Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Deep Learning for text data mining: Solving spreadsheet data classification.

Formal Supervisor: Prof. Reggie Davidrajuh
Supervisors at Avito LOOPS: Derek Gobel, CEO

Student: Aleksandr Kimashev

Stavanger 2017

1 Contents

List of frequently used abbreviations, terms:

- **Accuracy -** in this report, the ratio of the number of correct neural network responses to the whole number of samples.
- **Activation function -** in the simple neuron- function that activate it to output one, in a more general neuron function that compute neuron output, described in chapter **Simple neuron and activation function.**
- **Adam, Adagrad, Adamax, Adadelta, Nadam, RMSprop, Nesterov, Rprop** - advanced gradient descent algorithms, described in chapter **Training algorithm.**
- **Batch -** amount of data which are used for one change in the weights of an artificial neural network, described in chapter **Training algorithm.**
- **Convolutional network (CNN) -** family of advanced architecture neural networks, the basic idea is to use a mathematical convolution operation (filter) to sample, described in chapter **Convolutional and pooling.**
- **Error back propagation** - algorithm calculates the network output error and calculates the gradient vector as a function of weights, to chose best direction to change weights during network training, described in chapter **Training algorithm.**
- **False negative -** error for each class, showing how many examples of this class were falsely classified by other classes.
- **False positive -** classification error for each class, showing how many examples were falsely classified by this class.
- **FFNN -** fully connected feed forward artificial neural network, described in theory chapter **Feed forward network**
- **GD,SGD** - gradient descent and stochastic gradient descent algorithms, described in chapter **Training algorithm.**
- **GRU** - the newest type of recurrent network used in this research, described in chapter **Recurrent, LSTMs and GRUs.**
- **Hyper parameters -** not trainable artificial neural network parameters, which must be selected by the developer, as network architecture.
- **Hyperbolic tangent (Tanh) -** bounded activation function, normally used in output layer and in recurrent layers, described in chapter **Simple neuron and activation function.**
- **Layer -** in this work, bunch of neurons/units that have same inputs and output to another layer, with the same activation functions.
- **Loss function, error function -** function that used to estimate difference between neural network output and expected result, described in chapter **Loss functions.**
- **LSTM-** type of recurrent network used in this research, described in chapter **Recurrent, LSTMs and GRUs.**
- **Model -** artificial neural network in Keras.
- **Neuron** - smallest node of neural network, normally with non-linear activation function, described in chapter **Simple neuron and activation function.**
- **N-gram -**is a contiguous sequence of n items from a given sequence of text.
- **NLP -** natural language processing
- **NN -** any artificial neural network
- **One hot vector** - encode samples with a vector same long as dictionary, in which the right symbol corresponds to 1, and all the rest to 0, described in chapter **Vectirization.**
- **Overfitting** - Ability of a network or other classifier to remember training set without generalization, described in chapter **Overfitting.**
- **Pooling -** Sub-sampling is used to reduce the overall sample (usually image) size and increase the degree of invariance applied to it convolutional filters, described in chapter **Convolutional and pooling.**
- **Recurrent network -** family of advanced architecture neural networks, where neurons have back connections, described in chapter **Recurrent, LSTMs and GRUs.**
- **ReLU -** rectified linear unit- the most commonly used activation function in FFNN, described in chapter **Simple neuron and activation function.**
- **Sample -** In this work contents of one cell of spreadsheet.

- **Sigmoid -** sigmoid activation function, bounded activation function, normally used in output layer and in recurrent layers, described in chapter **Simple neuron and activation function.**
- **Synapse -** connection between neurons, with one parameter-weight. Most trainable parameters of neural network are synapses weights.
- **Test set -** set of samples that used only for evaluation neural network accuracy, network was newer trained on them, used in chapter **Overfitting.**
- **Trainable parameters -** artificial neural network parameters, which change in the learning process of the network, as weights oh synapses.
- **Training set -** set of samples that used to train neural network, a large enough network is able to simply remember all training samples, used in chapter **Overfitting.**
- **Unit -** node of advanced recurrent networks like GRU and LSTM, with few activation functions, described in chapter **Recurrent, LSTMs and GRUs.**

1 Introduction

Artificial neural networks are becoming more popular in recent times and are used in various industries - from economic analysis to image recognition. For example, many large online stores use them in order to offer their customers more suitable products for them. Such mathematical models are also widely used for speech recognition and reproduction, as well as for image recognition and processing.

In this work I describe only artificial neural networks and for simplicity I call them neural networks.

The boom for unmanned vehicles also opened a new field of application for neural networks. They are used for routing and navigation systems.

They were also attracted by the producers of anti-virus software. Such developments in the field of artificial intelligence can protect information from cybercriminals and identify illegal content on the Internet.

In this paper I consider the problem of using neural networks to classify and sort text information with given classes. For the classification of textual information, recurrent and convolutional networks are usually used [32][33][34][35][36][37][97][38][41][19][39][40][42][43]. As a minimum unit of information, letters, n-grams, sentence words, phrases can be considered. In my work, I look in detail at recurrent and convolutional networks, at the level of letters.

Working with neural networks is closer to the field of wandering search and description of a randomly found one. Here, most often do not prove, but find and convince in applicability. Many methods used in neural networks do not have a formally proven effectiveness.

The basic requirement for the network the network should generalize and not cram.

Improve the overall result can be due to the application of several different classifiers. From several classifiers it is possible to create a committee, which by voting will make decisions on classifying an object.

Main approach, to construct neural network architecture, that I used in this research was from simple to complex: increase the number of layers and neurons in layers until network became overfitting, and then to deal with overfitting.

2 Formulation of the problem

I developed this project for the Avito LOOPS company. This is a startup company, working on cutting edge Computer Science innovation for application in the Oil & Gas industry. This company is currently conducting a joint R&D project with a large, international oil company. Avito LOOPS develops software (using machine learning, artificial intelligence, etc.) to help teams collaborate smarter and faster.

Avito Loops has significant experience in text data mining and has already developed two text classifiers: one based on entity recognition, pattern matching and voting, the other based on machine learning and decision trees. This project's challenge was to develop a new classifier based on Deep Learning. Specific research goals are to investigate existing algorithms and implementations of Deep Learning, to understand their applicability to text mining, to design a solution that incorporates theoretical and practical aspects, to run classification experiments on different data sets so that the pros and cons of different techniques can be understood. Classification of the text was necessary for the spreadsheet columns classification.

Deep Learning is a type of algorithm suitable for the analysis of data in a broad range of applications including vision, speech and text. One of its important characteristics is the ability to work at progressively higher levels of abstraction: in a text example, this would mean to incrementally create abstractions for letters, short letter sequences, words and finally sentences.

In automatic text data mining,current challenge was to have a rich text type classification. By "rich" classification means to go beyond basic categories like "string", "number", "date", etc that are common in programming languages and databases. A "rich" classification should be able to detect categories of higher abstraction, like "person name", "job title", "project name", "activity description", "address", "equipment code", etc., so categories that humans recognise easily when reading text but that computer programs struggle with.

In this project, I solved the problem of classifying text columns in spread sheets, at the level of each cell, using neural networks. In general, the task can be reduced to the classification of individual cells.

The main problem of neural networks is the availability of data for training the network. For the final classification, five classes were selected, Interesting for Avito LOOPS and available for training data: first names, second names, codes, streets and job position titles. Also, if possible, I wanted to distinguish an unknown class[Figure 2-1]. The datasets used for learning and testing the network are described in detail in the chapter Datasets.

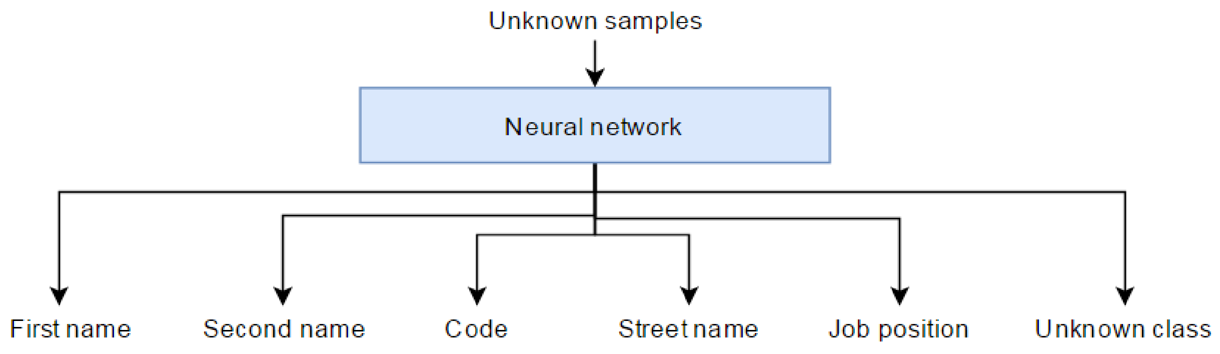


Figure 2-1 Behavior of the system being developed

3 Literature study

3.1 Literature study overview

In the beggining of this work, I now nothing about neural networks and I consider the literature review to be the most important chapter in my research, and although it turned out to be cumbersome, without it further work will not be clear. With this knowledge and some experience with neural networks, further work does not require much effort, only time to train networks.
Main topics described here:

- Description of an artificial neuron -**Simple neuron and activation function**.
- Description of the most simple neural network - **Feed forward network**.
- Network training algorithm and grade classification evaluation **Training algorithm, Loss functions, Additional improvements**.
- Description of the parameters of the neural network, which must be selected at the architecture level, and which do not change during the classical learning of the network - **Hyperparameters**.
- Some limitations of neural networks -**Limitations**.
- The problems of overfitting in neural networks and ways to deal with them -**Overfitting**.
- Advanced architecture of neural networks used in natural language processing - **Convolutional and pooling, Recurrent, LSTMs and GRUs**.
- Use of ensembles, busting and begging, to improve the quality of classification - **Additional NN optimization**
- Preparing data for the neural network -**Vectorization**

In this chapter, I spent some time getting to know the methods not used in this work, such as unsupervised learning, although in work these methods are not used they need to be mentioned in order for the theory to be complete. Also, methods like SGD were almost not used in the work, but they need to be described as an intermediate step to the more advanced algorithms like Adam, Nadam.
Neural networks are a section of artificial intelligence in which signals are processed in a similar way, as in neurons of living beings. The most important feature of the network is the parallel processing of information by all links. With a huge amount of inter-neural connections, this makes it possible to significantly speed up the process of processing information. In addition, with a large number of connections, the network acquires resistance to errors that occur on some lines. The functions of damaged connections are taken up by serviceable lines, as a result of which the network activity does not undergo significant changes.
Neural networks are represented as systems of interconnected neurons. Connections between neurons called synapses. Each synapse has one option - weight. These weights can be adjusted based on inputs and outputs. The collection of neurons organized into layers and divided into three main types: the input, hidden and output. An input layer which receives the information, few hidden layers (usually not more than 3 in feed forward networks) that it is processed and an output layer, which displays the result. The term "deep learning" came from having many hidden layers.
By increasing the number of hidden layers, we move from a shallow neural network, to a deep neural network. Deep neural networks are capable of significantly more complex behavior than their shallow counterparts. Each node, or neuron as it's called, processes input using an activation function.
Each layer can use any function to the previous layer to produce an output signal, typically a linear transformation, followed by non-linearity [21]. Synapses are in communication between neurons, they multiply the number of the input signal by its weights and send result to the neuron. Weight of the synapse characterizes the strength of the link between neurons. Neurons sum the outputs from synapses and apply an activation function (simple neuron returns 1 or 0 - result of activation function is compared to a threshold value, to decide is it enough to "activate" neuron and return 1). During training of a neural network this weights are changed to match the output parameters and the expected correct results (target).
Bias neurons output are always equal to one, and they will never have the input synapses. Weights of synapses from bias neurons are able to shift activation function to the right or left. Weights of synapses that connect ordinary neurons changes slope of the activation function. Also bias neurons helps when all input neurons receive the input of 0 and no matter what their weight, they will transfer to the next layer 0 but not in the case of the presence of a neuron bias.
Good way to think about neural network as about complicated regression function.
Linear regression selects the parameters of the linear function, so that the result best matches the expected result (observations). The coefficients are determined by minimizing some metric error between the

desired results and the obtained. In case of sigmoid function used for non-linearity, the each neuron behaves as logistic regression. Every neuron process is the sum of the weighted data and applied a non-linear function. Additional bias neuron every time output one. Neurons connections-synapses in the neural network have its own weights. Weights are randomly initialized, and then during training algorithm (for example error backpropagation) changed with some factor that determines training speed. Neural network result is more flexible than linear regression and can fit much more complicated data.

3.2 Simple neuron and activation function

Each neuron has three main parameters: the input data, output data and activation function. The input layer neurons take in information, in the form which can be numerically expressed. The following layers receive data from previous layers. The activation function converts the input data to the output. In some literature activation function called non-linearity [71] because normally in networks it should be non-linear.

Network that consists only of one neuron acts like regression.

In the process of learning a neural network, the main variable parameters are the weights of the synapses, but sometimes also some parameters of the activation function. Initially, weights are usually chosen randomly, by normal distribution with mat waiting at zero. Schematically, the work of the neuron is shown in the [Figure 3-1], synapses are represented as weights[57].

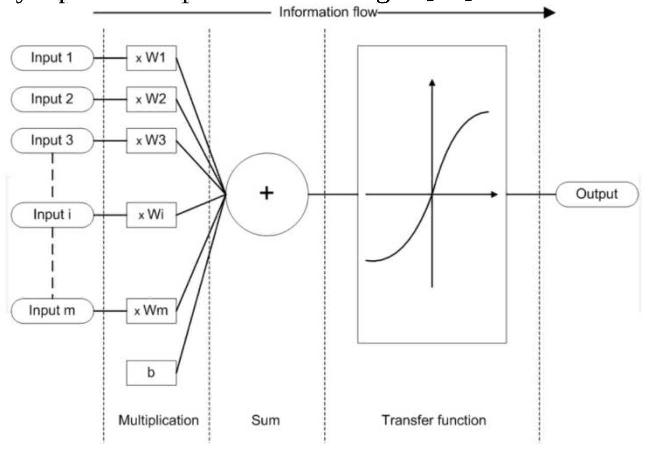


Figure 3-1 The scheme of the neuron's operation, input data (input) is multiplied by weights (W), summed, added bias (b) and the result is sent to the input of some activation function. Bias neuron here represented as b, but it can be represented as input one and weight[11]

Core part of neuron is activation function, that have some important properties[69]:

Nonlinear: When the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator. The linear activation function does not satisfy this property. When multiple layers use the identity activation function, the entire network is equivalent to a single-layer model.

Continuously differentiable: This property is necessary for enabling gradient-based optimization methods. The binary step activation function is not differentiable at 0, and it differentiates to 0 for all other values, so gradient-based methods can make no progress with it.

Range: When the range of the activation function is finite, gradient-based training methods tend to be more stable, because pattern presentations significantly affect only limited weights. When the range is infinite, training is generally more efficient because pattern presentations significantly affect most of the weights. In the latter case, smaller learning rates are typically necessary.

Monotonic: When the activation function is monotonic, the error surface associated with a single-layer model is guaranteed to be convex. Smooth Functions with a Monotonic derivative have been shown to generalize better in some cases.

Approximates identity near the origin: When activation functions have this property, the neural network will learn efficiently when its weights are initialized with small random values. When the activation function does not approximate identity near the origin, special care must be used when initializing the weights.

Linear activation function is possible but few neurons with linear activation function may be reduced to a single linear activation function neuron. The simplest neuron has a binary step activation function and can only output one or zero. But for the work of regular learning algorithms, that used gradients, instead of the binary step activation function, class sigmoidal (S-shaped) functions are usually used. Sigmoidal (S-

shaped) functions - continuous functions that have two horizontal asymptote and one point of inflection. The region of existence of the transfer functions is the entire real axis.
Most famous functions are: sigmoid, hyperbolic tangent and rectified linear unit[71][Figure 3-2].

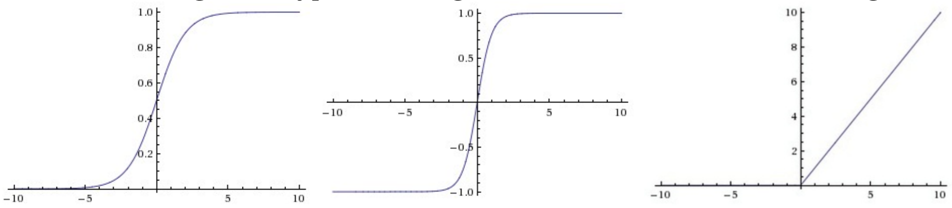


Figure 3-2 Sigmoid activation function (left), hyperbolic tangent activation function (middle) and rectified linear activation functioin (right)

Sigmoid function: $\boldsymbol{\sigma}(\boldsymbol{x}) = \frac{\mathbf{1}}{\mathbf{1}+\boldsymbol{e}^{-\boldsymbol{x}}}$, with deviate: $\boldsymbol{\sigma}'(\boldsymbol{x}) = \boldsymbol{\sigma}(\boldsymbol{x})(\mathbf{1} - \boldsymbol{\sigma}(\boldsymbol{x}))$

Hyperbolic tangent activation function: $\tanh(x) = 2\boldsymbol{\sigma}(\mathbf{2x}) - \mathbf{1} = \frac{\mathbf{2}}{\mathbf{1}+\boldsymbol{e}^{-\mathbf{2}\boldsymbol{x}}} - \mathbf{1} = \frac{\boldsymbol{e}^{\boldsymbol{x}}-\boldsymbol{e}^{-\boldsymbol{x}}}{\boldsymbol{e}^{\boldsymbol{x}}+\boldsymbol{e}^{-\boldsymbol{x}}}$, with deviate: $\tanh'(x) = 1 - \tanh'(x)^2$.

Rectified linear unit: $\boldsymbol{relu}(\boldsymbol{x}) = \mathbf{max}(\mathbf{0}, \boldsymbol{x}) = \begin{cases} \mathbf{0}\ \boldsymbol{for}\ \boldsymbol{x} < 0 \\ \boldsymbol{x}\ \boldsymbol{for}\ \boldsymbol{x} > 0 \end{cases}$, with deviate: $\boldsymbol{relu}'(\boldsymbol{x}) = \begin{cases} \mathbf{0}\ \boldsymbol{for}\ \boldsymbol{x} < 0 \\ \mathbf{1}\ \boldsymbol{for}\ \boldsymbol{x} > 0 \end{cases}$.

Sigmoid. Sigmoid returns real numbers in range [0,1], large negative numbers become 0 and large positive numbers become 1. The smoothness, continuity of function - important positive qualities. The continuity of the first derivative allows to train a network with gradient methods.
Sigmoid function is a nice interpretation of a real neuron behavior, but in practice currently it used only in output layer.
Sigmoid advantages:

- The derivatives are easy to calculate
- Activations won't keep increasing they are bounded, output always between 0 to 1.

Sigmoid disadvantages:

- Sigmoids saturate and kill gradients (vanishing gradient problem[49]). Near 0 or 1 function gradient is near zero. During backpropagation network training gradients show directions to the minimum of error function, and multiplication on this gradient will be almost zero. Therefore on each step weights will be almost not changed, and network training will be very slow. Also it is important that first initialization of weights should be more careful to prevent saturation. Too large initial weights will saturate and the network even before training.
- Function output are not zero-centered. That means that all data to the next layers will be positive, and gradients on the weights during gradient descent backpropagation can be positive or negative.
- Computationally expensive compared to some other activation functions

Improved **softmax** function can be used to solve multiclass problems (if the output of the neural network assumes more than two classes). Softmax ensures that all of the output values p are between 0 and 1, and that their sum is 1. This is a generalization of the Sigmoid to multiple variables. Softmax leads to model the joint distribution over the output variables p(x1, x2, x3, ..., xn) whereas using sigmoid leads to model the marginal distributions p(x1), p(x2), p(x3), ..., p(xn). Sigmoid should be used if every sample can be associated with multiple labels, if each example can only belong to one class, softmax should be used.
Hyperbolic tangent (Tanh).[53] The hyperbolic tangent is zero-centered, scaled sigmoidal function, that returns real numbers in range [-1,1] and can be a good alternative activation function when compared to a sigmoid. A tanh function will cause the derivatives to be much higher because of its range [-1,1] compared to a sigmoid's [0,1]. The derivative is also continuous and expressed in terms of the function itself. Use this function only with positive values is inappropriate since it significantly impairs the results of neural network.
Rectified linear unit (ReLU)[48]. Currently Rectified Linear Unit(ReLU) is the most popular activation function $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{max}(\mathbf{0}, \boldsymbol{x})$, that return input in positive area of the input parameters and zero in negative area of the input parameters.
ReLU advantages:

- Simple to compute (fast).
- Gradient of a ReLU can become a constant, that cause faster learning. It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form. Comparison between Tanh and ReLU convergence speed shown on figure.

- Sparse networks - sparsity arises when Wx + b is less than or equal to 0 and neuron output oi zero (such neurons have no influence on the output). The more nodes in the network that are sparse, the more sparse the overall network is. Sparse representations have been shown to be better for neural networks than dense representations.[59]

ReLU disadvantages:

- Large backpropagation gradient can change weights such that neuron will never be active any more. The boundary on the ReLU is zero to infinity and thus it may cause the activation to increase rapidly. This doesn’t happen as with a sigmoid. [70]

A plot from Krizhevsky et al. [Figure 3-3] (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the Tanh. [68]

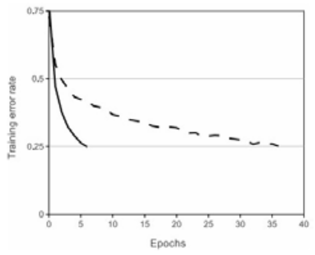


Figure 3-3 Training error rate by epochs for ReLU - solid line and Tanh - dashed line.

One attempt to solve ReLU "dying neurons problem" is leaky ReLU. Leaky ReLU use small negative slope in negative area of the input parameters instead of zero output. Some time slope is one of neuron parameter- Parameteric rectified linear unit (PReLU).[67]

Another ReLU improvements: Randomized leaky rectified linear unit (RReLU), Exponential linear unit (ELU), S-shaped rectified linear activation unit (SReLU)

The choice of activation function is determined by: [66]

1. The specific tasks.
2. Convenience implementation.
3. learning algorithm: some algorithms impose restrictions on the type of activation function. The most common type of non-linearity does not have a fundamental impact on the solution of the problem. However, the best choice may be to reduce the time of training in several times.

Base on different activation function and loss function single neuron can act like Binary Softmax classifier, Binary SVM classifier, Regularization interpretation [71]

One neuron can solve only linearly separable problem and divide space only with line or hyperplane (in the case of many parameters). (Hyperplane separating the different output values, called the decisive surface. [66]). The linearity of the division of space appears from the linear multiplication of input data by weight coefficients. Dependently on activation function this separation can be sharp or smooth [Figure 3-4].

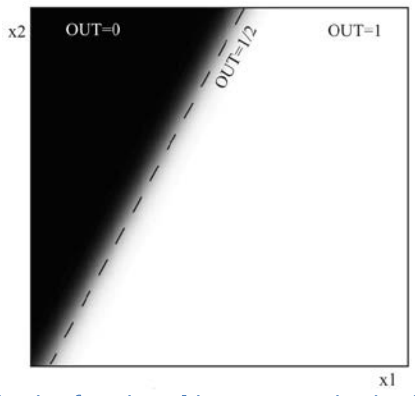


Figure 3-4 Behaviour of different activation functions: binary step activation function - dash line, smooth activation function - color transition.

In this paper, I use three activation functions: sigmoid, Tanh, ReIU, but Wikipedia shows a list of 19 different functions and since there are no specific rules, this list can expand. [69]

3.3 Feed forward network

Combination of neurons become a neural network. The way that neurons are connected to network called topology, architecture or graph of a neuron network. There are a lot of different possible ways to connect neurons in network, but all this ways divided on two main classes. In feed-forward networks (acyclic graph) information flows only in direction form input to output. In recurrent topology (semi-cyclic graph) information can flows not only in one direction from input to output but also in opposite direction.[12] Simplest type of neural network architecture is fully connected feed forward network[53][55][65]. Fully connected means that each neuron connected with all neurons from previous layer. Theoretically there are no limitations on number of layers, type of activation function or number of connections between neurons. The simplest network consist of only one neuron that can solve regression problems.

Formalization of feed-forward neural network (FNN)

Given an input x and setting of the parameters θ that determine weight matrices and the biases $(W_1, \dots, W_l, b_1, \dots, b_l)$, an FNN computes its output $f(x, \theta) = a_l$ by the recurrence[64]

$$s_i = W_i a_{i-1} + b_i$$
$$a_i = \Phi_i(s_i)$$

where $a_0 = x$ and $f(x, \theta) = a_l$. The vectors a_i are the activations of the neural network, and the activation functions $\Phi_i(\cdot)$ are some nonlinear functions, which are typically sigmoid or a tanh functions applied coordinate-wise.

The training objective is obtained by averaging the losses $L(y, z)$ over a set S of input-output pairs (aka training cases), giving the formula

$$h(\theta) = \frac{1}{|S|} \sum_{(x,y)\in S} L(y, f(x, y))$$

The loss function $L(y, z)$ quantifies how bad z is at predicting the target y. Note that L may not compare z directly to y, but instead may use z to parameterize some kind of predictive distribution, or compute some prediction vector $\hat{y}$ from z which is then directly compared to y.

Example of neuron feed-forward fully connected network with one hidden layer are shown on the [Figure 3-5]. [135]

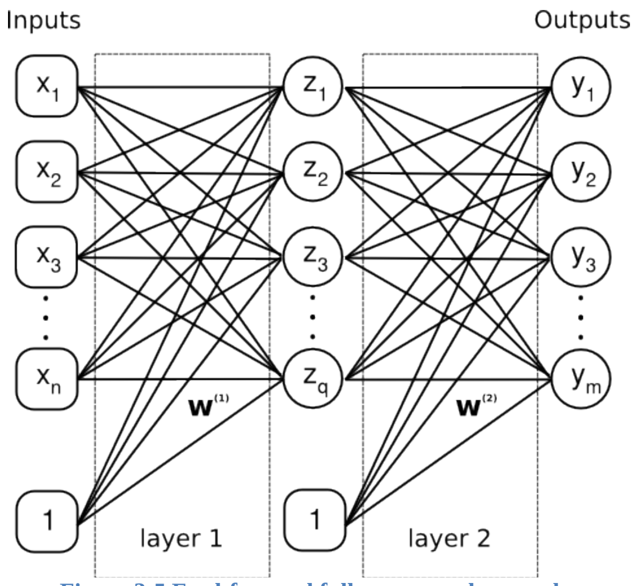


Figure 3-5 Feed-forward fully connected network.

Equation for output from this network:

$$y_m = f^{(2)}\left(\sum_{q=0}^{Q} w_{mq}^{(2)} f^{(1)}\left(\sum_{i=0}^{N} w_{qn}^{(1)} x_n\right)\right)$$

where $f^{(1)}$,$f^{(2)}$ and $w_{qn}^{(1)}$, $w_{mq}^{(2)}$are the activation functions and weights of the first and second layers, subscript mq- describe that synapse connect neuron m from precious layer and neuron q from next layer.

Even such simple network result equation is complicated.

Another way to represent behavior of neural network layer is matrix multiplication [lectures: Tensorflow and deep learning - without a PhD]

Hidden layer from previous example can be represented as [Figure 3-6]:

input neurons

$$\text{samples}\left|\begin{array}{ccccc} \text{bias} & \text{x1} & \text{x2} & \dots & \text{xn} \\ 1 & 0 & 0 & \dots & 1 \\ 1 & 0 & 1 & \dots & 0 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 0 \\ 1 & 1 & 0 & \dots & 1 \end{array}\right| \cdot \text{weights}\left|\begin{array}{cccc} \text{z1} & \text{z2} & \dots & \text{zq} \\ \text{w1-1} & \text{w1-2} & \dots & \text{w1-q} \\ \text{w2-1} & \text{w2-2} & \dots & \text{w2-q} \\ \dots & \dots & \dots & \dots \\ \text{wn-1} & \text{wn-2} & \dots & \text{wn-q} \\ \text{b1} & \text{b2} & \dots & \text{bq} \end{array}\right| \cdot \text{activation function}$$

Figure 3-6 Matrix representation of simple neural layer.

The complexity of the problems to be solved by the network, essentially depends on the number of layers. [66] [72]

Often, in order to demonstrate the limited capacity of single-layer network we should use XOR operation: This function of two arguments, that can be zero or one. It takes a value of 1 when one of the arguments is one, but not both, otherwise 0 [Figure 3-6].

points	x_1	x_2	XOR
A0	0	0	0
B0	1	0	1
B1	0	1	1
A1	1	1	0

Figure 3-7 Xor operation description.

Problem is to divide space in to two zones with output zero and output one. Such problem cannot be solved with one layer network with only one neuron. Actually it can be solved with one layer network with two neurons, but in this case result should be coded in vector with length(1 will be vectors [0,1] and [1,0] and zero vector[0,0]). As I mentioned before, one neuron can solve only linearly separable problem and divide space only with line or hyperplane. But with XOR problem it is impossible to solve it with only one line, like it shown on figure[Figure 3-8].

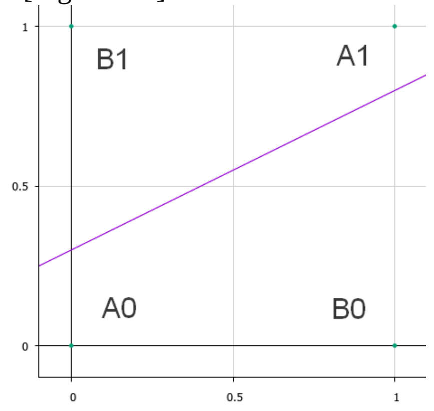
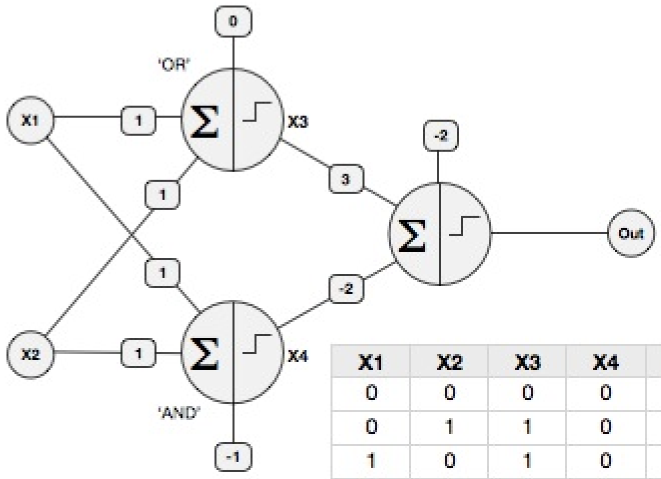
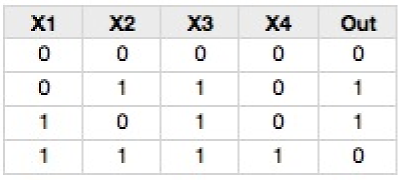


Figure 3-8 Impossibility to divide space by one line according to the XOR rule.

Such problem can be easily solved with two layers network. For simplicity, consider that neurons have binary step activation function, or sigmoid function with threshold value 0.5. Figure shown two layers network, that can solve XOR problem, with outputs of all neurons [Figure 3-9].





X1	X2	X3	X4	Out
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Figure 3-9 Neural network that can solve XOR problem. [73]

Each of the two neuron of the first layer forms a critical surface as an arbitrary line, and an output layer neuron integrates these two solutions to form a critical surface of the strip formed by parallel lines of the first layer neurons. The first layer divides the space into linearly separable. Also any successful synthesis is a non-linear coordinate transformation, after which the problem of classification is more solvable. Result of this division shown on figure[Figure 3-10].

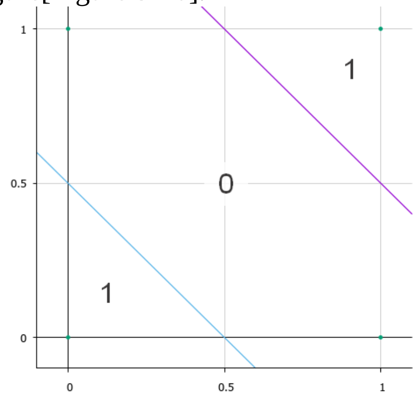


Figure 3-10 Separation of space by a two-layer neural network, with two neurons in first layer.

Binary step was used as activation function in this network. Such a network cannot be trained, by back-propagation algorithm.

Multilayer neural networks has more representing power than single-layer, only in case of the presence of non-linearity. For two layer network with one neuron in output layer: the crucial area is the intersection, union, inversion, or a combination of the fields generated by the neurons in the first layer. View function (intersection, union, inversion, or a combination thereof) is determined by the parameters of the neuron of the second layer (threshold and weights). The number of sides/hyperplanes in the region that divide space coincides with the number of neurons in the first layer [Figure 3-11]. The regions can be open or closed. If the region is closed, it always takes the form of a convex polygon.

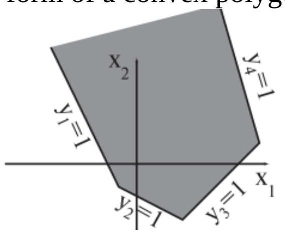


Figure 3-11 Separation of space by a two-layer neural network, with four neurons in first layer.

Three-layer network is the most common in-class networks and is capable of forming an arbitrary non-convex polygonal area multiply. Neurons of the first two layers create an independent arbitrary polygonal crucial area in the right quantity and in the relevant dimensions of the input space X. These areas combine neurons of the third layer in a desired combination. As for the two-layer network, permitted operations of intersection and union [Figure 3-12]. The weights can be negative, and the corresponding area can go with a minus sign, that implements the operation of inversion. The resulting open area may be:

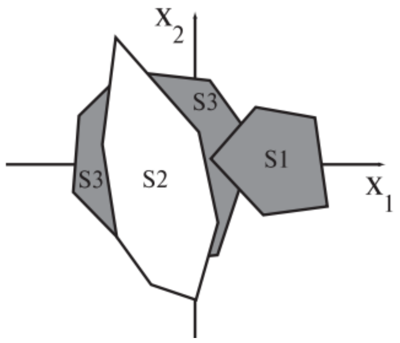


Figure 3-12 Separation of space by a three-layer neural network.

3.4 Training algorithm

Before the introducing training algorithms, I want to mention that even if problem can be solved in principle by using a neural network, it does not mean that this decision can be reached from an initial state of the network. The network topology affect the result much more than learning algorithm. If the network topology makes learning inconvenient, learning rate drops at times, and even dozens of times, and can make it simply impossible; neural network easily develops only those generalizations that are easy and convenient to make on the basis of its topology. All the rest, although possible in principle, but it is very unlikely or, or unreachable from the initial state of the network;

There are two learning paradigms: supervised learning and unsupervised learning. Also learning paradigms can be mixed, in this case supervised learning can use already pretrained network in unsupervised way, that extract features[52].

In supervised, case, the neural network has the correct answers (network output) for each input sample. The weights are adjusted so that the network has produced answers as close as possible to the known correct answers.

Training without a teacher - unsupervised learning, this type of training is not as common as supervised. There are no teachers, so the network does not get the desired result, or their number is very small. Basically, this type of training is inherent in neural networks whose task is to group the data according to certain parameters.

Also, the training can be done in three ways: a stochastic method (stochastic), a batch method (batch) and mini-batch method (mini-batch). There are many articles and studies on the subject matter which method is best and no one can come to a common response. Each method has its pros and cons.

Stochastic (also sometimes referred to online) method works on the following principle -if it have found the increment of the weight immediately update the corresponding weight. While weights change very often, which can contribute to a faster learning network. But training takes place on one example and requires significantly more weight changes, to achieve a certain accuracy.

The batch method summarize the increment of all weights on the current iteration (epoch), and then update all the weight using this amount. At the same time weights change only once after viewing all the examples, but this change is most true.

Mini-batch method is a golden mean and tries to combine the advantages of both methods. Here the principle is this: we are in a free manner distribute weight in groups and change their weight for the sum of all weights in increments of one group or another.

Supervised learning

Supervised learning is the machine learning task of getting function from supervised training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object and a desired output value. The complexity of the sample determines the number of training examples required to achieve the network's ability to generalize. Too few examples may cause a overfitting the network, when it is working well on the examples of learning sample, but it is bad - on test cases.

Supervised learning algorithms perform the following steps[74]:

- Determine the type of training examples.
- Gather a training set. The training set needs to be representative of the real-world use of the function.
- Determine the input feature representation of the learned function.
- Determine the structure of the learned function and corresponding learning algorithm.
- Complete the design. Run the learning algorithm on the gathered training set.

- Evaluate the accuracy of the learned function.

To make a network to give the right answers, it is necessary to train network, by changing weights (and neurons parameters). The optimization problems are not "convex functions" absolutely any algorithm can be mistaken. There are several methods for neuron network training.

Basic learning algorithm of the neural network is a back propagation method, which uses the gradient descent algorithm.

3.4.1 Gradient decent.

First lets discuss gradient decent [16]. Gradient descent is a way of finding a local minimum (or maximum) functions with the help of motion along the gradient. Understanding the gradient descent method is necessary to use the method of backpropagation[75].

A simple gradient descent for two dimensional function [Figure 3-13]:

- calculate slope at the current position x
- reduce x on the slope multiplied on rate (x = x - slope*r)
- repeat until the slope will not equal to 0

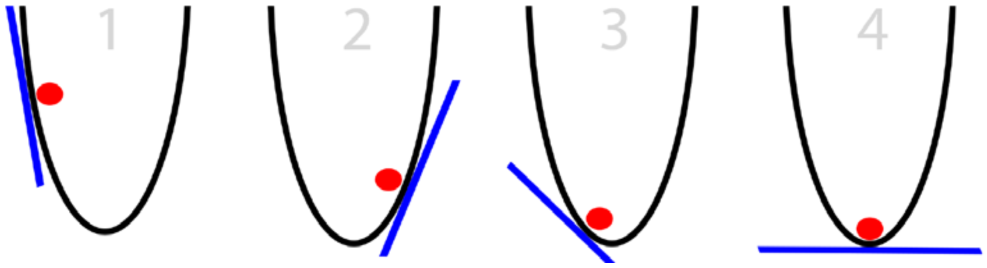


Figure 3-13 The process of finding the minimum with gradient descent. [76]

In the case of a linear model and the error function as a sum of square errors, such function surface will be a paraboloid, which has a unique minimum, and it allows to find a minimum analytically. In the case of non-linear model error surface it has a much more complex structure and may have local minima, flat sections, saddle points and long, narrow ravines. Determine the global minimum of a multidimensional function analytically impossible, and therefore the training of the neural network, in fact, is the procedure of studying the surface of the error function. Starting from a randomly chosen point on the surface of the error function, gradually learning algorithm finds the global minimum. In the end, the algorithm stops at a certain minimum, which can be a local or a global minimum.

Consider the function F (in our example it can be neuron network loss function, that describe dependence of errors on the selected weights and parameters), assuming for definiteness that it is dependent on three variables x, y, z. We compute its partial derivatives $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$ and form with them a vector, which is called the gradient of the function:

$$gradF(x,y,z) = \frac{df(x,y,z)}{dx}i + \frac{df(x,y,z)}{dy}j + \frac{df(x,y,z)}{dz}k$$

Where i, j, k - unit vectors parallel to the coordinate axes. In case of neural network our function will be in multidimensional space, where each weight and parameter will add new dimension. Partial derivatives characterize the variation of the function F for each independent variable separately. Formed through them gradient vector gives an idea of the behavior of the function in the neighborhood of (x, y, z). The direction of this vector is the direction of the most rapid increase of the function at that point. Opposite gradient direction, is the direction of the fastest decrease of the function. Gradient module defines the rate of increase and decrease of the function in the direction of the gradient. For all other areas of the speed of change of the function at the point (x, y, z) is less than the modulus of the gradient. In passing from one point to another as a gradient direction and its magnitude, in general, vary. The concept of gradient is naturally transferred to the function of any number of variables.

The main idea of the method of steepest descent is to move to a minimum in the direction of the fastest decrease of the function, which is opposite gradient direction [Figure 3-14].

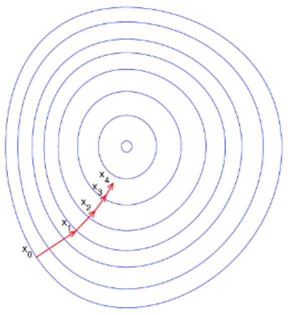


Figure 3-14 Steepest descent to move to a minimum.

This idea is implemented as follows:

- Choose any way the starting point, we calculate it the gradient of the function and make a small step backwards. As a result, we arrive at a point where the function value is smaller than the original.
- The new point will repeat the procedure: again calculate the gradient of the function, and take a step backwards.
- Continuing this process, we will move in the direction of decrease of the function. A special selection of the direction of movement at every step allows us to hope that in this case the approach to the smallest value of the function will be more rapid than in the method of coordinate descent.

Gradient descent method requires the calculation of the gradient of the objective function at each step. If it is given analytically, it is usually not a problem: for partial defining the gradient, you can obtain explicit formulas.

The calculation at each step of the gradient, allowing all the time to move towards more rapid decay of the objective function can at the same time slow down the calculation process. The fact that the gradient calculation - usually much more complex operation than the count of the function itself. So often used a modification of the gradient method, known as the steepest descent method. According to this method, after calculating the starting point of the gradient functions make towards opposite gradient direction not only a small step, but move up as long as the function decreases. After reaching the minimum point on the selected direction, again calculated gradient of the function and repeat the above procedure [Figure 3-15]. This gradient is calculated much less, only by changing the direction of motion.

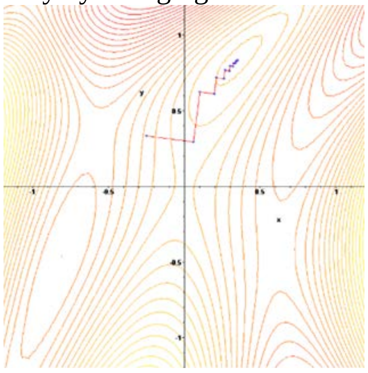


Figure 3-15 Steepest descent method behavior.

Formally to make just one step in the method of gradient descent (to make just one change network settings), it must be calculated for absolutely the entire set of training data. For each object of training data calculate the error and calculate the necessary correction network coefficients (but do not do this correction), and after the submission of all data to calculate the amount of the adjustment factor for each network (the sum of the gradients) and make the correction coefficients "one step". With a large set of training data, the algorithm will work very slowly, so in practice often make adjustments to network coefficients after each element of learning, where the gradient value approximated by the gradient of the cost function, calculated on only one element of training. This method is called stochastic gradient descent or operational gradient descent.

Problems of gradient descent[77]:

- Too large gradient. If it is too big, algorithm can jump point we needed. If it jump not very much, it's not scary. But it can jump even further away from minimum than it was before. To solve it we multiply gradients on learning rate, normally from 0 to 1.

- Too small gradient, so we algorithm change position very slow. Obviously, it is possible to increase the learning rate, and even increase it greater than 1. This is rarely used, but happens.
- Gradient descent method is faced with the problem of local and global minima. Getting into a local minimum algorithm often (depending on speed training or learning rate) can stay in it. Simplest solution is to use a random starting points. More advanced is momentum. If the algorithm uses a moment, then each weight change is added a change in the weights of the last iteration with a certain coefficient. Also momentum can help with finding minimum in ravine-type shape functions, where ordinary gradient descent will zig-zagging [Figure 3-16].

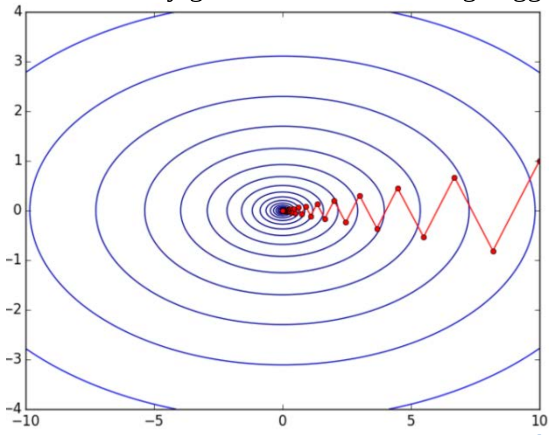


Figure 3-16 Gradient descent zig-zagging behavior in gully function.

Learning rate, is hyperparameter - value which is selected by the developer during trial and error. Too big learning rate can cause can fail to converge [Figure 3-17].

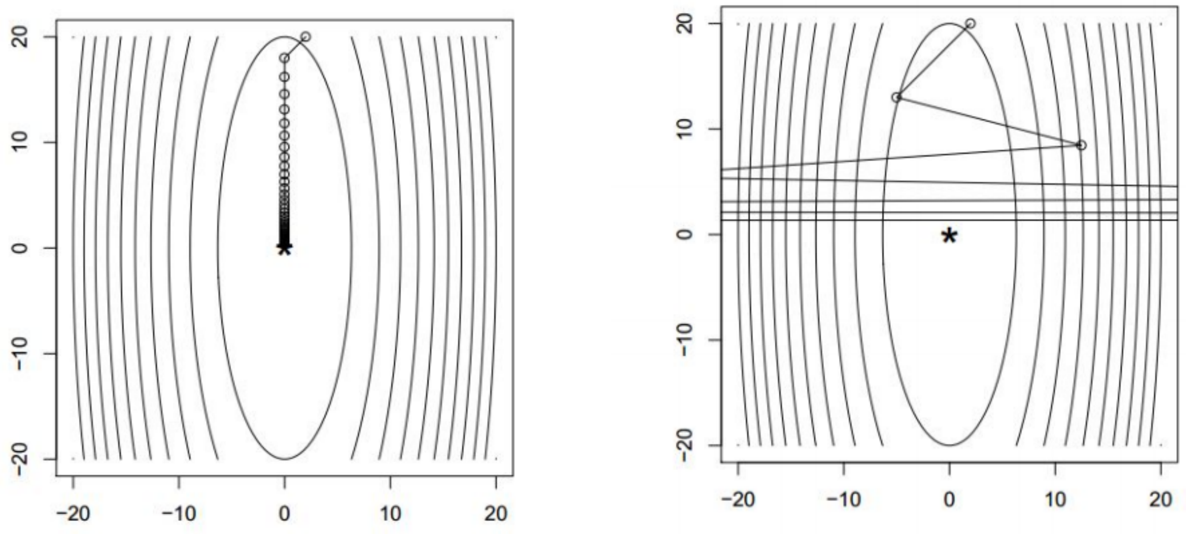


Figure 3-17 Too big learning rate, that failed converge, too small decrease training speed.

In the space of nonlinear functions is the point of zero gradient for all coordinates - that it is problem for the gradient descent. Simple gradient descent in this points will be stuck, but momentum can help in this points[78].

If the point of the gradient in all the coordinates 0, it can be [Figure 3-18]:

- Local minimum, if at all directions of the second derivative is positive.
- Local maximum, if in all directions, the second derivative is negative.
- Saddle point[54], if, for some areas of the second derivative is positive and negative for others.

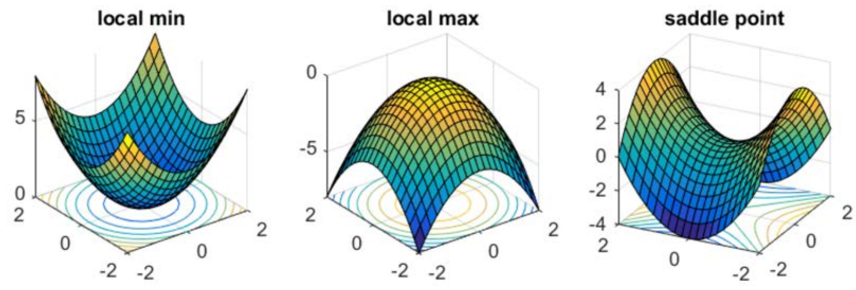


Figure 3-18 Local minimum, local maximum and saddle point, that has zero gradients[79].

The vast majority of points with zero gradient - this saddle point, rather than minima. It is easy to understand intuitively - to the point of zero gradient has a local minimum or maximum, the second derivative should be of the same sign in all directions, but the more measurements, the greater the chance that at least on some directional signs will be different. And so the most difficult points that meet - will saddle[63][62].

3.4.2 Back propagation gradient descent

Using the back-propagation algorithm[75] calculates the network output error and calculates the gradient vector as a function of weights. This vector indicates the direction of the shortest descent on the surface for a given point, so if you move in that direction, then the error is reduced. The sequence of these steps will eventually lead to a minimum of one type or another.

Simplified neuron network training scheme: [80]

1. Initialize weights and parameters of activation functions in a small non-zero values. Weights initialization will be different, for different activation functions.
2. Input iteration training set and calculate the output.
3. Calculate the error between network output and correct results.
4. Change the weight and parameters of activation function, so that the error decreased.
5. Repeat steps 2-4 for as long as the error does not stop or decrease is sufficiently low.

Each time when training algorithm use entire set of training data, called epoch.

Error or loss is the value that reflects the difference between the expected and received responses (distance). The error can be calculated in different ways. There is no any limitation and we are free to choose any method that will bring the best results.

For this example, let's assume that we minimize the Squared error (in my oppinion it is the simplest to understand formula), defined by the formula:

$$\varepsilon(\{w_{ij}\}) = \frac{1}{2}\sum_{k\in Outputs}(t_k - o_k)^2$$

where t_k - target output of the k-th neuron and o_k - output, which is computing network output. The squared error is a function of the weight coefficients. w_{ij} is the weight that connect neuron i from previous layer and neuron j from next layer. There are many methods for solving optimization problems.

The simplest method is to randomly search the weights w_{ij}. Next Idea is a random local search, randomly choose direction and calculate function in this one step, by this direction, if it leads down take a step.

A more effective method of gradient descent, whereby is a correction of each weights w_{ij} is performed in the direction opposite to the error function gradient. Now instead of randomly choose direction we can compute best direction from this point.

Movement in the direction opposite to the gradient will be carried out, if at each iteration to the coordinates of the current point w_{ij} we will add value directly proportional to the derivative of the coordinate w_{ij}, taken with the opposite sign.

$$\Delta w_{ij} = -\eta\frac{\partial\varepsilon}{\partial w_{ij}}$$

where η is a learning rate factor that specifies the speed of "movement". We are moving not only in the direction of decrease of the function, but at a rate directly proportional to the rate of decrease of the function.

For the output layer, squared error ε is a complex function which depends primarily on the output signals of the neuron, activation function o_j. The argument activation functions are sum of inputs S_j, which in turn depend on weights w_{ij}.

$$S_j = \sum_i w_{ij} x_i$$

$$\varepsilon = \varepsilon\left(S_j\left(w_{ij}\right)\right) = \varepsilon(o_j\,(S_j\,(w_{ij}\,)))$$

So deviate of error function with respect to weight w_{ij}, for the output layer are:

$$\frac{\partial\varepsilon}{\partial w_{ij}} = \frac{\partial\varepsilon}{\partial S_j}\frac{\partial S_j}{\partial w_{ij}} = \frac{\partial\varepsilon}{\partial o_j}\frac{\partial o_j}{\partial S_j}\frac{\partial S_j}{\partial w_{ij}}$$

deviate of error function with respect to output of neuron j, for squared error:

$$\frac{\partial\varepsilon}{\partial o_j} = -(t_j - o_j)$$

deviate of activation function with respect to sum neuron inputs, for sigmoid function:

$$\frac{\partial o_j}{\partial S_j} = o_j\,(1 - o_j)$$

and deviate of sum with respect to weight w_{ij}, that is equal x_i.

Denote the

$$\frac{\partial\varepsilon}{\partial o_j}\frac{\partial o_j}{\partial S_j} = \delta_j$$

The general formula of increment weights for sigmoid activation function and squared error, for output layer is:

$$\Delta w_{ij} = \eta o_j\,(1 - o_j)(t_j - o_j)x_i = \eta\delta_j x_i$$

In case neuron j is not on the last layer, and all next layers neurons that connected to neuron j called children(j):

$$\frac{\partial\varepsilon}{\partial S_j} = \sum_{k\in\text{children (j)}} \frac{\partial\varepsilon}{\partial S_k}\frac{\partial S_k}{\partial S_j}$$

Where $\frac{\partial\varepsilon}{\partial S_k}$ already calculated on next layer and is equal δ_k,

$$\frac{\partial S_k}{\partial S_j} = \frac{\partial S_k}{\partial o_j}\frac{\partial o_j}{\partial S_j} = w_{ij}\,\frac{\partial o_j}{\partial S_j}$$

$\frac{\partial o_j}{\partial S_j}$ - deviate of activation function.

The general formula of increment weights for sigmoid activation function and squared error, for not output layer is:

$$\Delta w_{ij} = -\eta\frac{\partial\varepsilon}{\partial w_{ij}} = -\eta\frac{\partial S_j}{\partial w_{ij}}\sum_k \frac{\partial\varepsilon}{\partial S_k}\frac{\partial S_k}{\partial S_j} = -\eta o_i o_j\,(1 - o_j)\sum_k \delta_k w_{ik}$$

Now we get generalized delta rule.

In 1949 Hebb D.O. formulated rule for binary step activation function, the following meanings:

The first rule of Hebb - If the signal is not correct and is zero, it is necessary to increase the weight of those inputs where was one.

The second rule of Hebb - If the signal is not correct and is one, it is necessary to reduce the weight of those inputs where was one.

Delta generally somewhat more general form of the Hebb rules:

$$w_j\,(t+1) = w_j\,(t+1) + \eta\varepsilon_i x_j$$

The components of the error vector defined as the difference between the expected and the actual value of the output neuron. Algorithm conventional delta-rule can also be used for network training with sigmoid activation, but the generalized delta rule effectively. In turn, the generalized delta rule can not be applied to the binary step of activation functions.

The algorithm is not universal and has few disadvantages:

- The training network weight values can result in the correction values become very large. large values of weights can lead to the fact that activation function deviation will be very small and network stop training. Usually to avoid this reduce learning rate η, but this increases the training time.
- Gradient descent method can get stuck in a local minimum and not hit the global minimum. The main difficulty is the right methods out of local minima in the training of neural networks: each

time leaving local minimum is searched again the next local minimum in the same manner backpropagation as long as find a way out will not succeed.

- Evidence of convergence suggests infinitesimal correction weights. In practice, if the step size is fixed and very small, the convergence is too slow, and if it is fixed and is too large, it may cause permanent paralysis or instability. A big learning rate convergence is faster, but there is a danger leap through a decision or go in the wrong direction.
- The complex landscape of the objective function: the plateau alternate with regions of strong nonlinearity. The derivative of the plateau is almost zero, and the sudden interruption, on the contrary, can send us too far.
- Some of the parameters are updated less often others, especially when there are informative, but few signs in the data that has a bad effect on the nuances of generalizing the network rules. On the other hand, giving too much importance to all general rarely seen signs can lead to retraining.

If network consist too many neurons, it lost property to generalize information. Network starts overfitting and remember all inputs, but any other inputs, even very similar, can be classified incorrectly.
Feed forward artificial neural networks with more sigmoidal layers poorly trained conventional methods that work well for networks with one hidden layer, because there is a problem of "vanishing" of the gradient, ie further of the output layer lower changes its weights, depend on activation function. Recurrent networks with pure gradient decent backpropagation training algorithm has bad converges.
There are advanced second-order algorithms (as Levenberg–Marquardt algorithm) [http://mechanoid.kiev.ua/neural-net-backprop3.html] that can find a good low and difficult terrain, but for a small number of weights. To use the method of the second order, it is necessary to count the Hessian - matrix of the derivatives with respect to each pair of pairs of parameters settings - and for Newton's method, and even reverse it. Also it can be used genetic algorithms [1][2] to train neural networks, but in this research I use only gradient based algorithms.

3.4.3 Optimization of gradient decent.

To make this chapter I used some referenses[77][81][82]. Normally gradient descent should input all training data to find best approximation function, that fit this data, and calculate average of gradients. But in practice it is not often possible of training time. So in neural network training algorithms input data often randomly divided on same size iteration sets- batches. In case weights changed after every new example, such gradient descent algorithm called stochastic (or "on-line") gradient descent SGD. SGD - stochastic movement is therefore not in the direction of the gradient of the error function (which includes the entire training set), but the error in the direction of the gradient of the random subsample. Consider that it add to this gradient normally distributed noise. This noise and allows algorithm to get out of local minima.
A compromise between computing the true gradient and the gradient at a single example, is to compute the gradient against more than one training example (called a "mini-batch") at each step. This can perform significantly better than true stochastic gradient descent because the code can make use of vectorization libraries rather than computing each step separately. It may also result in smoother convergence, as the gradient computed at each step uses more training examples[83]

3.4.3.1 Momentum

Method described in [7][8][83]. Stochastic gradient descent with momentum remembers the update weights at each iteration, and determines the next update as a convex combination of the gradient and the previous update. Each step algorithm position will be changed on this value:

$$\theta = \theta - (v_t + \gamma v_{t-1})$$

The name momentum stems from an analogy to momentum in physics: the weight vector, thought of as a particle traveling through parameter space, incurs acceleration from the gradient of the loss ("force"). Unlike in classical stochastic gradient descent, it tends to keep traveling in the same direction, preventing oscillations.

3.4.3.2 Nesterov accelerated gradient (Nesterov Momentum)

Method described in [7][8]. The algorithm with the accumulation of momentum. To not store the last n instances of change scales, the algorithm uses exponential moving average.

$$v_t = \gamma v_{t-1} + (1 - \gamma)x$$

Each step algorithm position will be changed on this value

$$\theta = \theta - v_t$$

To save the history, the algorithm multiplies the already accumulated value v_{t-1} by a factor γ and adds new value multiplied by $(1-\gamma)$. As γ closer to one, the accumulation window is bigger and stronger smoothing - the history affect more. If x become equal to 0, v_t attenuated exponentially for exponentially, hence the name of algorithm. Less γ, the algorithm more behaves like a normal SGD.
If at the time t under the algorithm point was a slope, and then he got to the horizontal or even opposite slope part of function, algorithm still continues to moving. However, every step algorithm loses $(1-\gamma)$ its speed.
Accumulated in the v_t value can greatly exceed the value of each of steps. Pulse accumulation already gives a good result, but algorithm calculates the gradient of the loss function at the point where algorithm should come. In this case function can increase speed if new derivation is bigger and decrease in another case.
Too high γ and learning rate can cause missing areas with opposite gradients. However, sometimes this behavior may be desirable.

3.4.3.3 Adagrad

Method described in [10][61]. Some parameters can be extremely informative, but they is rare changed. Adagrad algorithm family keep information about how often each parameter changed. An example of the algorithm can keep the sum of the squares of updates for each parameter. The magnitude of this value indicates the changing rate.

$$G_t = G_t + g_t^2$$

Each step algorithm position will be changed by this value

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t$$

Where G_t - sum of squares of the changes and ϵ - smoothing parameter is required in order to avoid division by 0. Frequently updated parameter will have bigger G_t, and large denominator.
Adagrad idea is to use something that would reduce the update for the parameters that already often updated. Not necessary to use exactly this formulas and metrics.
Adagrad no need to accurately select the learning rate. It should be big enough, but not too big to leap through a decision or go in the wrong direction.

3.4.3.4 RMSProp and Adadelta

Disadvantage of Adagrad is that G_t can be increased too much, that leads to too small updates and paralysis algorithm. RMSProp and Adadelta designed to correct this disadvantage[51].
Using the approach Adagrad, but instead G_t, averaged gradient of the square gradient. Using an exponential moving average.

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

Where $E[g^2]_t$ moving average at the time t. Then:

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

The denominator is the root of the mean squares of the gradient, hence RMSProp - root mean square propagation.

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

Adadelta differs in that to the numerator added a stabilizing member proportional $RMS[g]_t$.
Update the parameters takes three steps.

$$\Delta\theta = -\frac{RMS[g]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t - \frac{RMS[g]_{t-1}}{RMS[g]_t} g_t$$

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

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

Without explicit large $MS[\Delta\theta]_{-1}$, algorithm behavior is opposite Adagrad and RMSProp: first time we will stronger update the weights that are used frequently.
For RMSProp and Adadelta, as well as for Adagrad not need very accurately pick up the learning rate. Usually start η is from 0.1 - 1, γ is equal 0.9. The closer γ to 1, the longer RMSProp and Adadelta with big $RMS[g]_{-1}$ will strongly update parameters with rarely used weights.

If γ is approximately 1 and $RMS[g]_{-1}$ is equal zero, then Adadelta be long do not increase weights of rarely used parameters. That can lead to paralysis of the algorithm, or algorithm will first updates the neurons that encode the best parameters.

3.4.3.5 Adam

Adam - adaptive moment estimation, combines the idea of accumulation of the motion and the idea of a weaker update weights for typical parameters.

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

Also algorithm estimates the average dispersion in order to know the frequency of the gradient changes:

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

Same as $E[\Delta\theta^2]_t$, so there is no difference from RMSProp.

An important difference is in the initial calibration m_t and v_t, if the initial value is zero, it will spend a long time to accumulate them, especially with a large window $(0 \ll \beta_1 < 1, 0 \ll \beta_2 < 1)$

Algorithm artificially inflates m_t and v_t at the first steps (approximately $0 < t < 10$ for m_t and $0 < t < 1000$ for v_t)

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

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

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

Well-tuned Adam does not need gradient clipping (gradient clipping will clip the gradients between two numbers to prevent them from getting too large).

Authors Adam offered as the default values β_1 = 0.9, β_2= 0.999, $\epsilon = 10^{-8}$.

3.4.3.6 Adamax

Instead dispersion it calculate inertia moment of gradient distribution arbitrary power p. This works well when p, tending to infinity.

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p$$

To use it Adam, it is necessary to extract root: $u_t = v_t^{\frac{1}{p}}$

Result equation:

$$u_t = \lim_{p\to\infty} v_t^{\frac{1}{p}} = \lim_{p\to\infty} \left[\beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p\right]^{\frac{1}{p}} = \lim_{p\to\infty} \left[(1-\beta_2^p)\sum_{i=1}^{t} \beta_2^{p(t-i)} |g_i|^p\right]^{\frac{1}{p}}$$

$$= \lim_{p\to\infty} (1-\beta_2^p) \left(\sum_{i=1}^{t} \beta_2^{p(t-i)} |g_i|^p\right)^{\frac{1}{p}} = \lim_{p\to\infty} \left(\sum_{i=1}^{t} \beta_2^{p(t-i)} |g_i|^p\right)^{\frac{1}{p}}$$

$$= \max(\beta_2^{t-1}|g_1|, \beta_2^{t-2}|g_2|, \dots, \beta_2|g_{t-1}|, |g_t|)$$

It happened because when $p \to \infty$ in the sum will be dominated biggest term.

The remaining steps of the algorithm are the same as in Adam.

Additional motivation to increase the weight in the direction of a small gradient - it is better to get out of the saddle points, which are optimization almost always stuck. Conversely, algorithms such as RMSprop will see very low gradients in the saddle direction. [71]

3.4.3.7 Resilient propagation (Rprop)

Unlike the standard Backprop algorithm, Rprop uses only signs of partial derivatives to adjust the weighting coefficients[14]. The algorithm uses "learning by epoch", when the correction of weights occurs after the presentation of the network of all examples.

For each weights algorithm use its individual update-value $\Delta_{ij}^{(t)}$, which solely determines the size of the weight-update

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^{+} * \Delta_{ij}^{(t-1)}, & if \ \frac{\partial E^{(t-1)}}{\partial w_{ij}} * \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ \eta^{-} * \Delta_{ij}^{(t-1)}, & if \ \frac{\partial E^{(t-1)}}{\partial w_{ij}} * \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ \Delta_{ij}^{(t-1)}, & else \end{cases}$$

where $0 < \eta^{-} < 1 < \eta^{+}$

If at the current step the partial derivative with respect to the corresponding weight w_{ij} change its sign, then this indicates that the last change was large and the algorithm jumped through the local minimum and, therefore, the magnitude of the change is necessary reduce by η^{-} and return the previous value of the weighting factor.

$$\Delta w_{ij}{}^{(t)} = w_{ij}{}^{(t)} - \Delta_{ij}^{(t-1)}$$

If the sign of the partial derivative has not changed, then it is necessary to increase the correction value by η^{+} to achieve faster convergence. Fixing the factors η^{-} and η^{+}, we can drop hyperparameters.

The recommended values η^{-} = 0.5, η^{+}, = 1.2, but there are no restrictions on the use of other values for these parameters.

In order to prevent too large or small values of the balance, the correction value is limited from above by the maximum Δ_{max} and from the bottom by the minimum Δ_{min} values of the correction value, which by default, respectively, are set to 50 and 1.0E-6.

The initial values for all Δ_{ij} are set to 0.1. Again, this should only be considered as a recommendation, and in a practical implementation, you can specify a different value for initialization.

The following rule is used to calculate the correction value for the balance:

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)}, & if \ \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ +\Delta_{ij}^{(t)}, & if \ \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ 0, & else \end{cases}$$

If the derivative is positive, i.e. the error increases, then the weight coefficient decreases by the amount of correction, otherwise - increases. Then the weights are adjusted:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} + \Delta w_{ij}^{(t)}$$

Algorithm:

1 Initialize the correction amount Δ_{ij}
2 Calculate the partial derivatives for all examples.
3 Calculate the new value of Δ_{ij}.
4 Adjust the weights.
5 If the break condition is not satisfied, go to 2.
6 This algorithm converges 4-5 times faster than the standard Backprop algorithm.

3.5 Loss functions

Few loss function described [71] [84]. In supervised learning we should estimate difference between network results and correct answers. To estimate this error used loss function. The data loss takes the form of an average over the data losses for every individual example. $L = \frac{1}{N}\sum_i L_i$ where N is the number of training data and $f = f(x_i; W)$ is an activation function of output layer.

For different neural network problems, it can be used different functions:

Classification: For classification(assume that there is a single correct answer) problems used the SVM (e.g. the Weston Watkins formulation), sometimes squared and is the Softmax classifier that uses the cross-entropy loss:

$$L_i = \sum_{j \neq y_i} \max(0, f_i - f_{y_i} + 1)$$

$$L_i = \sum_{j \neq y_i} \max(0, f_i - f_{y_i} + 1)^2$$

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_i}}}\right)$$

The basic idea behind the structured SVM loss is to demand a margin between the correct structure y_i and the highest-scoring incorrect structure.
When the set of labels is very large (e.g. words in English dictionary, or ImageNet which contains 22,000 categories), it may be helpful to use Hierarchical Softmax [5]. That decomposes labels into a tree. The structure of the tree strongly impacts the performance and is generally problem-dependent.
Attribute classification: Binary classifier for every single attribute independently. The loss function then maximizes the log likelihood of probability.

$$L_i = \sum_j y_{ij} \log(\sigma(f_i)) + (1 - y_{ij})\log(1 - \sigma(f_i))$$

where the labels y_{ij} are assumed to be either 1 (positive) or 0 (negative), and $\sigma(\cdot)$ is the sigmoid function. Gradient on f:

$$\frac{\partial L_i}{\partial f_i} = y_{ij} - \sigma(f_i)$$

To classify among several classes (more than two) and provided that it can be selected only one class, the log likelihood of probability formula takes the form also known as the multiclass cross-entropy:

$$L_i = \sum_x p_i(x) \log(q_i(x))$$

Where $p_i(x)$ is probability of correct classification for the example x, that is one for correct class i and zero for all another, $q_i(x)$ is probability for class i according network output. In implementation part I used Keras/Tensorflow framework, and in Tensorflow this formula called categorical cross entropy.
Regression is the task of predicting real-valued quantities. For this task, it is common to compute the loss between the predicted quantity and the true answer and then measure the L2 squared norm, or L1 norm of the difference. The L2 norm squared would compute the loss for a single example of the form:

$$L_i = \|f - y_i\|_2^2$$

The reason the L2 norm is squared in the objective is that the gradient becomes much simpler, without changing the optimal parameters since squaring is a monotonic operation. The L1 norm would be formulated by summing the absolute value along each dimension:

$$L_i = \|f - y\_i\|_1 = \sum_j |f_j - (y_i)_j|$$

sum over all dimensions of the desired prediction, if there is more than one quantity being predicted.
Looking at only the j-th dimension of the i-th example and denoting the difference between the true and the predicted value by δ_{ij}, the gradient for this dimension is easily derived to be either δ_{ij}, with the L2 norm, or sign(δ_{ij},).
The L2 loss is much harder to optimize than a more stable loss such as Softmax. When faced with a regression task, first consider if it is absolutely necessary. Instead, have a strong preference to discretizing your outputs to bins and perform classification over them whenever possible.

3.6 Additional improvements of training algorithm

Certain optimizers can jump out of the global minimum, in order to solve this problem training with returns can be used. This slows down learning.
In training of deep networks, it is usually helpful to anneal the learning rate over time. With a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function. Slow decay wasting computation bouncing around chaotically with little improvement for a long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. There are three common types of implementing the learning rate decay: [71]

- Step decay: Reduce the learning rate by some factor every few epochs. Typical values might be reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs. The hyperparameters of step decay, involves (the fraction of decay and the step timings in units of epochs) are more interpretable.

- Exponential decay has the mathematical form $\alpha = \alpha_0 e^{-kt}$, where α_0 and k are hyperparameters and t is the iteration number or epoch number.
- 1/t decay has the mathematical form $\alpha = \alpha_0/(1 + kt)$ where α_0 and k are hyperparameters and t is the iteration number.

But changing parameters of algorithms with momentum can cause paralyze of such algorithms because it can be nowhere to increase speed.
Increasing the number of levels in any convolutional network with a certain number of layers, will lead to worse accuracy, it will train worse and accuracy decreases. Although deeper network has strictly greater representational power. And, generally speaking, it can be trivial to get a deeper model which is better than less deep, by adding a few identity layers, which simply passes the signal on without change. [9]
BatchNorm and ReLU help with vanishing gradient problem. There are several solutions for such deep networks training.
The network is gradually built up repeating blocks, after adding a new block to it temporarily connected fully connected feed-forward layers and trained, then fully connected feed-forward layers removed and new block connected.
Deep Residual Learning network able to train hundreds of layers (layer 152 [8]) For this, the network adds extra communication, bypassing several layers, gradients to better spread.
Also in practice used transfer learning [71] to train convolutional networks. It is common to train a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

3.7 Hyperparameters

Hyperparameters - are values that must be chosen manually, often by trial and error. Among these values are:

- Minibatch size
- Learning algorithm
- Parameters that are part of the learning algorithm like moment and the learning rate.
- Regularization, L1, L2, noise, ...
- The number of layers and neurons therein.
- Activation functions
- If a train CNN, the size of windows and fold pooling
- If there are ensemble of networks, the ensemble size and ways of combining

Selection of true hyperparameters is very important and will directly affect the convergence of neuron network.
The number of hidden layers and neurons in which we can calculate the brute force based on one simple rule - the more neurons, the more accurate the result and the exponentially more than the time you spend on its training.
Currently, there are no hard and fast rules for selecting any number of hidden layers in feed-forward fully connected networks, or to select the number of neurons in them. Although there are limitations to help decide.

1) If the function is defined on a finite set of points, then 3-layers network able to approximate it.
2) If the function is defined and continuous on a compact area, the 3- layers network able to approximate it.
3) The other functions that the neural network can be trained, can be approximated 4- layers network.

Thus, theoretically, the maximum number of layers that must be - four or two hidden layers. Although increasing the number of layers can help in solving real-world problems.
The number of neurons in the hidden layers.
Choosing the right number of neurons in hidden layers is very important. Too little - and the network will not be able to learn. Too much will cause an increase in network training time to actually unreal values. It may also lead to overfitting, the network will work fine on the training set, but very bad on the input examples are not included in it.
However, there are heuristics rules to choose number of neurons in hidden layers. By this rule, the number of neurons in the hidden layer of three layers network is:

$$k = \sqrt{mn}$$

where k - the number of neurons in the hidden layer, n - the number of neurons in the input layer, m - number of neurons in the output layer.
For 4- layers network:

$$k_1 = m\left(\frac{n}{m}\right)^{\frac{2}{3}}$$

$$k_2 = m\left(\frac{n}{m}\right)^{\frac{1}{3}}$$

where k_1- the number of neurons in the first hidden layer, k_2- the number of neurons in the second hidden layer.
The usual approach: increasing the number of layers and neurons in layers until network become overfitting, and then to deal with overfitting.

3.8 Limitations

Neural network computes linear functions, nonlinear functions of one variable, as well as all kinds of superposition - a function of the functions resulting from the cascade network connection[60].
The theorem of Kolmogorov -any multivariate continuous function can be represented as a superposition of one–dimensional functions:

$$f(x_1, \dots, x_n) = \sum_{q=0}^{2n+1} \Phi_q\left(\sum_{p=1}^{n} \Psi_{q,p}(x_p)\right)$$

where the functions Φ_q are continuous, and the function $\Psi_{q,p}$, in addition, also standard, ie, do not depend on the function f
[A. N. Kolmogorov. On the representation of continuous functions of many variables by superpositions of continuous functions of one variable and addition. Doklay Akademii Nauk USSR, 14(5):953 – 956, 1957]
Weierstrass Approximation Theorem: Any continuous function f can be approximated on a closed bounded interval [a, b] by polynomials with arbitrary accuracy. More precisely, for any ε > 0, there exists a polynomial p with

$$\max_{a \le x \le b} |f(x) - p(x)| < \varepsilon$$

Gorban's Theorem - Any polynomial in several variables can be obtained from one arbitrary nonlinear polynomial of one variable, using linear operations and superposition[60].
Neural networks allow with any accuracy calculate an arbitrary continuous function $f(x_1, \dots, x_n)$. Consequently, they can be arbitrarily closely approximate any continuous operation of the automaton[60].
Each layer linearly separates feature space derived from the previous layer, thus to obtain at the output plane of any other order is not possible.
No neural network can ever learn the function f(x) = x*x with 100% accuracy or on the x values from minus infinity to plus infinity , unless:

- an infinite number of training examples
- an infinite number of units
- an infinite amount of time to converge

3.9 Unsupervised learning

Chapter is based on[85][25]. During unsupervised learning a neural network watching input data, without advance information about what the output should correspond to those or other events. But since the data may contain certain patterns. It is possible to take a single-layer, initiate a network of random weights, when input data, determine the winner neuron and improve its weights. As a result, the neurons themselves get inputs between a main parameters contained in the input information. That can be used for clustering data.
For single-layer network can use deltarule. Deltarule very similar to the Hebb's rule, which has a very simple meaning: the connection of neurons are activated together, should be strengthened, and communication of neurons, triggered independently, should subside. But Hebb's rule originally formulated for unsupervised learning and allows neurons to be adjusted by the allocation factors.
For unsupervised learning used Kohonen neural network architectures. Furthermore, Kohonen networks can be used to reduce the dimensionality of data with minimum data loss.
The structure of the neural network comprises a single layer of neurons (Kohonen layer) without biases. Input variables are normalized. The number of neurons is equal to the number of clusters. The number of

input variables of the neural network is equal to the number of features that characterize the object, and studies based on its classification occurs to one of the clusters.
It is necessary to distinguish between the actual self-study and self-organizing neural network Kohonen. In a typical unsupervised network it has a strictly fixed structure. Self-organization has no permanent structure.
For self-study learning:
1. Choose amount of neurons in the Kohonen layer (K).
2. Random initialization of weighting coefficient values.
In case input data in [-1;1]: $|w_{ij}| \leq \frac{1}{\sqrt{M}}$, in case input data in [0;1]: $0.5 - \frac{1}{\sqrt{M}} \leq w_{ij} \leq 0.5 + \frac{1}{\sqrt{M}}$
where M - the number of features.
3. Calculation of Euclidean distances from the input vector to the center of the cluster:

$$R_j = \sqrt{\sum_{i=1}^{M} (x_i - w_{ij})^2}$$

4. Selected winner neuron j, the closest to the input vector. For the selected neuron are corrected weighting coefficients:

$$w_{ij}^{(q+1)} = w_{ij}^{(q)} + \nu \left(x_i - w_{ij}^{(q)} \right)$$

where ν - learning rate factor.
It repeated from step 3 till: exhausted predetermined limit number of periods of study, no significant change occurred in the weighting factors within the specified accuracy over the last epoch of training, exhausted predetermined limit physical training time.
In the case of a network of self-organizing Kohonen algorithm:
1. Set the critical distance corresponding to the maximum allowable Euclidean distance between the example inputs and weights of the neuron-winner. The initial structure does not contain neurons. When applied to the inputs of the network of the first example of a training sample created the first neuron with weights equal to the given input values.
2. On the new sample input, calculated Euclidean distances from the example to the center of each cluster and is determined by the neuron-winner with the lowest distance Rmin.
3. If Rmin is smaller than critical distance, made a correction weighting coefficients corresponding to the neuron-winner, otherwise the structure of the network adds a new neuron weights which are made numerically equal to the input values set an example.
4. The procedure is repeated with form 2. If during the last epoch of learning any clusters were not involved, the respective neurons are excluded from the Kohonen network structure.
Another modification of the self-study and self-correction algorithms provide the weighting factors, not only the neuron-winner, but all the other neurons.

3.10 Overfitting

Training of neural networks is often a serious problem, called overfitting - too big match of the neural network to a particular set of training examples, and the network loses the ability to generalize. Overfitting occurs when too long training, the number of training examples insufficient or too complex neural network structures.
Overfitting due to the fact that the choice of training set is random. On the training set of the neural network learning takes place. In the test set is related to tested model. These sets should not be crossed.
The difficulty of the algorithm is that we minimize the error is not that actually need to be minimized, we need to minimize network error on new observations, not on training set.
In other words, we would like to see the neural network has the ability to generalize the result to new observations.
For a number of steps, the prediction error is reduced on both test and train sets. Further parameters adjusted to the training set. However, at a certain stage the error on the test set begins to increase, and the error on the training set continues to decrease. But learning does not take place under the common data patterns, and only under a particular subset of the training. The accuracy in the test sample falls. This moment is the end of the real or learning from it and starts retraining. Test and teaching the set should not overlap.
Overfitting occurs when the neural network has too many parameters to be derived from the available options, as in the case of high-order polynomial [Figure 3-19]. Graphs polynomials can have different

shapes, and the higher the degree of (and thereby the more members included in it), so can be more complex, this form. If we have some data, we can set a goal to fit them polynomial curve (model) and thus obtain an explanation for the existing relationship. Data can be noisy, so you can not assume that the best model is given by a curve that passes exactly through all the points. low-order polynomial may be insufficiently flexible means

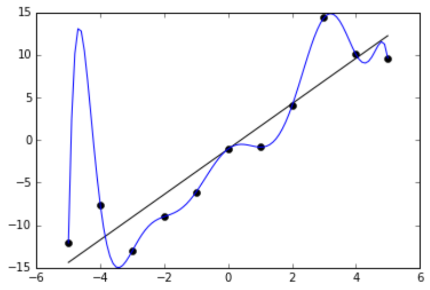


Figure 3-19 11 samples points, blue overfitted model (polynomial degree 10) and black generalized model (polynomial degree 1).

Noisy (roughly linear) data is fitted to both linear and polynomial functions. Although the polynomial function is a perfect fit, the linear version can be expected to generalize better. [86]
If the validation error increases(positive slope) while the training error steadily decreases(negative slope) then a situation of overfitting may have occurred [Figure 3-20].[86]

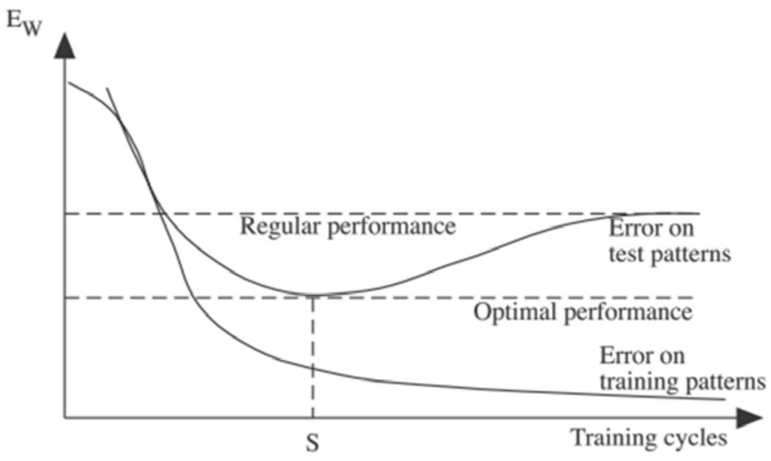


Figure 3-20 Overfitting in supervised learning (e.g., neural network). Point on the chart, after which the error of the test set begins to grow, with a decrease in the error in the training data, indicates the beginning of overfitting. [3].

Neural networks generally tend to reconfiguring. The network, which is very suitable for the training data, are unlikely to summarize the output of non-teaching. There are many ways to retrain the network restrictions (except for reducing the network), but the most common include averaging over multiple networks, regularization and use of the method of Bayesian statistics.
There are several ways of controlling the capacity of Neural Networks to prevent overfitting:[71] [86] [87][88]
In the process of network design, the ability of the network to be overfitted is an important point, after which the methods of against overfitting are usually used.
To solve the overfitting used methods of regularization, dropout, batch normalization, adding noise[46] to data and thinning of the neural network.

L1 and L2 regularization

Described in many sources[88][13][24][89]. Regularization of a model, it is a way to impose a fine to the objective function of the complexity of the model. From a Bayesian point of view - is a way to take into account some a priori information about the distribution of the model parameters.
Thus, to calculate the total gradient of the objective function is necessary to calculate the gradient regularization functions:

$$\frac{\partial C}{\partial \theta_i} = \eta \left(\frac{\partial E}{\partial \theta_i} + \lambda \frac{\partial R}{\partial \theta_i} \right)$$

where E - is the main objective function model R regularization function, λ - is the speed of learning and the regularization parameter, respectively.
regularization function L1 and its derivatives are as follows:

$$R_{L_1}(\theta) = \sum_{i=1}^{n} |\theta_i|$$

$$\frac{\partial R_{L_1}}{\partial \theta_i} = \frac{\theta_i}{\sqrt{\theta_i^2}} = \mathrm{sign}(\theta_i)$$

L2 regularization as follows:

$$R_{L_2}(\theta) = \frac{1}{2}\sum_{i=1}^{n} \theta_i^2$$

$$\frac{\partial R_{L_2}}{\partial \theta_i} = \theta_i$$

Both regularization method fine model of great importance to the balance in the first case, the absolute values of the weights in the second squares weights, so the distribution of the balance will be closer to normal with center at zero.

In comparison, final weight vectors from L2 regularization are usually diffuse, small numbers. In practice, if you are not concerned with explicit feature selection, L2 regularization can be expected to give superior performance over L1. :[71]

Max norm constraints. Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. In practice, this corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector $\vec{w}$ of every neuron to satisfy $\|\vec{w}\|_2 < c$. Typical values of cc are on orders of 3 or 4. [71]

Dropout

Described in many sources[16] [90]. Training a neural network usually produced by a stochastic gradient descent, randomly selecting one object from the selection. Dropouts regularization is that when you select another object changed the structure of the network: each neuron ejected from training with a certain probability. On each step we get a "new" network architecture [Figure 3-21].

On dropouts as if we average the huge mix of different architectures: it turns out that we each test case building a new model at each test case we take one model of a giant ensemble and teach one step, then the next example, we take a different model and teach it to one step, and then to average the output end, all these models.

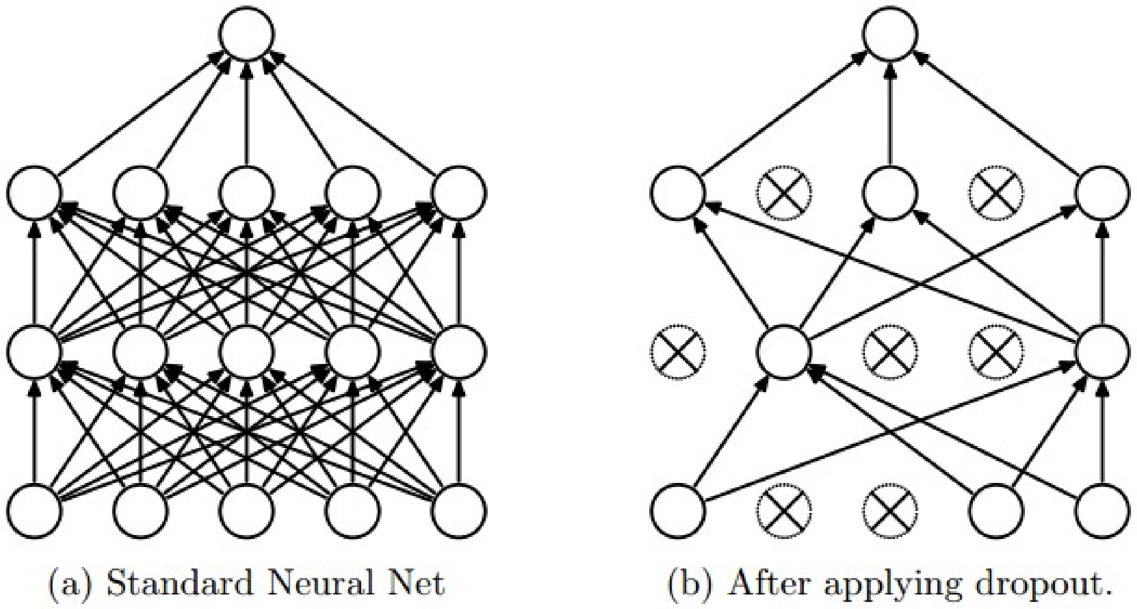


Figure 3-21 Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped. [16]

(However, the exponential number of possible sampled networks are not independent because they share the parameters.) During testing there is no dropout applied, with the interpretation of evaluating an averaged prediction across the exponentially-sized ensemble of all sub-networks. [71]

In practice: It is not very common to regularize different layers to different amounts (except perhaps the output layer) It is most common to use a single, global L2 regularization strength that is cross-validated. It is also common to combine this with dropout applied after all layers. The value of p=0.5p=0.5 is a reasonable default, but this can be tuned on validation data.[71]

Batch normalization

For this chapter I used [59][lectures: Tensorflow and deep learning - without a PhD] [91]. Batch-normalization is a method of accelerating deep learning, proposed by Ioffe and Szegedy. As the signal propagates through the network, even normalized at the input, passing through the inner layers, it can be highly distorted by both expectation and dispersion, which causes discrepancies between the gradients at different levels. Therefore, it is necessary to use stronger regularizers, thereby slowing down the pace of training.

Batch-normalization changes the input data in such a way as to obtain a zero expectation and a unit variance. Normalization is performed before entering each layer. This means that during the training we normalize batch_size examples, and during testing we normalize the statistics obtained on the basis of the entire training set, since we can not see the test data in advance.

Calculation of expectancy and variance for a specific batch $b = x_1, \dots, x_m$:

$$\mu_b = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_b^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_b)^2$$

The activation function is converted so that it has zero expectancy and a unit variance on the whole batch:

$$\widehat{x_i} = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}$$

Where $\epsilon > 0$ is a parameter that protects from division by 0. Finally, in order to get activation function y, we need to make sure that during normalization we did not lose the ability to generalize, and since we applied scaling and shift operations to the original data, we can allow arbitrary scaling and shifting of normalized values by obtaining the final activation function :

$$y_i = \gamma\widehat{x_i} + \beta$$

Where γ And β - trainable parameters of batch-normalization. This generalization also means that batch-normalization can be useful to apply directly to the input data of a neural network.

Batch-normalization, to deep convolutional networks, almost always successfully achieves its goal - to accelerate learning and prevent overfitting. Moreover, it can be an excellent regularizer, allowing to not so carefully choose the pace of training, the power of the L_2 -generalizer and dropout (sometimes the need for them completely disappears). Regularization is obtained because the result of the network operation for a particular example is no longer deterministic, which simplifies the generalization.

3.11 Convolutional and pooling

For this chapter I used [23][26][71][92][93][94][95]. Convolutional neural network (CNN) is a special architecture of artificial neural networks, the proposed by Yann LeCun for the effective image recognition[9], is a part of deep learning technologies. Today, the use of CNN is one of the main methods for extracting features from the audio, video and text data[19][20][45][47].

The basic idea in convolutional layer is to use a mathematical convolution operation (filter) to sample. Convolution is a two-dimensional matrix of coefficients.

The advantage of using such filters is: the number of output more if the image element is more like a filter applied to it. Using the convolution operation helps to get the output image, each pixel of which will correspond to the degree of similarity on a piece of image filter. In other words, we get the map features.

Sub-sampling (pooling) is used to reduce the overall size image and increase the degree of invariance applied to it filters. The architecture of CNN feature availability on the image is more important than the exact knowledge of its coordinates.

The convolutional neural network alternating convolution layers (C-layers), sub-sampling layers (S-layers) and fully connected feedforvard (F-layers) layer at the output.

This architecture contains three main paradigms:

- Local invariance.
- Shared weight.
- Down Sampling.

Local invariance- to the input of a single neuron does not feed the entire image (or the outputs of the previous layer), but only some of its area.
The concept of shared weights suggests that for a large number of links/synapses used a very small set of weights [Figure 3-22]. In the process of propagation in the C-layer, each image fragment is multiplied element-wise matrix on a small scale (core), the result is summed. This sum is a pixel of the output image, which is called the map features. If we have an 32x32 pixel image, each of the neurons of the next layer takes only a small entry section of the image size, for example 5x5, each of the fragments to be processed with one and the same set. Such sets of weights may be many, but each of them will be applied to the entire image. These sets of weights often called kernels. It is easy to calculate that even 10 neurons 5x5 input image size of 32x32 number of connections will be approximately 256,000, and the number of adjustable parameters, only 250!
Stride size defining by how much you want to shift your filter at each step. A larger stride size leads to fewer applications of the filter and a smaller output size.
Such a limitation on the weight of the network improves the properties of generalizations (generalization), which ultimately has a positive impact on the ability of the network to find the invariants in the image and respond mainly to them, ignoring other noise.
The number of cores (sets of weights) is determined by the developer and depends on a number of features to select. Another feature of the convolutional layer is that it can slightly reduces the image due to edge effects. Adding zero-padding is also called wide convolution, and not using zero-padding would be a narrow convolution
The essence of subsampling and S-layers is to reduce the image spatial dimension. The most common way to do subsampling it to apply a max operation to the result of each filter over a window. It provides a fixed size output matrix, which typically is required for classification.
Alternating layers allows to make maps of the features of maps of the features, which in practice means the ability to recognize complex features hierarchies.
Typically, after several layers, map features degenerates into a vector or a scalar, but it is becoming hundreds of such maps of features. Then they are served by few layers of fully connected network. The output layer of the network may have different activation functions.

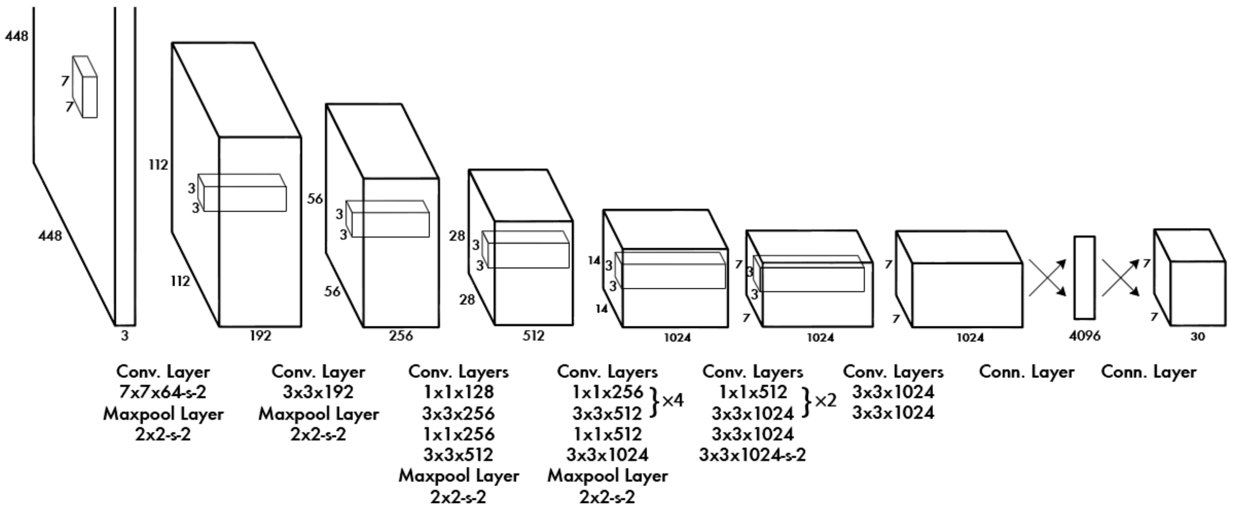


Figure 3-22 Scheme of the convolutional neural network for the classification of images [17].

Input to most natural language processing (NLP) tasks are sentences or documents. Like input featurescan be represented words or characters. In case of characters, convolution will get something like n-grams, without needing to represent the whole vocabulary.
Typically, words was embedded (low-dimensional representations) like word2vec or GloVe, but they could also be one-hot vectors that index the word into a vocabulary. Convolutional windows slide over full input rows. typically window size is 2-5 words.
In picture recognition often used different channels for input RGB (red, green, blue) colors. In NLP various channels can be word2vec and GloVe for example, or the same sentence represented in different languages, or phrased in different ways.
Convolutional Neural Network for NLP may look like this [Figure 3-23].

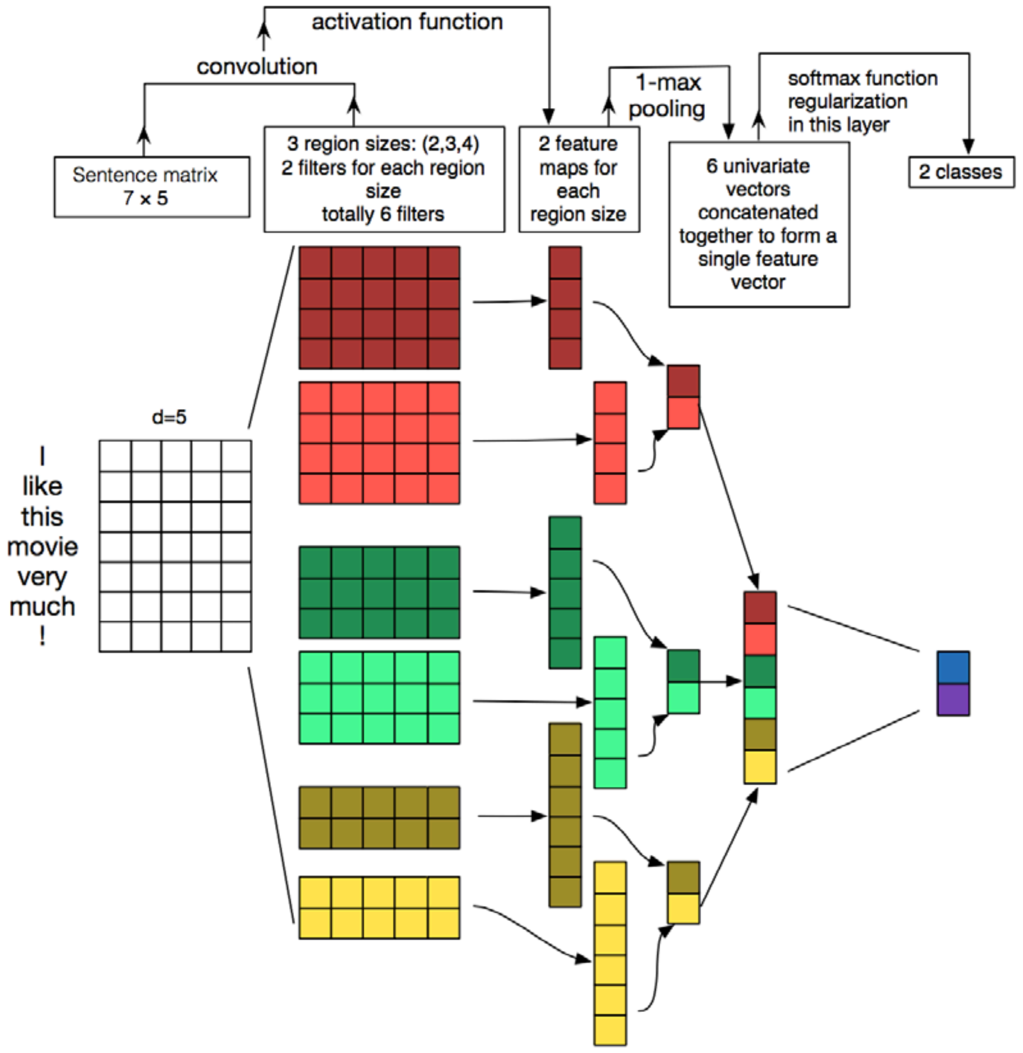


Figure 3-23 Example of convolutional Neural Network for NLP. Here are three filter region sizes: 2, 3 and 4, each of which has 2 filters. Every filter performs convolution on the sentence matrix and generates (variable-length) feature maps. Then 1-max pooling is performed over each map, i.e., the largest number from each feature map is recorded. Thus a univariate feature vector is generated from all six maps, and these 6 features are concatenated to form a feature vector for the penultimate layer. The final softmax layer then receives this feature vector as input and uses it to classify the sentence; here we assume binary classification and hence depict two possible output states. Source: Zhang, Y., & Wallace, B. (2015). A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification[96].

A big argument for CNNs is that they are fast. Very fast. Convolutions are a central part of computer graphics and implemented on a hardware level on GPUs.
As hyperparameters for convolutional networks for NLP will be:

- Narrow vs. Wide convolution
- Stride Size
- Pooling window
- Word/character-level

3.12 Recurrent, LSTMs and GRUs.

Recurrent Neural Networks [23][24][25] make more intuitive sense. They resemble how we process: Reading sequentially from left to right[58].
Recurrent Neural Network (RNN), in contrast to the Feedforward Neural Network (FNN), in addition to the trained set of parameters (weight and biases), uses the calculated values of previous network conditions.
Examples of the use of recurrent neural networks in the field of text processing in natural language:

- Simulation of vector representations of text information;

- Text generation;
- Analysis of the tone;
- Classification of texts.

A simple recurrent neural network is formally described as follows:

$$h^{(t)} = f(x^{(t)}, h^{(t-1)})$$
$$y^{(t)} = g(h^{(t)})$$

where $x^{(t)}$ is the input vector at time t (for example, a vector representation of the current word in the text fragment); $h^{(t)}$ is the state of the hidden recurrent neuron at a given time; $y^{(t)}$ is the output [Figure 3-24].

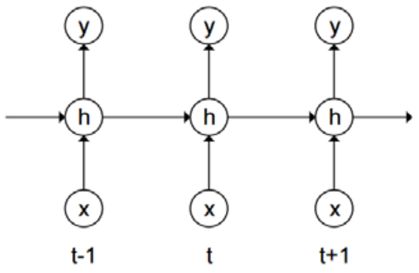


Figure 3-24 The scheme of the recurrent neural network.

The equation for calculating the value of the hidden neuron $h^{(t)}$ may vary in different versions, but has most frequently the form:

$$h^{(t)} = f(W^{hx}x^{(t)} + W^{hh}h^{(t-1)} + b^{h})$$

where W^{hx}, W^{hh}, b^{h} are trained parameters of recurrent neural network; f is nonlinear transformation (for example, a sigmoid function or the hyperbolic tangent).

The output function of the value $y^{(t)}$ depends on the specific tasks and can be the value of the hidden neuron. For example, in the classification problem is often used softmax function:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

This practice used fully connected feed forward layer. So,

$$y^{(t)} = \text{softmax}(W^{hc}h^{(t)} + b^{c})$$

where $y^{(t)}$ is vector of probabilities of belonging to each of the classes l; W^{hc} and b^{c} are trained parameters.

For more complex nonlinear transformations using deep neural network from multiple recurrent layers [Figure 3-25].

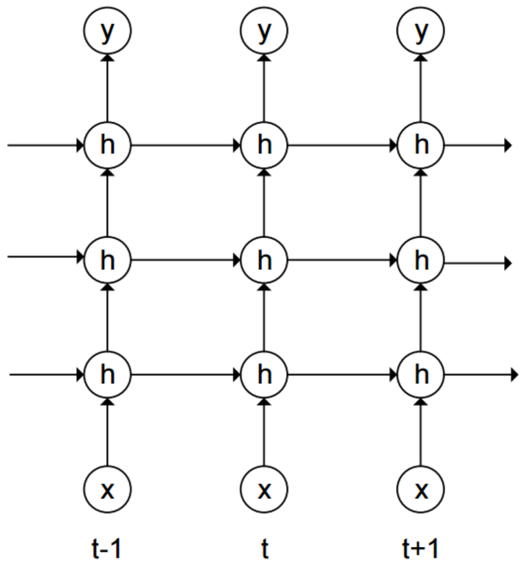


Figure 3-25 The scheme of the multiple recurrent neural network.

Depending on the desired result, various modifications of network used: one input to one output network - simple feed forward network, and some variations of recurrent networks[Figure 3-26].

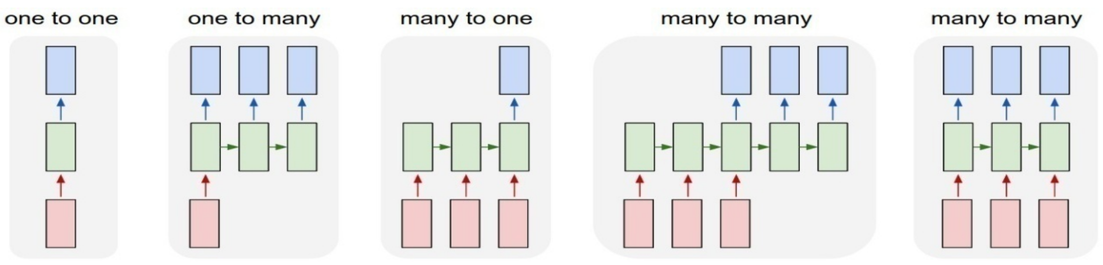


Figure 3-26 Various modifications of recurrent neural network [97].

Another improvement of the recurrent model is bidirectional [27], where the state of the hidden layer consists of two independent elements that are calculated using the $h^{(t)}{}_{\text{forward}}$ and $h^{(t)}{}_{\text{backward}}$ [Figure 3-27]

$$h^{(t)}{}_{\text{forward}} = f(W^{hx}{}_{\text{forward}}\ x^{(t)} + W^{hh}{}_{\text{forward}}\ h^{(t-1)} + b^{h}{}_{\text{forward}}\ )$$
$$h^{(t)}{}_{\text{backward}} = f(W^{hx}{}_{\text{backward}}\ x^{(t)} + W^{hh}{}_{\text{backward}}\ h^{(t-1)} + b^{h}{}_{\text{backward}}\ )$$
$$y^{(t)} = g(h^{(t)}{}_{\text{forward}}\ ,\ h^{(t)}{}_{\text{backward}}\ )$$

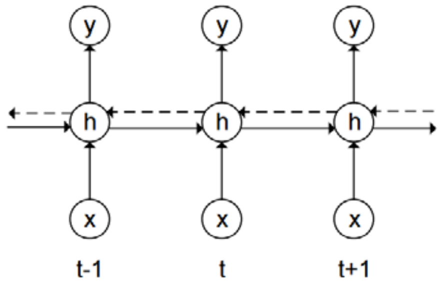


Figure 3-27 The scheme of the bidirectional recurrent neural network.

For training recurrent network used the same algorithm backpropagation, as for the feed forward networks. Let $\theta^{(t)}$ is a set of trained model parameters in time t. θ^{0} initialized with random values of small value (close to 0). Then, using the gradient descent, we have:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha\frac{\partial J}{\partial \theta}$$

Ideally, the partial values must take into account the use of the trained network parameters at all time points up to the entry.

Despite the universality of the model, it has a number of drawbacks. Thus, in determining the key of a long sentence, the model well, "remembers" Only few last tokens of the input text fragment. At each step, hidden layer updates and important information at the beginning of the sequence, may be lost. More advanced models are Gated Recurrent Unit (GRU) and Long Short Term Memory (LSTM).

Gated Recurrent Unit (GRU) and Long Short Term Memory (LSTM)

Recurrent networks described in many sources, for this chapter I used [22][50][98][99][100]. In 1997 Hohrayter Sepp and Jürgen Schmidhuber presented an approach called LSTM (Long Short-Term Memory). Recurrent Neural Network based on this approach have an improved (and more complex) method calculate the state of the hidden layer $h^{(t)}$. This method, in addition to input values, and the previous state of the network uses filters (gates), defining how information is used to calculate the output values for a current layer $y^{(t)}$. And the values of the hidden layer in the next step $h^{(t+1)}$. Calculation $h^{(t)}$ is called LSTM layer(LSTM unit) [Figure 3-28].

Along with the network state h, at each step calculated storage unit (memory cell) using the current input value $x^{(t)}$ and the value of the block in the previous step $c^{(t-1)}$.

Input filter (input gate) $i^{(t)}$ determines whether the value of the memory block in the current step should influence the result. Filter values range from 0 (completely ignore the input values) to 1, that provided by a sigmoidal function:

$$i^{(t)} = \sigma(W^{i}x^{(t)} + U^{i}h^{(t-1)})$$

For each hidden layer, it shares a unique weight **W** (weight between two time slot) and weight **U** (weight between hidden layers).

"Filter forgetting» (forget gate) allows to exclude when calculating the value of the previous step memory:

$$f^{(t)} = \sigma(W^{f}x^{(t)} + U^{f}h^{(t-1)})$$

Based on all data received at time t ($x^{(t)}, h^{(t-1)}, c^{(t-1)}$), calculated the state storage unit c (t) at the current step.

$$\tilde{c}^{(t)} = \tanh(W^c x^{(t)} + U^c h^{(t-1)})$$
$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}$$

Output filter (output gate) is similar to the previous two and looks like:

$$o^{(t)} = \sigma(W^o x^{(t)} + U^o h^{(t-1)})$$

The total value LSTM-layer is determined by the output filter and non-linear transformation on the state of the memory block.

$$h^{(t)} = o^{(t)} \circ \tanh(c^{(t)})$$

GRU (Gated Recurrent Unit), based on the same principles as the LSTM, but uses fewer filters and operations to calculate $h^{(t)}$. Filter update (update gate) $z^{(t)}$. and reset the filter status (reset gate) $r^{(t)}$. calculated using the following formulas:

$$z^{(t)} = \sigma(W^z x^{(t)} + U^z h^{(t-1)})$$
$$r^{(t)} = \sigma(W^r x^{(t)} + U^r h^{(t-1)})$$

The output value of $h^{(t)}$ is computed based on the intermediate value $\tilde{h}^{(t)}$, that, with the filter of reset state determines which values from previous step $h^{(t)}$ should be deleted.

$$\tilde{h}^{(t)} = \tanh(Wx^{(t)} + r^{(t)} \circ Uh^{(t-1)})$$

With the update filter and an intermediate value

$$h^{(t)} = z^{(t)} \circ h^{(t-1)} + (1 - z^{(t)}) \circ \tilde{h}^{(t)}$$

Similarities and differences between LSTM and GRU-layers (left LSTM, right GRU)

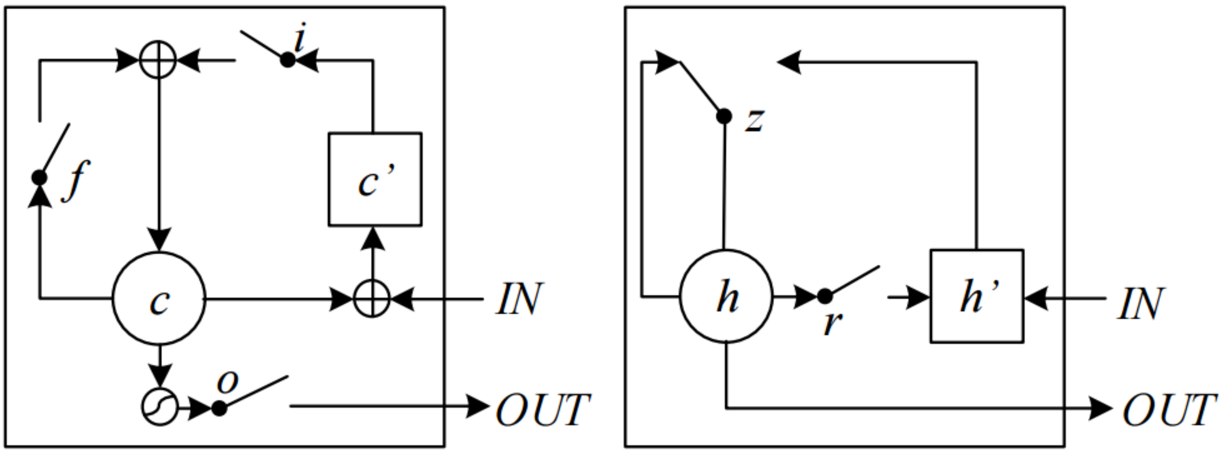


Figure 3-28 Illustration of (a) LSTM and (b) gated recurrent units. (a) i, f and o are the input, forget and output gates, respectively. c and c˜ denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and h˜ are the activation and the candidate activation. [6]

Like other recurrent neural network, LSTM and GRU (especially double and multi-layer) enough complicated to training. Significantly accelerate learning processes deep neural networks allow graphics processing units (GPU).

Recurrent Neural Network (as well as variations thereof GRU and LSTM), are quite effective in dealing with text information analysis tasks.

GRU less redundant, and trains by 20-30% faster than LSTM.

3.13 Additional NN optimization

Ensembles of Neural Networks

One way to improve the accuracy of the models is to provide ensembles of models - sets of models used to solve the same problem. Training of ensemble mean training a finite set of base classifiers and then combining the results of their prediction in a single forecast. The combined classifier will give a more accurate result, especially if:

- each base classifier itself has a good accuracy;
- they lead to different results (wrong on different sets).

Ensembles advantages:

- Statistical. Aggregate classifier "averages" the error of each of the basic classifications, therefore the impact of accidents on averaged result is significantly reduced.
- Computing. From different points easier to find the global minimum.

- Representative. It may also happen that the total result will be located outside of the set of results base, in this case, the construction of a combined result, extend the set of possible results.

Depending on how the ensemble is constructed, its use allows to solve one of two problems: the basic tendency of neural network architecture to not fit enough (this problem is solved by boosting), a tendency to overfitting (this problem is solved by bagging).
There are various universal voting scheme, for which the winner is the class:

- the maximum - with a maximum response of the ensemble members;
- averaging - with the highest average response of the ensemble members;
- The majority - with the largest number of votes of members of the ensemble.

Several possible ways of organizing ensemble of neural networks.
It is possible, to teach another neural network ("Head of the Committee"), the inputs of which will be predictions of all neural network, and the output will be final output.
Another approach could be the introduction of "specialization" of the concept of experts. For this inputs previously clustered into several (2-5) groups of similar input patterns.
Among the many ensemble classification methods, consider the most common are:

- begging (bootstrap aggregating)
- boosting

Begging
Bootstrap aggregating -bagging, it is the union of the results at different loadings. In the absence of a large training sample, it can be created a lot of random samples from the original simple replacement selection. Although the elements in the samples may be duplicated or overlap. The method is so called because it combines the results of the predictions of various classifiers trained on random subsets. Begging is useful only in case of different classifiers and instability when small changes in the initial sample leads to significant changes in the classification.
Busting
Busting (boosting, improvement) - this procedure is a serial of composition algorithms, where each following algorithm seeks to compensate for the shortcomings of the composition of previous algorithms. Busting is a greedy algorithm for constructing algorithms composition. Weighted voting does not increase the complexity of the algorithm is effective, but only smoothes answers basic algorithms. Efficiency boosting due to the fact that at least adding basic algorithms increase margins learning objects. Experiments have shown that sometimes boosting overfitted.

3.14 Vectorization

In my work, I classify the columns of tables. I assume that there is no relations between the cells of the column and they can be used in any order. Each cell in my work is one example. Since each cell contains only a few words and possibly special characters [101], I use information in each cell character-by-character in ASCII encoding. Many ideas of character-by-character processing of text can also be used in word-by-word processing.
To input samples to the neural network, several problems must be solved.
First, need to convert characters to numbers.
The simplest approach is to compose a dictionary of unique characters, and assign each character to a number from the dictionary. The disadvantage of this approach is the different "distance" between the character, for example the character "a", from the point of view of the neural network, can be more similar to "b" than to "c".
The second option is to encode characters with a vector same long as dictionary, in which the right symbol corresponds to 1, and all the rest to 0 (one hot vector) [Figure 3-29].

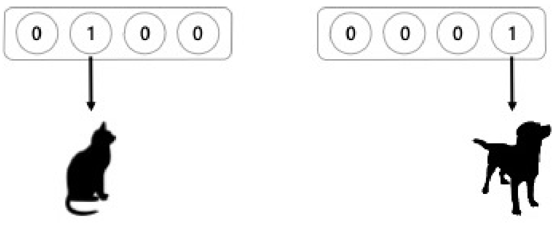


Figure 3-29 One hot vectors, for worlds cat and dog in case of four different animals.

Here all symbols are equidistant. The problem with this approach is in too long vectors of zeros, with a large dictionary. The simplest way to get the resultant word wind is to add character vectors, such a representation of a word is often called "neural bag of words / chars" [4][44] [Figure 3-30]. With this addition, words of different lengths yield identical vectors along the length. The advantage of this algorithm is the extreme simplicity of implementation, but information about the character order is lost. The results sometimes better than other more complex algorithms.

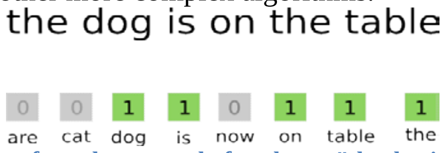


Figure 3-30 Bag of words approach, for phrase "the dog is on the table".

An interesting idea of the classification of words is suggested [102], the idea is to pre-cluster words into a number (several dozen) of clusters. Then each word is represented as a vector of distances to the center of each of the clusters. As a result, each word is encoded by a new vector, each element of which is easily explained in terms of the degree of relation to the clusters selected for clustering. Each element of such a vector has a simple and understandable explanation. And such a vector can already be used to feed another neural network or another method of machine learning to the input.

In the case of vectorization algorithms based on words, rather than individual characters, n-grams can be used for words. At the same time, a dictionary of all possible n-grams is made up, how much memory is enough (one letter, two, three, ...). Then any word is represented as a sequence of such n-grams.

The convolutional neural networks work on a similar principle, but instead of the n-grams dictionary, convolution filters are used. [103]

Reduce the dimension

To reduce the dimension, different algorithms of machine learning can be used, mapping one multidimensional space to another, and special methods for reducing the dimensionality of data, such as principal component analysis (PCA), independent component analysis (ICA), non-negative matrix factorization (NMF or NNMF), singular value decomposition (SVD)[12][104][105][106][107][108]

In this paper, I do not reduce the dimensionality of data, so I will briefly describe only the most popular of these SVD algorithm.

The idea of SVD is simple - any matrix (real or complex) is represented as a product of three matrices:

$$X = U\Sigma V^*$$

Where U is a unitary matrix of order m; Σ is a matrix of size m x n, on the main diagonal of which there are nonnegative numbers, called singular (the elements outside the main diagonal are equal to zero); V^* is a Hermitian transpose matrix of order n on V. m columns of the matrix U and n columns of the matrix V are called respectively the left and right singular vectors of the matrix X. For the problem of reducing the number of dimensions, it is the matrix Σ whose elements, raised to the second power, can be interpreted as the variance that " Component, and in descending order: σ1 ≥ σ2 ≥ ... ≥ σnoise. Therefore, when choosing the number of components for SVD guided by the sum of the variances given by the components considered.

The disadvantage of the method is that the singular expansion is rather slow; Therefore, when the matrices are too large, randomized algorithms are used.

Word embedding

Also, there are methods for vectorizing words based on relationships between words , on large volumes of texts, to create high-dimensional (50 to 300 dimensional) representations. Word embedding is the collective name for the set of language models and feature-learning techniques in natural language processing (NLP) where words or phrases from the vocabulary are mapped to vectors of real numbers [https://en.wikipedia.org/wiki/ Word_embedding]. Word embedding algorithms divided on two main ideas Continuous Bag-of-Words model, and a Continuous Skip-gram model [5][18][109][110].

The most famous implementation of the algorithm is word2vec, developed by Google in 2013. In the Internet you can download already trained word2vec models. To work with the text, the algorithm uses the sliding window over the text, that includes the central word, currently in focus, together with the four words and precede it, and the four words that follow it.

Word2vec uses a single hidden layer, fully connected neural network. The neurons in the hidden layer are all linear neurons.

In continuous bag-of-words model, the context words encoded in one-hot form and sent to the input layer. The training objective is to maximize the conditional probability of observing the actual output word (central window word) given the input context words. The skip-gram model is the opposite of the CBOW model. It is constructed with the focus word as the single input vector, and the target context words are at the output layer [Figure 3-31].

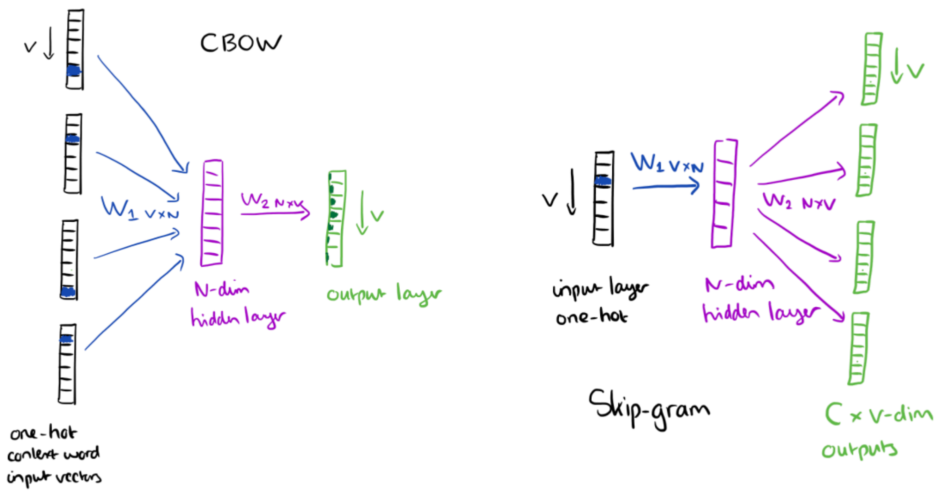


Figure 3-31 Continuous Bag-of-Words model (left) and Continuous Skip-gram model (right). [110]

Additional improvements

Also, as additional improvement can be used stemming and lemmatization. Some sources mention that phrase can be translated in different languages to increase amount of features. In this case I think that phonetic algorithms as metaphone can be used to get more features from words [111][112].

As additional information for a neural network, can be used special tokens (gender, number, class, etc.).

Data requirements

The basic requirement for training data, control and test sets should be representative. The statement "garbage in, garbage out" is well suited for neural networks. If the training data is not representative, then the model, at least, will not be very good, and in the worst case - useless. The neural network can be trained only on the data that it has.

The data must be balanced. Algorithms for training neural networks minimize the overall error, the proportions in which data of various types are presented are important. A network trained on 900 good and 100 bad examples will distort the result in favor of good observations, since this will allow the algorithm to reduce the overall error (which is determined mainly by good cases).

The network learns what is easiest to learn. As example, in case the network is designed to classify photos, cloud weather, or lighting, can become a key factor in network classification, than objects in photos. In such case, for the network to work correctly, it should be trained on data, where all the weather conditions and types of lighting which are interesting for feature network problems.

Also the set of local minima of the error function can be a property of the input, rather than the desired surface and the metric that we optimize. For example, we can take the function $y = |x|$.

And train the network to determine whether the point lies above or below this function on x is from -1 to 1 and y is from 0 to 1. As input data we will use the coordinates of points represented in the form of two numbers x and y. This requires a two-layer network, the first layer of two neurons, the second of one. Such small network will quickly learn and will fit examples well.

Then we take the same function, but the input data - 16 binary features. The first 8-bit decomposition of the first coordinate, the second 8 the same for the second. Analytically, the problem is solved similarly to the previous one, if the weights of the input parameters are additionally multiplied by the powers of two. But training the network on such data will be much more difficult. Even if that the final task and the metric are exactly the same.

In the case of the word embedding classification, vectors based on the sense of words are likely to be easier to accept by the neural network.

Correlation between input samples features can be problem too [lectures: Tensorflow and deep learning - without a PhD], Neuron network can solve this problem with addition layers, or such correlation can be removed with batch normalization.

Number of samples

The number of observations for the learning network is determined by heuristic rules which establish the relationship between the number of observations required and the size of the network. By different sources the number of observations should be 10 times bigger than network coefficients[31], or just bigger than network coefficients[113]. In fact, this number depends on the complexity of the display, which should play a neural network.

As the number of features used by the number of observations increases nonlinearly, so that even at a fairly small number of features, say 50, you may need a huge number of observations. This problem is called the "curse of dimensionality".

4 Examples of neural networks in natural language processing

In most cases, neural networks are used in NLP for various text classifications, such as Sentiment Analysis, Spam Detection or Topic Categoryization. Both convolutional and recurrent networks are used. Few recurrent networks examples:

- There are recurrent (LSTM) networks for classifying product reviews on Amazon [32].
- In medicine, recurrent networks are used in concept linking, or linking spans of text to concepts in a knowledge graph (KG)[33]
- Recurrent nerworks in spoken language understanding [34]
- LSTM Conversational modeling[35]
- Recurrent networks for searching words in continous strings[36]
- Character predictions by recurrent networks[37]
- LSTM Question answering [97]

Convolutional networks examples:

- Sentence classification[38][41][19]
- Modelling sentences [39]
- Text categorization [40]
- Character level text classification[42][43]

5 Framework/language

In general, the neural networks are not difficult to program, the main difficulty is very high requirements for code optimization. After several simple implementations of FFNN neural networks in the C# programming language, I wondered about the search for a faster neural network frameworks. Currently, there is a fairly large number of frameworks for working with neural networks. All frameworks are divided into two main groups [114]:

- Symbolic computation frameworks (CNTK [115], Theano [116], TensorFlow [117], MXNET [118]) are specified as a symbolic graph of vector operations, such as matrix add / multiply or convolution.
- Non-symbolic (imperative) neural network frameworks (Torch [119], Caffe [120])

My main requirement for the neural networks framework was using of Nvidia GPU. Nvidia has implemented a hardware-software parallel computing architecture that significantly improves computing performance through the use of Compute Unified Device Architecture (CUDA) graphics processors. Additionally, I wanted to have a framework that can run on Linux and Windows platforms. The programming language was not fundamental, when choosing a framework.

After reading the overview articles [121] [114], I stopped On the Theano framework, after getting to know the basic capabilities of the framework, I planned to use a high-level platform based on it - Lasagne. As an operating system, I tried using Windows 7 home premium, Windows 10 Professional and linuxmint 18.1cinnamon. Unfortunately installing the framework on each of these systems requires me 1-2 days. Also, in the process of compiling examples, I found that Theano very long compiles graphs for working with the GPU, especially in complex models. And as a result, learning models on the CPU is often faster than compiling and training models on the GPU. Perhaps during the installation, I made some mistakes.

The next candidate was the Tensorflow framework. Tensorflow like Theano can use the GPU capabilities. Installing Tensorflow on a computer with Windows 7 home premium took a few minutes. After compiling and learning a dozen models found on the Internet, I was satisfied with the performance of the framework. To speed up the development process, I switched to a high-level Keras platform using Tensorflow as a backend.

For work with neural networks I used the desktop NVIDIA GeForce GTX 670 4GB, Intel Core i7-3770K 3.50GHz, 8x2 GB DDR3-1333 DDR3

6 Datasets

As I mention above, the neural network can be trained only on the data that it has. So, I think, choosing training data is the most important part in working with neural networks. It is impossible to train network without training data and quality and amount of data will have most significant influence on results of training.

But collecting data for training the network is too time-consuming task and under this project I used datasets freely available in the interest and provided to me by the company Avito LOOPS.

Here I show the histograms of length of samples for the data sets used in the final classification. These histograms are needed to select the maximum length of the samples for trainig the neural network. The maximum length of samples affects the learning speed, the number of network parameters, the quality of the classification, and the size of the maximum batch (given by the amount of computer memory). All the histograms for clarity are made up to 60 chars. Some samples contains more than one word, and in this case space was additional char in histogram and for neural network.

The datasets available to me:

The Fuse Spreadsheet Corpus (fuse-binaries-dec2014.tar.gz (6.9 GB)) [122], containing 2,127,284 URLs that return spreadsheets (and their HTTP server responses), and 249,376 unique spreadsheets, contained within a public web archive of over 26.83 billion pages. Dataset contains not sorted spreadsheets in .xls (Excel 2003) and .xlsx (Excel 2007). The absence of a common data structure led to the fact that for reading the data I had to use three external applications, C# NPOI, VBA macros, and python (looks like good solution will be Apache POI), and reading all the spares on the columns took about 8 hours on my desktop. In the current research I decided not use this dataset, but the next stage of the work can be the sorting of this dataset with the help of neural networks obtained in this research.

The free and open global address collection (openaddr-collected-global.zip (8.6 GB))) [123], that contains .csv spreadsheets sorted by countries. Each spreadsheet has information about street names, house numbers and postal codes, combined with geographic coordinates from all world (LON, LAT, NUMBER, STREET, UNIT, CITY, DISTRICT, REGION, POSTCODE, ID, HASH). In this research I used only unique street names, that contains only latin letters, numbers, points, commas, dash and underscores. From this database I get 6357749 unique street names. After a quick survey of the base, I realized that the column with the street name is filled most fully, and in the case of Russian (Russian is my native language, so I easily can review such data) addresses it often contains information about the street position without a name or about the old and new name of the street. So for me It looks like database is has some noise. Figure [Figure 6-1] show histogram of street names that can be written in ASCII samples length.

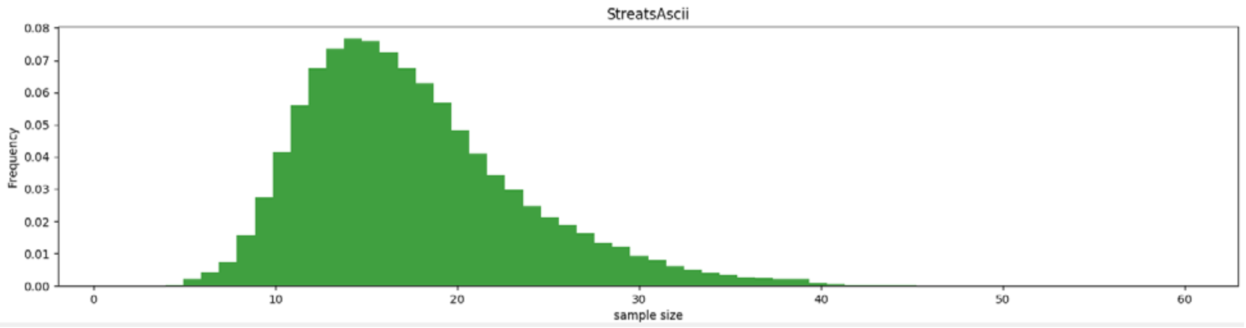


Figure 6-1 Histogram of ASCII street names samples length.

First, second names and job position titles. I got this datasets from the company Avito LOOPS. After removing duplicates I got 50193 unique first world names, 195153 unique second world names and 45665 unique job titles. Quick glance through first and second names, shows that some samples classified incorrect (applied to the Russian language and Russian names) , for example name " Julia" is a female first name in Russia, but according this dataset it is second name. Job titles contains a lot very similar samples as "3rd Party/Mtrl Supply Planner Additives" and "3rd Party/Mtrl Supply Planner Base Oils". Figures [Figure 6-2][Figure 6-3][Figure 6-4] show histogram of names and position titles samples lengths.

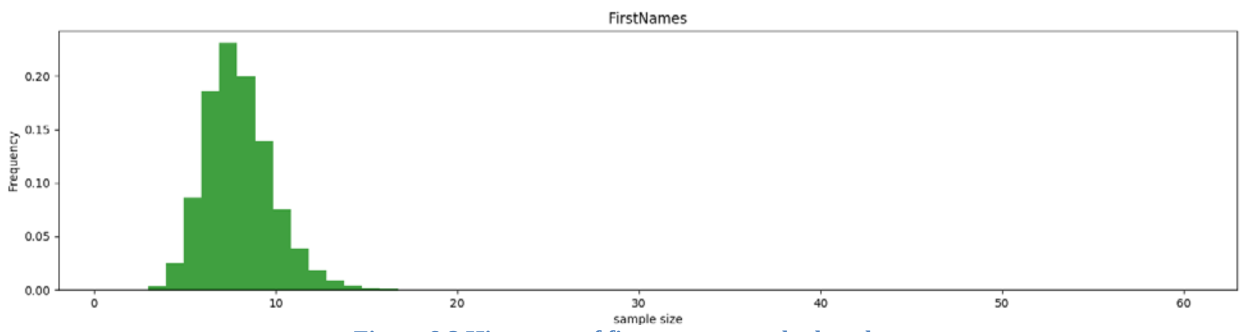


Figure 6-2 Histogram of first names samples length.

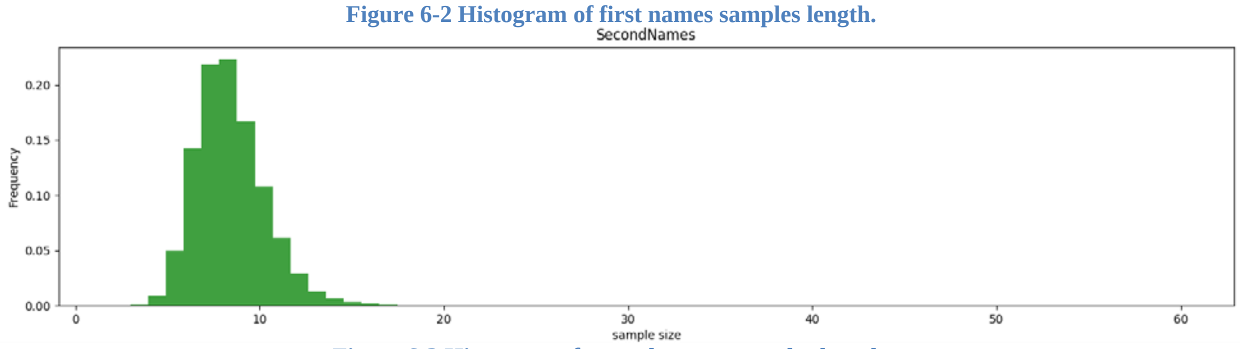


Figure 6-3 Histogram of second names samples length.

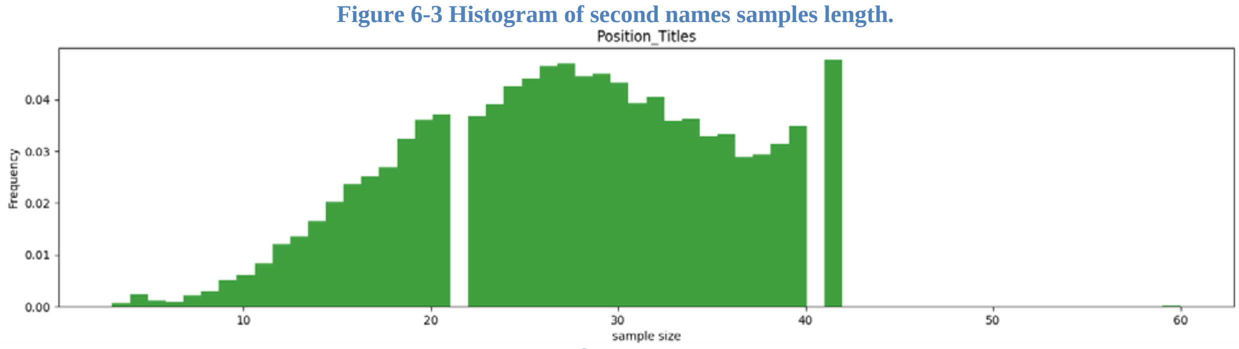


Figure 6-4 Histogram of position titles samples length.

Avito LOOPS provide me dataset with 1007 .csv spreadsheets. Some columns was classified on more than 20 classes, as date, location, code, activity descriptions,... But, after extracting samples from the spreadsheets by classes, I got very short lists of samples. So I was able to take only class **code**, from this dataset, that contains 518801 samples of equipment codes. Figure [Figure 6-5] show histogram of code samples length.

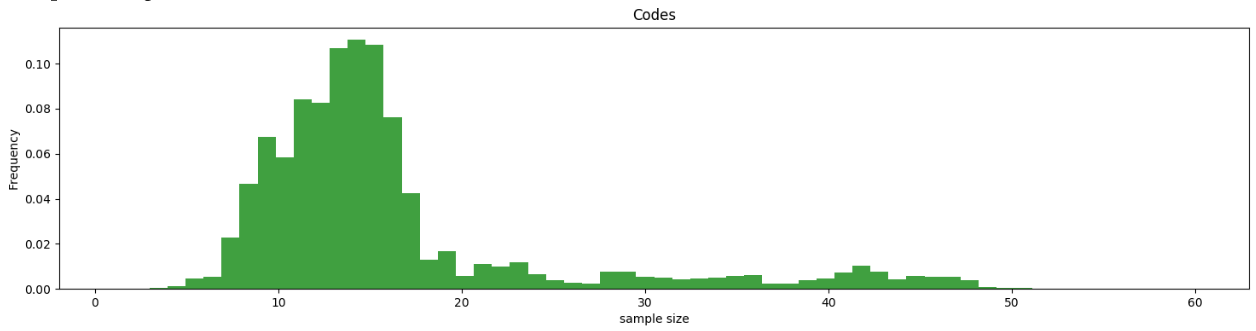


Figure 6-5 Histogram of code samples length.

Also In the beginning of this research, I used **generated** dataset [Figure 6-6] with random dates recorded in different formats as "9.17", "17-Sep", "17-Sep-96", "September-96", "September 17, 1996", "9.17.96 12:00 AM", "09.17.96 0:00", "17.09.1996",... 14 types of dates. Main advantage of this dataset, that I can generate as much data as I need (data is limited to listing all possible addresses for a certain period).

Also in the first experiments with neural networks I used not relevant with project datasets:

MNIST database of handwritten digits, has a training set of 60,000 examples, and a test set of 10,000 examples. [124]

Large Movie Review Dataset, dataset for binary sentiment classification containing set of 25,000 highly polar movie reviews for training, and 25,000 for testing [125]

Sentiment140, tweets dataset, for sentiment analysis. Dataset contain 1.6 million samples polarity of the tweet (0 = negative, 2 = neutral, 4 = positive), and tweet text. [126]

List of videocards 1447 unique videocard names [127]
All samples can be from 2-3 characters to 50 characters.

5.22	5.22.07	05.22.07	22-May	22-May-07	22-May-07	May-07
11.3	11.3.95	11.03.95	3-Nov	3-Nov-95	03-Nov-95	Nov-95
9.17	9.17.96	09.17.96	17-Sep	17-Sep-96	17-Sep-96	Sep-96
8.20	8.20.91	08.20.91	20-Aug	20-Aug-91	20-Aug-91	Aug-91
2.8	2.8.92	02.08.92	8-Feb	8-Feb-92	08-Feb-92	Feb-92

May-07	May 22, 2007	5.22.07 12:00 AM	5.22.07 0:00	5.22.2007	22-May-2007	22.05.2007
November-95	November 3, 1995	11.3.95 12:00 AM	11.3.95 0:00	11.3.1995	3-Nov-1995	03.11.1995
September-96	September 17, 1996	9.17.96 12:00 AM	9.17.96 0:00	9.17.1996	17-Sep-1996	17.09.1996
August-91	August 20, 1991	8.20.91 12:00 AM	8.20.91 0:00	8.20.1991	20-Aug-1991	20.08.1991
February-92	February 8, 1992	2.8.92 12:00 AM	2.8.92 0:00	2.8.1992	8-Feb-1992	08.02.1992

Figure 6-6 Examples of generated dates.

7 Design

By design in this research is understood choice of the best model. The choice of the best model went through the steps, based on the theory, by experimentally selecting the best hyperparameters.
For the implementation my neural networks I used Tensorflow/Keras frameworks. First I made few tests on GPU (graphics processing unit) and CPU (central processing unit). According this tests GPU give me better performance in 2-5 times than CPU on my desktop computer, but spend some time on compiling code in CUDA. All other examples I made on GPU.
Unfortunately, I received/found data for training network in parallel with work on the network. Many stages of work could be shorter if I had all the data right away.
Data between epochs was shuffled.
Examples not relevant with project problem
Here I want to make short description of examples, that I used to understand neural networks. This examples are not relevant with my main goal, but used the same approaches.
I assume that **MNIST** dataset and **XOR** example in neural networks are something the same as "hello world" in any programming language, and any newbie in neural networks will start from this examples. XOR problem is too simple and can be solved with back propagation on paper. But **MNIST** is already good example to understand basic neural network behavior. During working with this dataset I was trying feed forward networks and convolutional networks with dropout and batch normalization methods to solve overfitting problems. Best score on this models was over 99% accuracy. Examples implemented on Tensorflow.
As first natural language processing problem, for my neural networks I choose tweets sentiment analysis [128] This example use word lemmatizer for data preprocessing. Create lexicon based on a random one out of every 2500 samples. Data vectorized into the bag of words model. Model contain three feed forward layers. 10 epochs give around 74% accuracy. Example implemented on Tensorflow.
Sequence classification, Large Movie Review Dataset. [129] Dataset contains good or bad movie reviews. The problem is to determine whether a given movie review has a positive or negative sentiment. Keras provides access to the IMDB dataset the imdb.load_data() function allows to load the dataset in a format that is ready for use in neural network and deep learning models. So for this example I did not prepared data. This example shows word embedding, one dimensional convolutional layers and recurrent layers (long short term memmory), with best accuracy 86.36%. Example implemented on Tensorflow/Keras.
Also I was tried 33 different examples [130] provided by Keras. Most interesting is "Generates text from Nietzsche's writings" and almost the same example "Text Generation With LSTM Recurrent Neural Networks in Python with Keras"[131]. This examples works on character level, Each character vectorized in one hot vector. Recurrent network (long short term memmory) layers trained to predict next character.

7.1 Project problem solving overall.

During solving project problem I was working only on classification separate cells because I assume that there are no relations between cells in one column. To classify columns with such classifier we need to classify each cell separately, sum scores for all class and choose class with biggest score.
From theory part and examples of problems, that can be solved with neural network, I choose two main types of networks for this project: recurrent (simple recurrent, long short term memory LSTM, gated recurrent unit GRU) and one dimensional convolutional networks.
In this project I vectorize all samples on character level, because samples are not to big (it can be only 2-3 characters in one sample, and I do not have sample with length bigger than 50 characters). Samle can has lowercase latin letters and few special symbols, so all characters vectorized with one hot vectors- each sample vectorized in matrix with one hot vectors. As next improvement it can be interesting to use n-grams from samples with embedding algorithms and recurrent networks. But now I use the same vectors in recurrent and convolutional networks.
Since at the time I started developing the neural network, I had no experience with neural networks, the development of the final networks was carried out in four steps. The fourth step was the development and training of the final networks. Each step gave me a deeper understanding of the work of the neural network and the learning algorithms of the network.
At each step I tried to follow the basic rule: to complicate the network, increasing the number of layers and neurons in the layer, until the network starts overfitting, then applied the techniques against overfitting. If the overfitting succeeded to avoid, I continued to complicate the network.

But in the process of working with networks, I stopped at the maximum number of trainable parameters of the network, no more than 300,000, which was an average number of samples, at each step. Also with large networks, my computer did not do well.
This restriction is the most controversial and complex place in my work, described in more detail at the end of the second step.
In addition, despite the fact that the main parameter of network quality was the accuracy of classification, on a test set (an unknown network at the learning stage), it was important to get networks capable of generalizing information. Smaller networks are more generalized and, perhaps, having received a test set from another source, smaller networks would do better.
The possibility of classifying an unknown class is discussed in the Analysis chapter.

7.2 First step

First problem was to show that I understood framework and can train neural network on my own data. On this step I did not try to tune network. On this step I use Adam optimizer with default parameters (lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0 [132])
For the first classification problem I choose binary classification between videocard names (1447 unique videocard names) and generated dates (2100 dates in 14 different formats, 150 dates in each format- all types of dates are one class). Network architecture contain one recurrent (LSTM) layer with 10 units and one output neuron with sigmoid activation function [Figure 7-1]. Loss function is cross entropy, training algorithm is ADAM. From each sample I used first 40 characters. After one epoch with batch size equal to 5, I got accuracy on training set 0.9566. In this first example I do not use test set at all, goal of this example for me was to show that network can be trained on such data.

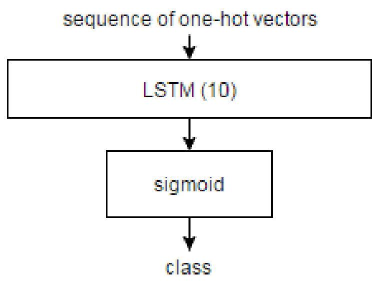


Figure 7-1 Neural network architecture.

Second classification problem was multiclass classification. During this classification I train network to classify 14 different formats of generated dates. Such classification is not always possible, because some dates are the same in different formats, like **17-Sep-96** can be: **9.17.96** and **09.17.96** but **17-Dec-96** will be **12.17.96** and **12.17.96** - in second sample it is impossible to choose class. As training set I use 1000 samples from all class, and as test set I used 100 samples from all class.
Network architecture contain one recurrent (LSTM) layer with 50 units and 14 output neuron with sigmoid activation functions [Figure 7-2]. Each output neuron calculate score for each class. Loss function is categorical cross entropy, training algorithm is ADAM. From each sample I used first 30 characters.
After 15 epoch with batch size equal to 50, I got accuracy on test set 0.8629.

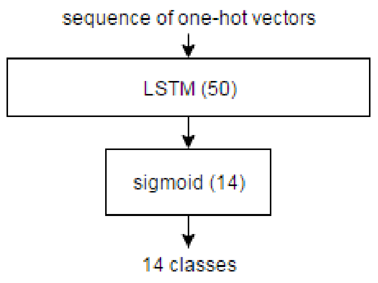


Figure 7-2 Neural network architecture.

Last classification problem in this step was binary classification problem on real data. In this problem I first time get some real and significant data for the project - classification between first and second names. To solve this problem I tried six different networks [Figure 7-3]

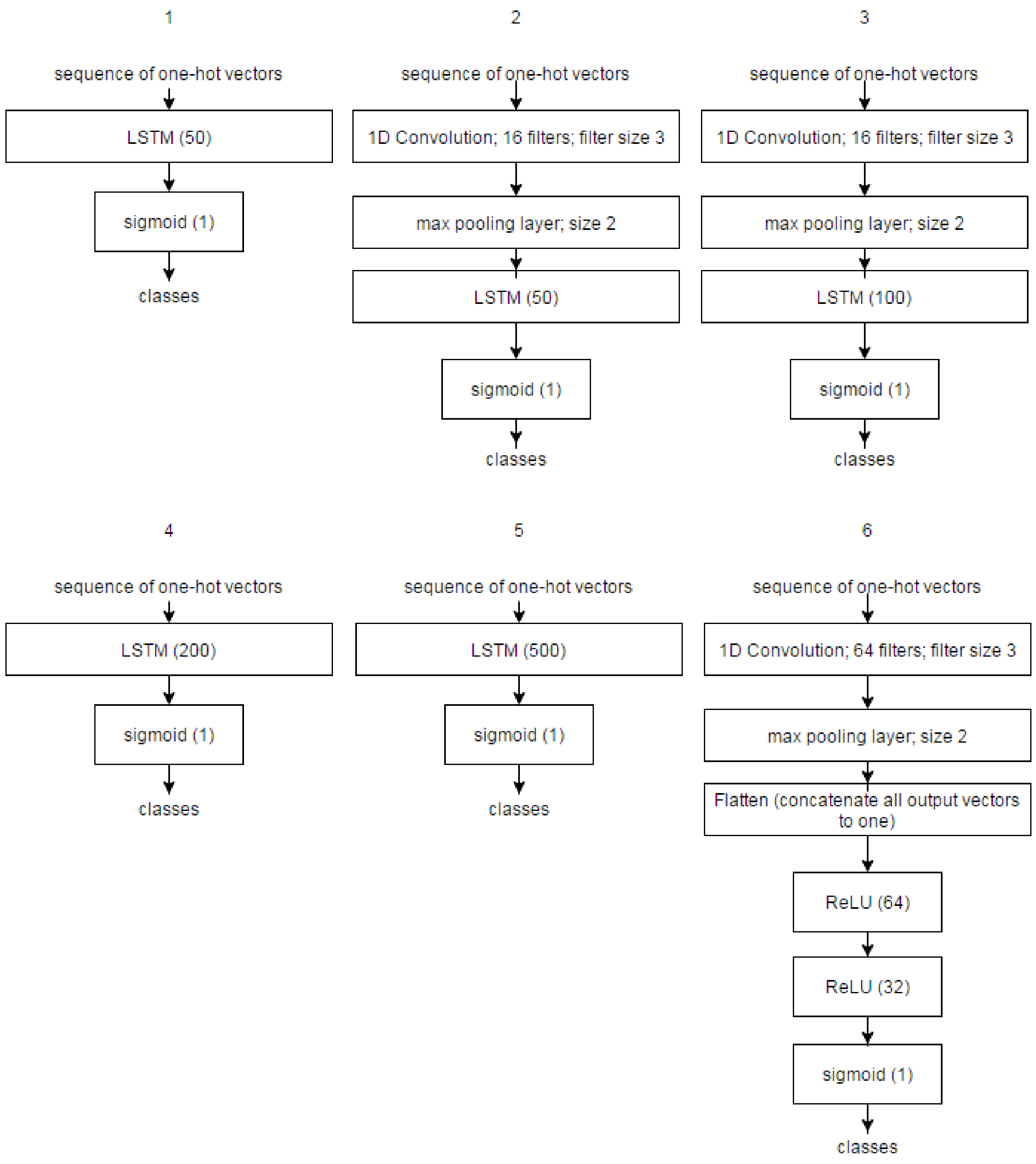

1
sequence of one-hot vectors
LSTM (50)
sigmoid (1)
classes
2
sequence of one-hot vectors
1D Convolution; 16 filters; filter size 3
max pooling layer; size 2
LSTM (50)
sigmoid (1)
classes
3
sequence of one-hot vectors
1D Convolution; 16 filters; filter size 3
max pooling layer; size 2
LSTM (100)
sigmoid (1)
classes
4
sequence of one-hot vectors
LSTM (200)
sigmoid (1)
classes
5
sequence of one-hot vectors
LSTM (500)
sigmoid (1)
classes
6
sequence of one-hot vectors
1D Convolution; 64 filters; filter size 3
max pooling layer; size 2
Flatten (concatenate all output vectors to one)
ReLU (64)
ReLU (32)
sigmoid (1)
classes

Figure 7-3 Network architectures for binary classification problem on real data.

In all networks convolution window on borders of sample start on the middle of window, that output same amount of neurons as input.
All networks has the same output layer with one sigmoid neuron.
All networks was trained with the same batch size 600, 15 epochs, loss function is cross entropy, training algorithm is ADAM. From each sample I used first 40 characters. In this example I assumed that data already shuffled, and first 200000 examples I used as training set, other 45347 as test set. Next I realized that it was mistake.
Accuracy on test set for each model after 15: 1: 0.8226, 2: 0.8234, 3: 0.8246, 4: 0.8364, 5: 0.8352 ,6: 0.8353.
Here I understand that some models has better accuracy before 15 epochs, but due to overfitting or incorrect hyperparameters (for example too high learning rate) accuracy went down.
Also important information, that small networks spend few minutes to train 15 epochs, but big networks need almost hour on the same amount of epochs. I did not collect time on this examples.

Manually looked through the network mistakes I realized that I get about 65% of false positive mistakes for second names. It can mean that network choose prefer to choose second name class every time when it is "not sure". Next I checked is this dataset balanced, and I realized that dataset contains 79% of second names. That means that even if network will chose second name every time with such unbalanced dataset, it will be 79% accuracy.
Correct accuracy of network 5 with balanced test set was: 0.7005 with 90% of false positive mistakes for second names.
As result of the first step I realized, that it is possible to train neural network on such data. But, to continue, I need to solve some problems.
Result problems of the first step:

1. I need to visualize results of each training step, to understand hyperparameters better
2. I need more structured approach to choose hyperparameters
3. I need estimate networks by types (like another hyperparameter)
4. To estimate network power ("size"), I need to check amount of network training parameters, not amount of neurons
5. Real datasets can be not balanced, test and train set should be balanced
6. Different models need different amount of epochs to training, even with the same training algorithm. So to choose best models I need train each network on different amount of epochs.
7. Training time is very important parameter. Small network can get lower accuracy, but trained very fast.
8. It is better to save network parameters/weights after each epoch, In this case if network accuracy will go down (because of overfitting or incorrect learning parameters) it will be easy to restore the best model. Restored model can be trained more with another parameters.

7.3 Second step

At this step I used two real datasets: first/second names and job titles, in total three classes. Total amount of samples was 291011. Also I made my experiments more strict, and I was working with problems from the first step.
I divided my program on three modules: data preparation, network design and network training.
Data preparation module made vectorization, divided data on training set and test set and balanced data. Solve problem 5 from previous step.
For balanced data I was trying three approaches:

- Reduce the number of examples in a larger class
- Repeat small class to make it larger (maybe with noise)
- Add class weights in training algorithm, to make smaller classes has more influence on loss function

According my experience, second approach has a little better convergence but I did not try it a lot, because duplicates do not change amount of new training samples, but increase time for each epoch of training. Also second approach need to choose what samples should be repeated, if you want to add only part of the samples.
On this step I used only class weights to balance data. After data was prepared, all networks was trained on the same training sets and was tested on the same test sets. Also In this module I check and remove duplicates (it was only one duplicate in names).
Network design module described different neural networks. Each network design module describe only one type of networks, that solve problem 3 from previous step. And now I am going from small networks to big networks (problem 4), that is my first solution for choosing hyperparameters more strictly (problem 2)
Network training module trained networks. This module save training statistics, as: training accuracy and training loss by batches and by epochs, testing accuracy and testing loss by epochs, training time and percentiles of neural network weights by epochs[lectures: Tensorflow and deep learning - without a PhD]. Such statistics give more information about changes in neural network on each step of training. Also now I can stop network on any step if I see that training don't give me correct results, or if I want to change some training parameters manually.
This module solve problems 1,6,7,8, from previous step.
As tested network types on this step, I choose simple recurrent networks, LSTM, GRU, bidirectional recurrent networks, convolutional networks with one window size and convolutional networks with different window sizes. All networks was trained about 30 epochs, or if last 5-7 epochs accuracy on test

set was not improved, I stopped training earlier. All models was trained with different training algorithms: Adam, Nadam (Adam with Nesterow momentum), RMSprop, SGD. Different learning rates, different momentums, learning rate annealing (with different decay coeffeicients) and gradient clipping in some models. Total numbers of trainable parameters in different models was from 2000 parameters to 1 640 771 parameters. Normally in this work I was guided by the rule that amount of parameters should be less than amount of samples (291011). But also I was tied some bigger models. I would like to note from myself that really important in this rule, in my opinion, is not the number of samples but the total number of features in data. More precisely, one might say complexity. So if bigger amount of parameters give beast accuracy on test set (network newer seen this before), it can be better solution, but it can be a lot minor features that can reduce the quality of the network on data from other sets.
To solve overfitting I used weight regularization, batch normalization, dropout and gaussian noise.
Also on this step I was tried my own learning rate reducer module, that reduce learning rate only if accuracy lowered. This approach should be studied deeper, because of it can cause training algorithm can stuck in local minima. In the end, I abandoned this approach and used the standard learning rate decay implemented in the Keras framework [133].

$$Lr_i = Lr_{i-1}\left(\frac{1}{1+ci}\right)$$

Where Lr_i and Lr_{i-1} are learning rate for updates i and i-1, i is number of update weights and c is annealing coefficient.
It is important to remember that using the learning rate decay imposes restrictions on the learning time, a high decay quickly reduces the learning rate [Figure 7-4].

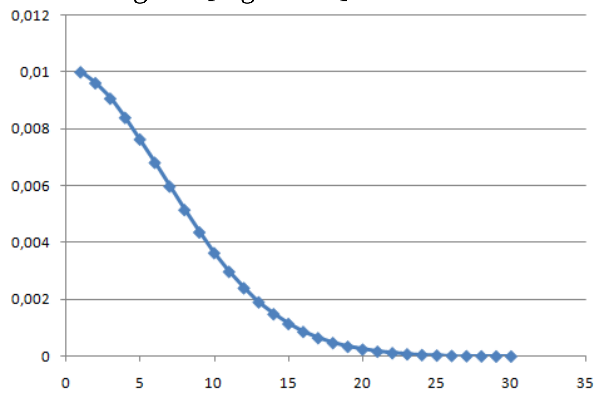


Figure 7-4 Learning rate decay by steps, start rate=0.01, decay=0.02.

As activation function improvements, I used sigmoid, ReLU and Leaky ReLU in simple neurons and recurrent units.
Unfortunately it is impossible to check all possible hyperparameters in the framework of this work. Each model with new parameters was trained from few hours till day. So I was increasing complexity of networks, but sometimes several hyperparameters changed at once, and some changes were spontaneous.
Trained models:

LSTM

I started this step with LSTM networks, as most often referred to in articles with natural language processing problems. On this step I was trying 16 long short term memory models. I started from small models (about 10 units) and finished with models that contains 400 units LSTM layer and three fully connected layers. Also I was trying models with two LSTM layers (100 units in each of them) that output data to three fully connected layers. Biggest model has 749443 trainable parameters.
From the beginning, according my theory part, I choose Nadam as my main training algorithm and reduced learning rate with increasing model complexity, but with LSTM layers with more than 200 units it is become necessary to use also gradient clipping. Figure [Figure 7-6] show loss by batches in first epoch and figure [Figure 7-6] by epochs, with too high learning rate Nadam without gradient clipping (with such high learning rate network cannot find a minimum, constantly jumping over it).

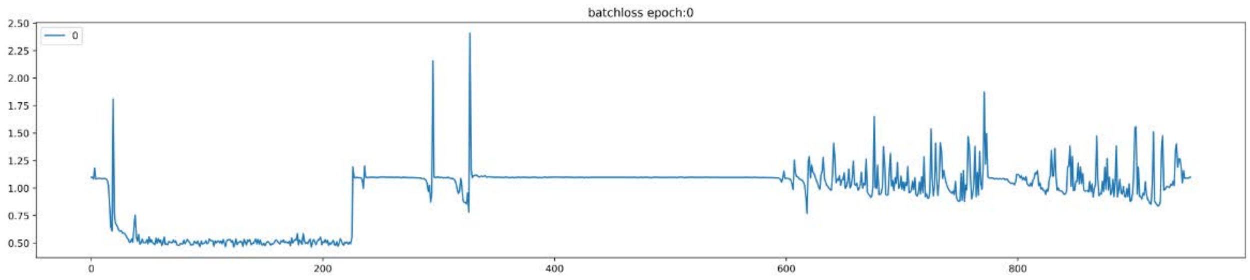


Figure 7-5 Training loss by batches for the first epoch, with too high learning rate for this model.

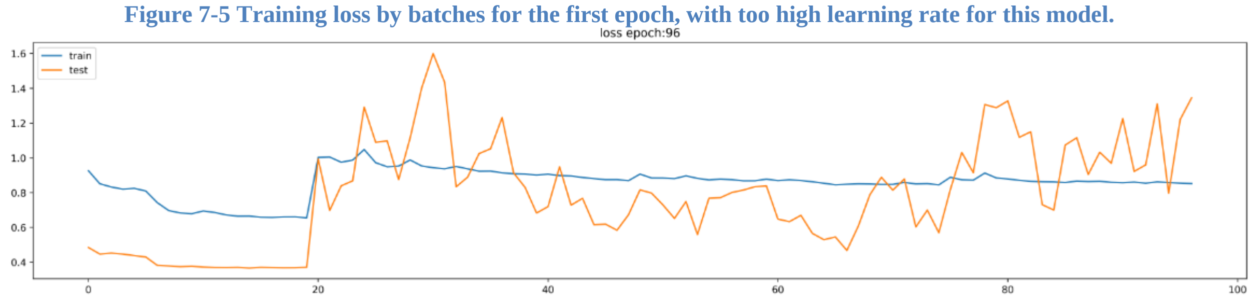


Figure 7-6 Training and test loss by epochs, with too high learning rate for this model.

Best test set accuracy I get with big model that contain 400 LSTM units in one layer that feed three layer feed forward fully connected layers 64x32x3 neurons- total amount of trainable parameters 749443, and accuracy 0.85833. To solve overfitting in this model I used gaussian noise and dropout.
Same model without fully connected layers has 722,803 trainable parameters and get accuracy 0.85033.
But model with 200 LSTM units in one layer has 201,403 trainable parameters and got test accuracy 0.8476 on 12 epoch, ten start overfitttng.
During this test I realized that It is not easy to overfit LSTM models. In small models training and test accuracy stops on some level [Figure 7-7]. In big models training accuracy continues to grow, but test accuracy lowered very slow, and methods against ovefritting straighten the line [Figure 7-8].

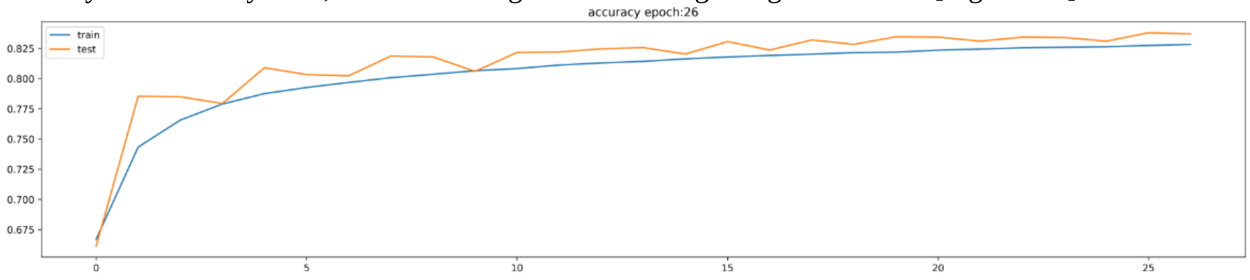


Figure 7-7 Behavior of one layer 50 LSTM units model with dropout (test accuracy is better, because of dropout)

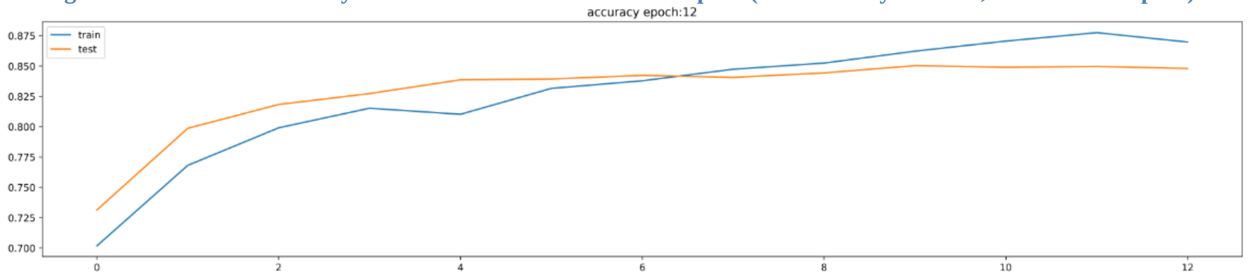


Figure 7-8 Behavior of one layer 400 LSTM units model with dropout and gaussian noise (test accuracy is better, because of dropout)

Simple recurrent networks

In this research I used seven simple recurrent networks, with amount of neurons form 50 in one layer (Trainable params: 5,203), to two recurrent layers network with 300 neurons in each layer, that feed feedforward layers (Trainable params: 295,331). It was surprise for me, but small network with 50 recurrent neurons and total 5,203 trainable parameters got best test accuracy from such type networks: 0.818. But all other networks got accuracy higher than 0.81. So all simple recurrent networks get almost the same accuracy from 0.81 to 0.82, and I decide to not use this type of networks.
(in all networks I fought with overfitting if this was necessary)

GRU

On this step I used six different gated recurrent unit networks. Because GRU is advanced and simplified LSTM, I was waiting very similar results and was trying only interesting models from LSTM. Biggest model has 337,443 trainable parameters, and got 0.848 accuracy on test set.
In general, I want to note the more predictable behavior of the GRU, It is overfitted faster and stronger [Figure 7-9] [Figure 7-10]. I compared same models with GRU and LSTM models and according my results GRU is one third faster convergence by time and by amount of epochs. Also GRU unit has smaller amount of trainable parameters than LSTM unit.

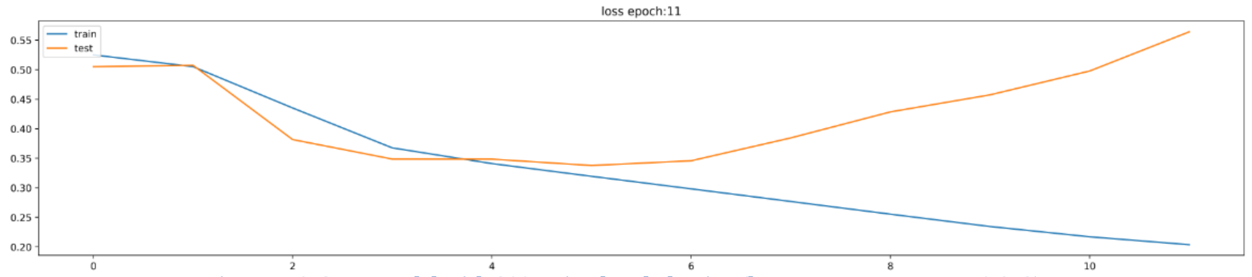


Figure 7-9 GRU model with 200 units, loss behavior (best test accuracy was 0.843)

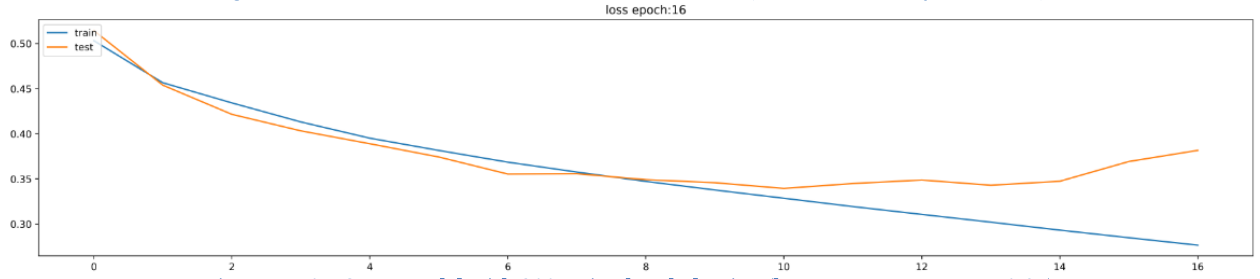


Figure 7-10 LSTM model with 200 units, loss behavior (best test accuracy was 0.84)

I consider that strong overtraining is an advantage of GRU, on LSTM charts it is not always clear what happens with the network.

Bidirectional recurrent networks

Here I was trying only two models with 50 bidirectional GRU units (30,453 trainable parameters) and with 50 bidirectional LSTM units(40,553 trainable parameters). And results should be compared with same amount of trainable parameters networks.
Bidirectional LSTM got test accuracy 0.8386, 80 one directional LSTM units (42,163 trainable parameters) got accuracy 0.8357.
Bidirectional GRU got test accuracy 0.8423, 80 one directional GRU units (31,683 trainable parameters) got accuracy 0.833.
According this examples bidirectional recurrent networks has little better accuracy in comparison with the same type and same size by parameters one directional networks.

Convolutional networks with one window size

In this examples I was using convolutional and multilayer convolutional networks with window size 3 characters. Smallest network has one convolutional layer with 64 filters, filter length was equal 3, total trainable parameters: 18,883. Biggest network has two conwolutional layers, that feeds feedforvard fully connected layers 64x32x3, 106,535 trainable parameters.
Convolutional networks are 2-3 times faster (by time), than recurrent on GPU. But they has bigger problems with overfitting.
In this examples small network has best test accuracy 0.84, all other networks has accuracy higher than 0.83, but was overfitted.
I did not spend too much time on this type of networks, because multylayer convolutional networks sounds unreasonable for me with such small samples (30-40 letters), and one layer convolutional network with one window size should not be better than networks with different window sizes.
But small convolutional network is very good choice to quickly assess data.

Convolutional networks with different window sizes

I used six networks from this type, all networks has the same windows sizes: 2,3,4,5 and four layers feed forward network in the end. All differences was amount filters, amount of neurons in feed forward networks and additional improvements as dropout and regularizations.
Here small and big models can has almost the same amount of trainable parameters because of different pooling lengths. Small model with 16 filters for each size, without pooling has 109,267 trainable parameters, big model with 128 filters for each size and pooling length 2 has 1 640 771 trainable

parameters. Best accuracy on this models was 0,855. Same128 filters for each size and pooling length 16 has 264,515 trainable parameters. Best accuracy on this models was 0.8483.
Model with 1 640 771 trainable parameters has size 18.8 Mb, total data size 3.34 Mb, this result is more like overfitted model for this data set and maybe on another datasets such network will have lover accuracy.
Convolutional networks can be easily overfitted after only one epoch and overfitting become biggest problem. Even if it is possible to solve overfitting, model still can not increase accuracy [Figure 7-11][Figure 7-12].

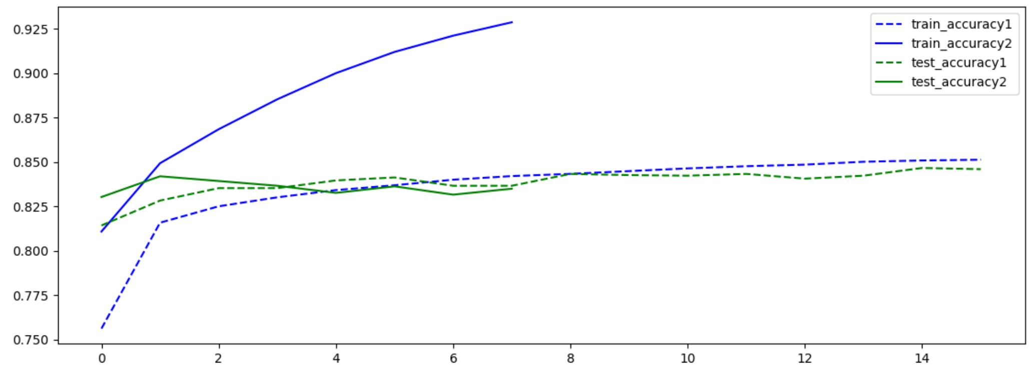


Figure 7-11 Model (1 640 771 trainable parameters) accuracy, solid lines for overfitted model (best accuracy 0.842 after two epochs), dashed lines for model with dropout (best accuracy 0,855 after 14 epochs)

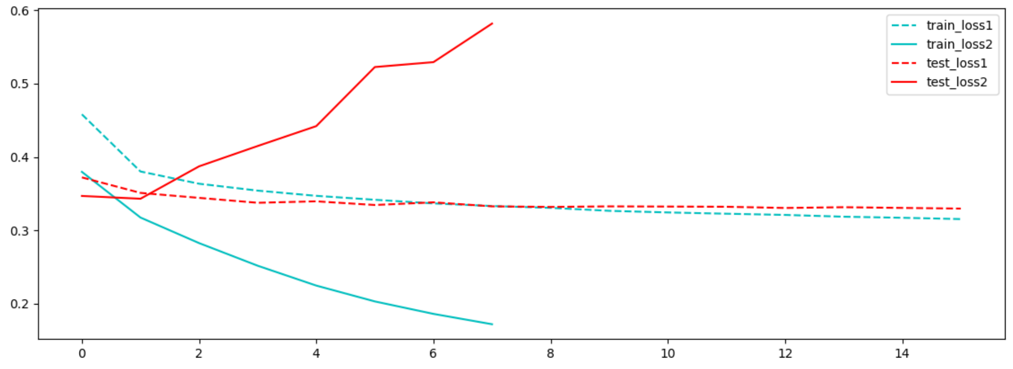


Figure 7-12 Model (1 640 771 trainable parameters) loss, solid lines for overfitted model (best accuracy 0.842 after two epochs), dashed lines for model with dropout (best accuracy 0,855 after 14 epochs)

Convolutional networks with different window sizes gave me best accuracy on this step.

Training estimation

The main problem of this step was to appear, what happens to the network after each batch and epoch. Without such an understanding, it is difficult to achieve good results. To understand changes in the network, I used graphics training loss by batches, training loss and accuracy by epochs, test loss and accuracy by epochs [Figure 7-13].

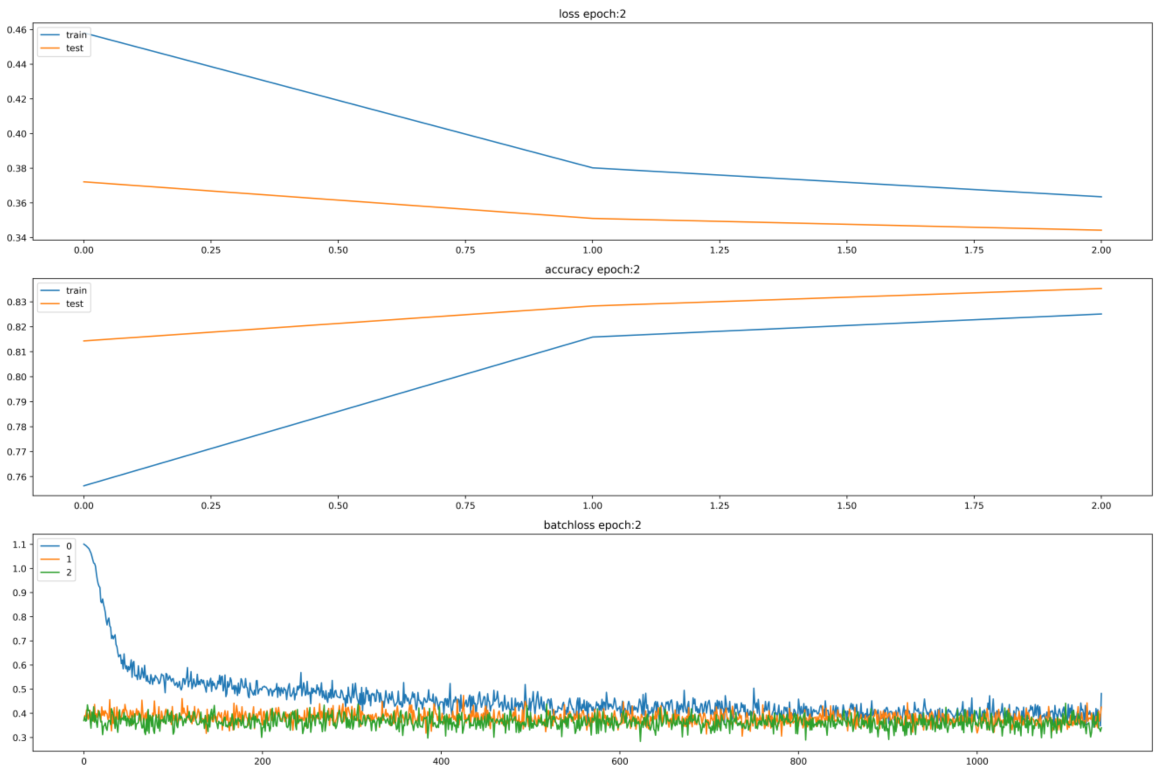


Figure 7-13 Grafical training representation.

Training loss by batches can help to quickly understand if the learning rate is too large, without waiting for the end of several eras. The behavior of the loss and accuracy can also talk about too much speed of training, and also about the overfitting, or can signal that the training has stopped at some minima. Ending of changes in accuracy and loss can also indicate that the training speed is reduced to almost zero, thanks to annealing, and it is possible to reduce the annealing rate.

Loss and accuracy are not equivalent and are built according to different equations, and although the goal of the learning function is to reduce the loss function, the network requirement is high accuracy on the test data.

Another way to visualize network learning is to plot the percentile scales for each layer [Figure 7-14]. For the first time I saw this technique in lectures [lectures: Tensorflow and deep learning - without a PhD]. In Tensorflow, Tensorboard was developed for such visualization, but at this stage I was unable to use this compiler in my work because of an error in source codes. Tensorboard used too much memory. Later with the help of the community I corrected the error in Tensorboard and used its visualization. To visualize the learning at this step, I used my weight percentile graphs in each layer.

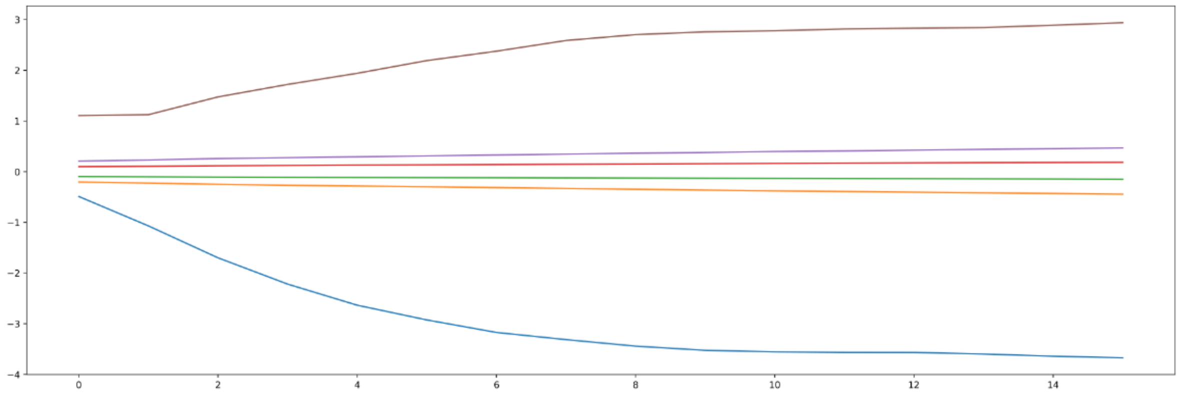


Figure 7-14 My own percentiles chart.

To see the quality of classification by classes I used false positive and false negative errors charts[Figure 7-15].

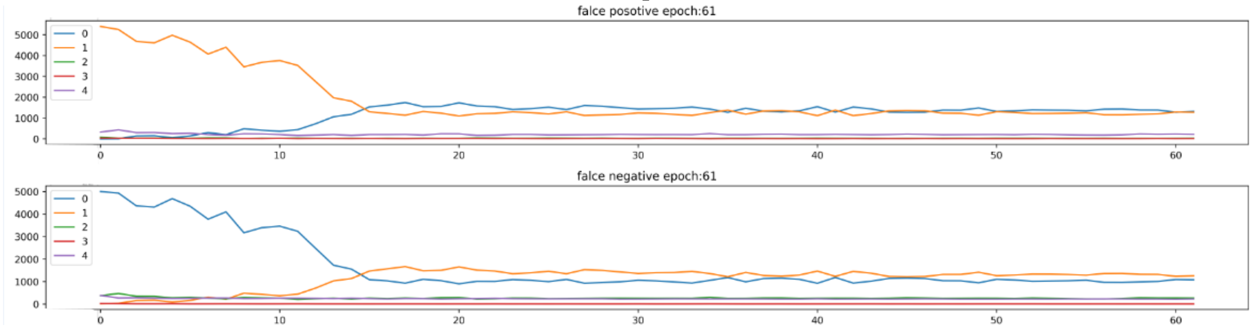


Figure 7-15 False positive and false negative errors for 5 classes, on test set visualization by epochs.

Second step results

As result of second step I choose few hiperparameters:

1. Three types of Networks that I will use in the feature work: GRU, bidirectional GRU and convolutional networks with different window sizes.
2. Batch size 500-1000
3. Adam as training algorithm
4. Keras learning rate decay
5. Simple ReLU in hidden layers
6. Important notes:
7. Complicated models should be trained with smaller learning rate
8. Gradient clipping can help if learning rate selected not correctly

From my point of view, although models with a number of parameters significantly exceeding the dataset, give the best results on the test sample (data that the network has never seen), however this result is more like overfitted model for this data set and maybe on another datasets such network will have lover accuracy. Unfortunately I did not found answer on this question, there are several studies devoted to the search for the optimal number of network parameters, but no one gives an exact answer [28][29][30]. A small number of weights increases the generalizing capacity of the network and lowers the accuracy[31].
In connection with the advice of the forums, the limitations of my computer and the data requirements, I decided not to exceed the number of parameters higher than the number of training examples.

7.4 Third step

In the previous step, the main attention was paid to the choice of the network, in this step attention is paid to the training of the selected networks. The purpose of this step is a deeper understanding of network behavior and achieving maximum accuracy. Great progress in understanding the network was the ability to use the tensorboard to visualize the histograms and percentile weights in each layer, after fixing the error in the source code for TensorFlow. I also added false positive and false negative errors to the learning log.
At this step I used same datasets: first/second names and job titles, in total three classes. Total amount of samples was 291011. So I tried to avoid networks with a number of parameters over 300,000. The data preparation module has been changed in this step and now only prepares the data for the network. Decisions about data balancing and data sharing on the test and training set are taken by the network design module - this makes it easier to send new data to the network.
In the previous step, I worked with samples of 50 characters length, in this step I used samples of 30 characters in length. This transition is due to the long of most work positions (the short position of the CEO is recognized by the neural network as the first name). On the contrary, the names do not exceed this length. Unfortunately, in this case, I got work positions duplicates: "3rd Party Logistics Coordinator", "3rd Party Logistics Coordinator Ukraine", "3rd Party Logistics Coordinator-Indonesi" - become the same.
As a result of changing the length of samples decreases the amount of trainable parameters in the same architecture networks and reduce accuracy.
A quick check by the convolutional network showed, that for samples with a length of 30 characters, network of similar architecture (but with fewer trainable parameters 176 963) reached an accuracy of 0.8452 at the 46th epoch, after which it began to be slightly overfited. The same network with 50

characters samples (264.515 trainable parameters) reached an accuracy of 0.8483 at the 18th epoch [Figure 7-16]. Direct comparison of accuracy between this and previous step is not correct.

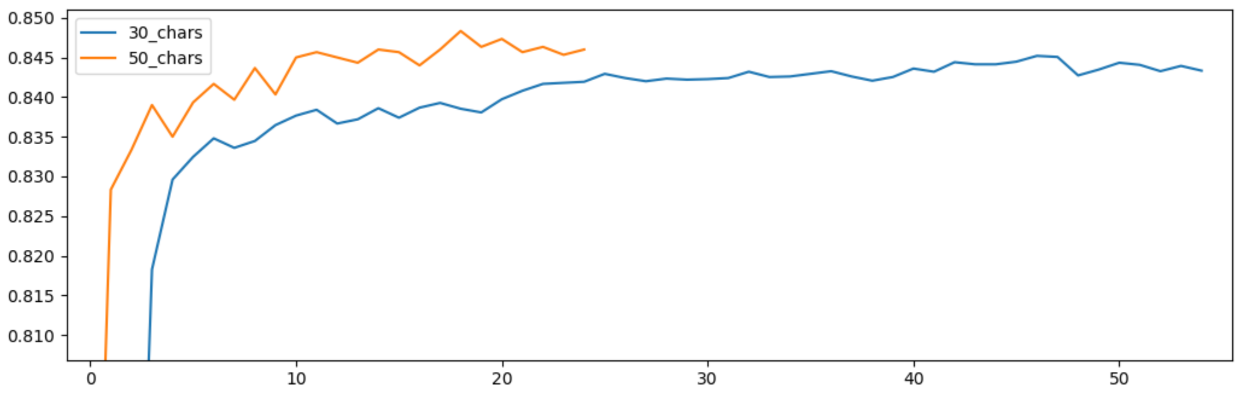


Figure 7-16 Test accuracy on the same architecture network with length of samples 30 and 50.

In addition, at this step, for balancing, I use an incremental coefficient for the first name class, and I do not increase the importance of the job titles class. As it seemed to me from the past experience, a big difference between job titles and names prevents the network from seeing the difference between first and second names, and with a complete balancing of the dataset, a network with a small number of parameters the first few epochs can not distinguish names.

This is the last step in the design of the network and at this stage it was desirable to get the maximum quality of classification on the data, due to the capabilities of the network itself, and not at the expense of long examples.

At this step, I worked only with GRU, bidirectional GRU and convolutional networks with different window sizes.

GRU

In this step, I tried 32 GRU network models ranging from small networks and gradually increasing the complexity of the network. If necessary, I added methods of combating retraining. Here I will show only the most interesting networks from my point of view. During the creation of networks, tensorboard was actively used. To complicate the GRU model, you can add the number of layers, the number of neurons in the layer and adding feedforward networks of different complexity to the end of the network.

Increase in the number of layers. To estimate main influence of increasing amount of layers I was working with 9 simple networks. On this step I did not used any algorithms against overfitting, and training algorithm was Adam with default parameters (lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0 [132]).

All models with best accuracy described on figures [Figure 7-17][Figure 7-18][Figure 7-19].

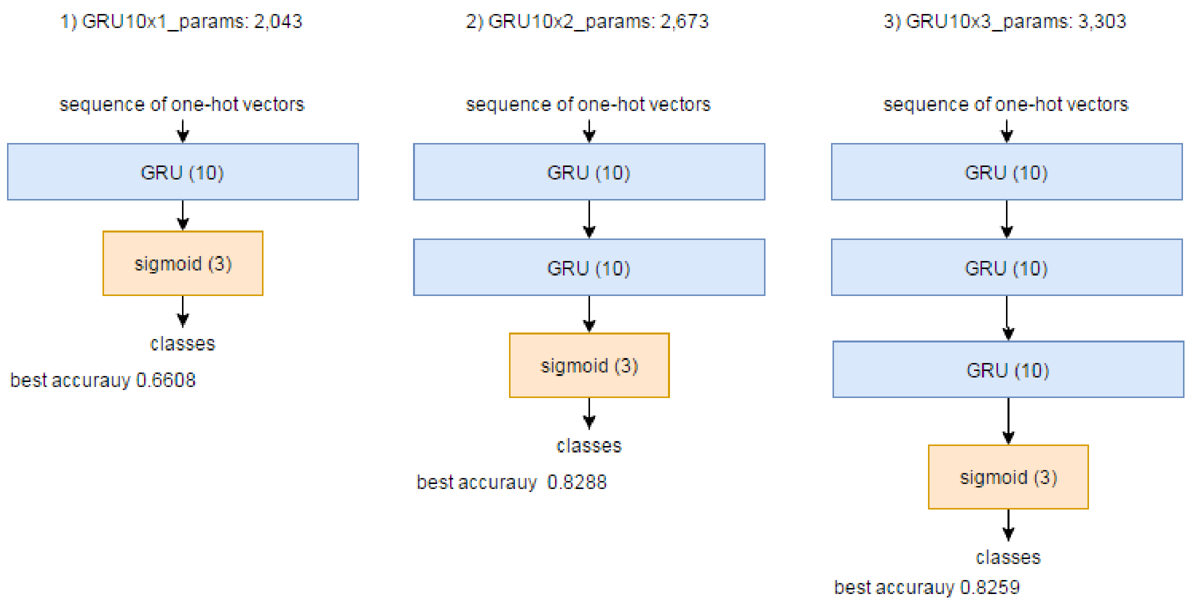


Figure 7-17 Architectures of GRU models with different amount of layers with best accuracy, the architecture of the models is briefly described in the title with the total number of parameters, the number of units/neurons is indicated in parentheses.

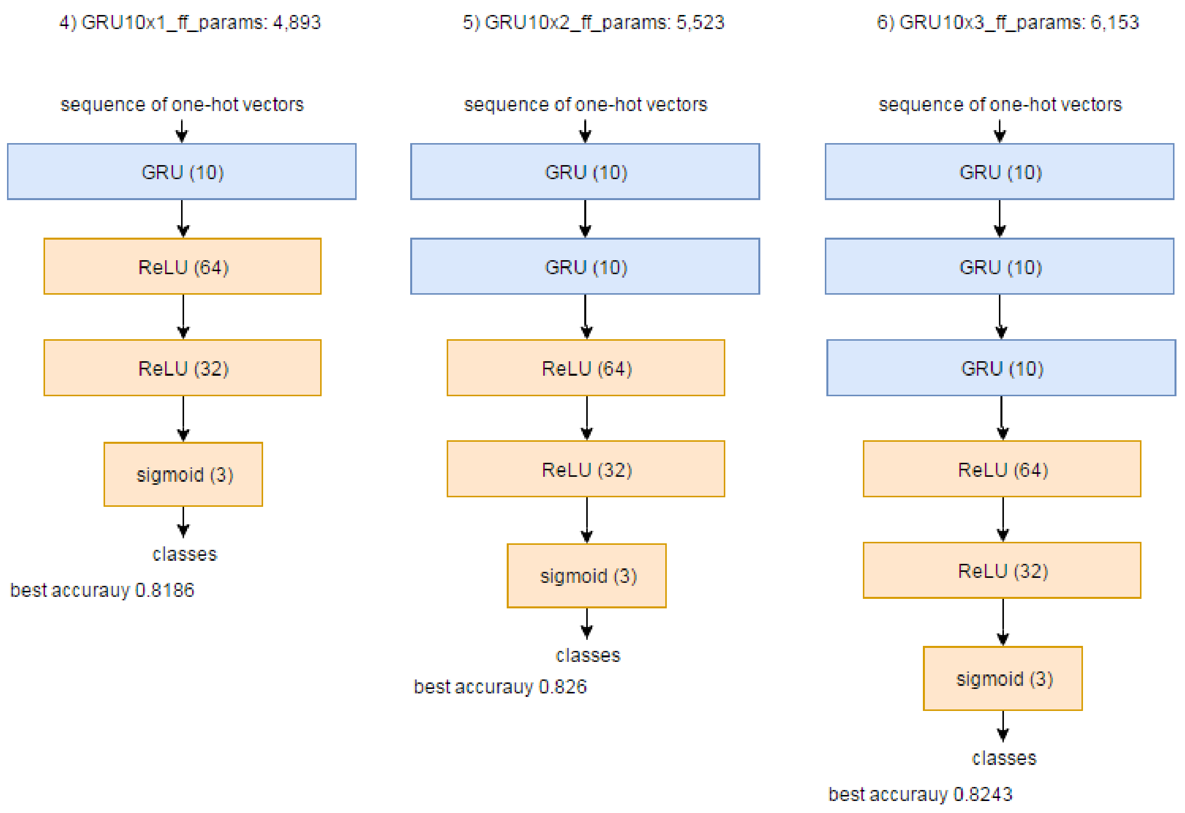

4) GRU10x1_ff_params: 4,893
5) GRU10x2_ff_params: 5,523
6) GRU10x3_ff_params: 6,153
sequence of one-hot vectors
GRU (10)
ReLU (64)
ReLU (32)
sigmoid (3)
classes
best accurauy 0.8186
best accurauy 0.826
best accurauy 0.8243

Figure 7-18 Architectures of GRU models with different amount of layers with best accuracy, the architecture of the models is briefly described in the title with the total number of parameters, the number of units/neurons is indicated in parentheses.

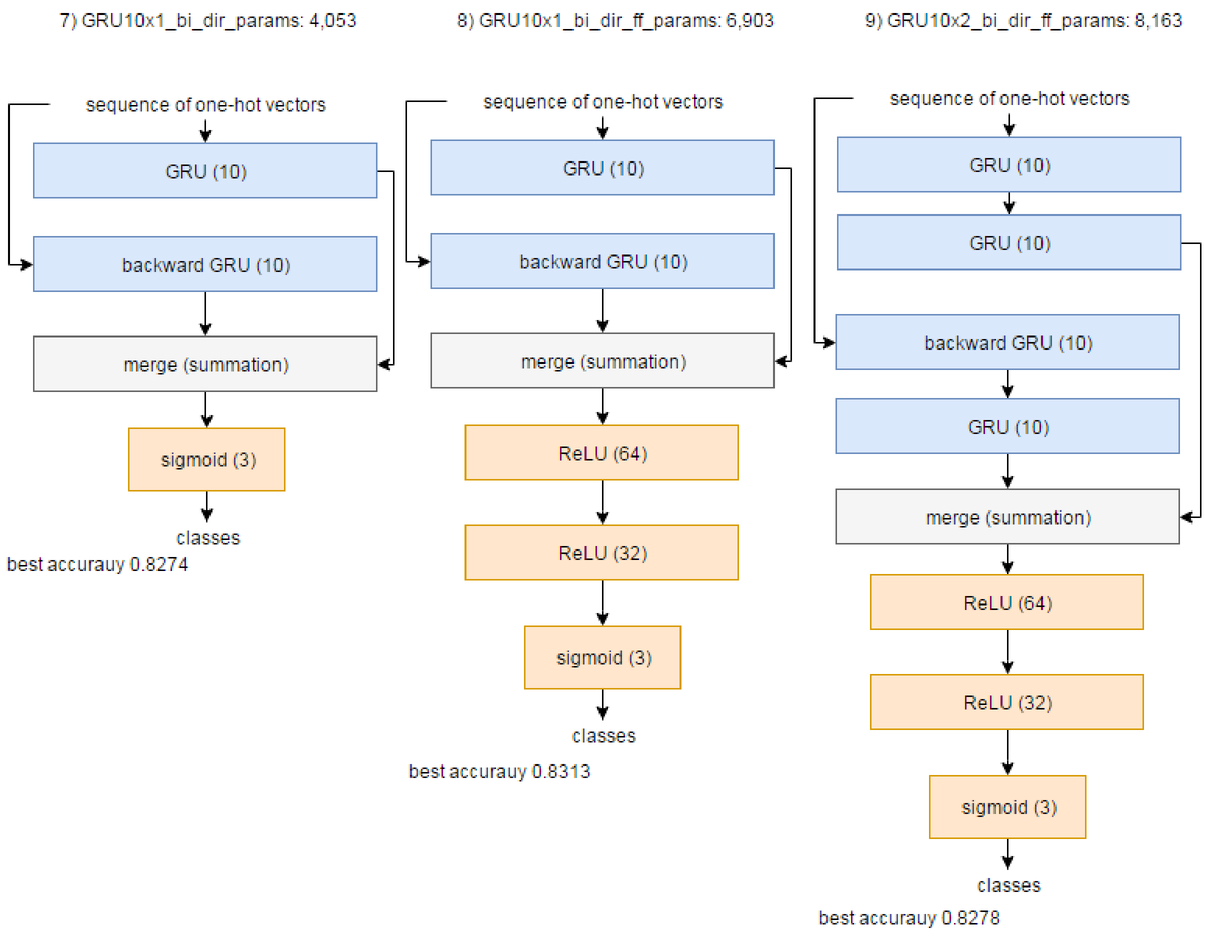


Figure 7-19 Architectures of GRU models with different amount of layers with best accuracy, the architecture of the models is briefly described in the title with the total number of parameters, the number of units/neurons is indicated in parentheses.

The accuracy of training models by epochs are shown in the graph [Figure 7-20].The noise on the graphs probably indicates the speed of training is too high, but the overall impact of the network architectures is.

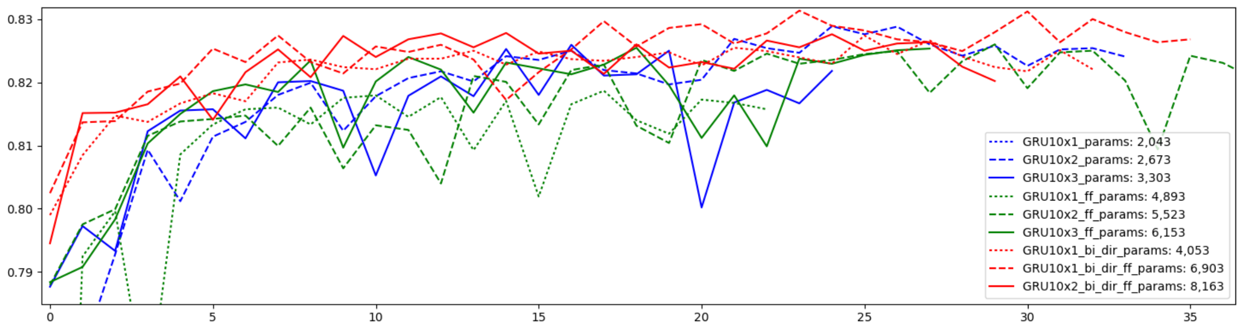


Figure 7-20 Models test accuracy by epochs.

As can be seen from the results of model training, the smallest model has an error of 0.6608, for 10 epochs, after checking for errors in classes I was convinced that this model could not learn to distinguish between first and second names even with not complete balancing.

It's also interesting to note that bidirectional networks charts has smaller noise and high learning rate not big problem for them.

The main trend is confirmed on the accuracy charts[Figure 7-20]: an increase in the number of parameters leads to an increase in accuracy. But we also see the influence of architecture, so the network "GRU10x3_ff_params: 6,153" with a large number of parameters, in general, loses in classification to "GRU10x1_bi_dir_params: 4,053".

The second conclusion is: increasng number of recurrent layers more than 2-3 does not give a perceptible gain in the quality of the network. Second layer in bidirectional network did not give me better performance because of owerfitting.

The third conclusion, for such small recurrent networks, the feed forward network 64x32x3 gives a slight increase in the quality of classification, with a significant increase in the number of parameters.

Increase in the number of neurons.

Another approach to increase neural network complexity is to increase number of neurons in layers. To estimate main influence of amount neurons, I was working with 7 simple networks. On this step I did not used any algorithms against overfitting, and training algorithm was Adam with default parameters (lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0 [132]).

For this experiments I used first model architecture from figure[Figure 7-17] with different amount of URU units.

Best accuracy for models by units GRU10: 0.662, GRU20: 0.66, GRU30: 0.8165, GRU40: 0.8378, GRU60: 0.8412, GRU80: 0.8442, GRU100: 0.837. Figure [Figure 7-21] show models test accuracy by epochs. As we see 20 gated recurrent units in one layer with 4,683 trainable parameters can not get difference between first and second names too. Even 60 gated recurrent units has problems with a division of names, first three epochs. 100 gated recurrent units got difference between names even after first epoch.

Also I tested 80 and 100 gated recurrent units on data without balancing and 80 units solve names difference only after 4 epochs, 100 units in this case did not get difference between names in first epoch.

So maybe 20 units with better balancing can get better score.

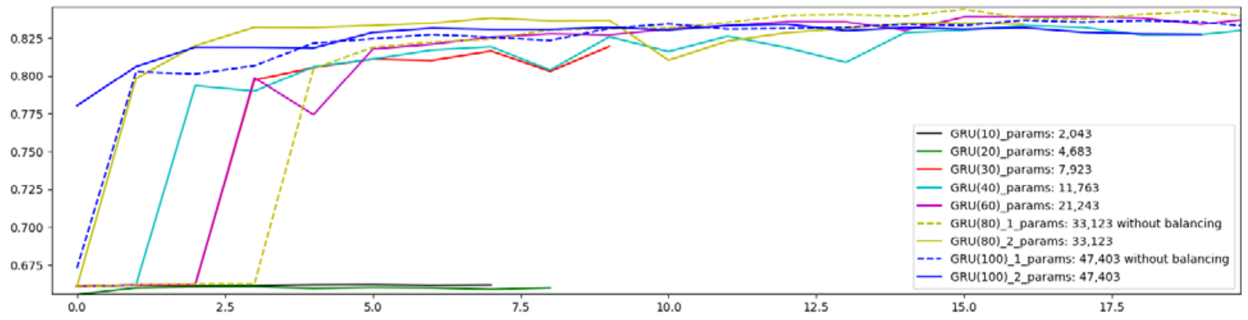


Figure 7-21 Models test accuracy by epochs.

So one layer model with 20 neurons and 4,683 trainable parameters got accuracy 0.66 and can not divide names classes, but two layers network with same 20 neurons with 2,673 trainable parameters got accuracy 0.8288. Difference between amount of trainable parameters, because of input and output layers has double difference in amount of synapses.

As result of numbers of layers and numbers of neurons testing I saw that GRU behaves very much like conventional feed forward networks and amount of layers can be much more important (like with HEX problem in theory part) than amount of neurons, but in case complexity of my problem two layers are enough.

From first two examples I choose two layers GRU with feed forward layers (GRU10x2_ff) [Figure 7-18] and two layers bidirectional network with feed forward layers (GRU10x2_bi_dir_ff) [Figure 7-19] [Figure 7-20] to improve them.

One directional GRU

On this stage I choose two layers GRU with feed forward layers and trying to get best accuracy from this network, remaining with the maximum number of trainable parameters below 300,000. During this work I was trying 20 different networks with different solutions against overfitting and gradually reducing start learning rate from 0.01 to 0.0002 (Adam with annealing) with networks grows. Best network from this type got accuracy 0.8525 after 319 epochs. First and last networks shown on figure [Figure 7-22]

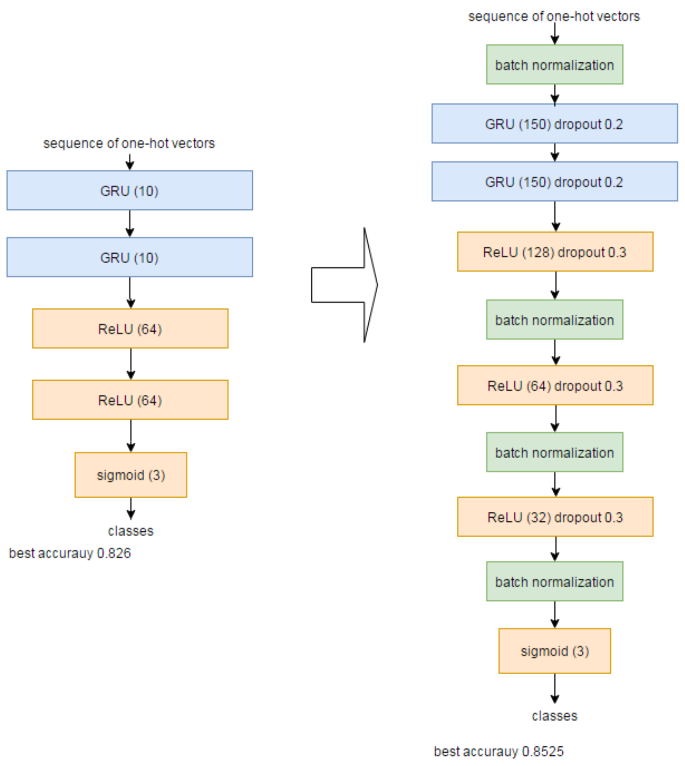


Figure 7-22 Started network and last network of this type. Dropout in GRU for all weights.

The process of network growth can be observed on eight networks whose accuracy graphs are shown[Figure 7-23].

Development process:

1. Picked up the selected network with started learning rate 0.01 (GRU10x2)
2. Increased the number of parameters and got overfitting (GRU40x2)
3. Solve overfitting (GRU40x2)
4. Increase amount of parameters without methods against overfitting and got overfitting (GRU60x2)
5. Solve overfitting (GRU60x2)
6. Increase amount of parameters with previous methods against overfitting and got overfitting (GRU90x2)
7. Increase amount of parameters with previous methods against overfitting and reduced started learning rate to 0.001 (GRU120x2)
8. Increase amount of parameters with previous methods against overfitting and reduced started learning rate to 0.0002 wits same decay =0.0002 (GRU150x2)

Here I did not explain solutions for overfitting on each step, because I was trying different methods, my last solution shown on figure [Figure 7-22]

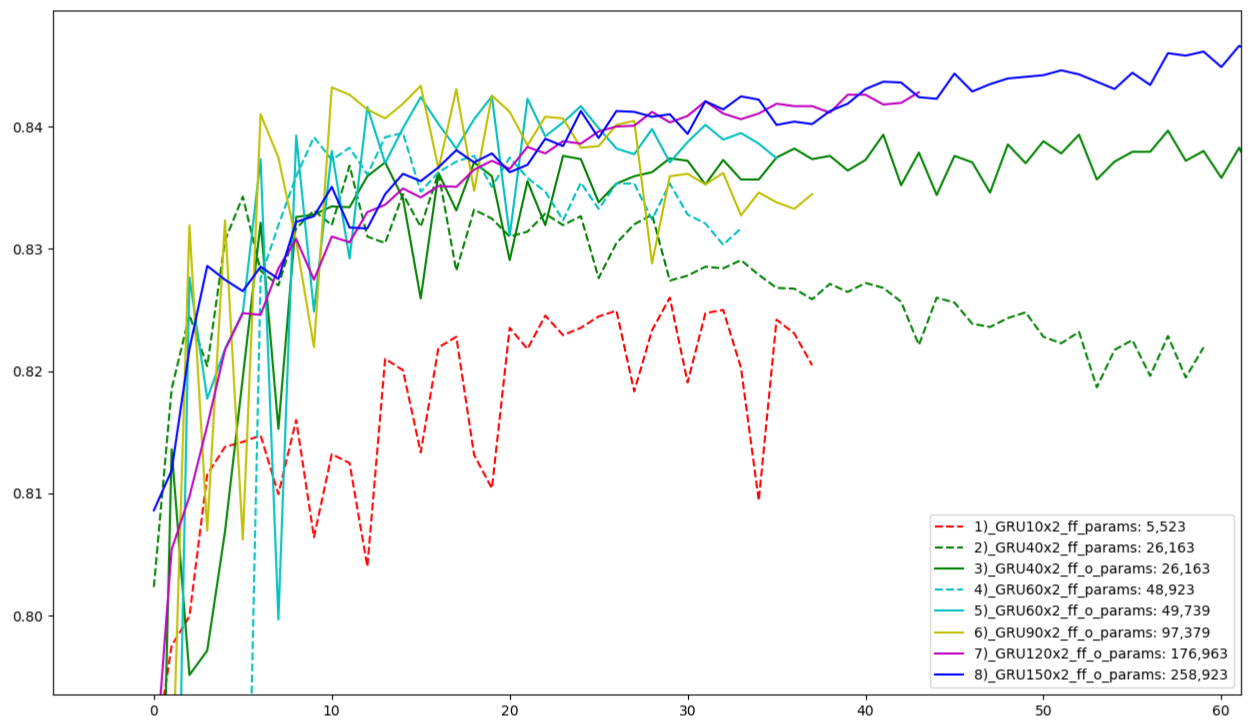


Figure 7-23 Test accuracy charts, with model improvement. Dashed chars explain models without methods against overfitting.

Last model was trained 324 epochs, that spend 8 hours and got best accuracy 0.852533 on 319 epoch, whole test accuracy chart[Figure 7-24].

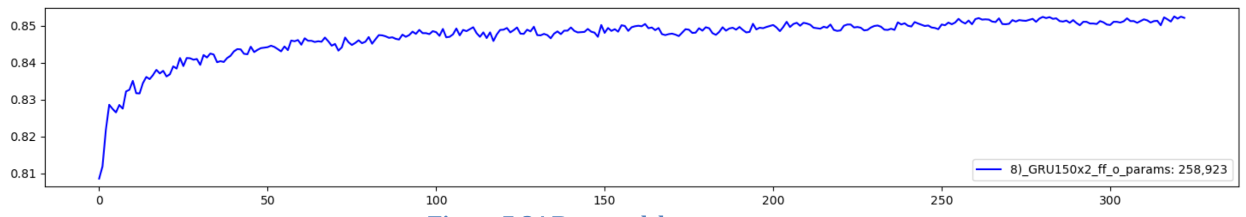


Figure 7-24 Best model test accuracy.

During this step I was using weights histograms, histograms give a lot of information about what is happening on the network. In my opinion most interesting histograms are output from recurrent network - that show GRU behavior and weights for first layer of feedforward network- that show (as I interpret it) feed forward response to recurrent part output.

Tensorboard screenshots histograms for same 8 models and my brief interpretation are shown [Figure 7-25][Figure 7-26][Figure 7-27][Figure 7-28][Figure 7-29][Figure 7-30][Figure 7-31] (histograms saved by time). *Unfortunately I have not seen any description of how such graphics are interpreted by other developers of networks and the interpretation offered here is purely my opinion. Also all the work with the charts is only visual, I did not evaluate skewness or kurtosis or any other properties mathematically.*

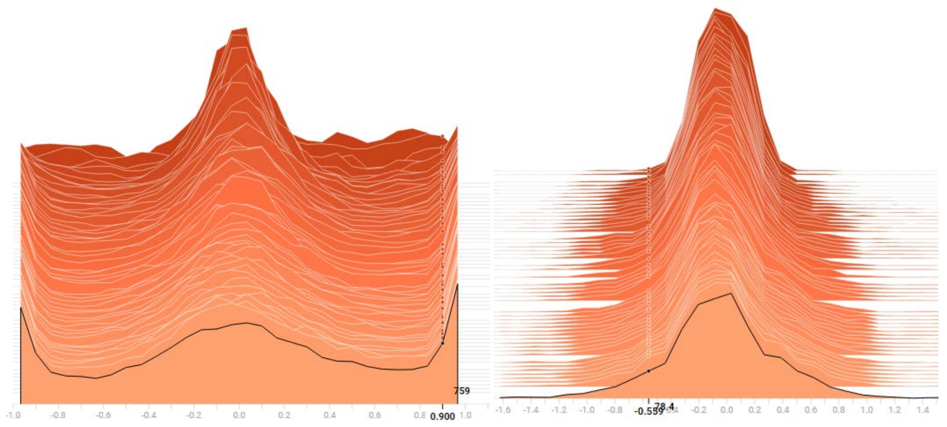


Figure 7-25 Model 2, GRU 40 units, 2 layers + FFNN 64x32x3 neurons by layer, without algorithms against overfitting. Histogram of output from recurrent layers (left); histogram of weights from first FFNN layer.

According figure [Figure 7-23] this model was overfitterd. GRU output distriduted mostly near -1,1 and 0, as we know GRU output is Hyperbolic tangent function, bounded from -1 to 1. Such behavior of output in my opinion means that each neuron has chosen interesting features from samples for itself and responds to them with a 1, on the remaining -1 or zero. I want the entire network to work.

Histogram of weights from first FFNN layer looks like slightly changed normal distribution. Weights were initially initialized by a normal distribution and if weights still looks same, that mean that this layer almost nothing learned.

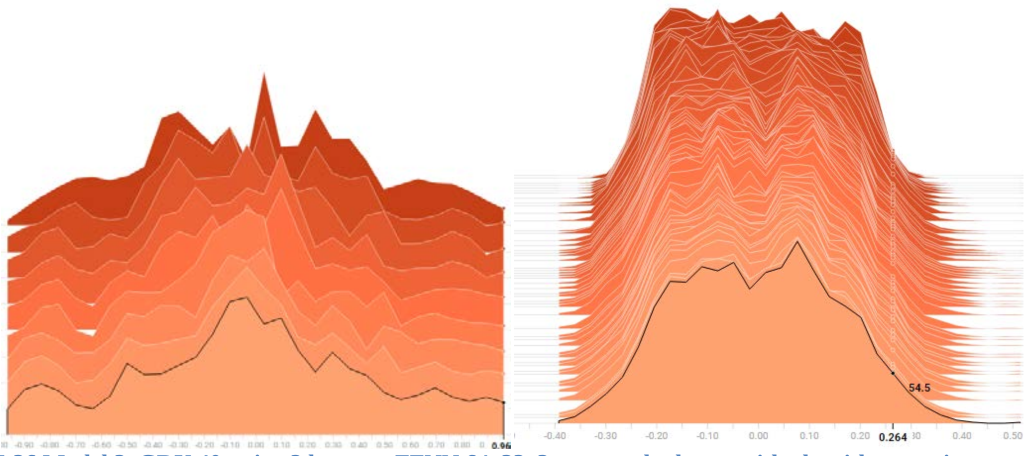


Figure 7-26 Model 3, GRU 40 units, 2 layers + FFNN 64x32x3 neurons by layer, with algorithms against overfitting. Histogram of output from recurrent layers (left); histogram of weights from first FFNN layer.

Overfitting solved [Figure 7-23], GRU output looks for me that network has potential to be trained more, I stopped too earlier.

Weights from first FFNN layer looks more like uniform between -0.2 and 0.2 with mean value near zero, that is looks good.

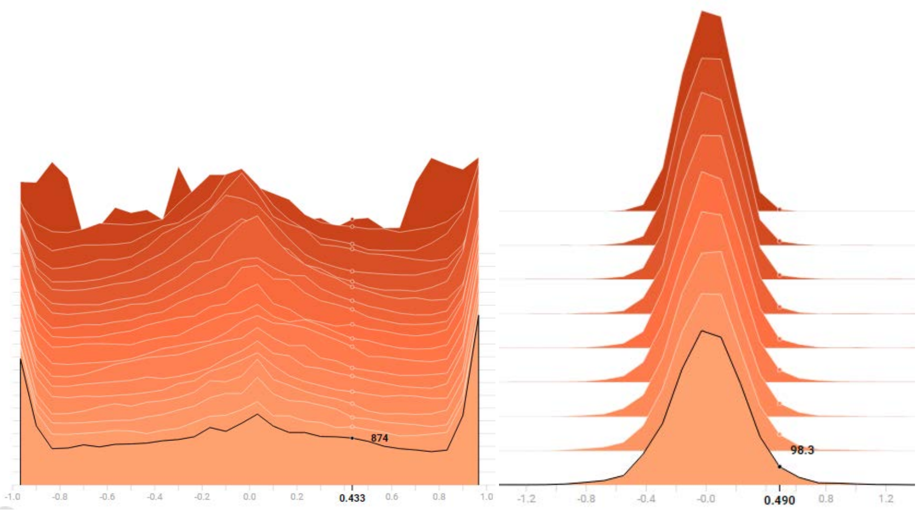


Figure 7-27 Model 4, GRU 60 units, 2 layers + FFNN 64x32x3 neurons by layer, without algorithms against overfitting. Histogram of output from recurrent layers (left); histogram of weights from first FFNN layer.

According figure [Figure 7-23] this model was overfitterd. Here we have exactly the same behaviour like model 2 [Figure 7-25], but overfitting is stronger and GRU output is more divided between -1 and 1. Histogram of weights from first FFNN layer looks like correct normal distribution.

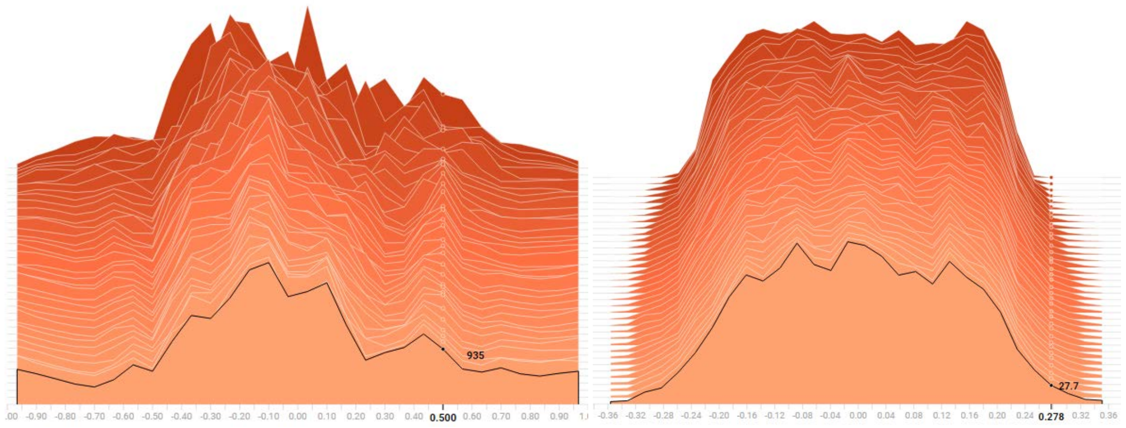


Figure 7-28 Model 5, GRU 60 units, 2 layers + FFNN 64x32x16x3 neurons by layer, with algorithms against overfitting. Histogram of output from recurrent layers (left); histogram of weights from first FFNN layer.

Overfitting solved [Figure 7-23], same as model 3, according GRU output network can be trained more. Weights from first FFNN layer looks even better and approach the uniform distribution, this layer works good.

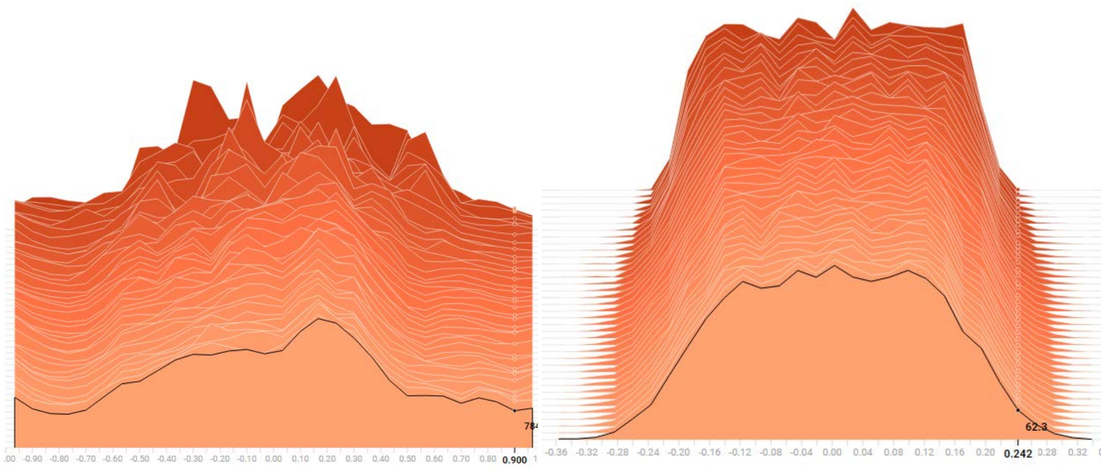


Figure 7-29 Model 6, GRU 90 units, 2 layers + FFNN 64x32x16x3 neurons by layer, with algorithms against overfitting. Histogram of output from recurrent layers (left); histogram of weights from first FFNN layer.

According figure[Figure 7-23], model looks like overfitted but histograms looks like for models 3 and 5, approaching the uniform distribution. For me it is can be because of to high learning rate, that is seen on figure[Figure 7-23] with too high accuracy changes.

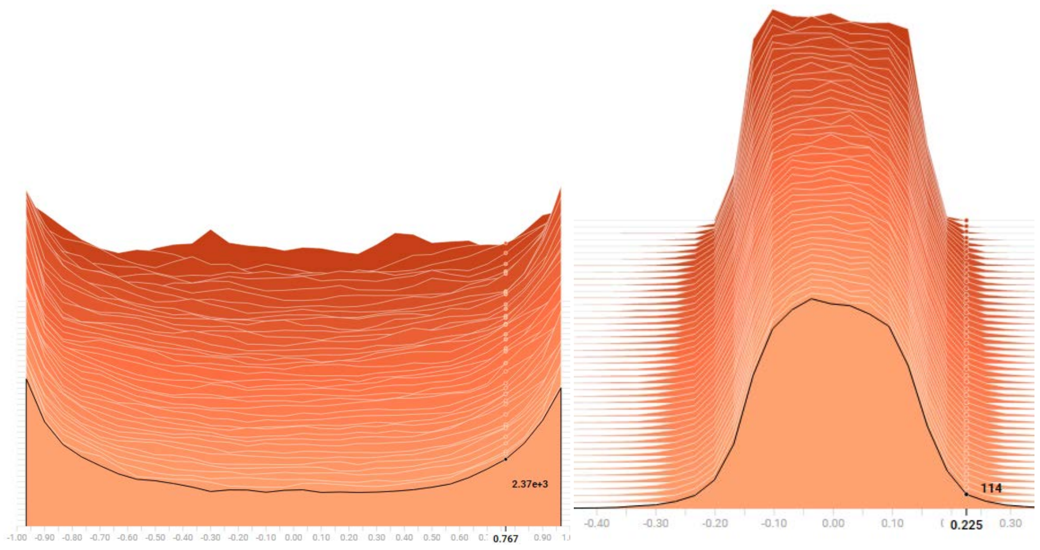


Figure 7-30 Model 7, GRU 120 units, 2 layers + FFNN 64x32x16x3 neurons by layer, with algorithms against overfitting. Histogram of output from recurrent layers (left); histogram of weights from first FFNN layer.

Model did not overfitted by accuracy chars[Figure 7-23], GRU output but it starts looks like overfitted models 2 and 4. I think that it happened because recurrent part of model already starts fit data, but FFNN with high dropout 0.3 still studying. Weights from first FFNN layer looks like layer learned something but not much. An interesting observation is that the FFNN weights at the beginning of the training had a distribution closer to uniform and in the learning process they return to the starting normal distribution - in my opinion that means that in the early stages of learning, fast ReLU functions from the feedforward layer took on the entire complexity of the task, but later the slower hyperbolic tangent and sigmoid functions from GRU were sufficiently trained that the first from the feedforward layer was not needed.

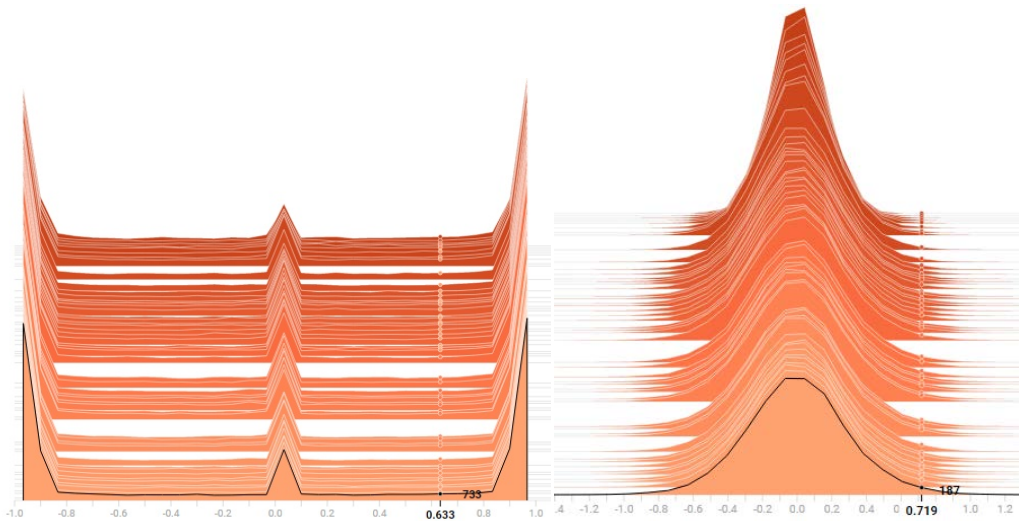


Figure 7-31 Model 8, GRU 150 units, 2 layers + FFNN 64x32x16x3 neurons by layer, with algorithms against overfitting. Histogram of output from recurrent layers (left); histogram of weights from first FFNN layer.

Model did not overfitted by accuracy chars[Figure 7-23]. Such high amount of recurrent units very much divided the features from samples, among themselves, output contains almost only -1, 0 and 1 and in my opinion, rather loose statement would be, it is first step to begins to approach to one hot input vectors. GRU layers become a property encoder. I think that zero output neurons are not good in this case, such neurons does not working with some samples.

I think that this network continues to learn thanks to feedforward part. That is the reason why I stopped on this network. Even if it has best accuracy on test set, I think that it can be not the best solution on another data.

Best histogram shape for GRU output, in my opinion, will be two maximums, at -1 and 1, such shape will correspond to a neuron with binary step activation function, such behavior is expected from sigmoidal functions.

Best shape for the histogram of weights from first FFNN layer, in my opinion will be uniform distributed weights near zero, or normal distribution with higher variance, than after initialization.

Bidirectional GRU

On this stage I choose two layers bidirectional GRU with feed forward layers and trying to get best accuracy from this network, remaining with the maximum number of trainable parameters below 300,000. I was trying 10 different networks, since bidirectional networks require more time for training, I stopped on network with 150,819 trainable parameters, that got 0.8474 accuracy on 51 epoch. For this network, I was guided by the rules developed earlier.

First and last networks shown on figure [Figure 7-32]

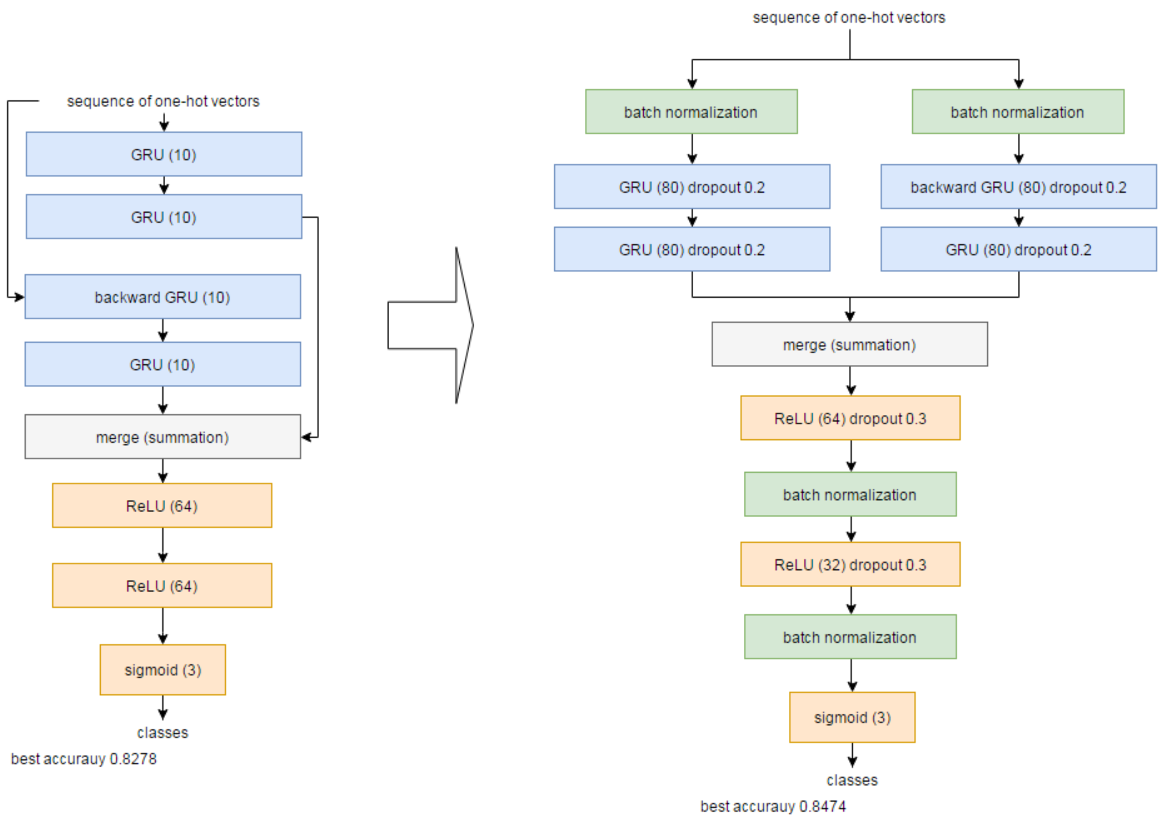


Figure 7-32 Started network and last network of this type. Dropout in GRU for all weights.

Last model was trained 62 epochs, that spend 8 hours and got best accuracy 0. 8474 on 51 epoch, whole test accuracy chart [Figure 7-33] and weights histograms [Figure 7-34]. According accuracy chart, model stopped in training without overfitting.

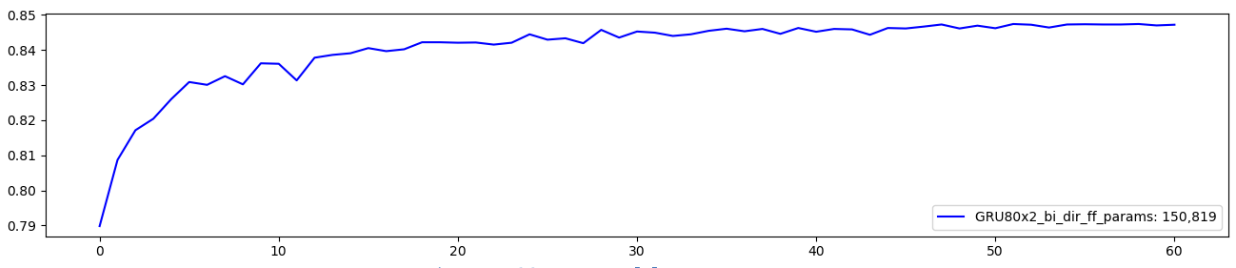


Figure 7-33 Best model test accuracy.

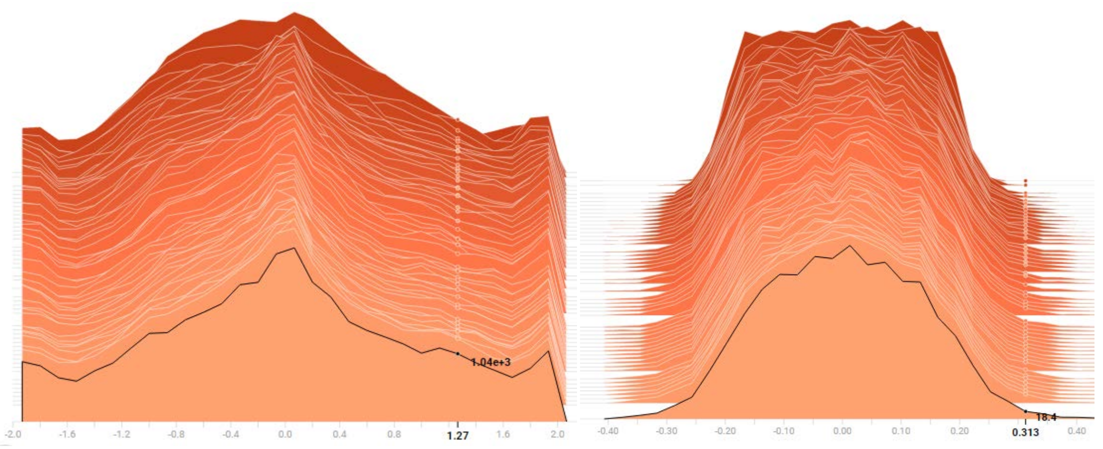


Figure 7-34 Bidirectional model GRU 80 units, 2 layers + FFNN 64x32x3 neurons by layer, with algorithms against overfitting. Histogram of merged output from recurrent layers (left); histogram of weights from first FFNN layer.

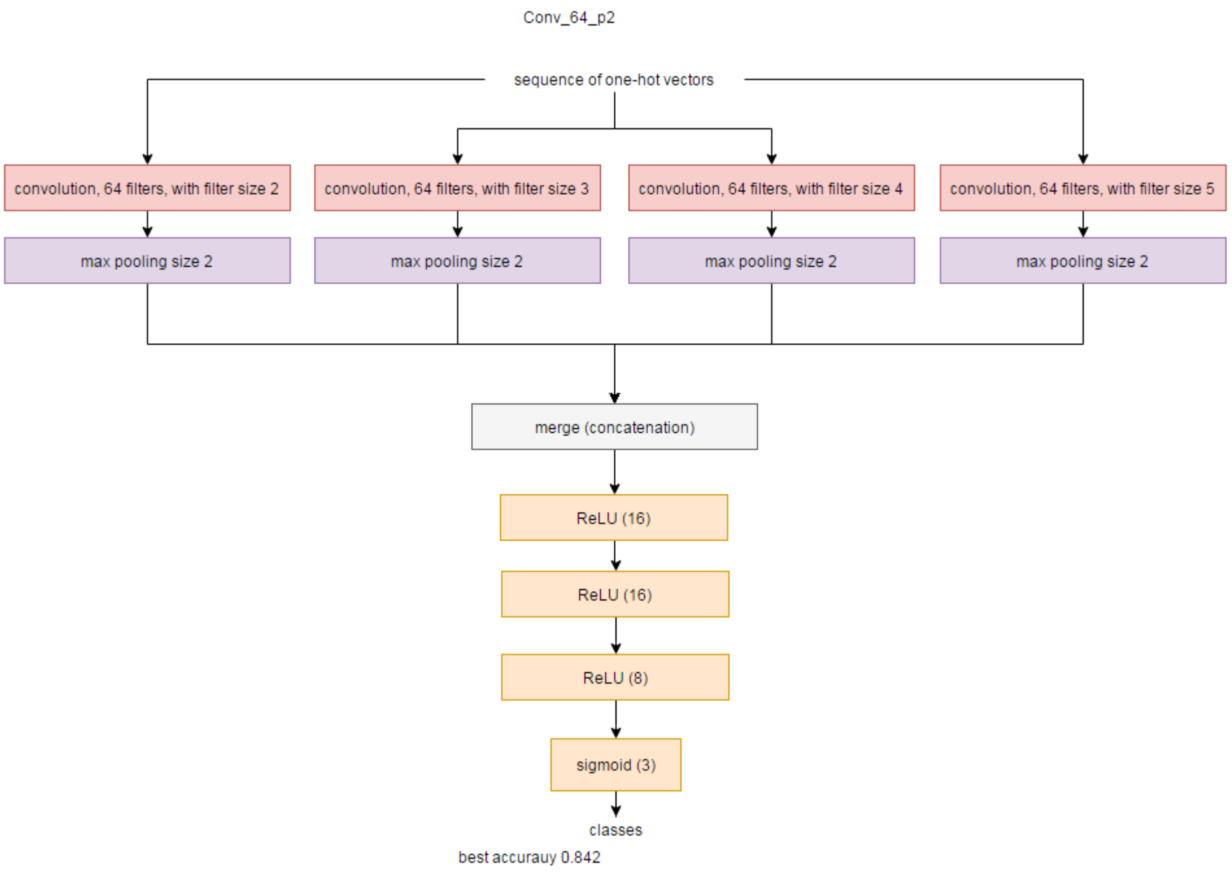
Model did not overfitted [Figure 7-33], GRU merged output from -2 to 2 is histogram of the sum and the distribution of each of the branches of the network can be very different, sum of uniform distributions should looks like triangle distribution, so, currently I assume that output from all brunches are more like uniform.

For me histograms looks good enough.

Convolutional network

Here I estimate convolutional network with feed forward layers and trying to get best accuracy from this network, remaining with the maximum number of trainable parameters below 300,000. I was trying 10 different convolutional networks, I stopped on network with 176,963 trainable parameters, that got 0.8427 accuracy on 61 epoch. For this network, I was guided by the rules developed earlier.

Here I will show two most interesting networks[Figure 7-35].


Conv_64_p2
sequence of one-hot vectors
convolution, 64 filters, with filter size 2
convolution, 64 filters, with filter size 3
convolution, 64 filters, with filter size 4
convolution, 64 filters, with filter size 5
max pooling size 2
max pooling size 2
max pooling size 2
max pooling size 2
merge (concatenation)
ReLU (16)
ReLU (16)
ReLU (8)
sigmoid (3)
classes
best accurauy 0.842

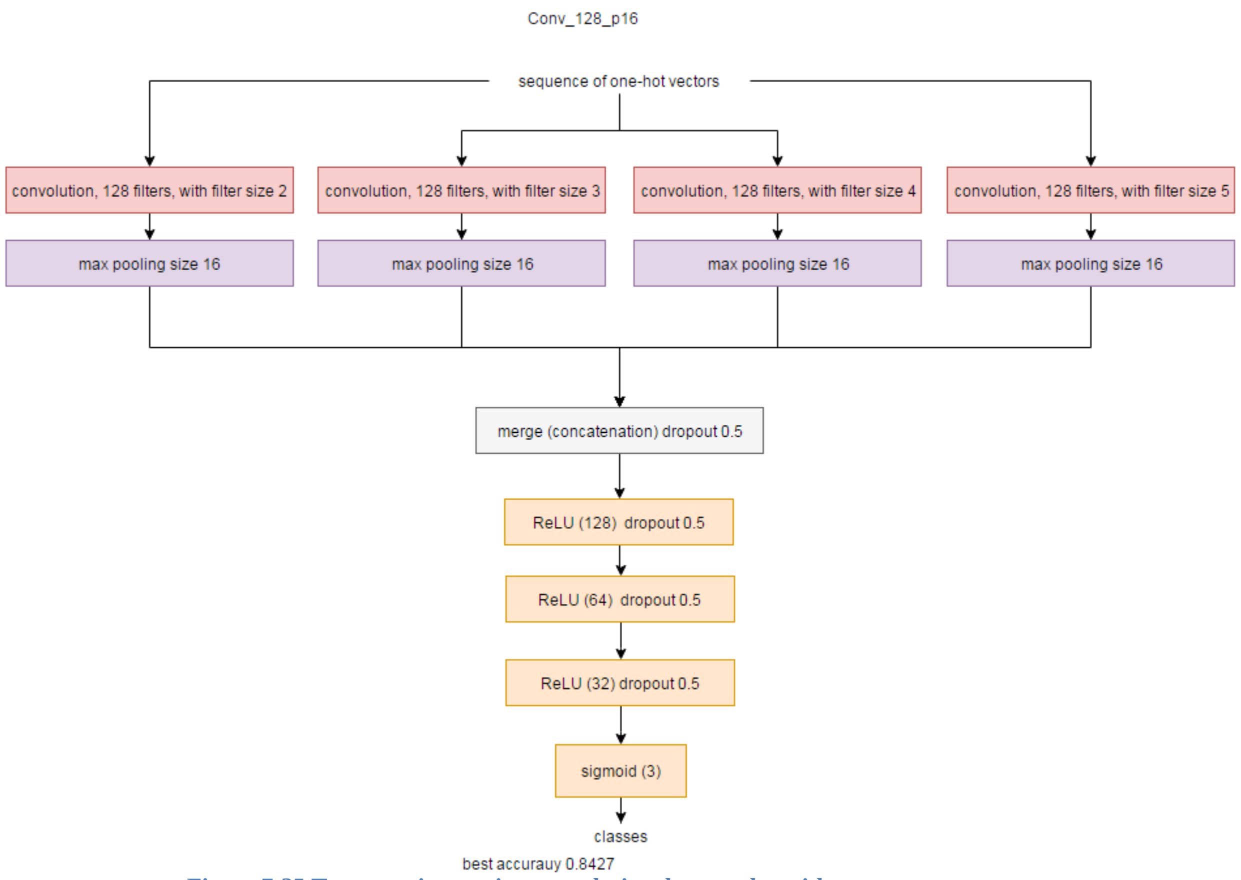

Conv_128_p16
sequence of one-hot vectors
convolution, 128 filters, with filter size 2
convolution, 128 filters, with filter size 3
convolution, 128 filters, with filter size 4
convolution, 128 filters, with filter size 5
max pooling size 16
max pooling size 16
max pooling size 16
max pooling size 16
merge (concatenation) dropout 0.5
ReLU (128) dropout 0.5
ReLU (64) dropout 0.5
ReLU (32) dropout 0.5
sigmoid (3)
classes
best accurauy 0.8427

Figure 7-35 Two most interesting convolutional networks, with accuracy on test set.

Even if both networks has similar best test accuracy, the number of parameters differs almost twice. The change in accuracy on the test data, in the learning process can be observed on the charts[Figure 7-36][Figure 7-37][Figure 7-38]. This is the first time in my work when a stronger model was trained slower, with the same training algorithm. It's also interesting that usually networks made a big jump in accuracy when they understood the difference between names. A convolutional network is much simpler than a recurrent and such a smooth accuracy growth, apparently due to the gradual finding of significant n-gramms.

Also first small network was owerfitted, but bigger network continue training.

According my experience big pooling window also prevent overfittng in comparison with model with the same amount of parameters, which sounds plausible, given that max pooling simply discards minor symptoms.

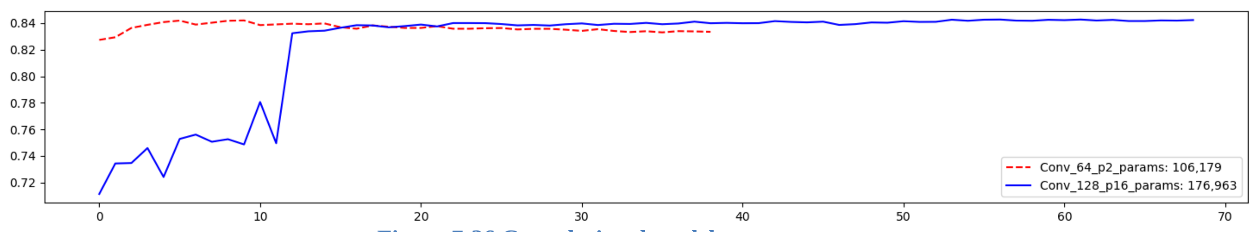


Figure 7-36 Convolutional models test accuracy.

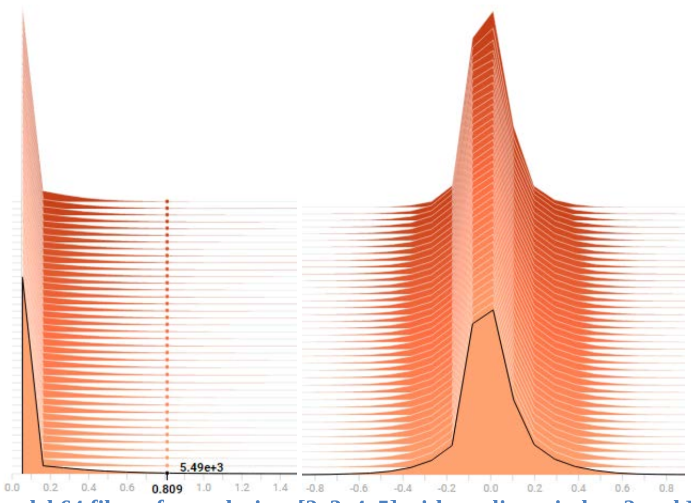


Figure 7-37 Convolutional model 64 filters for each sizes [2, 3, 4, 5] with pooling window 2 and FFNN 16x16x8x3 neurons by layer, with algorithms against overfitting. Histogram of merged output from convolutional layers (left); histogram of weights from first FFNN layer.

This network was overfitted [Figure 7-36]. Merged unbounded ReLU output histogram looks not interesting. Because small amount of neurons histogram of weights looks angular, but normal distributed- that is bad sign, this layer did not trained a lot.

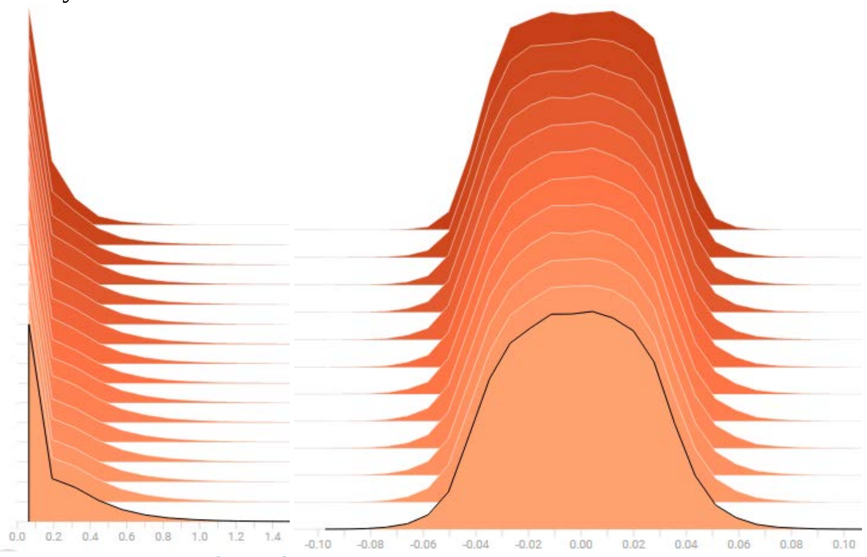


Figure 7-38 Convolutional model 128 filters for each sizes [2, 3, 4, 5] with pooling window 16 and FFNN 128x64x32x3 neurons by layer, with algorithms against overfitting. Histogram of merged output from convolutional layers (left); histogram of weights from first FFNN layer.

This network was not overfitted [Figure 7-36] and continued to trained. Merged unbounded ReLU output histogram has such additional angle in comparison with precious network because of dropout. Histogram of weights has not normal.

Additional improvements

First and simplest improvement of all networks, that I was trying to continue to train already trained networks with other training algorithms. I found that for these purposes, the simpler learning algorithms are well suited as SGD and RMSprop without any momentum. But continue of training my best recurrent network give me only 8 additional correct classified samples from test set that contain 15 000 samples.

Second idea to get better score is to made quorum of networks, and simple sum scores for each class from all networks. On this step I already has a lot of different networks, saved on different epochs, so to obtain a general classification, it was only necessary to classify the text data by different networks. In the first try I choose two best GRU networks, two best bidirectional GRU networks and three convolutional networks. This quorum gave me 0.8559 test accuracy on same test set with sample size 30 characters. Previous winner was two layers recurrent network (with biggest amount of parameters from described here networks: 258,923) with accuracy 0.8525.

The next attempt was to use the same network at different stages of training, to create a quorum, but in this case, on my examples I got the accuracy value close to the average between these networks

Development results:

Even the simplest network distinguishes work positions from names, and gets accuracy above 66%. On these data with the length of an example of 30 characters and networks of up to 300,000 parameters, it's not a problem to get accuracy of up to 84% on a simple non-optimized network, to get accuracy of up to 85%, you will need to work on optimization, to get accuracy over 85% Networks and many days of training these networks.
Also I want to mention that convolutional network has the best accuracy on samples with length 50 characters, but on the samples with 30 characters the winner was GRU network.
Based on the results of these 3 steps, I developed my own graph of the choice of the optimal network architecture, achieved some understanding of the internal operation of the network and the influence of some methods of dealing with overfitting on the result of the network state.
In the implementation of the final network I will use two layers GRU networks, bidirectional two layers GRU networks and convolutional networks.
Since the main difficulty was the separation of classes of names, I tried to train two networks, the first is trained on a class containing all the names, and the second only divides the names into two classes. But this approach did not give me any improvement in the quality of the classification and because of the complexity of implementation I declined it.

8 Implementation

For the replicability complete code of last solution (some interim solutions are not preserved) are available at: Appendix 1, run description.

The final implementation of the neural network is done in python, using tensorflow / keras frameworks. All visualization is done using the matplotlib library. Additional in the final implementation used libraries numpy and pickle. As a development environment, PyCharm was used.

Actually, the implication of a neural network is not a difficult task, even without using frameworks. And modern frameworks allow creating a simple neural network in a few lines of code. This work has developed an iterative and explicit separation between the design of the network and the implementation does not exist. By now I've tried more than a hundred models (considering different hyperparameters, architectures, learning algorithms, ...), each model was implemented. In this section, I will describe only the latest versions of the received networks.

Since the final goal of the work was a rich text type classification, rather than parsing spreadsheets, my experience in parsing is not part of the implementation section. In my experience above, parsing spreadsheets is better using Apache POI and Java.

Also, the entire classification is carried out at the level of individual cells in tables, not columns, which simplifies the work with data at the level of the neural network, allows you to classify columns of different lengths without additional costs and adds the additional ability to determine a column containing an unknown class.

The final network was trained at 5 classes of 300,000 samples from classes containing more samples. Used classes: First names, second names, codes, streets and job position titles. The maximum length of the example was 40 characters. Only classes of names were balanced.

In this paper, I do not use stemming and lemmatization because my examples mostly contain unique names and codes, besides a small noise in the data, for a neural network can be a plus. One of the main advantages of neural networks, in my opinion, is the ability to choose the most significant features from examples.

In some works, convolutional and recurrent networks are used sequentially. In my work, such a network architecture will not give any advantage, because the samples are too short.

Also, in this work for the test set, I took 5000 examples from each class to get a balanced test set. 5000 examples from the smallest class- unique job titles (45665 samples), is about 11%. In the process of working on this project, I did not meet a clear rule on the choice of the size of the training and test set, moreover, according to the law of large numbers [https://en.wikipedia.org/wiki/Law_of_large_numbers], I assume that for large data sets, the relative amount of test data can be reduced.

Cross-validation in neural networks is not used often, because of the huge computational costs. For each new sample, I will have to train the network again. In addition, during the training of the network, I changed some parameters, that is, the learning is not fully automated and even if all the random values used in the network (initialization of weights, dropout), are repeated exactly, each trained network becomes unique even with the same architecture.

In the development of networks, I chose networks with a size of up to 300,000 parameters, as the maximum number of examples in the class. But in my experience, the increase in the quality of classification by a neural network is increasing in leaps, an increase in the number of parameters depends more on the architecture of the network. So if I did not see improvements in the quality of classification, I did not increase the number of parameters, trying to keep the network as simple as possible.

In my experiments, the size of the batch varied from 100 to 1000 examples (too large batches require a lot of memory, so more than 1000 sometimes create an error). I obtained optimal results on a 500-sample batch.

The final learning algorithm was the Adam, with different learning rates and in the latest version, I myself change the learning rate, not using the standard methods of the framework (in which case I better understand the implementation).

In the process of training neural networks, I was guided by obtaining the highest quality of the test set classification. Therefore, the accuracy chart of the test set was the main factor. If in the last 5-10 epochs there was no improvement in the accuracy of the classification of the test set, I stopped training and tried to continue learning from the best point. To do this, I often took simpler learning algorithms, such as SGD with a low learning rate, but sometimes increased the learning rate, if the spasm did not seem high enough to me, or to accidentally get into another local minimum of the loss function.

Learning rate and annealing were selected in such a way that a small noise could be seen on the accuracy charts.

Since the project is only one module for classifying text data, no graphical interface or even an application programming interface was not supposed at this stage. At the moment, for classification, you need to have sample files, a trained network and run a script in the python.
Last implementation contains three modules: vectorization module, models description module, training module.

Vectorization module

The vectorization module loads text data and stores vectors ready for neural networks. In total, four folders are created with examples of length: 20, 30, 40, 50.
For the vectorization of all examples, first select unique characters, in my case, only 60 characters, of which a dictionary is made. Then each character of the example is encoded in one-hot vector with one corresponding to the position of the symbol in the dictionary. To save the finished ones, I used the library pickle. Most important part of vectorization module are shown in listing [Listing 8-1]

```
def vectors(class_, mlen):
    samplesamount = len(class_[1])
    cx1 = np.zeros((samplesamount, mlen, len(chars)), dtype=np.bool)
    for w, word in enumerate(class_[1]):
        fullword1 = word
        for c, char in enumerate(fullword1[:mlen]):
            cx1[w, c, char_indices[char]] = 1
    return cx1

maxlen=[20,30,40,50]
for mlen in maxlen:
    print(str(mlen))
    dataFolder = 'Prepared_vectors_'+str(mlen)+'/'
    os.makedirs(dataFolder)

    for class_ in classes:
        vector=vectors(class_,mlen)
        with open(dataFolder+class_[0]+'.pickle', 'wb') as f:
            pickle.dump(vector, f)

    with open(dataFolder+'chars.pickle', 'wb') as f:
        pickle.dump(chars, f)
    with open(dataFolder+'indices_char.pickle', 'wb') as f:
        pickle.dump(indices_char, f)
```

Listing 8-1 Main part of vectorization module.

Models description module

Model describes the models, selects training algorithms, divides the vectorized data into test and training sets and sends it all to the training module. Here I did not describe the trivial parts of the program, such as downloading files and splitting files into parts.
Models used in last classification described on figures[Figure 8-1][Figure 8-2].

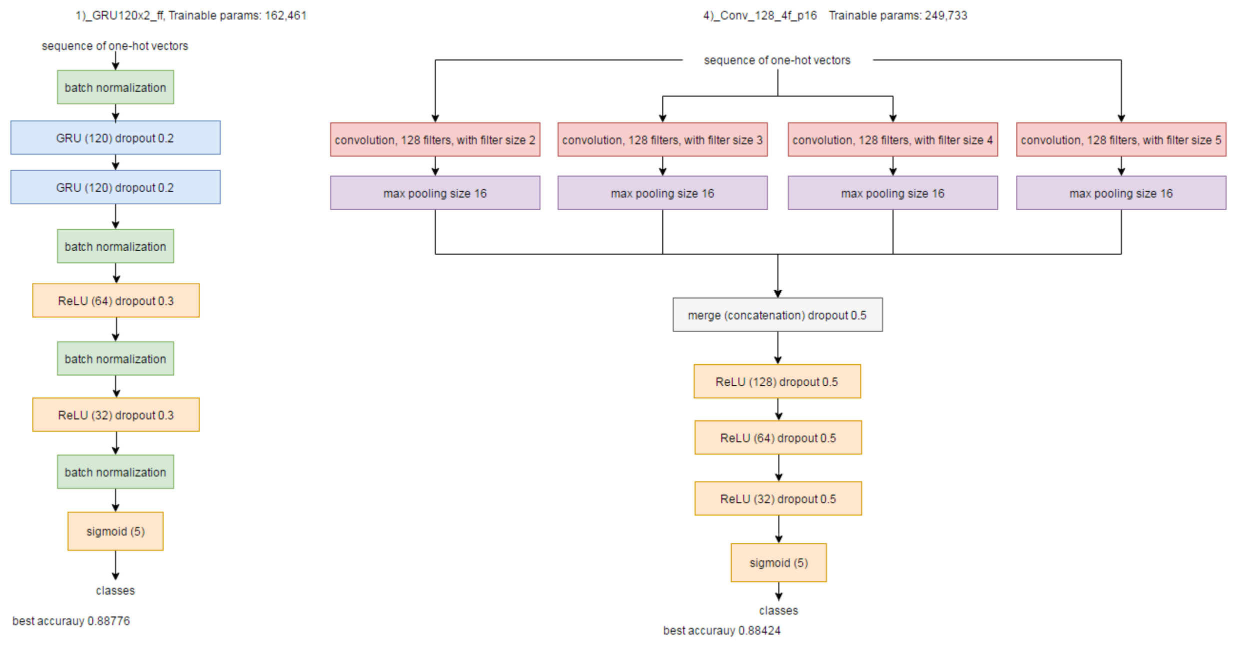


Figure 8-1 GRU and convolutional models, with accuracy on test set. The number of neurons/units is indicated in parentheses.

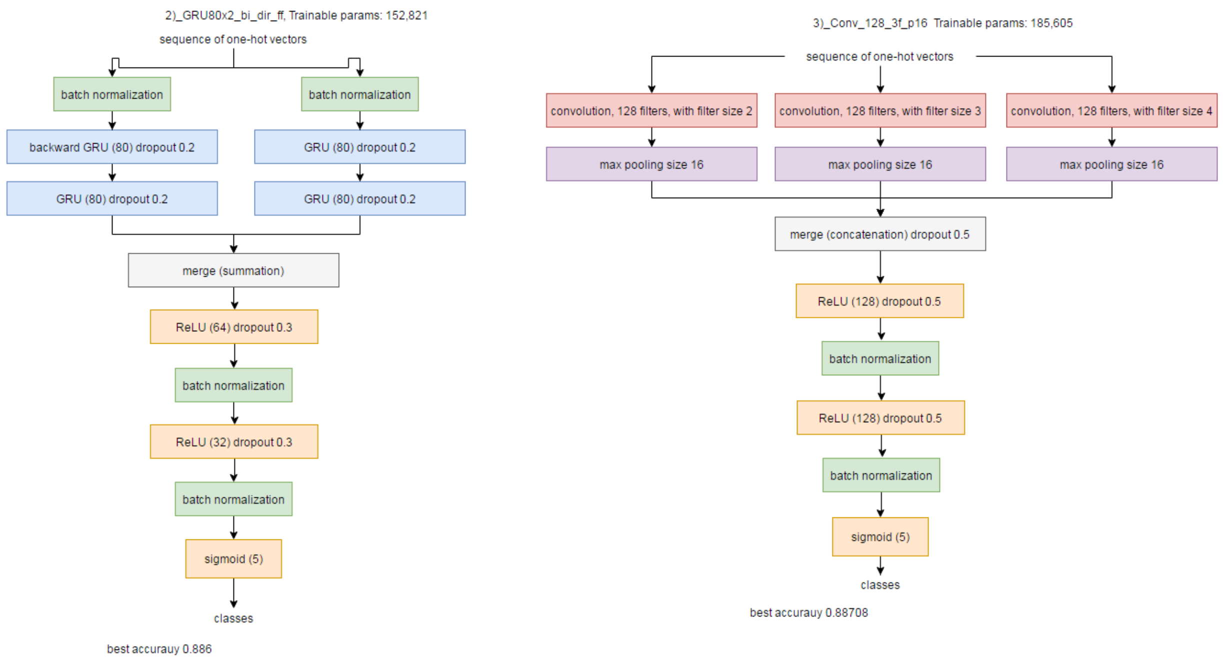


Figure 8-2 GRU and convolutional models, with accuracy on test set. The number of neurons/units is indicated in parentheses.

In all code listings I left only the basic terms.
All networks started with code [Listing 8-2]:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import GRU
from keras.optimizers import RMSprop, Nadam, Adam, SGD
from keras.layers.convolutional import Convolution1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.core import Flatten
from keras.layers.normalization import BatchNormalization
from keras.layers import Merge , Dropout
```

Listing 8-2 Code header

Next listings described last four models[Listing 8-3][Listing 8-4][Listing 8-5][Listing 8-6]:

```
model = Sequential()
model.add(BatchNormalization(input_shape=(maxlen, len(chars))))
model.add(GRU(120, dropout_W=0.2, dropout_U=0.2, return_sequences=True))
model.add(GRU(120, dropout_W=0.2, dropout_U=0.2))
model.add(BatchNormalization())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.3))
model.add(BatchNormalization())
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.3))
model.add(BatchNormalization())
model.add(Dense(classnum, activation='sigmoid'))
```

Listing 8-3 Model 1) GRU120x2_ff

```
model = Sequential()
left = Sequential()
left.add(BatchNormalization(input_shape=(maxlen, len(chars))))
left.add(GRU(80, dropout_W=0.2, dropout_U=0.2, return_sequences=True))
left.add(GRU(80, dropout_W=0.2, dropout_U=0.2))
right = Sequential()
right.add(BatchNormalization(input_shape=(maxlen, len(chars))))
right.add(GRU(80, dropout_W=0.2, dropout_U=0.2, return_sequences=True, go_backwards=True))
right.add(GRU(80, dropout_W=0.2, dropout_U=0.2))
model.add(Merge([left, right], mode='sum'))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.3))
model.add(BatchNormalization())
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.3))
model.add(BatchNormalization())
model.add(Dense(classnum, activation='sigmoid'))
```

Listing 8-4 Model 2) GRU80x2_bi_dir_ff

```
model = Sequential()
nb_feature_maps = 128
```

```
ngram_filters = [2, 3, 4]
conv_filters = []
for n_gram in ngram_filters:
        sequential = Sequential()
        conv_filters.append(sequential)
        sequential.add(Convolution1D(nb_filter=nb_feature_maps, filter_length=n_gram, activation='relu', input_shape=(maxlen, 
len(chars))))
        sequential.add(MaxPooling1D(pool_length=16))
        sequential.add(Flatten())
model.add(Merge(conv_filters, mode='concat'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(classnum, activation='sigmoid'))
```

Listing 8-5 Model 3) Conv_128_3f_p16

```
model = Sequential()
nb_feature_maps = 128
ngram_filters = [2, 3, 4, 5]
conv_filters = []
for n_gram in ngram_filters:
        sequential = Sequential()
        conv_filters.append(sequential)
        sequential.add(Convolution1D(nb_filter=nb_feature_maps, filter_length=n_gram, activation='relu', input_shape=(maxlen, 
len(chars))))
        sequential.add(MaxPooling1D(pool_length=16))
        sequential.add(Flatten())
model.add(Merge(conv_filters, mode='concat'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(Dense(classnum, activation='sigmoid'))
```

Listing 8-6 Model 4) Conv_128_4f_p16

Each model ends with the settings of the learning algorithm, annealing and balancing. Then it invokes the network training module. The approximate configuration code is shown in the listing[Listing 8-7]. Each trained network saved in different folders.

```
class_weight = {0: 4, 1: 1, 2: 4, 3: 1, 4: 1}
optimizer = Adam(lr=0.002)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
batch_size = 500

tr = TrainingClass()
tr.Training(model, batch_size, optimizer, os.getcwd()+'/model_11', trainX, trainY, testX, testY,
                                class_weight, epochnum=0, decay=0.01/70)
```

Listing 8-7 Example of training configuration

The accuracy of training of each of the networks is shown in the chart [Figure 8-3]. In the learning process, I sometimes changed the learning parameters. As a result, some networks were trained with new parameters. In this chart, I show only attempts that improve the result of the classification. After network training I chose best networks, that shown[Figure 8-1][Figure 8-2].

The important characteristic of the network is the speed of training. Average time for learning a network of one epoch (all data once):

• 1)_GRU120x2_ff	4.12 min
• 2)_GRU80x2_bi_dir_ff	7.82 min
• 3)_Conv_128_3f_p16	1.36 min
• 4)_Conv_128_4f_p16	1.53 min

The network was in the best classification condition, after which it compiled an ensemble of networks. As a result of the classification, each network issues score for each class and, summing up the outputs of all networks, I received a common score for the class. After that, I chose the class with the highest score. The result of this classification is shown as the result of the ensemble [Figure 8-3].

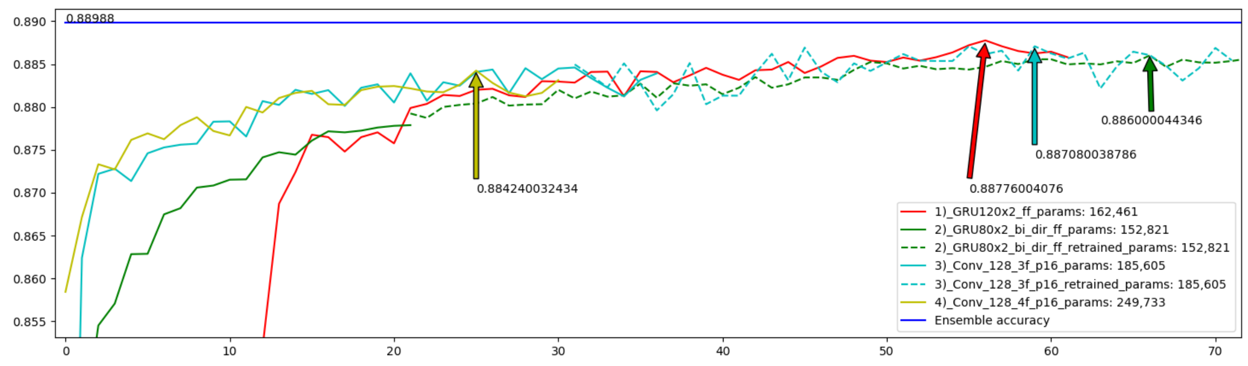


Figure 8-3 Accuracy charts of four networks, by epochs and accuracy of ensemble of this networks. Arrows shows best accuracy for each network.

Ensemble errors for classes [Figure 8-4]:

Class	False positive	False negative
First names	1299	1045
Second names	1259	1258
Codes	17	212
Streets	17	12
Position titles	161	226

Figure 8-4 Ensemble errors for classes.

As we can see Streets is the most correct classified class, and network has big problem to get difference between first and second names. But after a cursory review of the names, I realized that my own manual classification would only get worse.

Training module.

The task of the module was to manage the training of the network, collect the necessary statistics for visualization, save the network state after each epoch, and build network training schedules (including working with the tensorboard).

Listing [Listing 8-8] describe only important code from training module. Most inportant lines are *model.fit(...)* that starts training model and *lr=lr * (1. / (1. + decay*iteration))* that reduce leraining rate.

```
for iteration in range(1000):
        batchloss = []
        batchacc = []
        def plot_batch_loss(batch, logs):
                batchloss.append(logs['loss'])
                batchacc.append(logs['acc'])

        e_losses = [0]
        e_accs = [0]
        e_val_losses = [0]
        e_val_accs = [0]
        def plot_epoch_loss(epoch, logs):
                e_losses[0]=(logs['loss'])
                e_accs[0]=(logs['acc'])
                e_val_losses[0]=(logs['val_loss'])
                e_val_accs[0]=(logs['val_acc'])

        epoch_loss_callback = LambdaCallback(on_epoch_end=plot_epoch_loss)
        batch_loss_callback = LambdaCallback(on_batch_end=plot_batch_loss)

        hist=model.fit(trainX, trainY, batch_size=batch_size, nb_epoch=1, show_accuracy=True, verbose=0,
                class_weight=class_weight, validation_data=(testX, testY), shuffle=True, callbacks=
                                        [epoch_loss_callback, batch_loss_callback,tensorBoard])

        FalseNegetive, FalsePositive=self.ErrorByClass(model, testX, testY)

        model.save(resultFolder+'model_acc_'+str(e_val_accs[0])+'_epoch_'+str(epochnum+iteration)+'.h5')

        lr = K.eval(model.optimizer.lr)
        lr=lr * (1. / (1. + decay*iteration))
        K.set_value(model.optimizer.lr, lr)

def ErrorByClass(self, model, testX ,testY):
        ClassNotDetected=[0] * len(testY[0])
        ClassDetectedNotCorrect=[0] * len(testY[0])
        predictions=model.predict_classes(testX, verbose=0)
        for i,pred in enumerate(predictions):
                if testY[i][predictions[i]]==False:
                        ClassDetectedNotCorrect[predictions[i]]+=1
                        for k,_ in enumerate(testY[i]):
                                if testY[i][k]==True:
                                        ClassNotDetected[k]+=1
        return ClassNotDetected, ClassDetectedNotCorrect
```
Listing 8-8 Main part of training module.

9 Analysis

The resultant classifier in this work was an ensemble of four neural networks with a final classification accuracy of 0.88988. The maximum quality of classification by such networks only for first and second names was 0.745.

The main problem of this approach to classification are new classes that are not known for training. As I mentioned before - The basic requirement for training data, control and test sets should be representative. If the trained network receives a quote from the artwork or part of the program code on the python, the example will be classified as one of the known class networks.

Partial errors can be avoided by setting a threshold for the score for each class. The output in the network is limited by the sigmoid function from 0 to 1, and ideally the network issues one-hot vector, in which one indicates the class chosen for this sample. Really, the class is selected by the highest score i.e. if the classes have a scores: "0": 0.1, "1": 0.11, "2": 0.1, the highest score is 0.11 and the example will be classified as class 1. We can set a threshold for the maximum value of score 0.5, thanks that all samples with score lower than threshold should be classified as unknown classes. Such a payoff will naturally reduce the quality of classification of known classes. But, for the ensemble of networks, it is more preferable, because the unanimous choice of a class by all networks will greatly increase the score for this class. It will also be positively affected by the fact that the classification takes place within the separate cells, if the cells of one column are classified differently, this will be a good marker, meaning that there is a new class in the column. It is possible to improve the classification with a threshold value using the ReLU function in the last layer. But all these approaches will not give a guaranteed result.

For clarity, I show a schedule for classifying a test set from five classes of 5000 examples in each, the resulting ensemble of neural networks, using 30 different threshold values, from 0 to 4 [Figure 9-1]. Summing all errors, without threshold [Figure 8-4] was 2753 errors. Threshold less than 2 almost nothing changes, here an unknown class begins to appear - examples not defined for the network. 2 should be center of total bound of 4 networks, and here "total binary step" function will be changed. From threshold 2 to 3 false positive errors for second name and false negative errors for first name down, that increase size of unknown class to 5000 samples - network stopped recognize all first names. An unknown class grows not only due to errors, but also due to correctly classified examples. False positive first names and false negative second names errors down slower, form threshold 2 to 4. Error and unknown class match on threshold 4, and further raise the threshold no longer makes sense. The graph shows that the choice of the threshold value to 2 practically does not change the behavior of the classifications, but these values are before the transition through zero. In the situation where the error almost disappeared, except for the unknown class, 10,000 samples fell into the unknown class, most likely all the first and second names. Probably for columns this approach will be more applicable, but even now the threshold between 2 and 4 can be a plus.

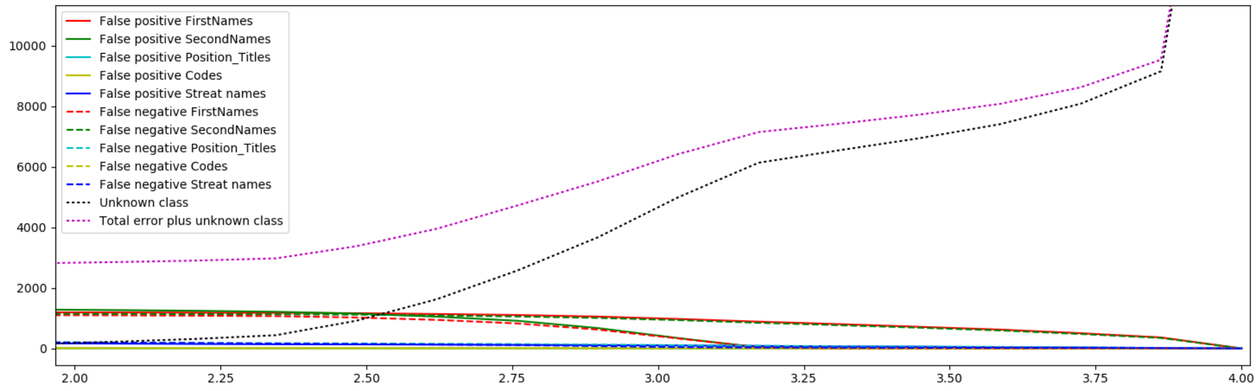


Figure 9-1 Amount of false positive, false negative errors and unknown class, for 25000 samples depending on the threshold value for maximum score.

The next step in working with new classes can be the threshold value for the probability of the class with the highest value. Probability I calculate as a score for this class divided by the sum of scores for all classes.

$$P_i = \frac{score_i}{\sum_0^k score_k}$$

For this example, I take thresholds from 0 to 1[Figure 9-2]. Starting with a threshold value of just under 0.5 to 0.9, errors associated with the classification of names are almost lost, and the size of an unknown class in this case becomes equal to 10,000 samples, which indicates that the network stops to classify first and second names. Further increase of the threshold value does not make sense, since the error becomes practically zero and the graphs of the unknown class and the total errors plus unknown class coincide.

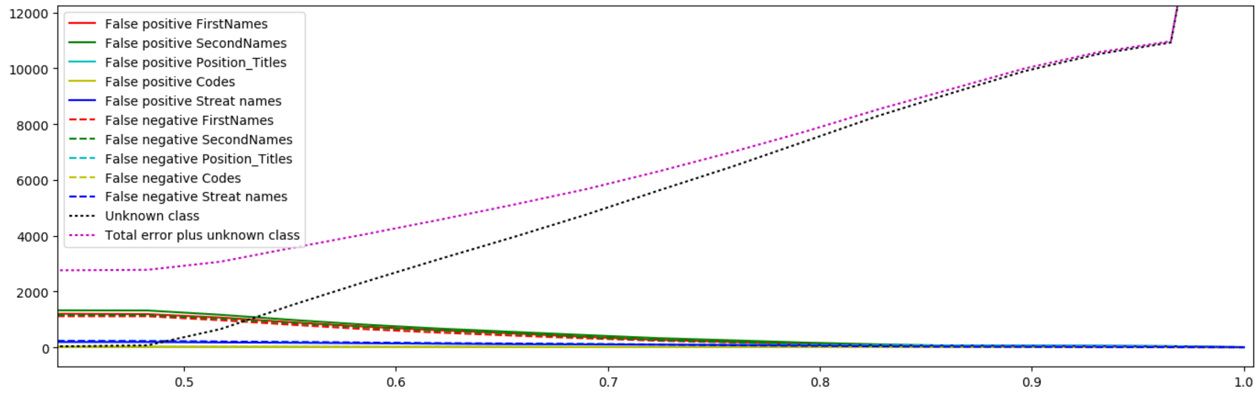


Figure 9-2 Amount of false positive, false negative errors and unknown class, for 25000 samples depending on the threshold value for probability.

To try this approach I prepared 5 additional unknown classes. To prepare unknown data I choose Hamlet, Romeo and Juliet [89], this my report, Student_guide_160913 and Regulations_160913 together, generated dates. During preparation I chose the text and divided it randomly into segments using length distributions from real datasets. Hamlet divided with first names distribution (almost same as second names), Romeo and Juliet divided with street names distribution, Student_guide_160913 and Regulations_160913 was divided with codes distribution, my report was devided as job titles, and generated dates has its own distribution.

Thanks to this approach, in the examples I got parts of words and whole words separated by dots, commas and sometimes numbers. Of course, the way to get an unknown class can be infinitely many, and this example shows only one of the possible outcomes of such a classification.

In total test set I got 10 classes, each class has 5000 samples. 5 classes that neural network was trained (but new for network test data) and 5 unknown classes.

At first I choose 30 thresholds from 0 to 1 [Figure 9-3]. As the result of classification I got 17000 false positive street names errors, 4400 false positive position titles and 3400 false positive codes. There are probably no additional errors with names, because the new unknown samples contain spaces. With an increase in the threshold above half, an unknown class appears in which all names immediately fall, as in the previous example, but the false positive street names error is still very high.

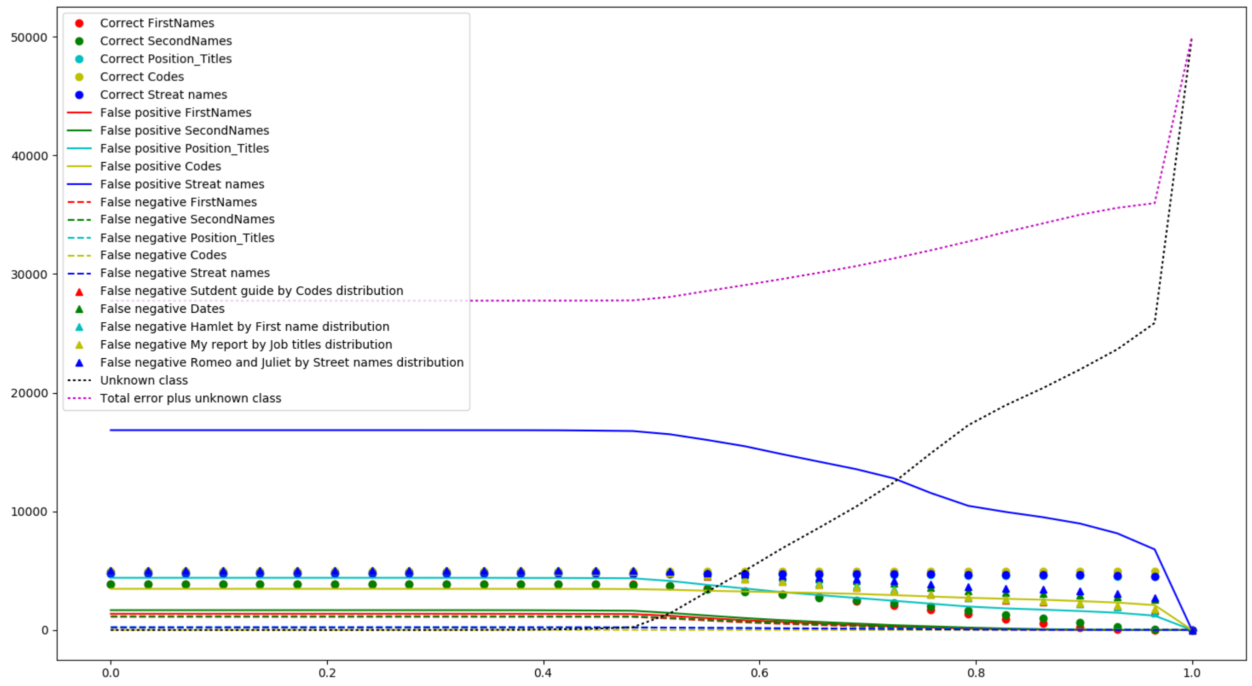


Figure 9-3 Amount of false positive, false negative errors, unknown class and correct classes, for 50000 samples depending on the threshold value for probability.

To check what happened with error near 1 threshold, I made one more chart with same data and with 25 thresholds between 0.99 and 1 [Figure 9-4]. Here we see that after threshold more than 0.995 false positive street names errors become less than correct classification for street names. And on the last step threshold 0.9996 false positive street names errors are equal 2000, but correct street names classification are still 3400.

At a high threshold value, networks can not define a complex classes, but simple classes are still well defined, so that although the network can not correctly determine part of the classes, some classes are defined correctly. Perhaps to improve the classification of unknown classes, do not stop at the best accuracy and continue to train the network, in which case the probabilities for already defined classes can grow and you can select a threshold value higher.

For these data, the probable probability value is above 0.9996, it can be useful for classifying codes, street names and job position titles.

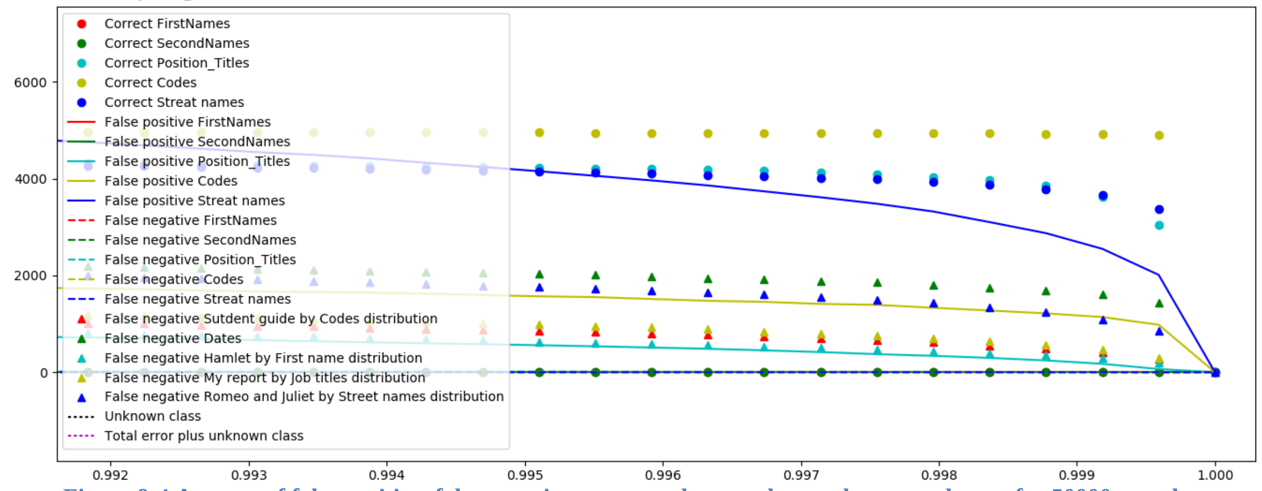


Figure 9-4 Amount of false positive, false negative errors, unknown class and correct classes, for 50000 samples depending on the threshold value for probability, from 0.98 to 1.

10 Conclusion, Discussion

The purpose of this work was a rich text type classification - detect categories of higher abstraction, like "person name", "job title", "project name", "activity description", "address", "equipment code", etc., so categories that people recognize, to classify the columns of spreadsheets. Neural networks are used in cases where the relationship between input and output data is not known, as in this case.

As classes for learning the neural network were chosen: First names, second names, codes, streets and job position titles. The most difficult to classify were first and second names. At a cursory look, I hardly found any known to me names, so I think my own manual classification would be about 50% on these two classes.

In the course of the work, more than a hundred different models of neural networks were compiled and trained and tested.

In the process I was using Tensorflow/Keras frameworks and trained network can be easily connected to any module in the python, in a few lines.

Also, a convenient system of scripts and graphs was created for the development and training of the network. Later, the training of such networks will take much less time.

The resultant classifier in this work was an ensemble of four neural networks with a final classification accuracy of 0.88988. In the ensemble, there are two convolutional networks, one recurrent and one bi-directional recurrent neural network. At first I planned to make a weighted vote in the ensemble of networks - networks with a higher accuracy would have a greater weight in voting, but in the end I came to the conclusion that the accuracy difference among the networks is too small and the weight of all networks in voting is equal to one.

A huge part of the work is devoted to improving the classification by a very small fraction of a percent, in general it is possible to take a standard model for a particular case and obtain a good classification result. Most further improvements results gain small increase in the quality of the classification. Even from existing models, the Improvement of model can be almost forever, improving the quality of classification by 1-2 examples from 25000, a few days of painstaking work. A good quotation of this approach is a quote: "After your network has converged the training is just beginning." Quotation was found by me on one of the forums devoted to neural networks, unfortunately I lost the link to the source.

All classifications are made for separate cells, not for columns.

The amount and quality of the data plays a major role in the neural network training and the problem of unknown classes is also a data problem. Of course, the network training, I would like to have more data for training, it would be guaranteed to improve the classification, and more classes data would avoid the problem of an unknown class more often. Also, real data sets have different amounts of data, and examples of different lengths, even with the length distributions shown in the datasets descriptions, some classification can be performed, for examples of different lengths.

At the stage of developing the model, I already chose some models of neural networks, which seemed to me to be the best for this work. In my opinion this is GRU and convolutional networks. Convolutional networks are much easier to train and trained few times faster (by time), but are quickly overfitted. Such a parameter as training time can be key in the work and if it is required to quickly train the network, I would choose convolutional networks that have a part of percent accuracy less than the recurrent networks.

At the stage of studying the theory, I often came across the phrase that the development of neural networks is an act of art, and now, having some experience with neural networks, I will agree with this statement. Any network trained on a cluster of high-performance computers or on a home computer, requires a huge amount of training time, and having more time it is always possible to make the network more complicated, even with the same number of trainable parameters. In this case, it is not possible for me to go through all the hyperparameters, even by some more advanced algorithm than brute force.

The main disadvantage of neural networks for solving similar problems, I think that to some extent I have to rely on luck when choosing hyper parameters.

I consider such results accuracy of 0.88988 is a successful achievement of the goal, but it can be increased with some improvements, that I did not try in this research. The behavior of the classifier on the unknown classes is significantly deteriorating, but this is contrary to the data requirement.

Some improvements I will describe here:

- Transfer learning - additional training of the network on new data. If just try to train the network on new data, then backpropagation effectively destroys all previously learned and training in fact already after a few bars goes from the very beginning (with wrong start initialization of weights).

Therefore, all the weights are fixed first, except for the weights of the last layer, last layer trained, the previous layers can be slowly released so that they also tune. Such additional training can be useful if new datasets appear, and it seems to me possible to learn the network on its own errors in the test set.

- Boosting and bagging, described here.
- Work on the formation of each batch, at the moment, the batches are formed randomly, and although the dataset is balanced, the balance of each individual batch is unknown. I think if you look at the formation of a batch, I would start with its balancing, although it might be worth starting with the batch, which contains more samples of complex classes like names, so that the network will see from the first changes the difference between such classes.
- Metaphone - a phonetic algorithm for indexing words by their sound, taking into account the basic rules of English pronunciation. Samples encoded with similar algorithms can also be used in network training. For example, two recurrent networks receive real examples and a metaphone output, and output the result to the general feedforward network. Similar advice using the same phrase in different languages for learning network I met in the literature [111]
- N-grams can be used in recurrent networks instead of characters.
- It can be used word embedding on n-grams level, to vectorize samples.
- Continue to train the network, after achieving the best accuracy for increasing the likelihood and the ability to use threshold values in the classification.

Novelty

Although the methods and approaches used in this work are not developed by me, nevertheless, all the researches I have found work with text information of a longer length. In NLP, normally, uses sentences or whole texts, as samples. The binary classification (positive / negative) is also often used. In my work neural networks classify short samples, often an samples is one word, and the number of final classes is 5. In addition, I made an attempt to classify an unknown neural network text class, which I have not seen in the literature.

The main uniqueness of this work is that at the time of writing the work, I was not able to find a ready-made solution or research for the classification of spreadsheets or such short samples.

11 References

1. Application of the genetic algorithm for training a neuron network in the objective identification of images A.M. Lipanov, A.V. Tyurikov, A.S. Suvorov, E.Yu. Shelkovnikov, P.V. Gulyaev
2. using genetic algorithms for training neural networks. Shumkov Eugene Alexandrovich, Chistik Igor Konstantinovich
3. Training of artificial neural networks Course "Neurointelligent systems", FBIC MIPT, 2013
4. Bag of Tricks for Efficient Text Classification. Armand Joulin, Edouard Grave, Piotr Bojanowski, Tomas Mikolov
5. Distributed Representations of Words and Phrases and their Compositionality. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean

7 Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio

8 On the importance of initialization and momentum in deep learning. I. Sutskever, J. Martens, G. Dahl, and G.E. Hinton.

9 Deep Residual Learning for Image Recognition. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

10 Systematic evaluation of CNN advances on the ImageNet. Dmytro Mishkin, Nikolay Sergievskiy, Jiri Matas

11 Notes on AdaGrad. Joseph Perla

12 Introduction to the Artificial Neural Networks. Andrej Krenker, Janez Bešter, Andrej Kos

13 MA 529 Stochastic Processes, Spring 2016 Instructor: Zhongqiang Zhang

14 L1 vs. L2 Regularization and feature selection. Paper by Andrew Ng, Presentation by Afshin Rostami

15 A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. Martin Riedmiller, Heinrich Braun

16 Dropout: A simple way to prevent neural networks from overfitting. Srivastava N., Hinton G., Krizhevsky A., Sutskever I., Salakhutdinov R.

17 Stochastic Gradient Descent for Non-smooth Optimization: Convergence Results and Optimal Averaging Schemes Ohad Shamir, Tong Zhang

18 You Only Look Once: Unified, Real-Time Object Detection. Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi

19 Efficient estimation of word representations in vector space. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean.

20 Convolutional Neural Networks for Sentence Classification. Yoon Kim

21 Exploring the Limits of Language Modeling. Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, Yonghui Wu

22 Practical Recommendations for Gradient-Based Training of Deep Architectures. Yoshua Bengio

23 LSTM: A Search Space Odyssey. Klaus Greff, Rupesh Kumar Srivastava, Jan Koutnik, Bas R. Steunebrink, Jurgen Schmidhuber

24 Recurrent Convolutional Neural Networks for Text Classification. Lai S., Xu L., Liu K., Zhao J.

25 Recurrent neural network regularization. Wojciech Zaremba, Ilya Sutskever, Oriol Vinyals

26 Short Text Clustering via Convolutional Neural Networks. Jiaming Xu, PengWang, Guanhua Tian, Bo Xu, Jun Zhao, FangyuanWang, Hongwei Hao

27 Text Understanding from Scratch. Xiang Zhang, Yann LeCun,

28 Towards End-to-End Speech Recognition with Recurrent Neural Networks. Alex Graves, Navdeep Jaitly

29 Fitnets: hints for thin deep nets. Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, Yoshua Bengio

30 Do Deep Nets Really Need to be Deep? Lei Jimmy Ba, Rich Caruana

31 Neural network optimum parameters determining under industrial process mathematical model construction. A.I. Gavrilov, P.V. Evdokimov

32 Applying Neural Networks. A practical Guide. Kevin Swingler

33 Predicting Amazon Product Review Helpfulness. James Wei, Jessica Ko, Jay Patel

34 Concept Linking for Clinical Text. Justin Fu

35 Using Recurrent Neural Networks for Slot Filling in Spoken Language Understanding. Grégoire Mesnil, Yann Dauphin, Kaisheng Yao, Yoshua Bengio, Li Deng, Dilek Hakkani-Tur, Xiaodong He, Larry Heck, Gokhan Tur, Dong Yu, Geoffrey Zweig
36 A Neural Conversational Model. Oriol Vinyals
37 A Recurrent Neural Network for Word Identification from Continuous Phoneme Strings. Robert B. Allen
38 Character-level Recurrent Text Prediction. Melvin Low
39 Convolutional Neural Networks for Sentence Classification. Kim, Y.
40 A Convolutional Neural Network for Modelling Sentences. Kalchbrenner, N., Grefenstette, E., & Blunsom, P.
41 Effective Use of Word Order for Text Categorization with Convolutional Neural Networks. Johnson, R., & Zhang, T.
42 Semantic Clustering and Convolutional Neural Network for Short Text Categorization. Wang, P., Xu, J., Xu, B., Liu, C., Zhang, H., Wang, F., & Hao, H.
43 Character-level Convolutional Networks for Text Classification. Zhang, X., Zhao, J., & LeCun, Y.
44 Text Understanding from Scratch. Zhang, X., & LeCun, Y.
45 Understanding bag-of-words model: a statistical framework Zhang Y., Jin R., Zhou Z.-H.
46 Gradient-based learning applied to document recognition. LeCun Y., Bottou L., Bengio Y., Haffner P.
47 Recurrent neural network language model training with noise contrastive estimation for speech recognition. Chen X., Liu X., Gales M.J.F., Woodland P.C.
48 Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts. Santos C.N., Gatti M.
49 Deep sparse rectifier neural networks. Glorot X., Bordes A., Bengio Y.
50 The vanishing gradient problem during learning recurrent neural nets and problem solutions. Hochreiter S.
51 Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. Hochreiter S., Bengio Y., Frasconi P., Schmidhuber J.
52 ADADELTA: an adaptive learning rate method. Zeiler M.D.
53 The Difficulty of Training Deep Architectures and the Effect of Unsupervised Pre-Training. D. Erhan, P.A. Manzagol, Y. Bengio, S. Bengio, P. Vincent.
54 Understanding the difficulty of training deep feedforward neural networks. X. Glorot and Y. Bengio.
55 Neural networks and principal component analysis: Learning from examples without local minima. P. Baldi and K. Hornik.
56 Constructing and training feed-forward neural networks for pattern classification. Xudong Jiang, Alvin Harvey Kam Siew Wah
57 Solving the problem of negative synaptic weights in cortical models. Christopher Parisien, Charles H. Anderson, Chris Eliasmith
58 Automation of monitoring of public opinion on the basis of intellectual analysis of messages in social networks. Budylsky Dmitry Viktorovich.
59 Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Sergey Ioffe, Christian Szegedy
60 A generalized approximation theorem and the computational capabilities of neural networks. A.N. Gorban
61 Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. John Duchi, Elad Hazan, Yoram Singer.
62 Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. Andrew M. Saxe, James L. McClelland, Surya Ganguli
63 Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho
64 Training Deep and Recurrent Networks with Hessian-Free Optimization. James Martens. Ilya Sutskever.
65 An Introduction to Neural Network Methods for Differential Equations Authors: Neha Yadav,Anupam Yadav,Manoj Kumar
66 Neural networks: basic models I. V. Zaentsev.
67 Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Kaiming He. Xiangyu Zhang. Shaoqing Ren. Jian Sun.

68 ImageNet Classification with Deep Convolutional Neural Networks. Alex Krizhevsky. Ilya Sutskever. Geoffrey E. Hinton

Links:

69 https://en.wikipedia.org/wiki/Activation_function
70 http://www.machinephilosopher.com/activation-function-neural-network/
71 http://cs231n.github.io
72 http://www.aiportal.ru/articles/neural-networks/decision-xor.html
73 http://stackoverflow.com/questions/5794954/determining-bias-for-neural-network-perceptrons/5798467
74 https://www.saylor.org/site/wp-content/uploads/2011/11/CS405-6.2.1.2-WIKIPEDIA.pdf
75 https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
76 http://www.stokastik.in/building-a-multi-class-text-classifier-from-scratch-using-neural-networks/
77 https://habrahabr.ru/post/272679/
78 https://habrahabr.ru/post/282900/
79 http://www.offconvex.org/2016/03/22/saddlepoints/
80 https: //habrahabr.ru/post/313216/
81 https://habrahabr.ru/post/318970/
82 http://sebastianruder.com/optimizing-gradient-descent/
83 https://en.wikipedia.org/wiki/Stochastic_gradient_descent
84 https://malaikannan.wordpress.com/2016/09/13/cross-entropy/
85 http://neuronus.com/nn/38-theory/961-nejronnye-seti-kokhonena.html
86 https://en.wikipedia.org/wiki/Overfitting
87 http://wiki.technicalvision.ru/index.php/Проблема_переобучения_модели_и_метод_регуляризации
88 http://cs.nyu.edu/~rostami/presentations/L1_vs_L2.pdf
89 https://habrahabr.ru/post/175819/
90 http://www.nanonewsnet.ru/articles/2016/kak-obuchaetsya-ii
91 https://habrahabr.ru/post/309302/
92 https://geektimes.ru/post/74326/
93 http://deeplearning.net/tutorial/lenet.html
94 http://www.360doc.com/content/16/0303/19/2459_539162206.shtml
95 http://colah.github.io/posts/2014-07-Understanding-Convolutions/
96 http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/
97 http://ben.bolte.cc/blog/2016/keras-language-modeling.html
98 http://eric-yuan.me/rnn1/
99 https://deeplearning4j.org/lstm
100 http://datareview.info/article/znakomstvo-s-arhitekturoy-lstm-setey/
101 https://offbit.github.io/how-to-read/
102 https://habrahabr.ru/post/277563/
103 http://servponomarev.livejournal.com/8599.html
104 https://en.wikipedia.org/wiki/Principal_component_analysis
105 https://en.wikipedia.org/wiki/Independent_component_analysis
106 https://en.wikipedia.org/wiki/Non-negative_matrix_factorization
107 https://en.wikipedia.org/wiki/Singular_value_decomposition
108 https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/
109 https://iksinc.wordpress.com/2015/04/13/words-as-vectors/
110 http://www.kdnuggets.com/2016/05/amazing-power-word-vectors.html/2
111 http://www.drdobbs.com/the-double-metaphone-search-algorithm/184401251
112 http://www.gedpage.com/soundex.html
113 https://basegroup.ru/community/glossary/representativeness
114 http://blog.revolutionanalytics.com/2016/08/deep-learning-part-1.html
115 https://github.com/Microsoft/CNTK/
116 https://github.com/Theano/Theano
117 https://www.tensorflow.org
118 Https://github.com/dmlc/mxnet
119 http://torch.ch
120 http://caffe.berkeleyvision.org

121 https://habrahabr.ru/company/microsoft/blog/313318/
122 http://static.barik.net/fuse/
123 https://openaddresses.io
124 http://yann.lecun.com/exdb/mnist/
125 http://ai.stanford.edu/~amaas/data/sentiment/
126 http://help.sentiment140.com/
127 http://www.videocardbenchmark.net/gpu_list.php
128 https://pythonprogramming.net/data-size-example-tensorflow-deep-learning-tutorial/
129 http://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/
130 https://github.com/fchollet/keras/tree/master/examples
131 http://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/
132 https://keras.io/optimizers/
133 http://machinelearningmastery.com/using-learning-rate-schedules-deep-learning-models-python-keras/
134 http://shakespeare.mit.edu/
135 http://pypr.sourceforge.net/ann.html

Additional materials:

lectures:

- Tensorflow and deep learning - without a PhD (https://github.com/martin-gorner/tensorflow-mnist-tutorial)
- Stanford University CS224d: Deep Learning for NLP 2016 (https://www.youtube.com/watch?v=Qy0oEkCZkBI&list=PLdkuolCDcHLocBoeunRB6LejpSZc_y8wM)
- CS231n Convolutional Neural Networks for Visual Recognition (http://cs231n.github.io)

Lecture slides:

- CSCI 315: Artificial Intelligencethrough Deep Learning W&L Winter Term 2016 Prof. Levy
- Recurrent neural networks Ekaterina Lobacheva lobacheva.tjulja@gmail.com Deep Learning CMC MSU, 2016

Internet resources:

- https://keras.io
- https://www.tensorflow.org
- http://neuralnetworksanddeeplearning.com/
- http://machinelearningmastery.com/
- https://blog.keras.io/
- https://habrahabr.ru/

12 Appendix 1 (program demo)

To demonstrate the program, I attach to this report the latest version of the most important scripts without models (**OnlyScriptsDemo.rar**). Same scripts and best models can be downloaded by link: **https://github.com/AleksandrData/PojectDemo**

The program requires installed frameworks: tensorboard and keras.

Demo version does not include scripts working with an unknown class, since this work is not complete.

Full demo version contains the program files:

- 1_PrepareVectors.py
- 2_Multiclass.py
- 3_predictions_one_word.py
- Training.py

And data files:

- Codes.txt
- FirstNames.csv
- Position_Titles.csv
- SecondNames.csv
- StreatsAscii.txt

Data files contain 30 samples of each type (only for demonstration of work). The sizes of the complete data sets are too large and some of the data is provided to me by Avito LOOPS company. Of course, it is not possible to fully train the network on such data.

Scripts contains minor changes to work with small sets of data.

Operating procedure:

The first script is **1_PrepareVectors.py**. As a result of the work, this script creates four folders with prepared vectors for all classes: **Prepared_vectors_20, Prepared_vectors_30, Prepared_vectors_40, Prepared_vectors_50**. The number in the folder name means the maximum length of the samples. Also, the folders contain dictionaries used to convert samples into vectors.

Second is the model selection script **2_Multiclass.py**. This script contains last four models from the report.

```
17          maxlen = 40
```

Specify what vector folder use for training.

```
35          testAmount = 5
```

Indicates size of the test data (changed for a small number of examples)

```
57          modelNum = 1
```

Chooses a model for learning

Lines 59-170 describe the models of neural networks, learning algorithm and data balancing.

The zero model allows you to continue learning previous models.

Selected model sent to the training class from the **Training.py** file.

The main line in this class is:

```
73          hist = model.fit (trainX, trainY, batch_size = batch_size, nb_epoch = 1, show_accuracy = True, verbose = 0, class_weight
                        = class_weight,
                        Validation_data = (testX, testY), shuffle = True, callbacks =
                        [Epoch_loss_callback, batch_loss_callback, tensorBoard])
```

That starting the learning process of the model for one epoch, and the lines:

```
94          lr = K.eval (model.optimizer.lr)
95          lr = lr * (1. / (1. + decay * iteration))
96          K.set_value (model.optimizer.lr, lr)
```

That changing learning rate after each epoch (annealing algorithm)

The rest of the code just outputs and saves various information about the training. Training does not provide a stop before 1000 iterations in the code, but it saves the model after each iteration. During training I manually stop algorithm, based on results. After each iteration, training algorithm saves:

- Charts form of images and data: res.png, acc.txt, batchloss.txt, loss.txt;
- Output to the console: time_err.txt;
- Model with the epoch number and accuracy on the test set: model_acc_0.2_epoch_0.h5;
- And the logs folder with data for tensorboard.

An example training console output contains a description of the model:

```
--------------------------------------------------
____________________________________________________________________________________________________
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
batchnormalization_1 (BatchNorma (None, 40, 46)        184         batchnormalization_input_1[0][0]
____________________________________________________________________________________________________
gru_1 (GRU)                      (None, 40, 120)       60120       batchnormalization_1[0][0]
____________________________________________________________________________________________________
gru_2 (GRU)                      (None, 120)           86760       gru_1[0][0]
____________________________________________________________________________________________________
batchnormalization_2 (BatchNorma (None, 120)           480         gru_2[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 64)            7744        batchnormalization_2[0][0]
____________________________________________________________________________________________________
dropout_1 (Dropout)              (None, 64)            0           dense_1[0][0]
____________________________________________________________________________________________________
batchnormalization_3 (BatchNorma (None, 64)            256         dropout_1[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 32)            2080        batchnormalization_3[0][0]
____________________________________________________________________________________________________
dropout_2 (Dropout)              (None, 32)            0           dense_2[0][0]
____________________________________________________________________________________________________
batchnormalization_4 (BatchNorma (None, 32)            128         dropout_2[0][0]
____________________________________________________________________________________________________
dense_3 (Dense)                  (None, 5)             165         batchnormalization_4[0][0]
====================================================================================================
Total params: 157,917
Trainable params: 157,393
Non-trainable params: 524
____________________________________________________________________________________________________
TensorBoard: tensorboard --logdir=C:\Users\User\Desktop\diplom\py\1PojectDemo/GRU120x2_ff/logs
```

And information on training:

```
Falce negetive: 5 5 0 5 5
Falce positive: 0 0 20 0 0
Epoch: 0, total train time: 26.755398817501494sec, with val_acc: 0.2

Lr: 0.00200000009499
Falce negetive: 5 5 0 5 5
Falce positive: 0 0 20 0 0
Epoch: 1, total train time: 47.03809079943137sec, with val_acc: 0.2

Lr: 0.00199966681719
Falce negetive: 5 0 4 5 5
Falce positive: 0 19 0 0 0
Epoch: 2, total train time: 66.94418693646054sec, with val_acc: 0.24

Lr: 0.00199900058082
Falce negetive: 5 0 4 5 5
Falce positive: 0 19 0 0 0
Epoch: 3, total train time: 87.05136487033272sec, with val_acc: 0.24
```

The last script **3_predictions_one_word.py** uses already trained models to classify new samples.
Line

```
13      words = ' alex ',' dog ',' agriculture ',' clock ',' jan ',' lopamudra ',' department of housing and ',' to understand their
                applicabil ']
```

Contains a list of words for classification. It is important that words do not have new symols, which were not in the dictionary for vectorization. Otherwise, you need to replace such symbols.
For each example, the program output a score for classes into the console. Then you can select the highest score or use algorithms to identify an unknown class.
Script output is:

```
classes:
['firs names', 'second names', 'position titles', 'codes', 'street names']
```

```
classification score:
--- alex; predicted:[  3.95075822e+00   1.56203949e+00   2.43370421e-03   1.50835770e-03   3.76382768e-02]
--- dog; predicted:[ 2.15022826  2.95039797  0.05574801  0.00852747  0.07484355]
--- agriculture; predicted:[ 1.29598784  2.22561455  0.03362644  0.00748577  0.17331529]
--- clock; predicted:[ 0.76049161  3.10469842  0.00836152  0.004857    0.07136095]
--- jan; predicted:[  3.99312258e+00   9.91214156e-01   1.67067209e-03   9.79841920e-04   2.45420095e-02]
--- lopamudra; predicted:[  3.61467838e+00   1.54853678e+00   3.60016245e-03   2.55763740e-03   4.55116108e-02]
--- department of housing and; predicted:[  1.76077492e-06   1.68220220e-06   3.95434856e+00   8.99853883e-04
   7.67063303e-03]
--- to understand their applicabil; predicted:[  2.19813955e-04   2.83384155e-02   3.04266739e+00   7.34444242e-03
   1.52156258e+00]
```