ECONTECHMOD. AN INTERNATIONAL QUARTERLY JOURNAL – 2018, Vol. 07, No. 2, 3-8

Forecasting economic and financial indicators by supply of deep and recovery neural networks

N. Boyko, A. Ivanets, M. Bosik

Lviv Polytechnic National University, Lviv, Uktaine; e-mail: nataliya.i.boyko@lpnu.ua

Received February 18.2018: accepted May 20.2018

Abstract. This paper studies the potential of the application of the Recurrent Neural Networks, as well as the Deep Neural Networks in the field of the finances and trading. In particular, their use in the stock price predicting software. The concepts of the RNNs and DNNs are provided and explained thoroughly. Both techniques RNNs and DNNs are utilized in the implementation of the stock price predicting software. Two separate versions of the software are created in order to demonstrate the main differences between the algorithms, as well as to determine the best of the two. Each version is thoroughly examined. The comparison of each of the algorithms is performed and highlighted. Examples of the implementations of the software, utilizing each of the algorithms on big volumes of stock data, for stock price prediction are provided. The article summarizes the concept of stock price prediction backed by the popular machine learning algorithms and its application in the nowadays world.

Keywords: neural network, deep, recurrent, activation function, feedforward, neuron, hidden layer, stock price prediction.

INTRODUCTION

Over the past two decades Machine Learning has become one of the mainstays of information technology and with that, a rather central, albeit usually hidden, part of our life [6]. With the ever-increasing amounts of data becoming available there is good reason to believe that smart data analysis will become even more pervasive as a necessary ingredient for technological progress [2-5].

The extremely large amounts of digital data collected everyday provide us with the opportunities we could only dream of before. The other contributing factor is the constant increases in the computing power which actually allow us to perform these extremely complex calculations. These events allowed us to rediscover the true power of the machine learning in today’s complex world. Such progress in this field has led to many new discoveries and found useful ways of application of the various machine learning algorithms, such as: image recognition, speech recognition, natural language processing, event prediction, etc [20].

The process of prediction of the stock prices using the ML techniques is the subject of this research.

The purpose of the article is to study the ways of utilization of the RNNs and DNNs in the real world applications, as well as to assess the effectiveness of each of them [8-11].

THE PRELIMINARY SEARCH, ANALYSIS OF THE PROBLEM

Research task is to demonstrate the power of the RNNs and DNNs and their practical application in the field of the finances and trading, as well as to provide a software which will help with the stock price prediction. In order to achieve the research goal, the following problems had to be solved:

1) A thorough review of both techniques RNNs and DNNs;

2) Implementation of the stock price predicting software backed by the RNNs and DNNs;

3) Testing of the implemented pieces of software;

4) Comparison of the algorithms and determination of the best one for the practical use in the real world.

RECURRENT NEURAL NETWORKS

The human brain is a recurrent neural network (RNN): a network of neurons with feedback connections. It can learn many behaviors/sequence processing tasks/algorithms/programs that are not learnable by traditional machine learning methods. This explains the rapidly growing interest in *artificial* RNNs for technical applications: general computers which can learn algorithms to map input sequences to output sequences, with or without a teacher. They are computationally more powerful and biologically more plausible than other adaptive approaches such as Hidden Markov Models (no continuous internal states), feedforward networks and Support Vector Machines (no internal states at all) [5].

Early RNNs of the 1990s could not learn to look far back into the past. Their problems were first rigorously analyzed on Schmidhuber's RNN long time lag project by his former PhD student Hochreiter(1991). A feedback network called "Long Short-Term Memory" (LSTM, Neural Comp., 1997) overcomes the fundamental problems of traditional RNNs, and efficiently learns to solve many previously unusual tasks involving [12-14]:

1. Recognition of temporally extended patterns in noisy input sequences
2. Recognition of the temporal order of widely separated events in noisy input streams
3. Extraction of information conveyed by the temporal distance between events
4. Stable generation of precisely timed rhythms, smooth and non-smooth periodic trajectories

5. Robust storage of high-precision real numbers across extended time intervals.

The fundamental feature of a Recurrent Neural Network (RNN) is that the network contains at least one feed-back connection, so the activations can flow round in a loop. That enables the networks to do temporal processing and learn sequences, e.g., perform sequence recognition / reproduction or temporal association/prediction. Recurrent neural network architectures can have many different forms. One common type consists of a standard Multi-Layer Perceptron (MLP) plus added loops. These can exploit the powerful non-linear mapping capabilities of the MLP, and also have some form of memory. Others have more uniform structures, potentially with every neuron connected to all the others, and may also have stochastic activation functions. For simple architectures and deterministic activation functions, learning can be achieved using similar gradient descent procedures to those leading to the back-propagation algorithm for feed-forward networks. When the activations are stochastic, simulated annealing approaches may be more appropriate. We will look at a few of the most important types and features of recurrent networks [9].

Let's consider the simplest form of fully recurrent neural network which is an MLP with the previous set of hidden unit activations feeding back into the network along with the inputs (fig. 1):

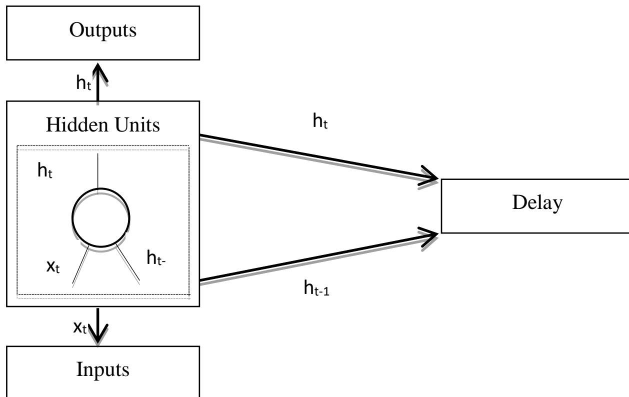

Fig. 1. The simplest form of a fully recurrent neural network

It's also worth noting that the time t has to be discretized, with the activations updated at each time step. The time scale might correspond to the operation of real neurons, or for artificial systems any time step size appropriate for the given problem can be used. A delay unit needs to be introduced to hold activations until they are processed at the next time step [10].

Since one can think about recurrent networks in terms of their properties as dynamical systems, it is natural to ask about their stability, controllability and observability: Stability concerns the boundedness over time of the network outputs, and the response of the network outputs to small changes (e.g., to the network inputs or weights). Controllability is concerned with whether it is possible to control the dynamic behaviour. A recurrent neural network is said to be controllable if an initial state is steerable to any desired state within a finite number of time steps. Observability is concerned with whether it is possible to observe the results of the control applied. A recurrent network is said to be observable if the state of the network can be determined from a finite set of input/output measurements [6-8].

Consider the following visualizations of the recurrent neuron and the actual recurrent network, which give a better idea of what processing takes place behind the scenes (fig. 2):

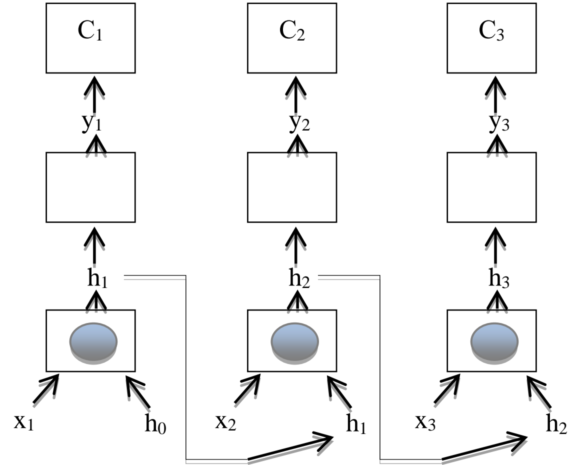

Fig. 2. An unfolded Recurrent Neural Network

A Recurrent Neural Network (RNN) (fig. 2) can be unfolded based on the Recurrent Neurons. Each of the recurrent neurons receives some input x_t (Input at time t) and the previous state h_{t-1}(state at time $t-1$), and then calculates the next state based on the previously mentioned input data according to the following formulas:

$$h_t = \sigma_h(W_h * x_t + U_h * h_{t-1} + b_h) \quad (1)$$
$$y_t = \sigma_y(W_y * h_t + b_y),$$

where x_t is input vector; h_t is hidden layer vector; y_t is output vector; W, U and b are parameter matrices and vector; σ_h and σ_y are activation functions. We can see clearly that the same formula that was used to calculate a new state in a single recurrent neuron remains valid in the unfolded RNN, which makes sense. It is also worth pointing out that the weights in the final RNN are shared over time!

Applications of the RNNs include adaptive robotics and control, handwriting recognition, speech recognition, keyword spotting, music composition, attentive vision, protein analysis, stock market prediction, and many other sequence problems [18-19].

LONG SHORT TERM MEMORY NETWORKS (LSTM)

Long Short Term Memory networks or LSTMs are a special kind of Recurrent Neural Networks (RNNs), capable of learning long-term dependencies, which address the vanishing/exploding gradient problem and allow learning of the long-term dependencies. They were introduced by Hochreiter and Schmidhuber (1997), and were refined and popularized by many people in following work. This type of networks was praised for their pretty simple structure and wide range of applications. They work tremendously well on a large

variety of problems, and are now widely used, and just recently risen to prominence with the state-of-the-art performance in speech recognition, language modelling, translation, image captioning. LSTMs are explicitly designed to avoid the long-term dependency problem, as remembering information for long periods of time is practically their defau[illegible] not something they struggle to learn!

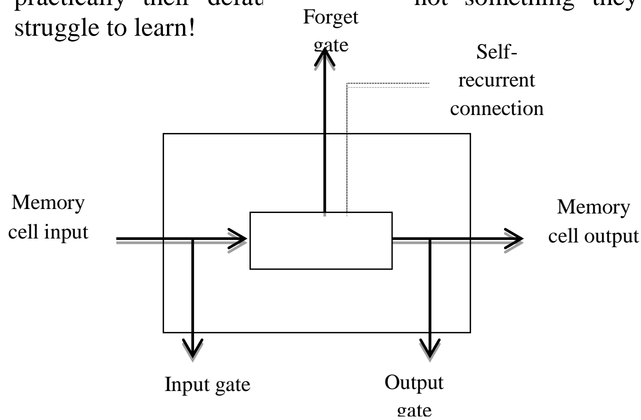

Fig. 3. General Structure of an LSTM

Central idea of the LSTMs lies in their memory cells (blocks) which can maintain its state over time, consisting of an explicit memory (aka the cell state vector) and gating units which regulate the information flow into and out of memory (see fig.5).

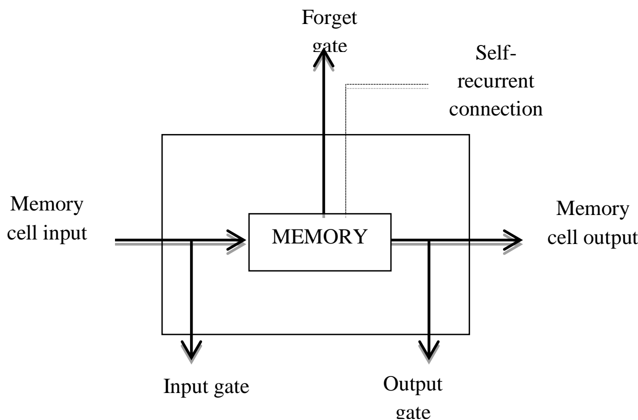

Fig. 4. LSTM Memory cell

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single layer with tanh activation function (hyperbolic tangent). LSTMs also have this chain-like structure, but the repeating module has a different structure, main difference of which is instead of having a single neural network layer, there are four, interacting in a very special way.

Applications are similar to those of typical RNNs, but with more complications.

STOCK PRICE PREDICTION USING RNN

```
# inputs.shape = (number of examples, number of input, dimension of
each input).
    self.learning_rate = tf.placeholder(tf.float32, None,
name="learning_rate")
    # Stock symbols are mapped to integers.
    self.symbols = tf.placeholder(tf.int32, [None, 1],
name='stock_labels')
    self.inputs = tf.placeholder(tf.float32, [None, self.num_steps,
self.input_size], name="inputs")
    self.targets = tf.placeholder(tf.float32, [None, self.input_size],
name="targets")

    def _create_one_cell():
      lstm_cell = tf.contrib.rnn.LSTMCell(self.lstm_size,
state_is_tuple=True)
      if self.keep_prob < 1.0:
        lstm_cell = tf.contrib.rnn.DropoutWrapper(lstm_cell,
output_keep_prob=self.keep_prob)
      return lstm_cell

    cell = tf.contrib.rnn.MultiRNNCell(
      [_create_one_cell() for _ in range(self.num_layers)],
      state_is_tuple=True
    ) if self.num_layers > 1 else _create_one_cell()

    if self.embed_size > 0:
      self.embed_matrix = tf.Variable(
        tf.random_uniform([self.stock_count, self.embed_size], -1.0,
1.0),
        name="embed_matrix"
      )
      sym_embeds = tf.nn.embedding_lookup(self.embed_matrix,
self.symbols)

      # stock_label_embeds.shape = (batch_size, embedding_size)
      stacked_symbols = tf.tile(self.symbols, [1, self.num_steps],
name='stacked_stock_labels')
      stacked_embeds = tf.nn.embedding_lookup(self.embed_matrix,
stacked_symbols)

      # After concat, inputs.shape = (batch_size, num_steps, lstm_size
+ embed_size)
      self.inputs_with_embed = tf.concat([self.inputs,
stacked_embeds], axis=2, name="inputs_with_embed")
    else:
      self.inputs_with_embed = tf.identity(self.inputs)

    # Run dynamic RNN
    val, state_ = tf.nn.dynamic_rnn(cell, self.inputs, dtype=tf.float32,
scope="dynamic_rnn")

    # Before transpose, val.get_shape() = (batch_size, num_steps,
lstm_size)
    # After transpose, val.get_shape() = (num_steps, batch_size,
lstm_size)
    val = tf.transpose(val, [1, 0, 2])

    last = tf.gather(val, int(val.get_shape()[0]) - 1, name="lstm_state")
    ws = tf.Variable(tf.truncated_normal([self.lstm_size,
self.input_size]), name="w")
    bias = tf.Variable(tf.constant(0.1, shape=[self.input_size]),
name="b")
    self.pred = tf.matmul(last, ws) + bias
    self.last_sum = tf.summary.histogram("lstm_state", last)
    self.w_sum = tf.summary.histogram("w", ws)
    self.b_sum = tf.summary.histogram("b", bias)
    self.pred_summ = tf.summary.histogram("pred", self.pred)

    # self.loss = -tf.reduce_sum(targets *
tf.log(tf.clip_by_value(prediction, 1e-10, 1.0)))
    self.loss = tf.reduce_mean(tf.square(self.pred - self.targets),
name="loss_mse")
    self.optim =
tf.train.RMSPropOptimizer(self.learning_rate).minimize(self.loss,
name="rmsprop_optim")
    self.loss_sum = tf.summary.scalar("loss_mse", self.loss)
    self.learning_rate_sum = tf.summary.scalar("learning_rate",
self.learning_rate)
    self.t_vars = tf.trainable_variables()
    self.saver = tf.train.Saver()
```

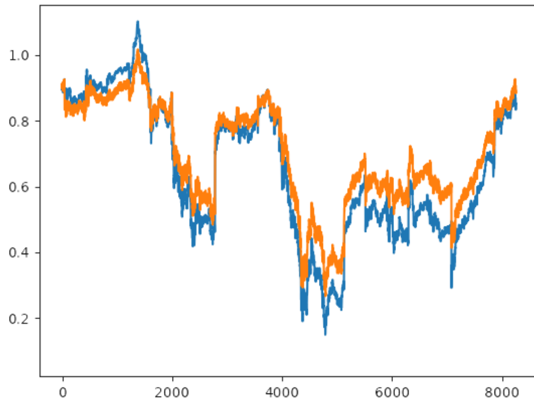

Fig. 5. Prediction results

DEEP NEURAL NETWORKS

A neuron is an information-processing unit that is fundamental to the operation of a neural network.

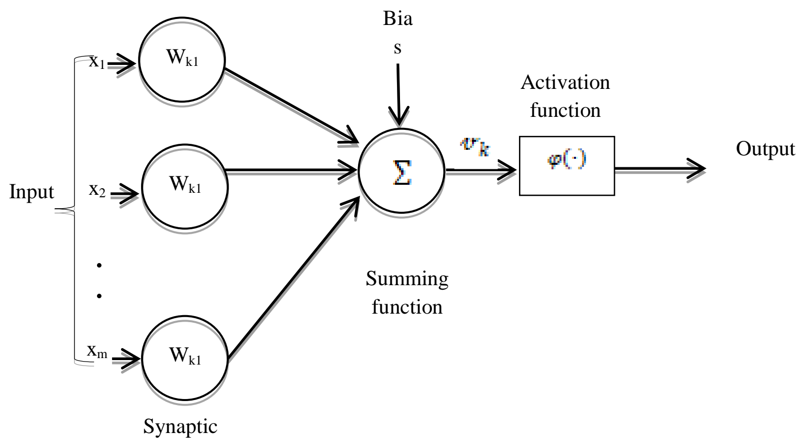

Fig. 6. Model of neuron

In mathematical terms:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$
$$y_k = \varphi(u_k + b_k), \qquad (2)$$

where $x_1, x_2, \ldots, x_m$ are the input signals; $w_{k1}, w_{k2}, \ldots, w_{km}$ are the weights of neuron k; b_k is the bias; φ is the activation function.

Multilayer perceptron is a class of feedforward artificial neural network. The network consists of a set of sensory units that constitute the input layer, one or more hidden layers of computation nodes, and an output layer of computed nodes.

Hidden neurons enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns.

The model of each neuron in the network unites a nonlinear activation function. Activation function must be smooth(differentiable everywhere). The two common activation functions are hyperbolic tangent and the logistic function. Alternative activation functions have been proposed, including the rectifier and softplus functions.

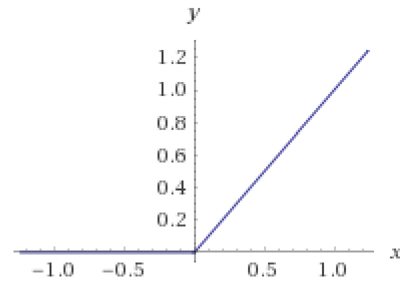

Fig. 7. Plot of the rectifier

The common method for training a neural network is error back-propagation. Learning occurs in the perceptron by changing connection weights after each piece of data is processed, based on the amount of error in the output compared to the expected result. Each layer of nodes trains on a distinct set of features based on the previous layer's output. The further you advance into the neural net, the more complex the features your nodes can recognize, since they aggregate and recombine features from the previous layer.

The network is fully connected if every node in each layer of the network is connected to every other node in the adjacent forward layer. If some of the communication links are missing from the network, the network is partially connected.

MLPs are widely used for pattern classification, recognition, prediction and approximation. Therefore, one of the problems deep learning solves best is in processing and clustering the world's raw, unlabeled media, discerning similarities and anomalies in data that no human has organized in a relational database or ever put a name to.

COMPARING ALGORITHMS

Deep neural network allow signals to transfer one way only: from input to output. There are no feedback, the output of any layer does not affect that same layer. DNN takes a fixed size input and generates fixed-size outputs. Feedforward neural networks are ideally suitable for modeling relationships between a set of predictor or input variables and one or more output variables.

Recurrent neural nets are neural networks that keep state between input samples. They remember previous input samples and use those to help classify the current input sample. They are mostly useful when the order of your data is important. Generally speaking, problems related to time-series data (data with a timestamp on them) are good candidates to be solved well with recurrent neural nets. Also RNN can handle arbitrary input/output lengths.

Unlike a traditional deep neural network, which uses different weights at each layer, a RNN shares the same weights across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn.

DNN and RNN have different models of neuron. RNN use LSTM blocks, neurons that output influences

their input. DNN have an input layer of source nodes that projects onto an output layer of neurons, but not vice versa.

STOCK PRICE PREDICTION USING DNN

Deep learning model for predicting the S&P 500 index based on price 500 units per minute.

We need two cells in order to fit our model: *X* contains the network's inputs (the stock prices of all S&P 500 constituents at time $T = t$) and *Y* the network's outputs (the index value of the S&P 500 at time $T = t + 1$).

The model consists of four hidden layers. The first layer contains 1024 neurons, slightly more than double the size of the inputs. Subsequent hidden layers are half the size of the previous layer, which means 512, 256 and finally 128 neurons.

Cells (data) and variables (weights and biases) need to be combined into a system of sequential matrix multiplications.

The hidden layers of the network are transformed by activation function — ReLU.

The optimizer takes care of the necessary computations that are used to adapt the network's weight and bias variables during training. Here the Gradient descent optimizer is used.

```
# Placeholder
X = tf.placeholder(dtype=tf.float32, shape=[None, n_stocks])
Y = tf.placeholder(dtype=tf.float32, shape=[None])

# Model architecture parameters
n_stocks = 500
n_neurons_1 = 1024
n_neurons_2 = 512
n_neurons_3 = 256
n_neurons_4 = 128
n_target = 1

# Layer 1: Variables for hidden weights and biases
W_hidden_1 = tf.Variable(weight_initializer([n_stocks, n_neurons_1]))
bias_hidden_1 = tf.Variable(bias_initializer([n_neurons_1]))
# Layer 2: Variables for hidden weights and biases
W_hidden_2 = tf.Variable(weight_initializer([n_neurons_1,
n_neurons_2]))
bias_hidden_2 = tf.Variable(bias_initializer([n_neurons_2]))
# Layer 3: Variables for hidden weights and biases
W_hidden_3 = tf.Variable(weight_initializer([n_neurons_2,
n_neurons_3]))
bias_hidden_3 = tf.Variable(bias_initializer([n_neurons_3]))
# Layer 4: Variables for hidden weights and biases
W_hidden_4 = tf.Variable(weight_initializer([n_neurons_3,
n_neurons_4]))
bias_hidden_4 = tf.Variable(bias_initializer([n_neurons_4]))

# Output layer: Variables for output weights and biases
W_out = tf.Variable(weight_initializer([n_neurons_4, n_target]))
bias_out = tf.Variable(bias_initializer([n_target]))
# Hidden layers
hidden_1 = tf.nn.relu(tf.add(tf.matmul(X, W_hidden_1), bias_hidden_1))
hidden_2 = tf.nn.relu(tf.add(tf.matmul(hidden_1, W_hidden_2),
bias_hidden_2))
hidden_3 = tf.nn.relu(tf.add(tf.matmul(hidden_2, W_hidden_3),
bias_hidden_3))
hidden_4 = tf.nn.relu(tf.add(tf.matmul(hidden_3, W_hidden_4),
bias_hidden_4))

# Output layer (must be transposed)
out = tf.transpose(tf.add(tf.matmul(hidden_4, W_out), bias_out))
# Cost function
mse = tf.reduce_mean(tf.squared_difference(out, Y))
# Optimizer
learning_rate = 0.01
opt = tf.train.GradientDescentOptimizer(learning_rate).minimize(mse)
# Initializers
sigma = 1
weight_initializer = tf.variance_scaling_initializer(mode="fan_avg",
distribution="uniform", scale=sigma)
bias_initializer = tf.zeros_initializer()
# Make Session
net = tf.Session()
# Run initializer
net.run(tf.global_variables_initializer())
# Number of epochs and batch size
epochs = 10
batch_size = 256

for e in range(epochs):

  # Shuffle training data
  shuffle_indices = np.random.permutation(np.arange(len(y_train)))
  X_train = X_train[shuffle_indices]
  y_train = y_train[shuffle_indices]
  # Minibatch training
  for i in range(0, len(y_train) // batch_size):
     start = i * batch_size
     batch_x = X_train[start:start + batch_size]
     batch_y = y_train[start:start + batch_size]
     # Run optimizer with batch
     net.run(opt, feed_dict={X: batch_x, Y: batch_y})
# Prediction
pred = net.run(out, feed_dict={X: X_test})
print(pred)
# Print final MSE after Training
mse_final = net.run(mse, feed_dict={X: X_test, Y: y_test})
print(mse_final)
```

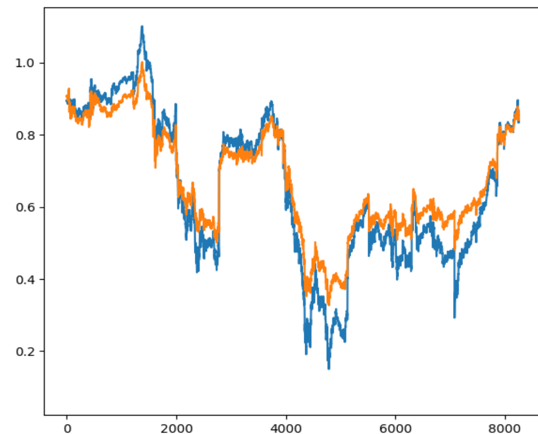

Fig. 8. Predicted results

CONCLUSIONS

Thanks to the constant advances in the field of the Machine Learning and increases in computing power, new horizons are opening for the scientists all over the world. Existing machine learning algorithms already found their place in today's complex world. The most common applications such as image recognition, language modelling, state-of-the-art classification and decision making systems are already making a huge impact on our lives, and the power of their impact will only grow bigger, as the amount of data collected every day increases. Based on the results from the implemented pieces of software we've seen a clear picture of what each kind of the network is capable of, and how effective each of them is. It has also been demonstrated with a good example that the RNN-based or DNN-based built on LSTMs networks work the best for the kind of problems we were solving,

as they provide a "memory" of all of the states and based on those states make a better, more real-world-like, predictions; while the ones that are based on the DNNs with no internal LSTM layers produce more noisy and less consistent results. Despite the good results produced by the implementations in our experiments, one can't be too sure to fully rely on them without any side input from the human; however one can use them as a great tool to boost the efficiency and increase the profits in some sort of the financial institution.

REFERENCES

1. **Boyko N. 2016** Basic concepts of dynamic recurrent neural networks development / N. Boyko, P. Pobereyko // ECONTECHMOD : an international quarterly journal on economics of technology and modelling processes. – Lublin: Polish Academy of Sciences, Vol. 5, № 2. – P. 63-68.
2. **Leskovec J. 2014** Mining of massive datasets / J. Leskovec, A. Rajaraman, J.D. Ullman. – Massachusetts: Cambridge University Press,. – 470 p.
3. **Mayer-Schoenberger V. 2013** A revolution that will transform how we live, work, and think / V. Mayer-Schoenberger, K. Cukier. – Boston New York,. – 230 p.
4. **Boyko N. 2016** A look trough methods of intellectual data analysis and their applying in informational systems / N. Boyko // Computer sciences and informatopn technologies CSIT 2016 : Proceedings of XI International scientific conference CSIT 2016 : proceedings. – Lviv: Publ ofv Lviv Polytechnik. – P. 183-185.
5. **Maass W. 2002** Real-time computing without stable states: a new framework for neural computations based on perturbations / W. Maass, T. Natschger, H. Markram / Neural Computation : proceedings. – Switzerland: Institute for Theoretical Computer Science, Vol. 11. – P. 2531–2560.
6. **Schrauwen B., Verstraeten D., Campenhout J.V. 2007** An overview of reservoir computing theory, applications and implementations / B. Schrauwen, D. Verstraeten, J.V. Campenhout // Proc. of the 15th European Symp. on Artificial Neural Networks : proceedings. – Belgium: Bruges,. P. 471–482.
7. **Coombes S. 2005** Waves, bumps, and patterns in neural field theories / S. Coombes // Biological Cybernetics : proceedings. – Nottingham: University of Nottingham, Vol. 93, № 2. – P. 91–108.
8. **Antonopoulos N. 2010** Cloud Computing: Principles, Systems and Applications / Nick Antonopoulos, Lee Gillam. — L.: Springer. – 379 p.
9. **Shyshkin V.M. 2011** Safety of clod computig – problems and possibilities of risk-analisys[Tekst] / V.M. Shyshkin// International scientific conference "Automated management systems and modern information technologies" – Tbilisi: Publication House "Technical University". – P. 142 (In Russian).
10. **Nandkishor G. 2012** Use of cloud computing in library and information science field / Nandkishor Gosavi, Seetal S. Shinde, Bhagyashree Dhakulkar // International Journal of Digital Library Services. – Vol. 2, iss. 3. – P. 51–60. – Mode of access : http://www.ijodls.in/uploads/3/6/0/3/3603729/vol._2_july_-_sept._2012_part-2.pdf.
11. **Sangeeta N. 2013** Dhamdhere. Cloud Computing and Virtualization / Sangeeta N. Dhamdhere. – 385 p.
12. **Monirul Islam M. 2013** Necessity of cloud computing for digital libraries: Bangladesh perspective / M. Monirul Islam // International Conference on Digital Libraries (ICDL): Vision 2020: Looking Back 10 Years and Forging New Frontiers – P. 513–524.
13. **Avetysov M. A.** Cloud computing for libraries/ M. A. Avetysov. – Access mode: http://www.aselibrary.ru/blogs/archives/997/.
14. **Mell P. 2011** The NIST Definition of Cloud Computing : Recommendations of the National Institute of Standards and Technology / Peter Mell, Timothy Grance. –. Access mode: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.
15. Microsoft support cloud computing». – Access mode: http://www.dw.com/uk/microsoft-531253 (in Ukrainian).
16. **Hewitt C. 2008** ORGs for Scalable, Robust, Privacy-Friendly Client Cloud Computing / Carl Hewitt // IEEE Internet Computing, Vol. 12, N 5, P. 96–99.
17. **Foster I. 2001** The Anatomy of the Grid: Enabling Scalable Virtual Organizations / I. Foster // International Journal of High Performance Computing Applications, Vol. 15, N 3, P. 200–222.
18. **Matov O.Ia. 2004** Information technology and the development of GRID systems in high-performance, globally-distributed computing infrastructures corporate cooperation / O.Ia. Matov // Registration, storage and processing of data,, V. 6, № 1, P. 85–98.
19. **Graham S. 2005** Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI, SAMS / S. Graham. – 816 p.
20. **Foster I. 2008** Cloud computing and grid computing 360-degree compared / I. Foster, Y. Zhao, I. Raicu, S. Lu // Grid Computing Environments Workshop, 2008, GCE'08., P. 1-10.