on validation dataset has started to increase) are usually saved and later on used for making predictions on testing dataset.

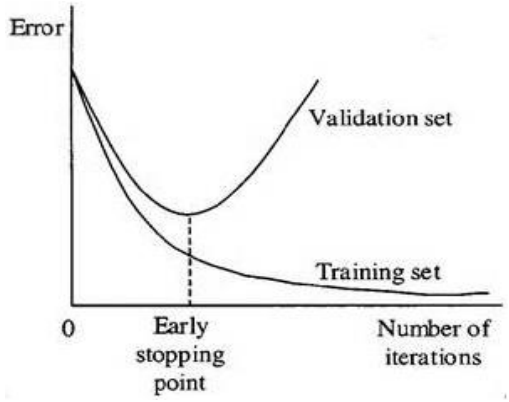

Figure 2.2.17. The sign of overfitting. Image adapted from [59].

Dropout is a second common used regularization method. During training, some numbers of layers outputs are randomly ignored by setting the weight matrices that connect them to the next layer on zero. The percentage of ignored layers usually takes values between 0.1 and 0.5. *Dropout* makes the model more robust and helps in sparse data representation.

Weight decay is a regularization term that penalizes big weights. It is also known under name L2 regularization. The loss function is penalized with the squared sum of the weights:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2 + \frac{\lambda}{2} * \sum_{i=1}^{K} w_i^2 \tag{2.2.74}$$

Where λ stands for the weight decay parameter.

The more training examples there are, the weaker *weight decay* term should be and weights can freely grow trying to fit examples. The more parameters there are the higher regularization should be as it prevents overfitting by preventing the weights to grow too large. During training, after each update, the weights are multiplied with the parameter slightly less than one:

$$w_{ij} = \lambda * w_{ij} - \alpha * \frac{\partial L}{\partial w_{ij}} \tag{2.2.75}$$

In contrast to L2 regularization, L1 regularization penalizes the sum of absolute weights:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2 + \frac{\lambda}{2} * \sum_{i=1}^{K}|w_i| \tag{2.2.76}$$

L2 regularization is computationally more efficient due to having analytical solutions. On the other hand L1 regularization is robust to outliers and can be used in feature selection.

In [62], batch normalization has recently been proposed as the method to prevent overfitting and coordinate the update of weights in the model. It does this by normalizing the activations of each layer:

$$input'(x) = \frac{input(x) - mean(x)}{standard\ deviation(x)} \quad (2.2.77)$$

Therefore, inputs to each layer have mean value zero and standard deviation one. Batch normalization has been proposed as the solution for "covariate shift" problem - to enable efficient training and evaluation of the model by maintaining the same distribution of input values to each layer [62].

The overfitting can also occur when the model is too complex (consists of too many layers and units per layer) compared to the number of training data examples and features. In that case, the neural network has a capacity of remembering minor details and loses the possibility to generalize. The same happens if the neural network has been trained for a lot of epochs without employing any regularization method.

2.2.5.4 Weights initialization

As mentioned in Section 2.2.3.3 neural networks weights initialization can have a significant influence on avoiding vanishing and exploding gradient problem and network's faster convergence. Beside weights initialization with zeros or sampling from standard normal distribution two other methods have been recently proposed:

1) Xavier initialization – In 2010 Xavier Glorot and Yoshua Bengio published the paper titled "Understanding the difficulty of training deep feedforward neural networks" [42] and suggested that the network's weights should be chosen from a random uniform distribution that is bounded between $\pm\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}$, where n_i represents the number of incoming networks connections and n_{i+1} number of outgoing networks connections. This weight initialization method is suitable only for activation functions symmetric about a given value, such as hyperbolic tangent and logistic sigmoid. They have shown that Xavier initialization method maintains the weight gradients variance almost unchanged during forward and backward propagation on an example of five-layer neural network.
2) Kaiming initialization – The Xavier initializer is suitable only for activation functions symmetric about a zero and with the output ranges from -1 to +1. Meanwhile, *ReLu* activation function has been proven as beneficial and in 2015 Kaiming He proposed the solution for weights initialization in case of network layer using *ReLu* activation function: Weight matrix elements should be randomly chosen from a standard normal distribution and divided with the $\sqrt{\frac{2}{n}}$, where *n* stands for the number of incoming connections. The proposed method has been tested on a 30-layer convolutional neural network and successfully converged, without significant variance fluctuations during forward and backward propagation.

3. Experimental setup

3.1 Experiment 1: Stock price prediction

3.1.1 Data description

The four companies *International Business Machine (IBM)*, *Intel*, *Nvidia* and *Genomic Health* data has been used for the experiment. The data has been downloaded from the *Yahoo Finance* website [2]. Table 3.1.1 shows the start date of the data record and number of dates available in the downloaded file for each company.

Table 3.1.1. Start date of the data record and number of available dates for each of the four companies

Company	The data record's first day	Number of available dates
IBM	02.01.1962	14507
Intel	17.03.1980	9942
Nvidia	22.01.1999	5182
Genomic Health	29.09.2005	3495

The data matrix is organized in the way that each row represents one date and each column one price feature. The price consists of six features: open, high, low, close, adjusted close price and volume. The opening price is the price of the first trade for the particular stock on a trading day. The closing price is the last price at which a stock was traded before the market closed that day. The highest price and the lowest price represent the highest and the lowest value that stock has reached that trading day. Therefore, the trading can be made multiple times in one day, not necessary from day to day. The adjusted closing price reflects the closing price of a stock in relation to other stock price attributes. It is considered as more realistic stock price feature than the closing price.A volume is a total number of buyers and sellers exchanging shares on a trading day.

In Figure 3.1.1 *IBM's* open price, high price, low price, close price and adjusted close price features are shown. In Figure 3.1.2 *IBM's* volume feature is represented. In Figure 3.1.3 *Intel's* first five stock price features are plotted and in Figure 3.1.4 *Intel's* volume feature is plotted. Figure 3.1.5 and Figure 3.1.6 illustrate *Nvidia's* and *Genomic Health's* open price, high price, low price, close price and adjusted

close price features, respectively, while Figure 3.1.7 and Figure 3.1.8 illustrate *Nvidia's* and *Genomic Health's* volume feature. The x-axis represents date of record, while y-axis represents features values.

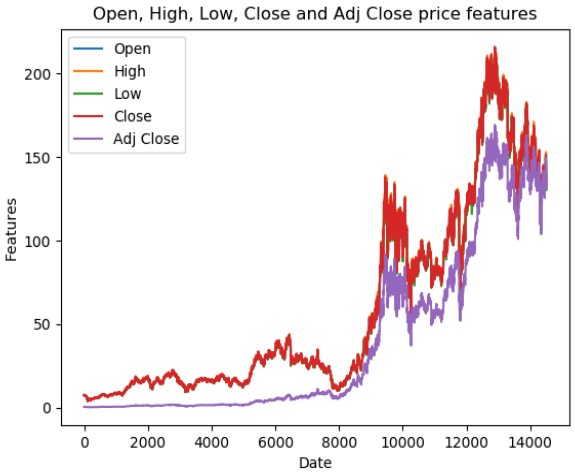

Figure 3.1.1. IBM's open, high, low, close and adjusted close price features

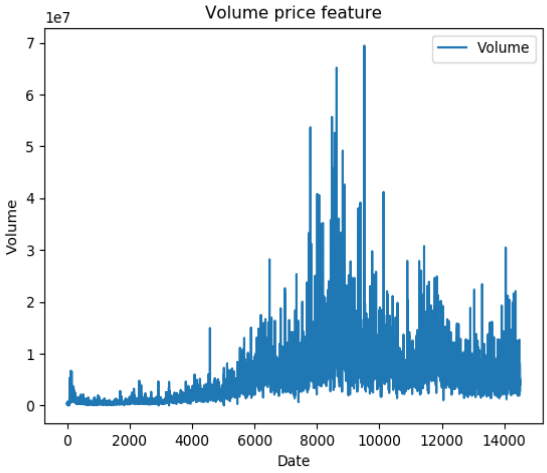

Figure 3.1.2. IBM's volume price feature

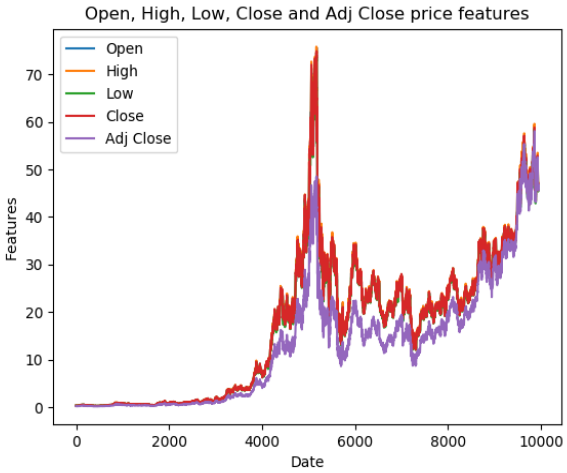

Figure 3.1.3. Intel's open, high, low, close and adjusted close price features

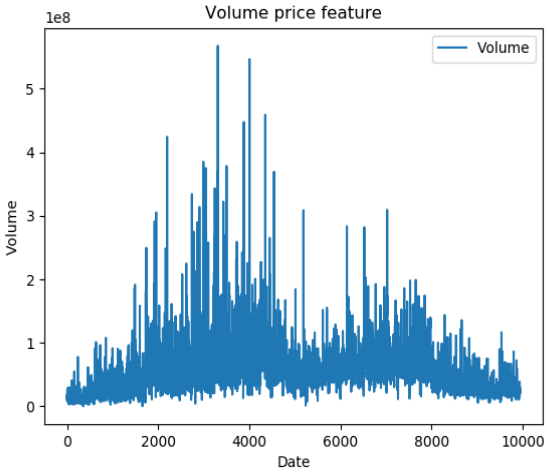

Figure 3.1.4. Intel's volume price feature

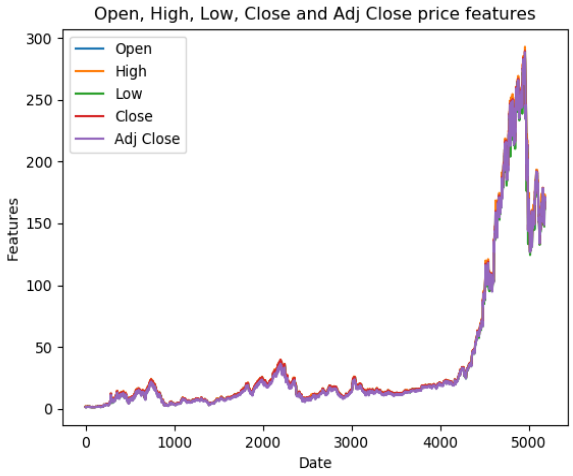

Figure 3.1.5. Nvidia's open, high, low, close and adjusted close price features

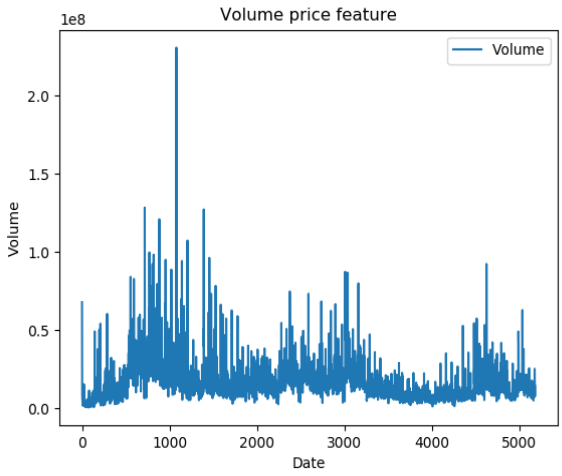

Figure 3.1.6. Nvidia's volume price feature

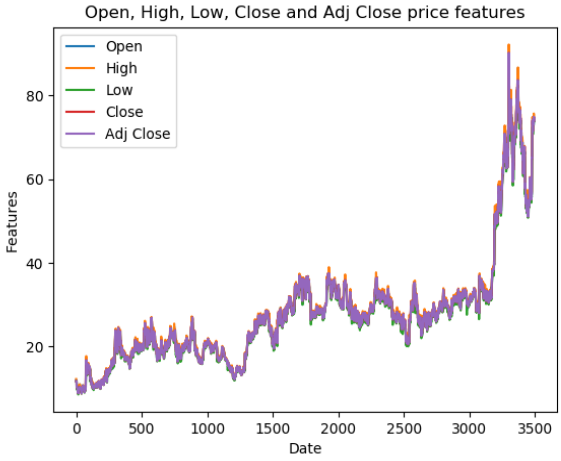

Figure 3.1.7. Genomic Health's open, high, low, close and adjusted close price features

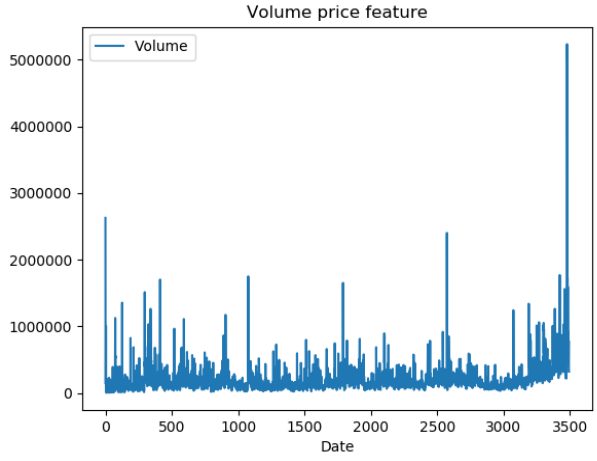

Figure 3.1.8. Genomic Health's volume price feature

The total data has been split on training and testing dataset in ratio 0.9. The volume feature hasn't been included because it degrades network's performance and acts as random noise. Normalization as preprocessing step has been applied on the dataset, scaling all the feature's values in a range between zero and one. The window for making predictions consists of 50 successive dates and it is used for forecasting opening and adjusted closing price tomorrow, day after tomorrow and two days after tomorrow.

3.1.2 Network architecture

The LSTM network consists of one input layer, two hidden layers and one output layer. Ouput activation function in recurrent layers is hyperbolic tangent (Formula 2.2.3), recurrent function is logistic sigmoid (Formula 2.2.4), Xavier initialization is used for recurrent weight initialization (Section 2.2.5.3). *Dropout* of 20% randomly chosen outputs has been applied after each recurrent layer (Section 2.2.5.2). The algorithm tries to minimize root mean squared error loss function (Formula 2.2.21) with *warm-up* period of five time steps by employing Adam optimizer for weights update (Section 2.2.5.1). The network has been trained for 100 epochs with *batch* size of 32 and early stopping as the second regularization method has been applied (Section 2.2.5.2).

In Figure 3.1.9 network's architecture has been illustrated. Connections between network's layers can be easily visualized from this graph.

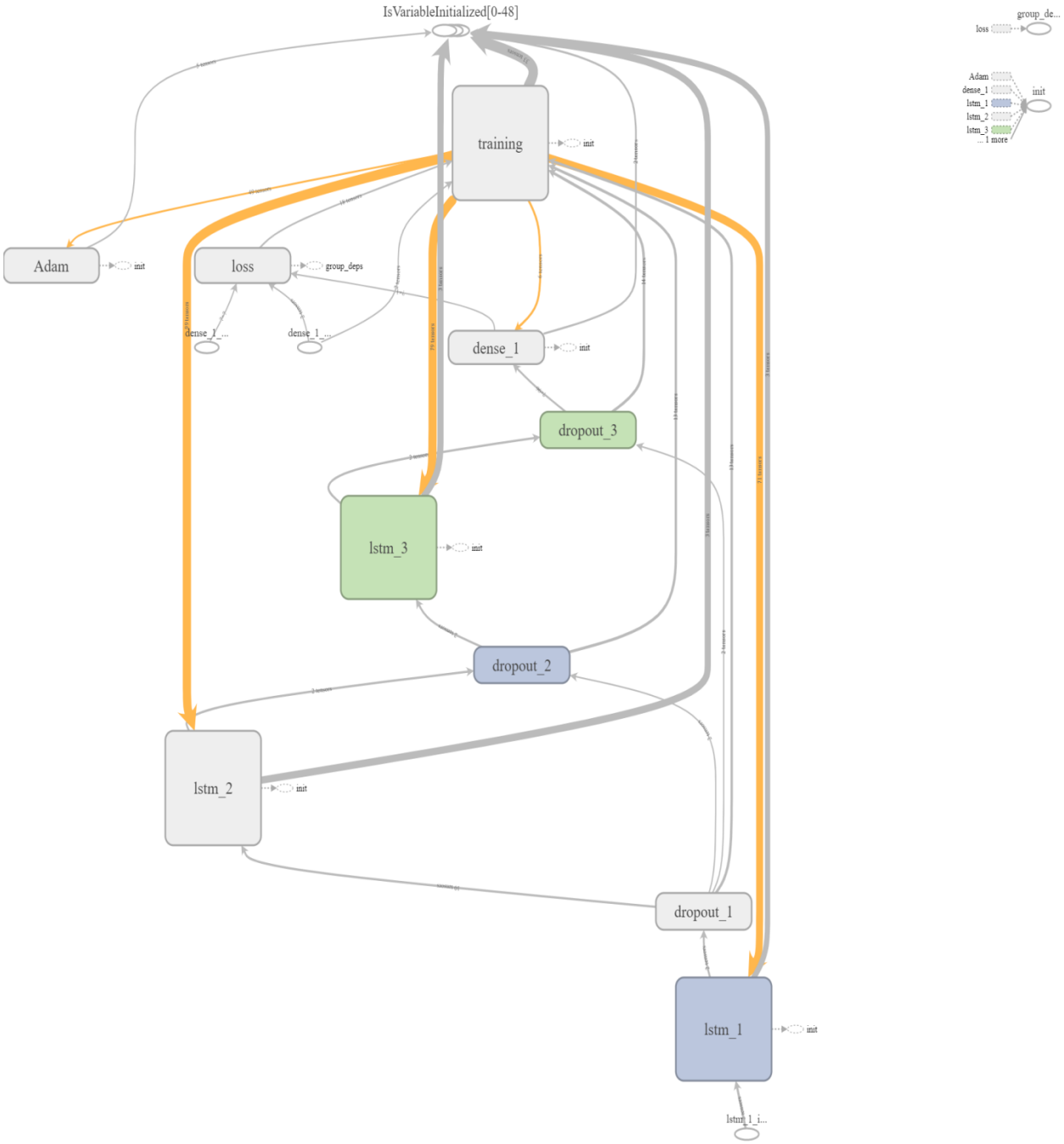

Figure 3.1.9. Network architecture

3.2 Experiment 2: Anomaly detection in ECG time signals

3.2.1 Data description

Electrocardiograms are widely used as an inexpensive and noninvasive method for detecting anomalies in a heart's rhythm. It is a test that records heart's electrical activity through small electrode patches attached to the skin of chest, arms and legs. An ECG can be used to measure and detect the underlying rate and rhythm mechanism of the heart, the orientation of the heart in the chest cavity, the evidence of damage to the various parts of the heart muscle, the effects of cardiac drugs and the function of implanted pacemakers. Some of the cardiovascular abnormalities in a heart's rhythm that can be detected from ECG signal are:

- Atrial premature beat: The heart has four chambers. The upper two are called the "atria" and the lower two are called the "ventricles". Premature atrial contraction occurs when the heart's electrical system triggers the early or extra beat in the atria. It is a common arrhythmia type and most of the time it doesn't require any treatment.
- Paced beat: A pacemaker is a small device that helps heart beats more regularly. The pacemaker rhythm can easily be recognized on the ECG. It shows pacemaker spikes, vertical signals that represent electrical activity of the pacemaker.
- Right bundle branch block beat: Bundle branch block beat occurs when there is a delay in electrical impulses that coordinates heartbeats. The delay can occur on the path that sends electrical impulses to the right or the left lower chamber of the heart. Therefore, right or left bundle branch block beat can be detected.
- Premature ventricular contraction: Premature ventricular contraction occurs when the heart's electrical system triggers the early or extra beat in the ventricle. Like atrial premature beat, premature ventricular contraction doesn't require any special treatment.

The database used in this experiment is MIT-BIH arrhythmia database, downloaded from the PhysioBank [4, 5]. In 1975 Massachusetts Institute of Technology (MIT) obtained the long-term ECG recordings (Holter recordings) by the Arrhythmia Laboratory of Boston's Beth Israel Hospital (BIH) for research purpose. Long-term Holter recording refers to recording of ECG signal with duration of 24 hours. MIT-BIH arrhythmia database consists of 48 half-hour excerpts of two-channel studied by the BIH Arrhythmia Laboratory between 1975 and 1979. In most cases, one channel is modified limb lead II (MLII) and the other is V1. The 23 recordings in the "100 series" were randomly chosen from 4000 Holter recordings and the other 25 recordings in the "200 series" were chosen to represent rare, but meaningful combination of abnormalities in a heartbeat signal. The digitization rate is 360 samples per

second per channel [5]. Twenty different types of heartbeats (one type denotes normal beat and other nineteen types stand for abnormal beats) are present in MIT-BIH arrhythmia database. In this work, the detection of four different anomaly types - atrial premature beat, paced beat, right bundle branch block beat and premature ventricular contraction has been performed.

In Figure 3.2.1 the atrial premature beat has been emphasized in red color on the electrical signal measured by modified limb lead II. In Figure 3.2.2 the same heartbeat type has been underlined by red color on the electrical signal measured by V1 lead. In Figures 3.2.3-3.2.4, 3.2.5-3.2.6, 3.2.7-3.2.8 paced beat, right bundle branch block beat and premature ventricular contraction have been shown in red color on the electrical signal measured by MLII and V1 leads, respectively.

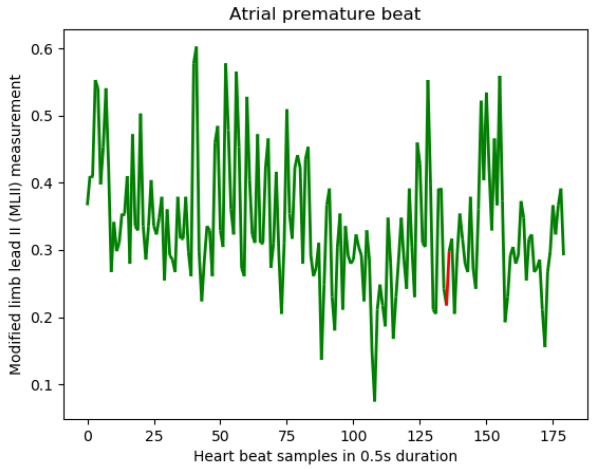

Figure 3.2.1. Atrial premature beat in electrical signal measured by modified limb lead II

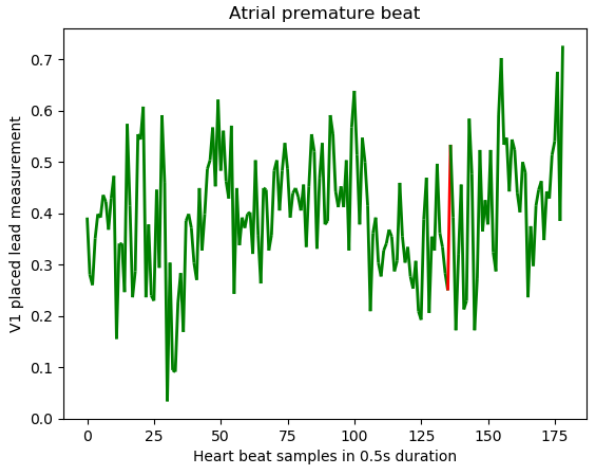

Figure 3.2.2. Atrial premature beat in electrical signal measured by lead V1

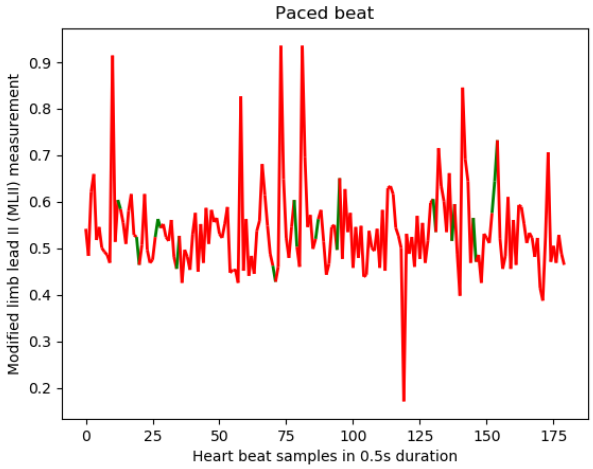

Figure 3.2.3. Paced beat in electrical signal measured by modified limb lead II

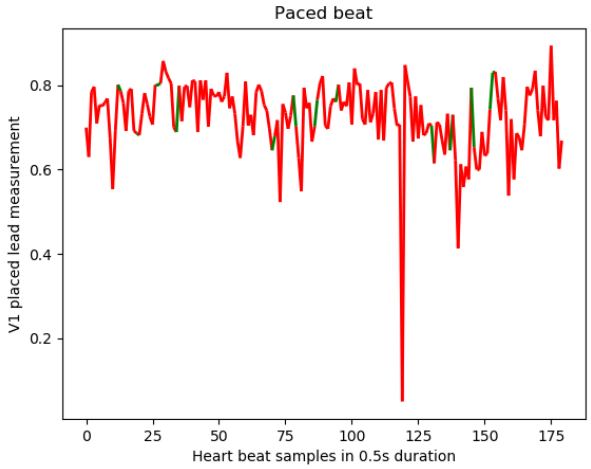

Figure 3.2.4. Paced beat in electrical signal measured by lead V1

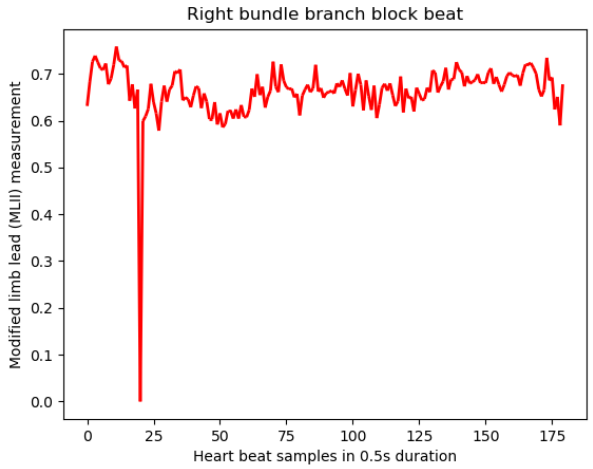

Figure 3.2.5. Right bundle brunch block beat in electrical signal measured by modified limb lead II

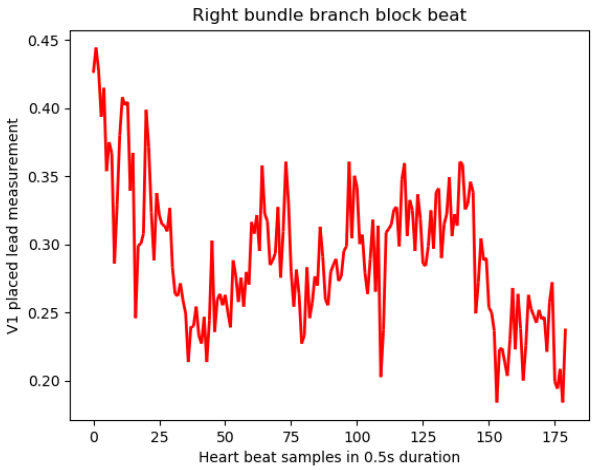

Figure 3.2.6. Right bundle brunch block beat in electrical signal measured by lead V1

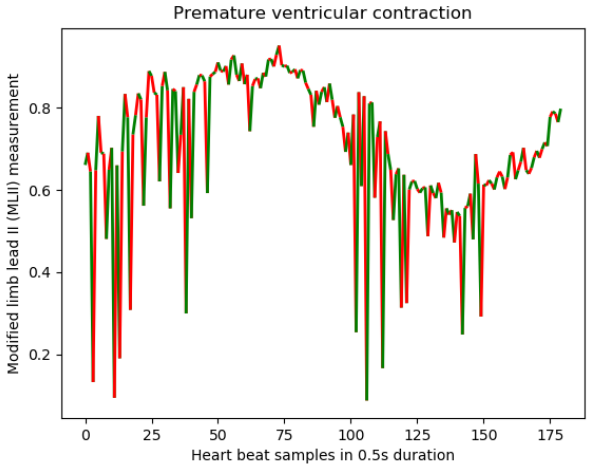

Figure 3.2.7. Premature ventricular contraction in electrical signal measured by modified limb lead II

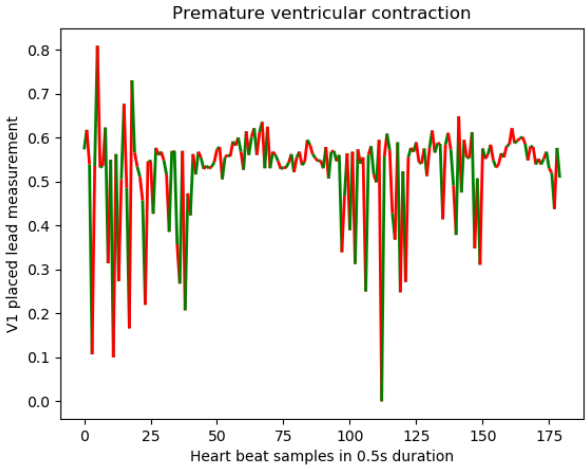

Figure 3.2.8. Premature ventricular contraction in electrical signal measured by lead V1

Electrical signals measured by modified limb lead II and lead V1 have been used for anomaly detection. In Table 3.2.1 number of beats without abnormalities, atrial premature beats, paced beats, right bundle brunch block beats and premature ventricular contraction beats in the test dataset have been shown. Training and testing dataset have been split in ratio 0.8. Data normalization has been applied on features scaling them in a range between zero and one. The window consists of 50 successive heartbeats and it is used for anomaly detection as well as anomaly type detection.

Table 3.2.1. Number of characteristic beats in a test dataset

Number of beats / **Heartbeat type**	
Heartbeat without anomaly	14637
Atrial premature beat	516
Paced beat	1385
Right bundle brunch block beat	1437
Premature ventricular contraction beat	1446

3.2.2 Network architecture

Network architecture similar to the one described in Section 3.1.2 has been used for the anomaly detection in ECG time signal. The LSTM network consists of one input layer, two hidden layers and one output layer. Output activation function in recurrent layers is hyperbolic tangent (Formula 2.2.3), recurrent function is logistic sigmoid (Formula 2.2.4), Xavier initialization is used for recurrent weight initialization (Section 2.2.5.3). *Dropout* of 20% randomly chosen outputs has been applied after each recurrent layer (Section 2.2.5.2). For binary classification problem logistic sigmoid activation function in output layer has been used and algorithm was minimizing binary cross-entropy. For multiclass classification problem softmax activation function (Formula 2.2.13) has been used and algorithm was minimizing categorical cross-entropy. The accuracy (Formula 2.2.67) and F1-score (Formula 2.2.71) have been applied as classification metrics. The network has been trained for 100 epochs with *batch* size of 32, Adam optimizer has been employed for weights update (Section 2.2.5.1) and early stopping as the second regularization method has been applied (Section 2.2.5.2).

The network architecture is the same as network architecture used for stock price prediction and it is illustrated in Figure 3.1.9.

4. Results

4.1 Experiment 1: Stock price prediction

The prediction results obtained using a neural network whose architecture is described in Section 3.2 over the dataset described in Section 3.1 are illustrated in Figures 4.1.1- 4.1.24. The forecasts have been made for opening and adjusted closing price one day in advance, two days in advance and three days in advance for each of four companies using a window of size 50. The real stock price trends are plotted in red color, while predicted stock prices have been plotted in blue color. In Figure 4.1.1 *IBM's* opening price one day in advance for each date in a testing dataset has been plotted, in Figure 4.1.2 opening price two days in advance is shown, and in Figure 4.1.3 opening price three days in advance has been plotted. Figures 4.1.4 - 4.1.6 illustrate *IBM's* adjusted closing price one day, two days and three days in advance. Figures 4.1.7- 4.1.9, 4.1.13 – 4.1.15, 4.1.19 – 4.1.21 illustrate *Intel's*, *Nvidia's* and *Genomic Health's* opening price one day, two days and three days in advance, respectively. Figures 4.1.10 - 4.1.12, 4.1.16 – 4.1.18, 4.1.22 – 4.1.24 show *Intel's*, *Nvidia's* and *Genomic Health's* adjusted closing price one day, two days and three days in advance, respectively. The Table 4.1.1 summarizes root mean squared errors for each opening price prediction. In Table 4.1.2 root mean squared errors for adjusted closing price predictions have been listed.

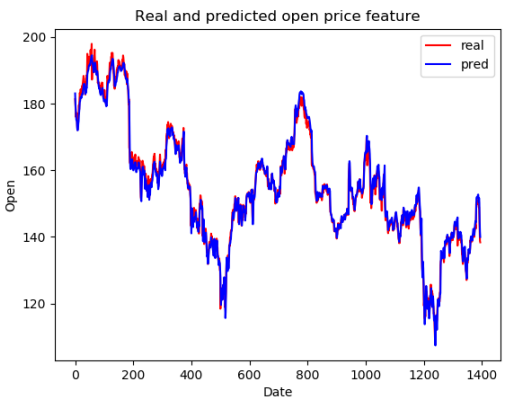

Figure 4.1.1. IBM's open price feature one day in advance

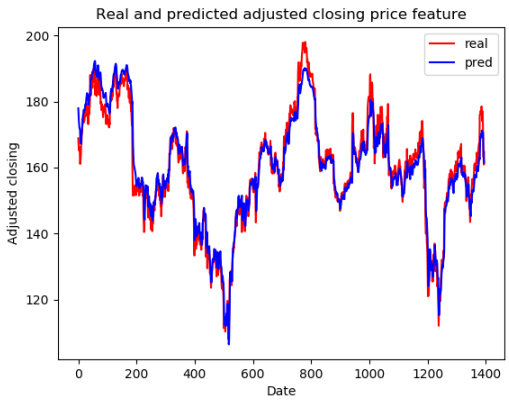

Figure 4.1.4. IBM's adjusted close price feature one day in advance

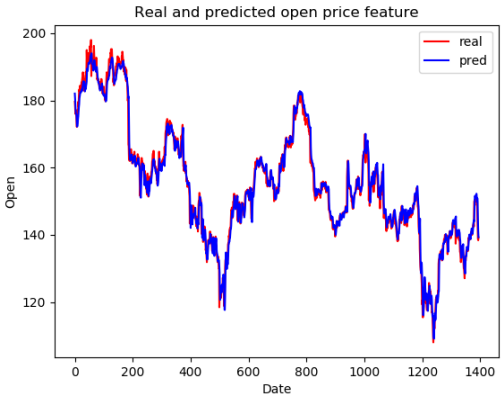

Figure 4.1.2. IBM's open price feature two days in advance

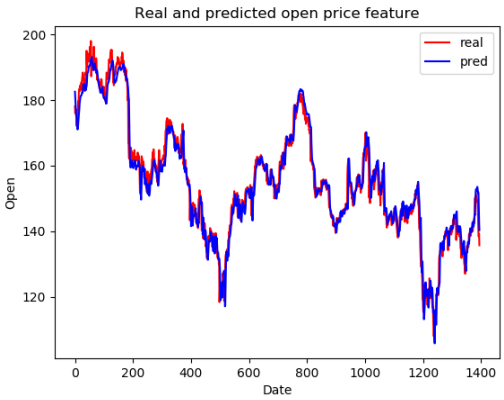

Figure 4.1.3. IBM's open price feature three days in advance

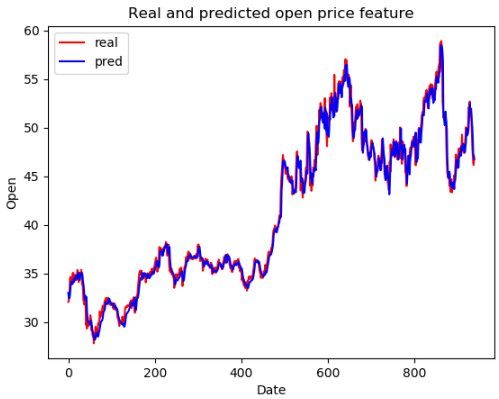

Figure 4.1.7. Intel's open price feature one day in advance

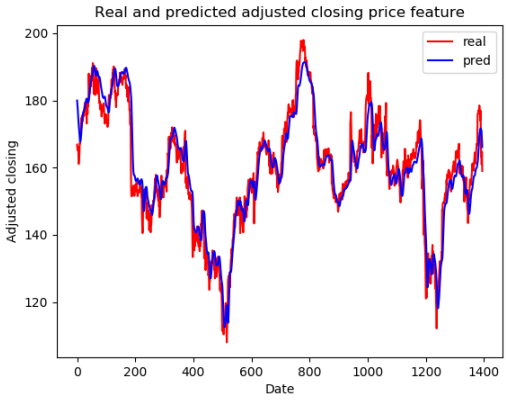

Figure 4.1.5. IBM's adjusted close price feature two days in advance

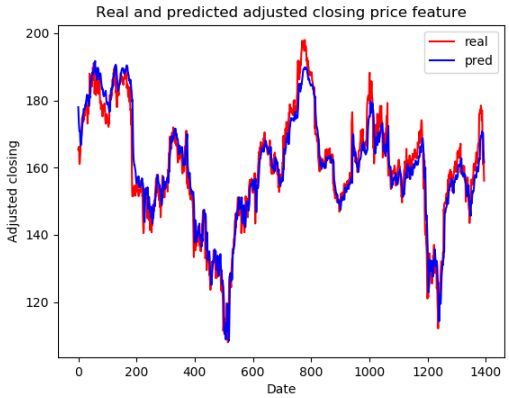

Figure 4.1.6. IBM's adjusted close price feature three days in advance

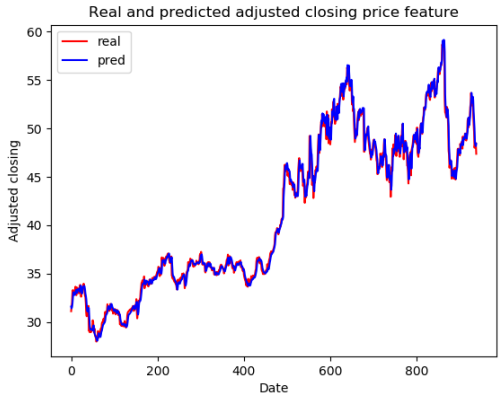

Figure 4.1.10. Intel's adjusted close price feature one day in advance

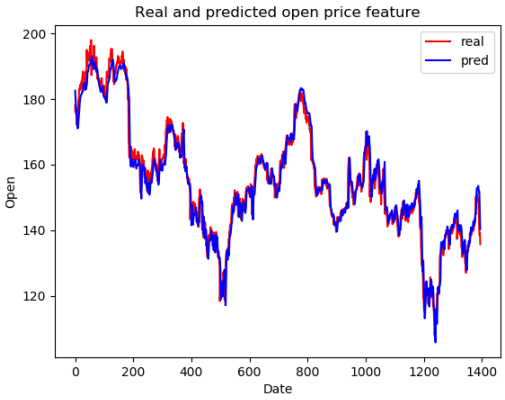

Figure 4.1.8. Intel's open price feature two days in advance

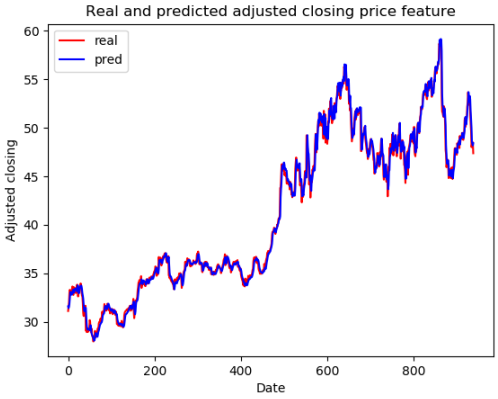

Figure 4.1.11. Intel's adjusted close price feature two days in advance

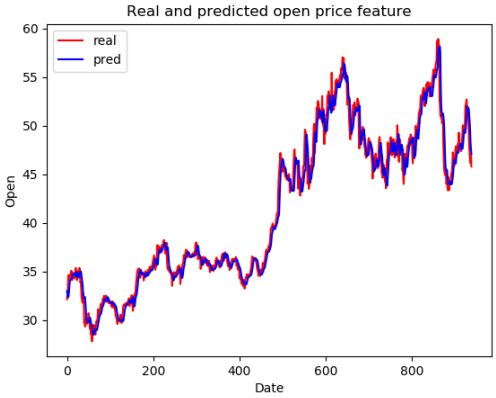

Figure 4.1.9. Intel's open price feature three days in advance

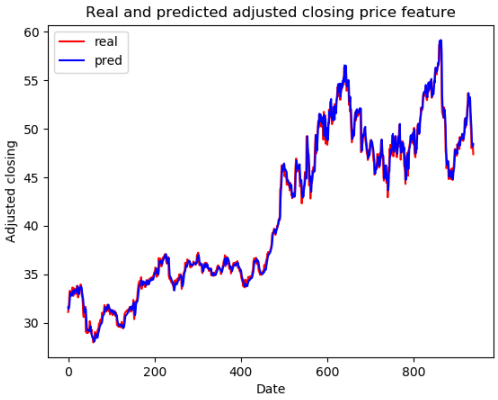

Figure 4.1.12. Intel's adjusted close price feature three days in advance

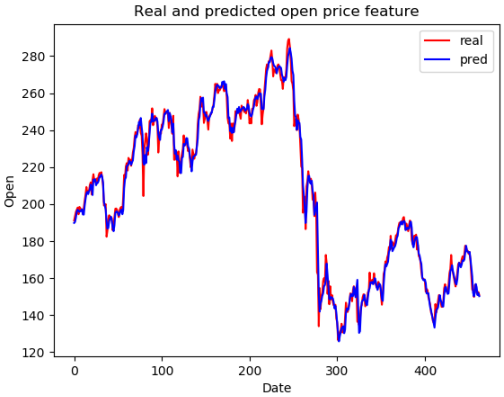

Figure 4.1.13. Nvidia's open price feature one day in advance

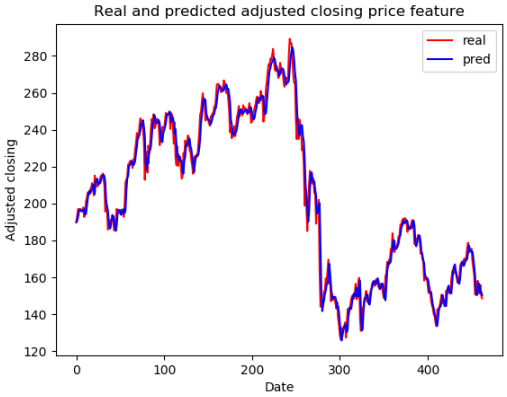

Figure 4.1.16. Nvidia's adjusted close price feature one day in advance

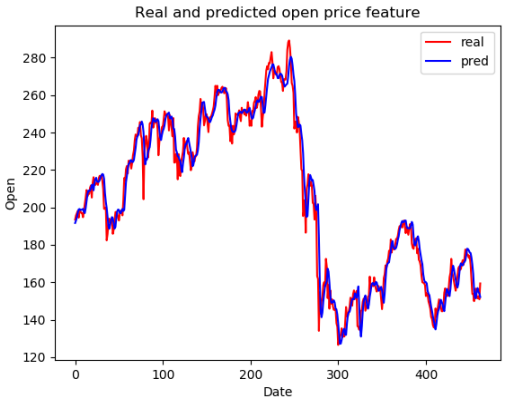

Figure 4.1.14. Nvidia's open price feature two days in advance

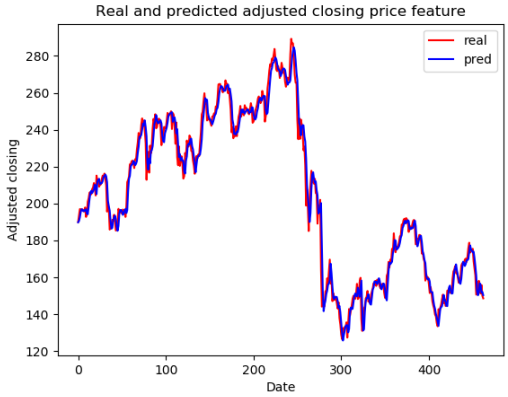

Figure 4.1.16. Nvidia's adjusted close price feature two days in advance

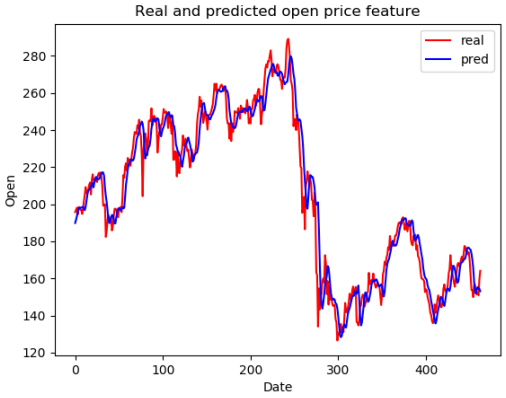

Figure 4.1.15. Nvidia's open price feature three days in advance

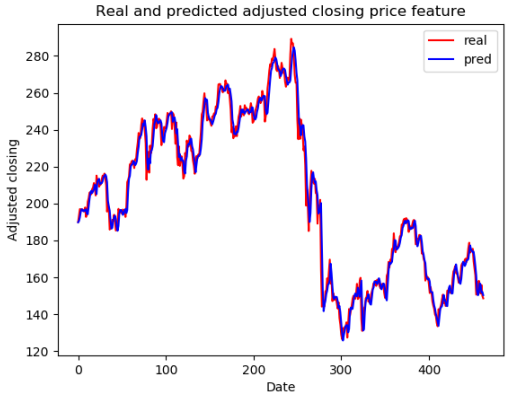

Figure 4.1.17. Nvidia's adjusted close price feature three days in advance

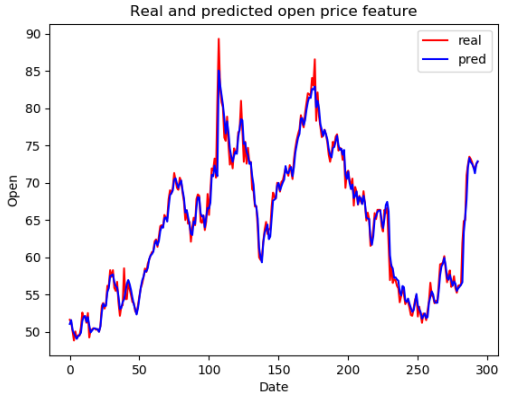

Figure 4.1.19. Genomic Health's open price feature one day in advance

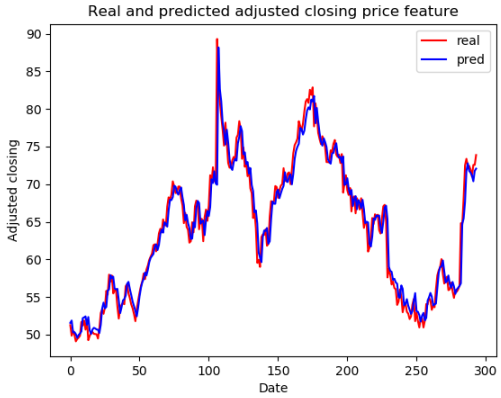

Figure 4.1.22. Genomic Health's adjusted close price feature one day in advance

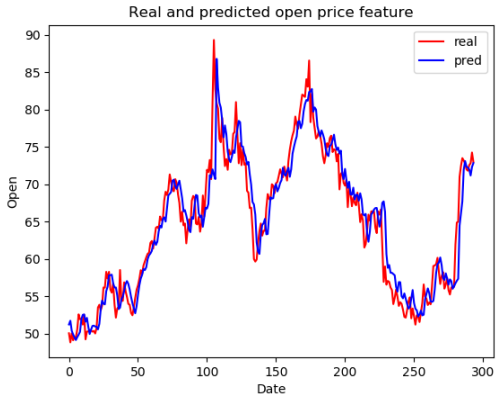

Figure 4.1.20. Genomic Health's open price feature two days in advance

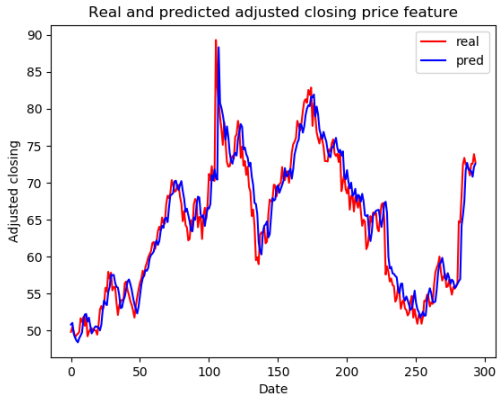

Figure 4.1.23. Genomic Health's adjusted close price feature two days in advance

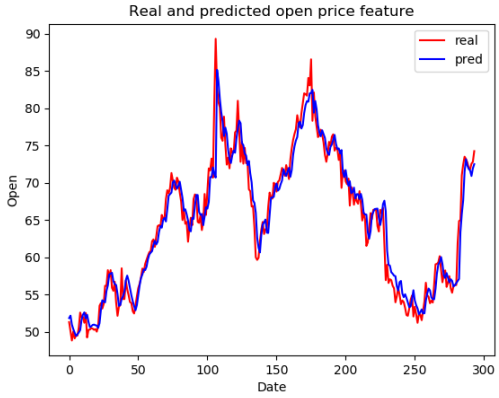

Figure 4.1.21. Genomic Health's open price feature three days in advance

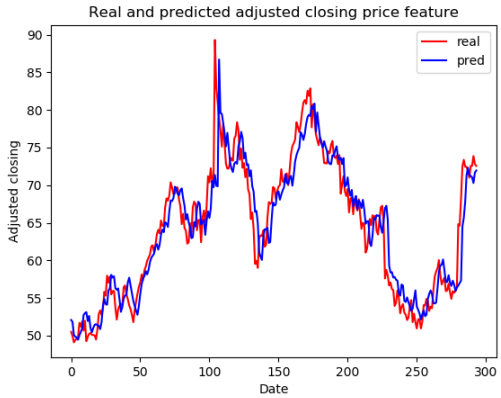

Figure 4.1.24. Genomic Health's adjusted close price feature three days in advance

Table 4.1.1. Root mean squared errors for opening price feature prediction

Prediction days / **Company**	One day	Two days	Three days
IBM	1.7698	2.9541	3.4244
Intel	0.6278	0.9164	1.1958
Nvidia	4.4937	7.8715	9.8871
Genomic Health	1.1408	2.3349	2.8907

Table 4.1.2. Root mean squared errors for adjusted closing price feature prediction

Prediction days / **Company**	One day	Two days	Three days
IBM	3.7000	5.1422	5.0144
Intel	0.7162	0.9765	1.0146
Nvidia	6.1180	7.5121	7.4076
Genomic Health	2.0402	2.7046	3.1497

4.2 Experiment 2: Anomaly detection in ECG time signals

The LSTM network has been applied to two types of problems concerning anomaly detection in ECG time signals. The first problem refers to anomaly binary classification, while the second problem refers to anomaly multiclass classification. In the first problem the LSTM network whose architecture has been described in Section 3.2.2 learned to classify heartbeats from the dataset described in Section 3.2.1 in two classes, normal heartbeat and the heartbeat with anomaly. The overview of data has been provided in Table 3.2.1. For the binary classification problem heartbeats without anomaly, atrial premature beats, paced beats and premature ventricular contraction beats (without right bundle brunch block beats) have been given as input to the neural network. Predictions have been made for each beat (360 samples per second). However only signals during the first second, tenth second, twenty second and thirty second have been plotted here. In Figure 4.2.1 actual anomalies have been enhanced in red on electrical signal measured by modified limb lead II during the first second. In Figure 4.2.2 predicted anomalies have been underlined in blue on the same electrical signal. Figure 4.2.3 shows actual anomalies in red color during the tenth second, while Figure 4.2.4 shows predicted anomalies in blue color during the same time frame. Figures 4.2.5-4.2.6 and Figures 4.2.7-4.2.8 illustrate the real and predicted anomalies during the twenty and thirty second, respectively. The mean squared error, accuracy, confusion matrix and F1-score have been measured on the test dataset.

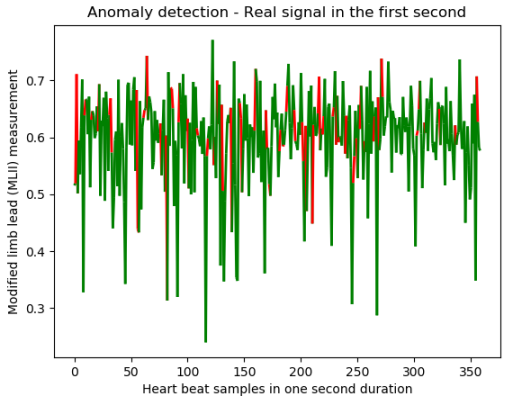

Figure 4.2.1. Real anomalies on the electrical signal measured by modified limb lead during the first second

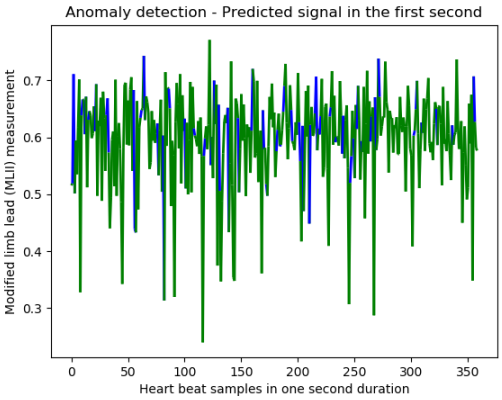

Figure 4.2.2. Predicted anomalies on the electrical signal measured by modified limb lead during the first second

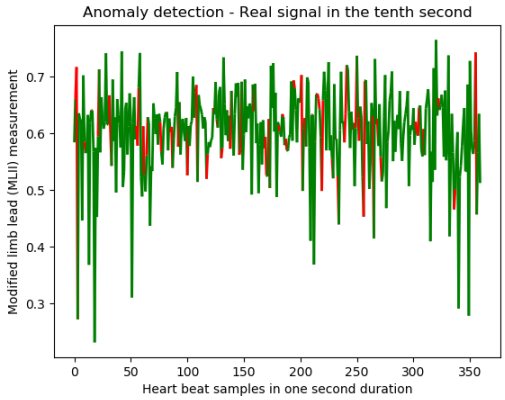

Figure 4.2.3. Real anomalies on the electrical signal measured by modified limb lead during the tenth second

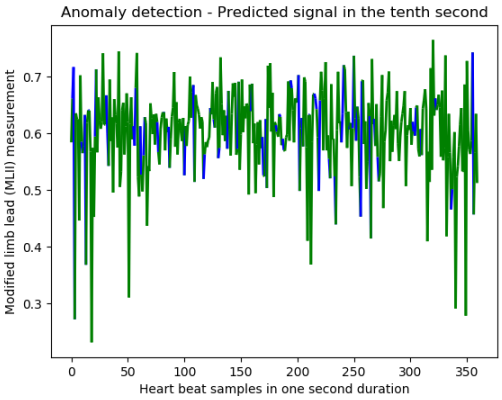

Figure 4.2.4. Predicted anomalies on the electrical signal measured by modified limb lead during the tenth second

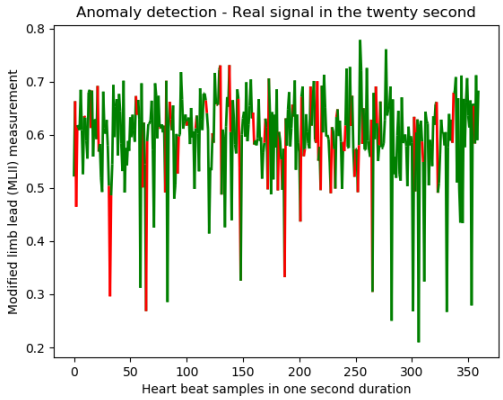

Figure 4.2.5. Real anomalies on the electrical signal measured by modified limb lead during the twenty second

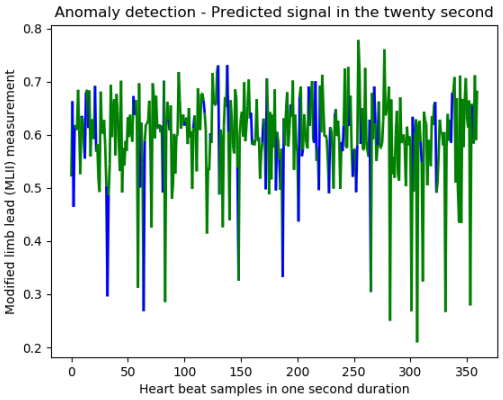

Figure 4.2.6. Predicted anomalies on the electrical signal measured by modified limb lead during the twenty second

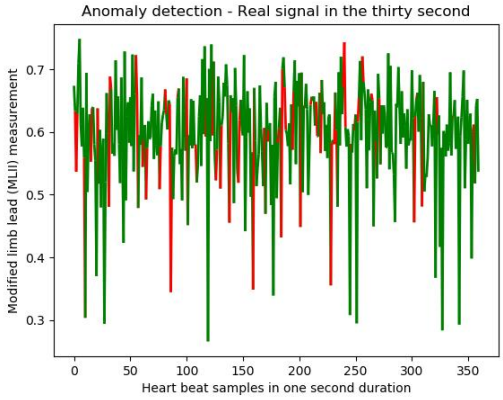

Figure 4.2.7. Real anomalies on the electrical signal measured by modified limb lead during the thirty second

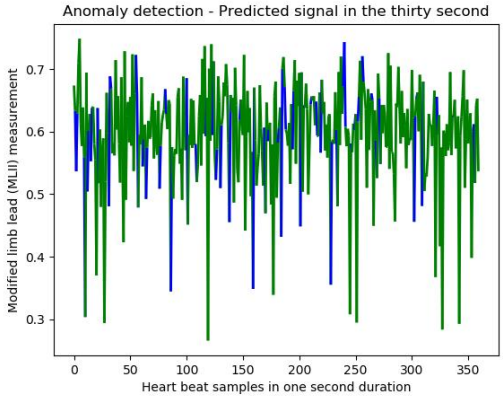

Figure 4.2.8. Predicted anomalies on the electrical signal measured by modified limb lead during the thirty second

The mean squared error measured on test dataset is 0.0513, the accuracy is 98.4418% and F1-score is 0.9537.

The confusion matrix is $cm = \begin{bmatrix} 14601 & 44 \\ 236 & 3088 \end{bmatrix}$.

The second problem was anomaly multiclass classification. The neural network's task was to learn to distinguish between heartbeats without anomaly, atrial premature beats, paced beats, premature ventricular contraction beats and right bundle brunch block beats whose numbers have been provided in Table 3.2.1. The network architecture is the same as in the binary classification problem, except for the number of neurons in an output layer and type of loss function (described in Section 3.2.2).

Prediction has been made for each beat as in the previous case. However, only signals in the first and twenty second have been shown. In Figure 4.2.9 real atrial premature beats during the first second have been underlined in red color, while in Figure 4.2.10 predicted atrial premature beats during the first second have been enhanced in blue color. In Figures 4.2.11 - 4.2.12 real and predicted atrial premature beats have been plotted during the twenty second, respectively. In Figures 4.2.13 - 4.2.14 real and predicted paced beats have been shown during the first second, while in Figures 4.2.15 - 4.2.16 real and predicted paced beats have been shown during the twenty second. In Figures 4.2.17 – 4.2.18 and in Figures 4.2.19 – 4.2.20 real and predicted right bundle branch block beats in red and blue color have been illustrated during the first and the twenty second, respectively. In Figures 4.2.21 - 4.2.22 and in Figures 4.2.23 - 4.2.24 real and predicted ventricular contraction beats during the first and during the twenty second have been shown. As in the previous case the mean squared error, accuracy, confusion matrix and F1-score have been measured on the test dataset.

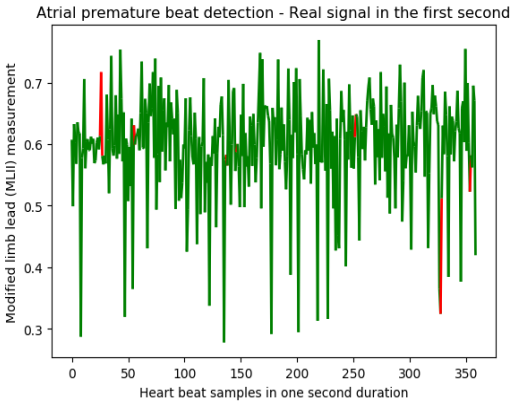

Figure 4.2.9. Real atrial premature beats on the electrical signal measured by modified limb lead during the first second

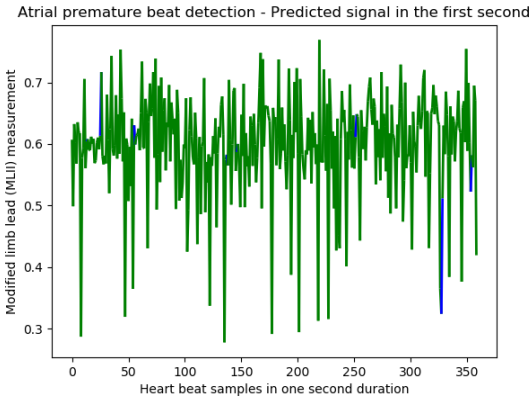

Figure 4.2.10. Predicted atrial premature beats on the electrical signal measured by modified limb lead during the first second

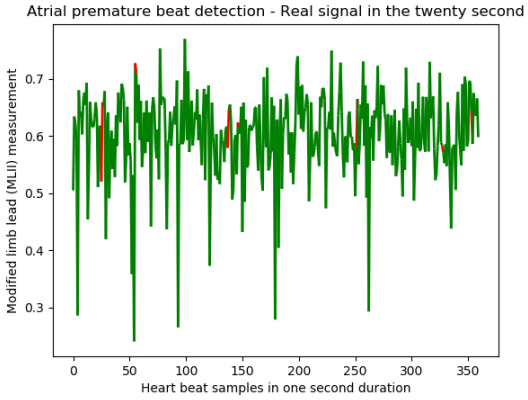

Figure 4.2.11. Real atrial premature beats on the electrical signal measured by modified limb lead during the twenty second

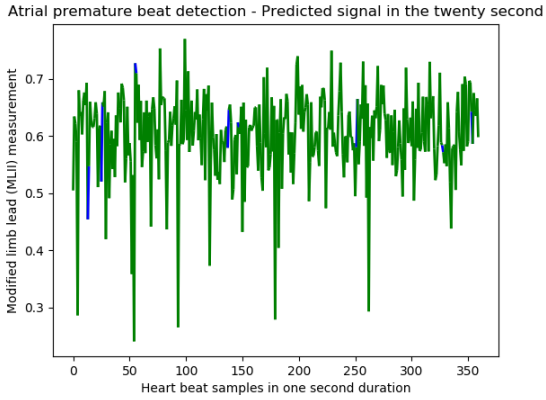

Figure 4.2.12. Predicted atrial premature beats on the electrical signal measured by modified limb lead during the twenty second

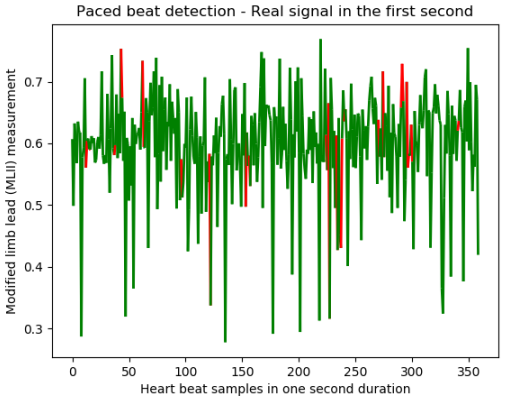

Figure 4.2.13. Real paced beats on the electrical signal measured by modified limb lead during the first second

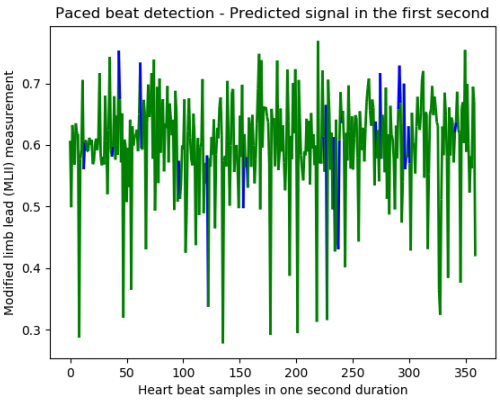

Figure 4.2.14. Predicted paced beats on the electrical signal measured by modified limb lead during the first second

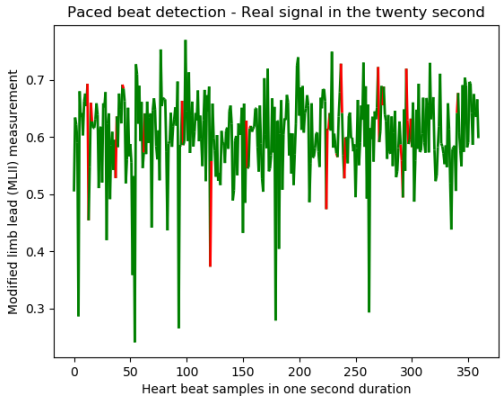

Figure 4.2.15. Real paced beats on the electrical signal measured by modified limb lead during the twenty second

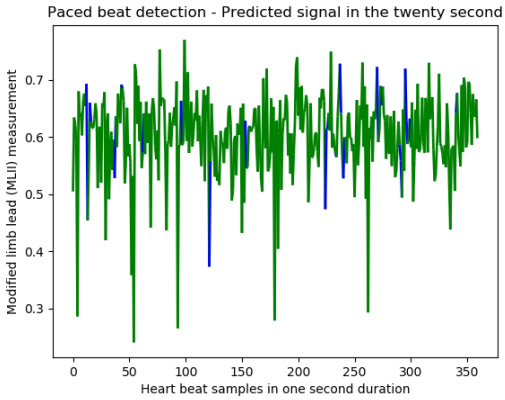

Figure 4.2.16. Predicted paced beats on the electrical signal measured by modified limb lead during the twenty second

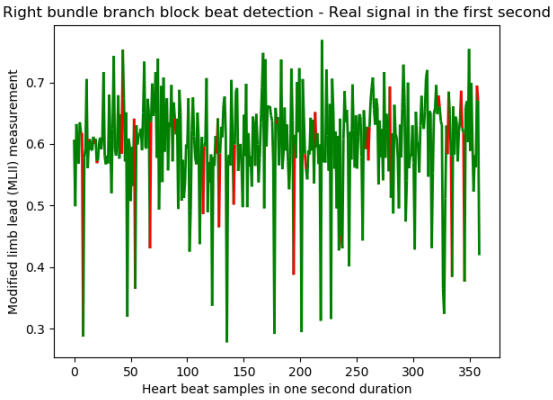

Figure 4.2.17. Real right bundle branch block beats on the electrical signal measured by modified limb lead during the first second

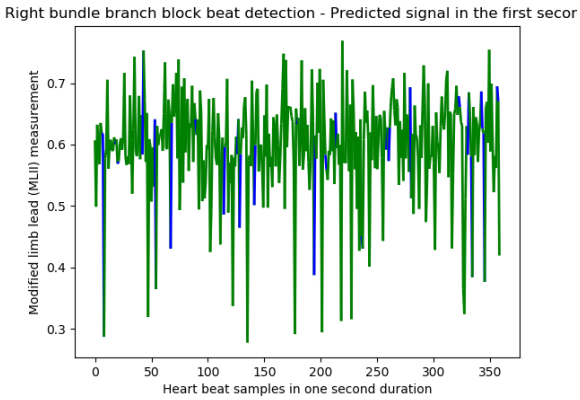

Figure 4.2.18. Predicted right bundle branch block beats on the electrical signal measured by modified limb lead during the first second

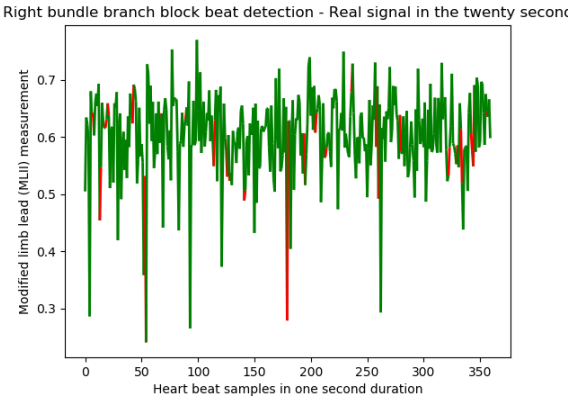

Figure 4.2.19. Real right bundle branch block beats on the electrical signal measured by modified limb lead during the twenty second

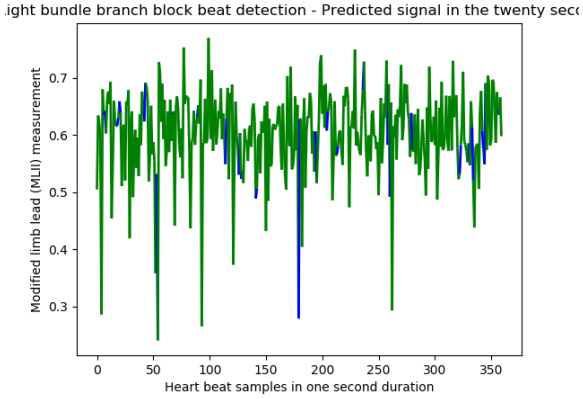

Figure 4.2.20. Predicted right bundle branch block beats on the electrical signal measured by modified limb lead during the twenty

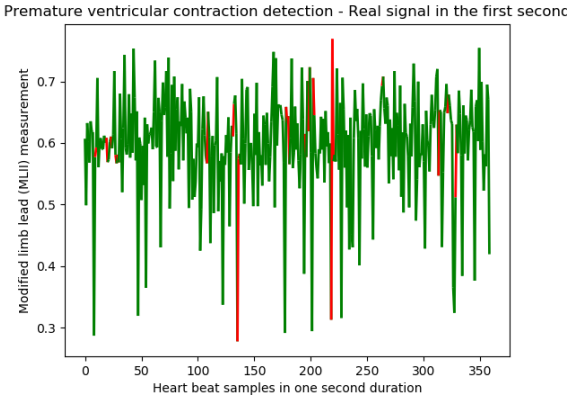

Figure 4.2.21. Real premature ventricular contraction beats on the electrical signal measured by modified limb lead during the first second

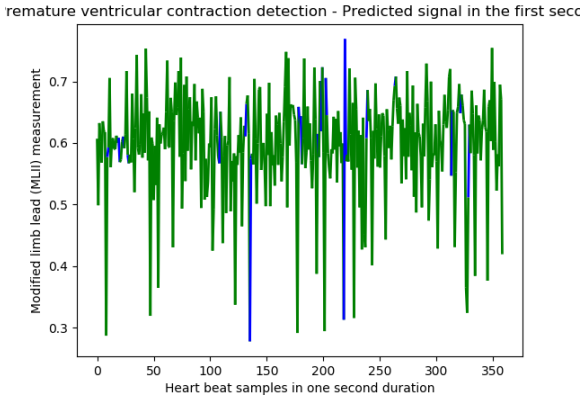

Figure 4.2.22. Predicted premature ventricular contraction beats on the electrical signal measured by modified limb lead during the first second

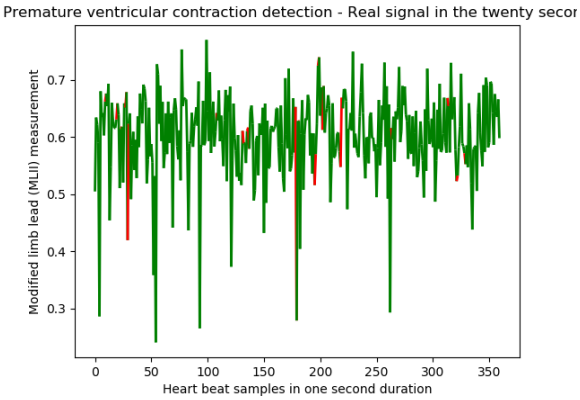

Figure 4.2.23. Real premature ventricular contraction beats on the electrical signal measured by modified limb lead during the twenty second

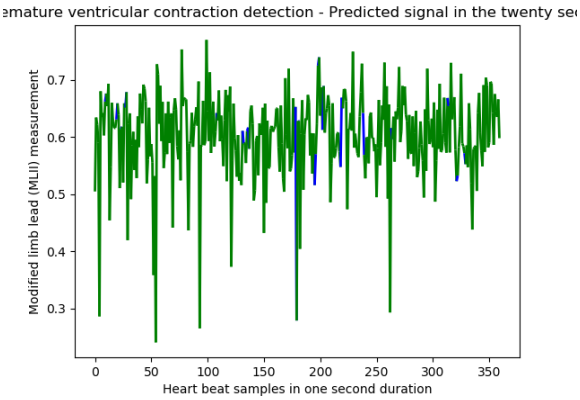

Figure 4.2.24. Predicted premature ventricular contraction beats on the electrical signal measured by modified limb lead during the twenty second

The mean squared error measured on test dataset is 0.0627 , the accuracy is 98.0694% and F1-score is 0.9807 .

The confusion matrix is $cm = \begin{bmatrix} 14610 & 3 & 1 & 21 & 2 \\ 137 & 353 & 0 & 0 & 26 \\ 4 & 0 & 1380 & 1 & 0 \\ 98 & 0 & 3 & 1331 & 5 \\ 2 & 72 & 0 & 0 & 1372 \end{bmatrix}$

5. Discussion

The long-short term memory network has been applied to two time-sequence datasets. The tasks of predicting the opening and adjusted closing stock price features one day, two days and three days in advance, binary anomaly classification in ECG time signals and multiclass anomaly classification in ECG time signals have been successfully performed. This can be confirmed by careful observation of Figures 4.1.1 – 4.1.24 and Figures 4.2.1 – 4.2.24, low root mean squared error values listed in Table 4.1.1 and Table 4.1.2, low mean squared error values and high F1-score for binary and multiclass anomaly classification problems. This section describes problems whose overcoming would improve system's performance and compare results of this work with previous projects in the same field. It is divided into two parts: the first one investigates the stock price prediction and the second refers to anomaly detection.

5.1 Experiment 1: Stock price prediction

One of the main challenges of the stock price prediction problem using recurrent neural networks is the dataset structure. The stock market is open only on workdays. Therefore the information about stock price is available only for five days in the week with the gap of two days. Even though the stock market is closed during the weekend, the significant changes in stock price happen. However, those changes aren't recorded in the *Yahoo Finance* dataset. As the recurrent neural network requires information at discrete time steps, the problem occurs when the stock price on Monday should be predicted. The second problem is the insufficient amount of training examples, especially if the company has been recently established. This can be noticed by comparing *IBM's* and *Nvidia's* root mean squared errors listed in Table 4.1.1. The *IBM* has 14507 available dates, while *Nvidia* has 5182 available dates. *IBM's* root mean squared errors are almost two and a half times less than *Nvidia's* root mean squared errors. Therefore, the training and testing datasets have been split in ratio 0.9 to make more examples available for network's training.

The stock price prediction with LSTM networks has been investigated in the paper "Stock Price Prediction Using Long Short Term Memory" [14]. The LSTM network for making predictions on data of the recently established company (only ten years old) has been trained for 50 epochs on the dataset with 32 batches. Its prediction is plotted in Figure 5.1.1. The prediction illustrated in Figure 5.1.2 (same as Figure 4.1.13) looks significantly more precise than the one plotted in Figure 5.1.1.

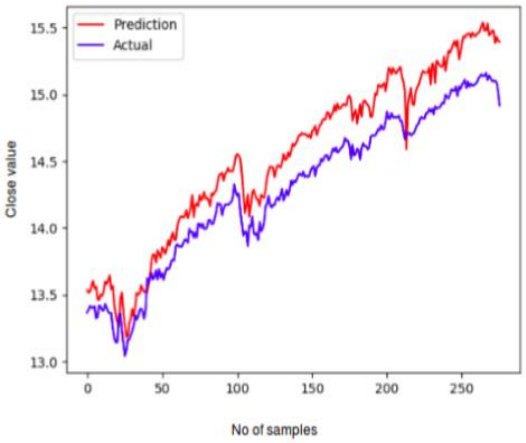

Figure 5.1.1. Stock price prediction using LSTM.
Image adapted from [14].

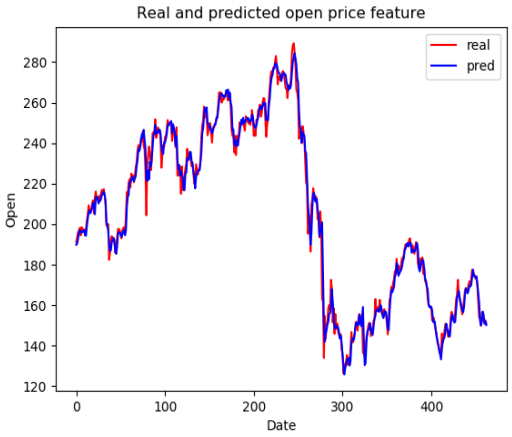

Figure 5.1.2. Nvidia's open price feature one day in advance

On the other hand, the LSTM network in [14] has been trained on the data of the company established more than 25 years ago. In Figure 5.1.3 predictions made in [14] have been plotted. Figure 5.1.4 (same as Figure 4.1.1) illustrates forecasts from this work. For the larger size dataset obtained results look visually similar.

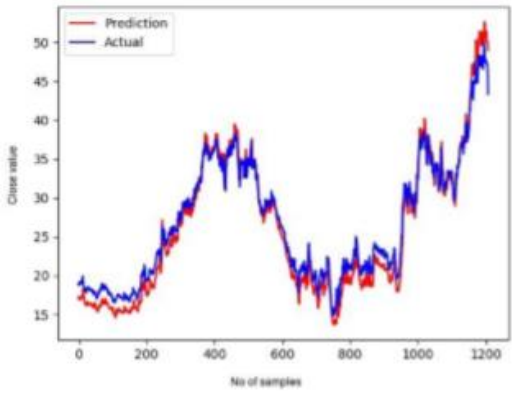

Figure 5.1.3. Stock price prediction using LSTM.
Image adapted from [14].

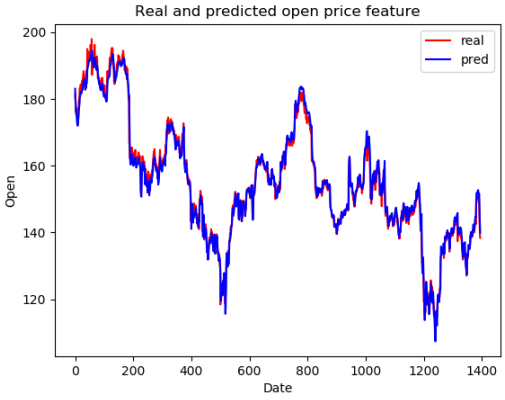

Figure 5.1.4. IBM's open price feature one day in advance

5.2 Experiment 2: Anomaly detection in ECG time signals

The biggest challenge of the anomaly detection problem was the dataset size and hardware limitation. The part of the MIT-BIH dataset that includes beats with abnormalities, atrial premature beats, paced beats, right bundle brunch block beats and premature ventricular contraction beats consists of 97152 examples. The network has been trained on the AMD A8-6410 APU with AMD Radeon R5 processor and 4GB RAM. Extensive dataset and absence of Graphics processing unit (GPU) resulted in a very long and slow network's training. The network has been trained during 100 epochs and *batch* of 32 examples has been applied. The early stopping patience parameter has been set on ten epochs. Even though the accuracy and F1-score have reached high values, the results could be even better because the network's training hasn't been stopped by the early stopping, but when the final number of epochs has been reached after 15 hours of training.

The evaluation metrics used in anomaly detection problem are accuracy and F1-score. It is important to choose the right metrics as the number of beats without abnormalities is 14637 and the number of beats with abnormalities is 4784. The accuracy of 75.3669% could be reached only by classifying all beats into normal beats class. Logically, it is more dangerous to classify beat with abnormality into wrong class than opposite. Therefore, high F1-score value is of great importance.

In paper "Automated detection of cardiac arrhythmia using deep learning techniques" the anomaly binary classification problem has been tried on the MIT-BIH arrhythmia database [19]. The accuracy of 77.4% has been achieved using three-layer LSTM network, while the highest accuracy of 83.7% has been reached by applying three-layer CNNs with GRU. It is still 14% lower than the one obtained in this project.

6. Conclusion

In this thesis LSTM networks have been successfully applied on two time – sequence datasets proving their superiority in drawing conclusions from time - series data. The thesis has been started by introducing multilayer perceptron, the base neural network, and mathematical explanation of forward and backward propagation. Next, the recurrent neural network, problems it encounters and LSTM as the proposed solution for them have been described. LSTMs complex architecture and their ability to learn long-term dependencies have been explained in detail. Two real-world datasets have been selected for conducting the experiments on them using LSTM networks. The LSTM networks have been challenged on regression problem that concerned stock price prediction one day, two days and three days in advance, binary classification problem for anomaly detection in ECG time signals and multiclass classification problem for anomaly type detection in ECG time signals. Based on the results it can be concluded that LSTMs are effective time series modelers and anomaly detectors.

The future work could include testing different neural network's architectures, optimizing learning parameters and trying regularization methods for obtaining higher accuracy and lower values of the loss function. The stock price prediction problem could be more efficiently solved by combining dataset downloaded from the *Yahoo Finance* website with the datasets obtained from various available sources. Using GPU for the neural network's training would fasten training and enable training for more training epochs. Lastly, it would be interesting to try GRUs as they simplify the LSTM architecture. GRUs are computationally more efficient than LSTMs. However, it has been found that they produce similar results to LSTMs.

References

[1] B. Wang, H. Huang and X. Wang, "A novel text mining approach to financial time series forecasting", *Neurocomputing*, 83(6): 136-45, 2012.

[2] https://finance.yahoo.com/quote/YHOO/history/ (01.09.2019)

[3] E. Braunwald, *A Textbook of Cardiovascular Medicine*, *Fifth Edition*, p.108, Philadelphia, W.B. Saunders Co., 1997, ISBN 0-7216-5666-8.

[4] G. B. Moody and R .G. Mark, "The Impact of the MIT-BIH Arrhythmia Database", *IEEE Eng in Med and Biol* 20(3):45-50 (May-June 2001). (PMID: 11446209)

[5] Goldberger AL, Amaral LAN, Glass L, Hausdorff JM, Ivanov PCh, Mark RG, Mietus JE, Moody GB, Peng C-K, Stanley HE, "Components of a New Research Resource for Complex Physiologic Signals" (2003).Circulation. 101(23):e215-e220.

[6] Z. Guo, H. Wang, Q. Liu and J. Yang, "A Feature Fusion Based Forecasting Model for Financial Time Series", *Plos One*, 2014; 9(6):172–200.

[7] V. Cherkassky, *The nature of statistical learning theory*, Springer, 1997.

[8] J. Roman and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns*", Proceedings of the Twenty-Ninth Hawaii International Conference*, vol. 2. IEEE, pp. 454–460, 1996.

[9] O. Hegazy, O. S. Soliman and M. A. Salam, "A machine learning model for stock market prediction", *arXiv preprint arXiv*:1402.7351, 2014.

[10] K. Kim and I. Han, "Genetic algorithms approach to feature discretization in artificial neural networks for the prediction of stock price index," *Expert systems with Applications*, vol. 19, no. 2, pp. 125–132, 2000.

[11] J. Heaton, N. Polson and J. Witte, "Deep learning in finance", *arXiv preprint arXiv*:1602.06561, 2016.

[12] H. Jia, "Investigation into the effectiveness of long short term memory networks for stock price prediction", *arXiv preprint arXiv*:1603.07893, 2016.

[13] M. Roondiwala, H. Patel and S. Varma, "Predicting Stock Prices Using LSTM", *International Journal of Science and Research (IJSR)* 6(4), April 2017.

[14] N. Raghav, K.R. Uttamraj, R. Vishal and Y.V. Lokeswari, "Stock Price Prediction Using Long Short Term Memory", International Research Journal of Engineering and Technology (IRJET), Volume: 05 Issue: 03, March 2018.

[15] N. Safdarian, N. Jafarnia and G. Attarodi, "A new pattern recognition method for detection and localization of myocardial infarction using T-wave integral and total integral as extracted features from one cycle of ECG signal", *Journal of Biomedical Science and Engineering*, 7(10):818-824, 2014.

[16] L.N.Sharma, R.K.Tripathy and D.Samarendra, "Multiscale energy and eigenspace approach to detection and localization of myocardial infarction", *IEEE Transactions on biomedical engineering* 62(7):1827-37, 2015.

[17] R. Acharya, H. Fujita, S. Lih Oh, Y. Hagiwara, J. Hong Tan and Muhammad Adam, "Application of deep convolutional neural network for automated detection of myocardial infarction using ECG signals", *Information Sciences* 415: 190-198, 2017.

[18] T. Reasat and C. Shahnaz, "Detection of inferior myocardial infarction using shallow convolutional neural networks", *arXiv preprint arXiv*:1710.01115, 2017.

[19] G. Swapna, K.P.Soman, R. Vinayakumar, "Automated detection of cardiac arrhythmia using deep learning techniques", *International Conference on Computational Intelligence and Data Science (ICCIDS)* 132 (2018) 1192–1201, 2018

[20] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *Bulletin of Mathematical Biophysics*, Vol 2, 1943, p.115-133

[21] F. Rosenblatt, A. Stieber, R. A. Wolf, A. E. Hurray and J. C. Hay, *MARK I PERCEPTRON OPERATORS' MANUAL*, CORNELL AERONAUTICAL LABORATORY, INC. BUFFALO 21, NEW YORK, February 1960

[22] M. Minsky and S. Papert, *Perceptrons*, Massachusetts Institute of Technology, Third printing, 1988.

[23] P.Werbos, *Beyond regression: new tools for prediction and analysis in the behavioral science,* Thesis (Ph. D.) - Harvard University, 1975.

[24] Y. Bengio, "A Connectionist Approach to Speech Recognition", *International Journal of Pattern Recognition and Artificial Intelligence 07(04)*, November 2011.

[25] J. Schmidhuber and S. Hochreiter, "LONG SHORT-TERM MEMORY", *Neural Computation 9(8):*1735-1780, 1997.

[26] G. Hinton, S. Osindero and Yee-Whye Teh, “A Fast Learning Algorithm for Deep Belief Nets”, *Neural Computation* 18, 1527–1554, 2006.

[27] D. C. Ciresan, U. Meier, L. M. Gambardella, J. Schmidhuber, “Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition”, *Neural Computation*, Volume 22, Number 12, December 2010

[28] G. Hinton, L. Deng, D. Yu, G. E. Dahl, Abdel-rahman Mohamed, N. Jaitly, A. Senior, P. Nguyen, T. N. Sainath, B. Kingsbury, “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups”, IEEE Signal Processing Magazine, Volume: 29, Issue: 6, November 2012.

[29] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov, “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors”, *arXiv:*1207.0580v1 [cs.NE] 3 Jul 2012

[30] K.Hornik, “Approximation capabilities of multilayer feedforward networks”, *Neural Networks*, Volume 4, Issue 2, 1991.

[31] T. Donkers, J. Ziegler, B. Loepp, “Sequential User-based Recurrent Neural Network Recommendations”, *RecSys '17: Proceedings of the 11th ACM Conference on Recommender Systems*, 2017

[32] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, Y. LeCun, “What is the best multi-stage architecture for object recognition?”, *2009 IEEE 12th International Conference on Computer Vision*, 10.1109/ICCV.2009.5459469, 2009.

[33] V. Nair, G. Hinton, “Rectified linear units improve restricted Boltzmann machines*”, Proceedings of the 27 th International Conference on Machine Learning*, Haifa, Israel, 2010.

[34] X. Glorot, A. Bordes and Y. Bengio , “Deep Sparse Rectifier Neural Networks”, *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) 2011, Fort Lauderdale, FL, USA. Volume 15 of JMLR: W&CP 15*.

[35] A. L. Maas, A. Y. Hannun, A. Y. Ng, “Rectifier Nonlinearities Improve Neural Network Acoustic Models”, *Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.*

[36] R. Pascanu, T. Mikolov, Y. Bengio, “On the difficulty of training recurrent neural networks”, *Proceedings of the 30th International Conference on Machine Learning*, Atlanta, Georgia, USA, 2013. JMLR:W&CP volume 28

[37] F. A. Gers, J. Schmidhuber, F. Cummins, "Learning to Forget: Continual Prediction with LSTM", *Proc. ICANN'99 Int. Conf. on Artificial Neural Networks* (Edinburgh, Scotland), vol. 2, p. 850-855, IEE, London, 1999

[38]https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html (01.09.2019)

[39] K. Cho, B. Merrienboer and C. Gulcehre, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, October 25-29, 2014, Doha, Qatar

[40] F.A. Gers and J. Schmidhuber, "Recurrent nets that time and count", Neural Networks, 2000. IJCNN 2000, *Proceedings of the IEEE-INNS-ENNS International Joint Conference*, Volume: 3

[41] https://www.techopedia.com/definition/32512/overfitting (01.09.2019)

[42] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS) 2010, Chia Laguna Resort, Sardinia, Italy. Volume 9 of JMLR: W&CP 9.

[43] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (01.09.2019)

[44] https://r2rt.com/non-zero-initial-states-for-recurrent-neural-networks.html (02.09.2019)

[45]https://www.oreilly.com/library/view/artificial-intelligence-by/9781788990547/97eeab76-9e0e-4f41-87dc-03a65c3efec3.xhtml (03.09.2019)

[46]http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning-part-3/ (03.09.2019)

[47]https://www.slideshare.net/grigorysapunov/deep-learning-cases-text-and-image-processing/ (03.09.2019)

[48] https://ai.googleblog.com/2016/09/a-neural-network-for-machine.html (03.09.2019)

[49]https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6 (03.09.2019)

[50] https://www.datacamp.com/community/tutorials/neural-network-models-r (03.09.2019)

[51]https://stats.stackexchange.com/questions/343078/whats-relationship-between-linear-regression-recurrent-neural-networks (03.09.2019)

[52] http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (03.09.2019)

[53]https://www.analyticsvidhya.com/blog/2017/12/introduction-to-recurrent-neural-networks/ (03.09.2019)

[54] https://skymind.ai/wiki/lstm (03.09.2019)

[55] https://towardsdatascience.com/grus-and-lstm-s-741709a9b9b1 (03.09.2019)

[56] http://arunmallya.github.io/writeups/nn/lstm/index.html#/ (03.09.2019)

[57] https://isaacchanghau.github.io/post/lstm-gru-formula/ (03.09.2019)

[58] https://excelsior-cjh.tistory.com/185 (03.09.2019)

[59] https://elitedatascience.com/overfitting-in-machine-learning (03.09.2019)

[60] https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec (04.09.2019)

[61] http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning-part-4/ (04.09.2019)

[62] S.Ioffe and C.Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*", arXiv:1502.03167v3[cs.LG]2 Mar 2015*

Abbreviations

LSTM	Long short term memory
ECG	Electrocardiogram
ANN	Artificial neural network
SVR	Support vector regression
RNN	Recurrent neural network
CNN	Convolutional neural network
NN	Neural network
SVM	Support vector machine
GPU	Graphics processing unit
CPU	Central processing unit
MLP	Multilayer perceptron
FNN	Feedforward neural network
MSE	Mean squared error
RMSE	Root mean squared error
MAE	Mean absolute error
GRU	Gated recurrent unit
RTRL	Real time recurrent learning
BPTT	Backpropagation through time
TP	True positive
TN	True negative
FP	False positive
FN	False negative
MIT	Massachusetts Institute of Technology
BIH	Boston’s Beth Israel Hospital
MLII	Modified limb lead II