Ministry of Education and Science of the Republic of Kazakhstan
Suleyman Demirel University

Bek Bazatbekov

Realization of Monoplan, NetLines, NetSphere algorithms

THESIS

Presented in Partial Fulfillment for the
Degree of Master of Science in Mathematics
(degree code: 6M060100)
Department of Mathematics and Natural Sciences
Faculty of Engineering and Natural Sciences

Supervisor: **Kaisar Tulenbayev**

Kaskelen, 2019

Abstract

In this work, I consider a learning algorithms for classification tasks, called Monoplane, NetLines and NetSphere, which are all adapted for binary real input patterns. Algorithms generally helpful in classification data by self-constructing neural network, it generates new neurons to fix previous errors and stop new compilations when the output will have minimum error percent. To make realization of algorithms, that automatically construct neural networks by using appropriate methods, like backpropagation, gradient boosting, perceprtons, adaptive boosting and e.t.c To use algorithms (Monoplane, NetLines, NetSphere) to make classification of data by self-constructing neural networks.

Аңдатпа

Бұл жұмыста мен Monoplan, Net Lines және Net Sphere деп аталатын жіктеу есептерін оқыту алгоритмдерін қарастырамын. Алгоритмдер, әдетте, дербес нейрондық желіні құру арқылы деректерді жіктеу пайдалы, ол алдыңғы қателерді түзету және шығу ең аз қателер пайызы болған кезде жаңа компиляция тоқтату үшін жаңа нейрондарды генерациялайды. Кері тарату, градиентті бустинг, перцепртондар, адаптивті бустинг және т. б. сияқты тиісті әдістерді пайдалана отырып, нейрондық желілерді автоматты құру алгоритмдерін жүзеге асыру.

Аннотация

В этой работе я рассматриваю алгоритмы обучения задач классификации, называемые Monoplan, NetLines и NetSphere, которые адаптированы для двоичных вещественных входных паттернов. Алгоритмы, как правило, полезны в классификации данных путем самостоятельного построения нейронной сети, она генерирует новые нейроны, чтобы исправить предыдущие ошибки и остановить новые компиляции, когда выход будет иметь минимальный процент ошибок. Реализация алгоритмов автоматического построения нейронных сетей с использованием соответствующих методов, таких как обратное распространение, градиентный бустинг, перцепртоны, адаптивный бустинг и др. т. с использовать алгоритмы (Monoplan, NetLines, NetSphere) для классификации данных с помощью самостроящихся нейронных сетей.

Acknowledgements

First of all, I would appreciate my adviser, for helping me in every way he could. Moreover, academic stuff of university who aided us and never turned back.

To my family

Contents

1. Introduction

At a time when the neural network industry is developing at a high speed and it is necessary to find and develop new approaches to solving problems. The usual methods such as the method of back propagation of errors or methods of convolutional neural networks can not always provide accurate and fast answers, and therefore, in this work, I suggest the use of algorithms for automatic construction of networks that have a common idea proposed by Torres-Moreno [4]. Algorithms Monoplan, NetLines, NetSphere are intelligent algorithms for classifications. The best part of the algorithms is that they are trainees, reduce the time required for testing at the stages of learning neurons and do not have many layers(have only one), which greatly facilitates the work of computer technology and offloads the necessary computer memory. The main purpose of the work is to compare the already established methods of neural network training with the methods used in the algorithms of Monoplan, NetLines and NetSphere. Identification of positive and negative sides and implementation of these algorithms. To realize the idea of neural network, let's begin from its biological prototype. The idea of a detailed device of the brain appeared only about a hundred years ago. In 1888, Spanish doctor Ramoni Kayal experimentally showed us that the consistency of brain tissue is a interconnected homogeneous nodes - neurons. Later studies using an electron microscope showed that all neurons, regardless of type, have a similar organizational structure. From the central part of the neuron, the soma, there are tree-like processes, the dendrites. They have a role of carrier or receptors, which carries the signals between neurons. The task of the axon, the biggest sprout, is to transmit a signal of activity from the soma to other neurons. The junction of the axon with the dendrites of other neurons is divided by a small, about 200 nm, distance. This gap is called synapse. Signals between neurons are transmitted chemically and electrically. At rest, the neuron's protoplasm is negatively

charged, with a potential of about 70 mV. As signals arrive from dendrites, the charge is depolarized to a potential of 60 mV, there is a diffusion of positively charged sodium ions Na + into the soma, the neuron "works" - its charge rises sharply to a positive one, and then the excitation propagates through other axons. Then the core gradually comes to its original state. The neurons of the brain "work" out of sync, like digital devices. The response rate for different neurons can vary from 1 to 100 Hz. The speed of propagation of the signal lies in the range from 0.5 to 2 m / s. In addition, it was noted that the strength of the synaptic connection for different neurons is different and moreover, it is not constant. If the neuron "works" more often than the others, then the strength of its synaptic connection increases. Such adaptability of the synapse is called the Hebbian rule. The overall, the brains structure is very complex. The number of neurons can be estimated as approximately 1011. Each neuron has an average of 104 synaptic connections, and the whole brain has 1015 connections. From all of the above, it can be seen that the brain is a very complex system with many parameters, connections and external sensors, and we are not yet able to understand how the thinking process takes place.

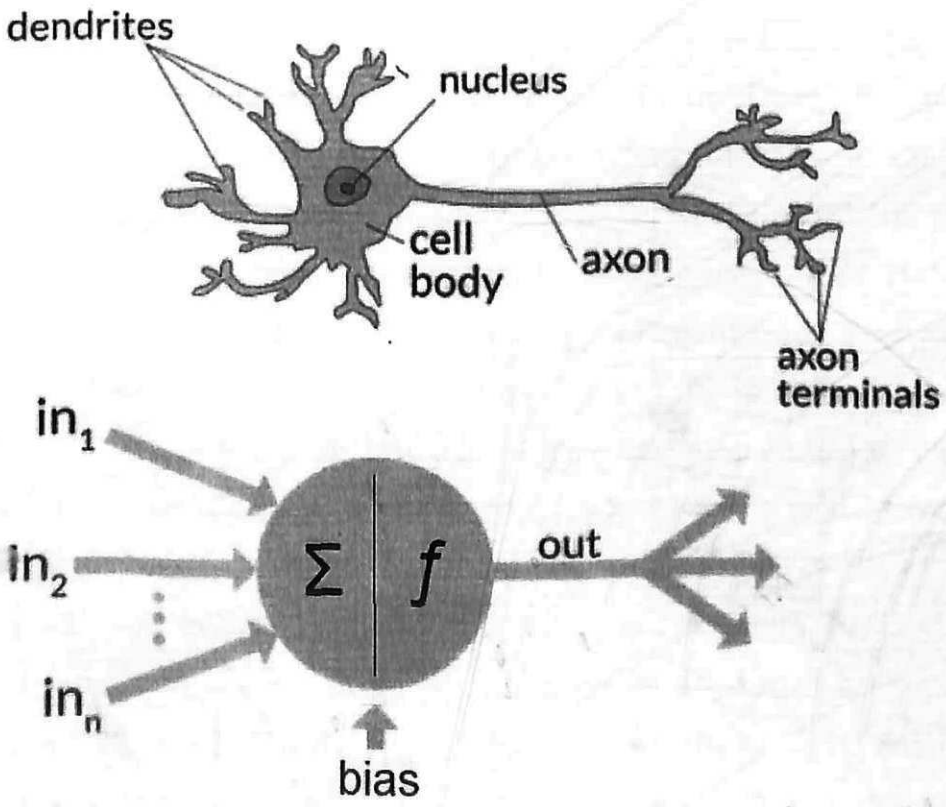

Figure 1.1: Biological prototype

2. Neural Network

2.1 Neural network theory

From the late 80s to the present, the theory of neural networks is experiencing a real boom in the number of scientific publications. In modern times, neural networks are increasingly used to solve a variety of practical problems. Moreover, it is safe to say that hardware and software devices built on the basis of neural networks are being actively introduced into our life and are increasingly used in medical diagnostics, industry, financial system and in everyday life - where it is necessary to solve problems, before that subject only to man.

There are two types of neural network training – with and without a teacher. In the case where we already got the teacher, has a ready answer for the input vector, in case without a teacher the neural network is self-learning. Each type of training has its own tasks and they almost do not intersect. To date, a lot of neural network architecture and methods of their training have been considered. But the main ones for training with a teacher it is "algorithm of back propagation of error", and for training without a teacher it is algorithms of Heb - BA and Kohonen. These types are very similar to biological reality, for example – a child is trained with or without a teacher? What the child is shown to his parents and teachers, what he is learning himself. There are also half-hearted examples of training, such as playing ball in the yard, the kid sees how it is received from others and the process of learning it takes place independently. In order to solve the problems with the help of neural networks it is necessary to: 1. Consider in detail the task and analyze it, think about whether it would be easier to solve it using other methods; 2. To determine the set of inputs. The set of inputs is selected from the beginning issued the smallest mistake; 3. You must select output variables, which will be the right output for this neural network; 4. Determine

what will be the number of layers and nodes in them; 5. The new network will be generated and trained by this examples.

The idea of artificial neural networks (ANN) arose as an attempt to describe the processes of perception of information occurring in the human brain. Like the human brain, ANN consists of a set of interconnected elements - neurons that mimic the neurons of the brain. There are various principles of networking, or otherwise, their architecture. In relation to neural networks, the term cytoarchitecture is sometimes used. A classic example of network architecture neurons of one neural network are arranged in one layer and its output becomes the input of the next layer. However, there are networks with more complex cytoarchitecture, for example, recursive networks. In this article only multilayered direct distribution networks will be considered, i.e. networks in which the excitation is transmitted only in one direction sequentially, from layer to layer. One of the models of a multilayered neural network is the back-propagation network error. This network model got its name due to a specific learning algorithm. Considered in many sources, it is given too in detail, which sometimes makes it difficult to grasp the essence of the algorithm. The purpose of this article is to show the advantage of using vector arithmetic to understand the principles of learning the neural network back propagation of an error. In addition, it allows you to more clearly present the principle of functioning of multilayer neural networks in general.

2.2 Neural Network

We have a relatively small number of classes (10 to be exact). The size of the images in the training set is 28x28 pixels. For these reasons, the neural network architecture will be quite simple: two convolutional layers and one maximum join layer (max pooling layer) to extract the characteristics of the original image; multilayer perceptron (MLP), whose task is to classify the previously obtained image characteristics.

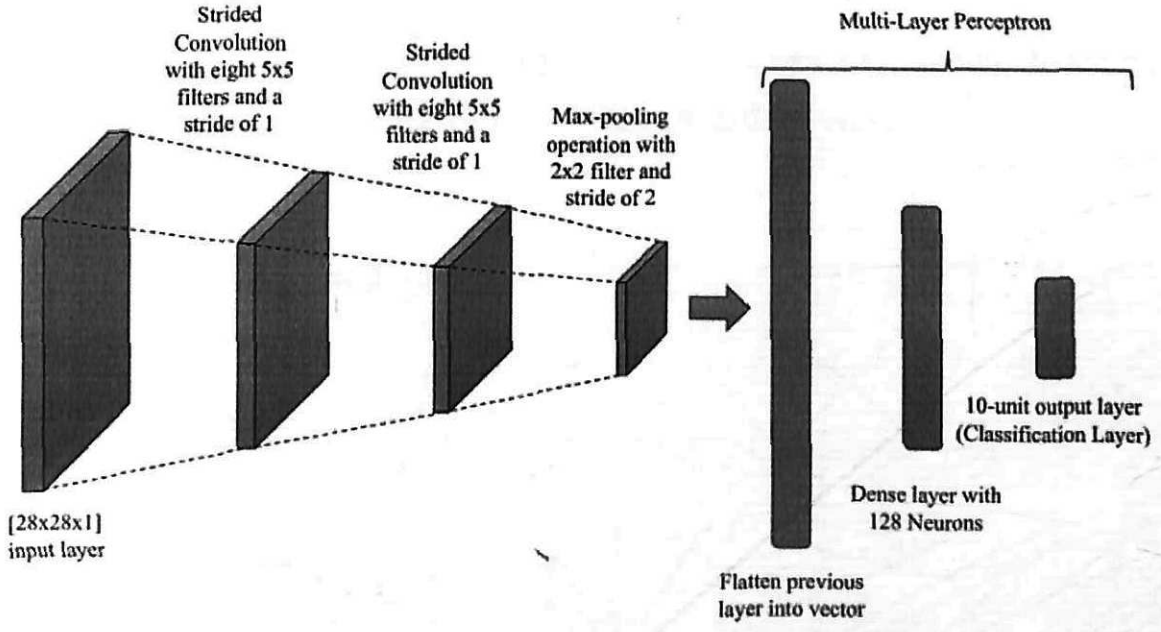

Figure 2.1: Neural network architecture

In neural network programming we will follow several steps:

Step 1. Data extraction

A collection of training and control handwritten numbers can be found here. The files in this collection have preserved the images and corresponding markers in the form of tensors. This makes working with data much easier than with real images. To extract data from files, we will use the following functions:

Step 2. Parameter initialization

Let's write functions to initialize filter parameters for convolutional layers and also consider the weights so they become fully connected layers. It will provide

```
def extract_data(filename, num_images, IMAGE_WIDTH):

    print('Extracting', filename)
    with gzip.open(filename) as bytestream:
        bytestream.read(16)
        buf = bytestream.read(IMAGE_WIDTH * IMAGE_WIDTH * num_images)
        data = np.frombuffer(buf, dtype=np.uint8).astype(np.float32)
        data = data.reshape(num_images, IMAGE_WIDTH*IMAGE_WIDTH)
        return data

def extract_labels(filename, num_images):

    print('Extracting', filename)
    with gzip.open(filename) as bytestream:
        bytestream.read(8)
        buf = bytestream.read(1 * num_images)
        labels = np.frombuffer(buf, dtype=np.uint8).astype(np.int64)
    return labels
```

Figure 2.2: Retrieving training data and the corresponding markers

the learning process uniform, we will perform initialization, in which the average value of the parameters will be zero, and the standard deviation — one.

```
def initializeFilter(size, scale = 1.0):
    '''
    Initialize filter using a normal distribution with and a
    standard deviation inversely proportional the square root of the number of units
    '''
    stddev = scale/np.sqrt(np.prod(size))
    return np.random.normal(loc = 0, scale = stddev, size = size)

def initializeWeight(size):
    '''
    Initialize weights with a random normal distribution
    '''
    return np.random.standard_normal(size=size) * 0.01
```

Figure 2.3: Functions of initialization of neural network parameters

Step 3. Back propagation of error.

In order to train the neural network, functions need to be developed to propagate the error back through the pudding and convolution layers.(Figure 3.3)

Step 4. Network construction

In network construction, we need to write a function which will combine both operations of forward and reverse propagation of signals in convolutional layers. This function will take the input image and produce gradients and loss characteristics.(Figure 3.5)

```
def conv(image, label, params, conv_s, pool_f, pool_s):

    [f1, f2, w3, w4, b1, b2, b3, b4] = params

    conv1 = convolution(image, f1, b1, conv_s)
    conv1[conv1<=0] = 0

    conv2 = convolution(conv1, f2, b2, conv_s)
    conv2[conv2<=0] = 0

    pooled = maxpool(conv2, pool_f, pool_s)
    (nf2, dim2, _) = pooled.shape
    fc = pooled.reshape((nf2 * dim2 * dim2, 1))
    z = w3.dot(fc) + b3
    z[z<=0] = 0

    out = w4.dot(z) + b4

    probs = softmax(out)
    loss = categoricalCrossEntropy(probs, label)
    dout = probs - label
    dw4 = dout.dot(z.T)
    db4 = np.sum(dout, axis = 1).reshape(b4.shape)

    dz = w4.T.dot(dout)
    dz[z<=0] = 0
    dw3 = dz.dot(fc.T)
    db3 = np.sum(dz, axis = 1).reshape(b3.shape)

    dfc = w3.T.dot(dz)
    dpool = dfc.reshape(pooled.shape)
    dconv2 = maxpoolBackward(dpool, conv2, pool_f, pool_s)
    dconv2[conv2<=0] = 0

    dconv1, df2, db2 = convolutionBackward(dconv2, conv1, f2, conv_s)
    dconv1[conv1<=0] = 0

    dimage, df1, db1 = convolutionBackward(dconv1, image, f1, conv_s)

    grads = [df1, df2, dw3, dw4, db1, db2, db3, db4]

    return grads, loss
```

Step 5. Neural network training

To make it able for neural network to learn the way to solve classification problems, we will use the Adam optimization algorithm. Adam — adaptive moment estimation, another optimization algorithm. It combines both the idea of accumulating motion and the idea of a weaker update of weights for typical features.

To evaluate the trained parameters of a neural network, its memory should be evaluated. This will give an understanding of how well the network is able to solve the problem of classification. This, in fact, is an assessment of the effectiveness of the neural network. Memory — a characteristic of the precision of the class definition.(Figures 3.6;3,7)[5]

```
def adamGD(batch, num_classes, lr, dim, n_c, beta1, beta2, params, cost):

    [f1, f2, w3, w4, b1, b2, b3, b4] = params

    X = batch[:,0:-1]
    X = X.reshape(len(batch), n_c, dim, dim)
    Y = batch[:,-1]

    cost_ = 0
    batch_size = len(batch)

    df1 = np.zeros(f1.shape)
    df2 = np.zeros(f2.shape)
    dw3 = np.zeros(w3.shape)
    dw4 = np.zeros(w4.shape)
    db1 = np.zeros(b1.shape)
    db2 = np.zeros(b2.shape)
    db3 = np.zeros(b3.shape)
    db4 = np.zeros(b4.shape)

    v1 = np.zeros(f1.shape)
    v2 = np.zeros(f2.shape)
    v3 = np.zeros(w3.shape)
    v4 = np.zeros(w4.shape)
    bv1 = np.zeros(b1.shape)
    bv2 = np.zeros(b2.shape)
    bv3 = np.zeros(b3.shape)
    bv4 = np.zeros(b4.shape)

    s1 = np.zeros(f1.shape)
    s2 = np.zeros(f2.shape)
    s3 = np.zeros(w3.shape)
    s4 = np.zeros(w4.shape)
    bs1 = np.zeros(b1.shape)
    bs2 = np.zeros(b2.shape)
    bs3 = np.zeros(b3.shape)
    bs4 = np.zeros(b4.shape)

    for i in range(batch_size):

        x = X[i]
        y = np.eye(num_classes)[int(Y[i])].reshape(num_classes, 1)
        grads, loss = conv(x, y, params, 1, 2, 2)
        [df1_, df2_, dw3_, dw4_, db1_, db2_, db3_, db4_] = grads

        df1+=df1_
        db1+=db1_
        df2+=df2_
        db2+=db2_
        dw3+=dw3_
        db3+=db3_
```

2.3 Perceptron

The perceptron is based on a mathematical model of information perception by the brain. There are 3 types of perceptron, which are called reacting elements, associative elements and sensors. Then, consider that S-elements moved in A-elements, so it could be represented as S-A. Why are A-elements called associative? The fact is that A-elements are aggregators of signals from sensor elements. For example, we have a group of sensors, each of which recognizes a piece of the letter "D" in the study picture. However, only their combination (that is, when several sensors gave a signal equal to 1) can excite the A-element as a whole. The other letters A-element does not react, only the letter "D". That is, it is associated with the letter "D". Hence the name.

There is another example. In fact, your eyes are made up of an incredible number of S-elements (sensors) that capture incident light (about 140,000,000). And you have some A-element that recognizes a specific part of the face. And then you saw a man on the street. Some A-elements that have recognized specific parts of the face are excited. Next, the signals that produced the excited A-elements are sent to the summator (R-element), the action of which you already know. However, to get to the R-element, they go through AR links, which also have weights. However, here they can already take any values (as opposed to S-A links).

Final chord. The R-element stacks the weighted signals from the A-elements with each other and, if a certain threshold is exceeded, generates an output signal equal to 1. This means that in the General flow of information from the eyes we recognize the person's face.

If the threshold is not exceeded, the perceptron output is -1. That is, we have not isolated a person from the General flow of information.

Since the R-element determines the output of the perceptron as a whole, it is called a reacting element. Similar names. But this is not the same as a perceptron with one hidden layer, although it may seem so. It is this type of perceptrons that we will study in more detail. This type of perceptron as an elementary perceptron, too often mean when they talk about the perceptron at all.

Its key feature is that each S-element uniquely corresponds to one A-element, all S-A links have a weight equal to +1, and the threshold A of elements is 1.

I'll explain. Take a picture of a perceptron in a General sense and convert it into a picture of a single-layer perceptron.

Based on the key features of a single-layer perceptron sensor can be uniquely associated with only one associative element. Look at the white sensor in the picture (top left corner). It transmits signal to light green (first) and gray (fourth) associative elements. Disorder. The sensor can only transmit a signal to one A-element. Remove the unnecessary connection. The same operation is carried out with other sensors.

Each sensor can only transmit a signal to one A-element. However, this statement does not prohibit the situation when several sensors transmit a signal to one A-element, as shown in the picture above (1, 2 and 3 A-elements).

Further, S-A links always have a weight of one, and the threshold of A-elements is always +1. On the other hand, we know that sensors can only signal 0 or 1.

Consider the first S-element in the last picture. Let it generate a signal equal to one. The signal passes through the S-A link and does not change, since any number multiplied by 1 is equal to itself. The threshold of any element is equal to 1. Since the sensor produced a signal equal to 1, the A-element was definitely excited. This means that it gave a signal equal to 1 (since it can also generate only 1 or 0 at its output). Then this single signal is multiplied by an arbitrary weight A-R connection and falls into the corresponding R-element, which sums up all the received weighted signals, and if they exceed its threshold, gives +1. Otherwise, the output of this R-element is -1.

Nothing on reminds? That's right, apart from sensory elements and S-A connections, we have just described the scheme of the artificial neuron. This is no accident. A single-layer perceptron is indeed an artificial neuron with a slight difference. In contrast to the artificial neuron, the single-layer perceptron inputs can take fixed values: 0 or 1. An artificial neuron can have any input values.

In the perceptron, R-elements sum the weighted inputs and, if the weighted sum is above a certain threshold, give 1. Otherwise, the outputs of the R-elements would be -1. It is easy to guess that this behavior is easily set by the activation function called the single jump function. The difference is that the single jump function gives 0 if the threshold is not exceeded, and here gives -1, but it is not

essential.

Thus it becomes clear that part of the single-layer perceptron (highlighted in black rectangle in the picture above) can be represented as an artificial neuron, but in any case do not confuse these two concepts. First, no one has canceled the S-elements, which in the artificial neuron is not there. Secondly, in a single-layer perceptron, S-elements and A-elements can take only fixed values 0 and 1, whereas in an artificial neuron there are no such restrictions.[5]

Multilayer perceptron

Under multilayer perceptron understand two different types: multilayer perceptron Rosenblatt and multilayer perceptron rumelhart.

Multilayer perceptron Rosenblatt Contains more than 1 layer of A-elements. The multilayer perceptron by Rumelhart is a special case of the multilayer perceptron by Rosenblatt, with two features:

S-A relationships can be of arbitrary weight and learn along with the A-R relations. Training is carried out by a special algorithm, which is called training by the method of back propagation of the error. This method is the cornerstone of learning all multilayer perceptron. Before starting the practice, I would like

to tell you more about what tasks perceptron is able to solve and what is not. I will also give a few handy analogies to understand whether it is possible to use a perceptron in this problem or not. This is extremely important, because the perceptron, as the simplest kind of neural network, not so much and knows how.

The perceptron very well solve the problem of classification. What does that mean? If you have a group of objects (for example, cats and dogs), the perceptron after training will be able to indicate to which group the object belongs (to cats or dogs).

Under the field of sensors refers to the set of all S-elements. Under classification – invented us classes (those same cats and dogs). Under the "non-empty set of elementary perceptrons conducting a successful classification" means that there is at least one perceptron, completed the classification of objects.

Consider the following example.

I want perseptron to learn to distinguish between cats and dogs. This is a difficult task, but we will make it much easier.

We will have 3 of the sensor: the length of the legs, the color and the shape

of the muzzle. Since S-elements can take the values 0 or 1, let's assume that the values 1 will correspond to short legs, mixed color and rounded muzzle, respectively. Values 0 will indicate the sign of the dog on this S-element (long legs, solid color and elongated muzzle). So we got the sensor field. If you like, it can be represented as a set of possible values 0 and 1 for each S-element. For example, an absolute cat should trigger all S-elements 1,1,1.

```
def convolutionBackward(dconv_prev, conv_in, filt, s):
    (n_f, n_c, f, _) = filt.shape
    (_, orig_dim, _) = conv_in.shape
    dout = np.zeros(conv_in.shape)
    dfilt = np.zeros(filt.shape)
    dbias = np.zeros((n_f,1))
    for curr_f in range(n_f):
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:

                dfilt[curr_f] += dconv_prev[curr_f, out_y, out_x] *
                conv_in[:, curr_y:curr_y+f, curr_x:curr_x+f]

                dout[:, curr_y:curr_y+f, curr_x:curr_x+f] +=
                dconv_prev[curr_f, out_y, out_x] * filt[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
        dbias[curr_f] = np.sum(dconv_prev[curr_f]
    return dout, dfilt, dbias

def nanargmax(arr):
    idx = np.nanargmax(arr)
    idxs = np.unravel_index(idx, arr.shape)
    return idxs

def maxpoolBackward(dpool, orig, f, s):
    (n_c, orig_dim, _) = orig.shape

    dout = np.zeros(orig.shape)

    for curr_c in range(n_c):
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:

                (a, b) = nanargmax(orig[curr_c, curr_y:curr_y+f, curr_x:curr_x+f])
                dout[curr_c, curr_y+a, curr_x+b] = dpool[curr_c, out_y, out_x]

                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return dout
```

Figure 2.4: The function of back propagation of the gradient through the convolutional and pudding layers

```
def conv(image, label, params, conv_s, pool_f, pool_s):

    [f1, f2, w3, w4, b1, b2, b3, b4] = params

    conv1 = convolution(image, f1, b1, conv_s)
    conv1[conv1<=0] = 0

    conv2 = convolution(conv1, f2, b2, conv_s)
    conv2[conv2<=0] = 0

    pooled = maxpool(conv2, pool_f, pool_s)
    (nf2, dim2, _) = pooled.shape
    fc = pooled.reshape((nf2 * dim2 * dim2, 1))
    z = w3.dot(fc) + b3
    z[z<=0] = 0

    out = w4.dot(z) + b4

    probs = softmax(out)
    loss = categoricalCrossEntropy(probs, label)
    dout = probs - label
    dw4 = dout.dot(z.T)
    db4 = np.sum(dout, axis = 1).reshape(b4.shape)

    dz = w4.T.dot(dout)
    dz[z<=0] = 0
    dw3 = dz.dot(fc.T)
    db3 = np.sum(dz, axis = 1).reshape(b3.shape)

    dfc = w3.T.dot(dz)
    dpool = dfc.reshape(pooled.shape)
    dconv2 = maxpoolBackward(dpool, conv2, pool_f, pool_s)
    dconv2[conv2<=0] = 0

    dconv1, df2, db2 = convolutionBackward(dconv2, conv1, f2, conv_s)
    dconv1[conv1<=0] = 0

    dimage, df1, db1 = convolutionBackward(dconv1, image, f1, conv_s)

    grads = [df1, df2, dw3, dw4, db1, db2, db3, db4]

    return grads, loss
```

Figure 2.5: Forward and reverse propagation of signals in convolutional layers

```
def adamGD(batch, num_classes, lr, dim, n_c, beta1, beta2, params, cost):

    [f1, f2, w3, w4, b1, b2, b3, b4] = params

    X = batch[:,0:-1]
    X = X.reshape(len(batch), n_c, dim, dim)
    Y = batch[:,-1]

    cost_ = 0
    batch_size = len(batch)

    df1 = np.zeros(f1.shape)
    df2 = np.zeros(f2.shape)
    dw3 = np.zeros(w3.shape)
    dw4 = np.zeros(w4.shape)
    db1 = np.zeros(b1.shape)
    db2 = np.zeros(b2.shape)
    db3 = np.zeros(b3.shape)
    db4 = np.zeros(b4.shape)

    v1 = np.zeros(f1.shape)
    v2 = np.zeros(f2.shape)
    v3 = np.zeros(w3.shape)
    v4 = np.zeros(w4.shape)
    bv1 = np.zeros(b1.shape)
    bv2 = np.zeros(b2.shape)
    bv3 = np.zeros(b3.shape)
    bv4 = np.zeros(b4.shape)

    s1 = np.zeros(f1.shape)
    s2 = np.zeros(f2.shape)
    s3 = np.zeros(w3.shape)
    s4 = np.zeros(w4.shape)
    bs1 = np.zeros(b1.shape)
    bs2 = np.zeros(b2.shape)
    bs3 = np.zeros(b3.shape)
    bs4 = np.zeros(b4.shape)

    for i in range(batch_size):

        x = X[i]
        y = np.eye(num_classes)[int(Y[i])].reshape(num_classes, 1)
        grads, loss = conv(x, y, params, 1, 2, 2)
        [df1_, df2_, dw3_, dw4_, db1_, db2_, db3_, db4_] = grads

        df1+=df1_
        db1+=db1_
        df2+=df2_
        db2+=db2_
        dw3+=dw3_
        db3+=db3_
```

Figure 2.6: Software implementation of the optimization algorithm Adam(part 1)

```
def train(num_classes = 10, lr = 0.01, beta1 = 0.95, beta2 = 0.99, img_dim = 28, img_depth = 1,
          f = 5,
          num_filt1 = 8, num_filt2 = 8, batch_size = 32, num_epochs = 2, save_path = 'params.pkl'):

    m =50000
    X = extract_data('train-images-idx3-ubyte.gz', m, img_dim)
    y_dash = extract_labels('train-labels-idx1-ubyte.gz', m).reshape(m,1)
    X-= int(np.mean(X))
    X/= int(np.std(X))
    train_data = np.hstack((X,y_dash))

    np.random.shuffle(train_data)

    f1, f2, w3, w4 = (num_filt1 ,img_depth,f,f), (num_filt2 ,num_filt1,f,f), (128,800), (10, 128)
    f1 = initializeFilter(f1)
    f2 = initializeFilter(f2)
    w3 = initializeWeight(w3)
    w4 = initializeWeight(w4)

    b1 = np.zeros((f1.shape[0],1))
    b2 = np.zeros((f2.shape[0],1))
    b3 = np.zeros((w3.shape[0],1))
    b4 = np.zeros((w4.shape[0],1))

    params = [f1, f2, w3, w4, b1, b2, b3, b4]

    cost = []

    print("LR:"+str(lr)+", Batch Size:"+str(batch_size))

    for epoch in range(num_epochs):
        np.random.shuffle(train_data)
        batches = [train_data[k:k + batch_size] for k in range(0, train_data.shape[0], batch_size)]

        t = tqdm(batches)
        for x,batch in enumerate(t):
            params, cost = adamGD(batch, num_classes, lr, img_dim, img_depth, beta1, beta2,
                                  params, cost)
            t.set_description("Cost: %.2f" % (cost[-1]))


    with open(save_path, 'wb') as file:
        pickle.dump(params, file)

    return cost
```

Figure 2.7: Software implementation of the optimization algorithm Adam(part 2)

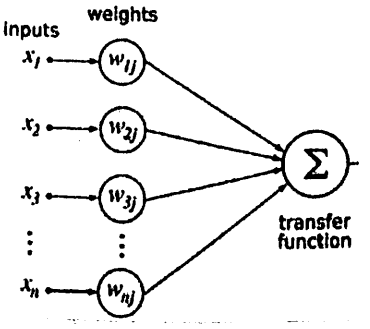

Figure 2.8: Perceptron

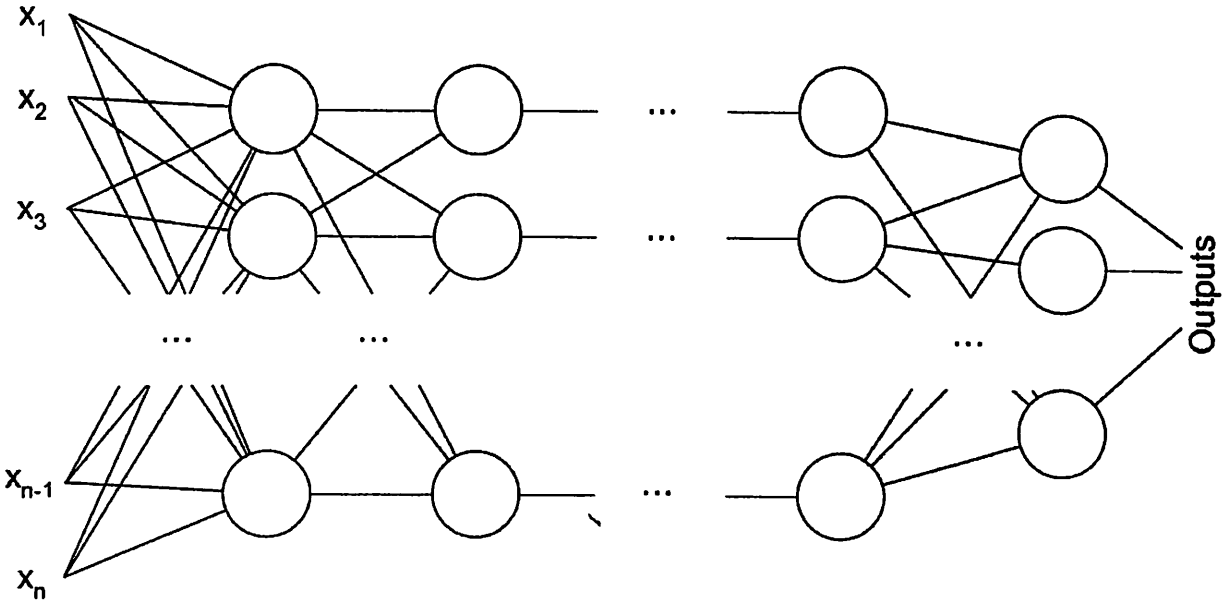

Figure 2.9: Multilayer perceptron

3. Convolutional neural networl

3.1 Simple Convolutional neural networks

Convolutional neural networks work on the basis of filters that are engaged in the recognition of certain characteristics of the image (for example, straight lines). A filter is a collection of cores; sometimes a single core is used in a filter. Kernel is a common matrix of numbers called weights, which are "trained" (adjusted if you prefer) to search the images for certain characteristics. The filter moves along the image and determines whether some desired feature is present in a particular part of the image. To obtain an answer of this kind, a convolution operation is performed, which will give us the sum of the all products of the filter elements of matrix of the signals which has been inputed. 3.1)

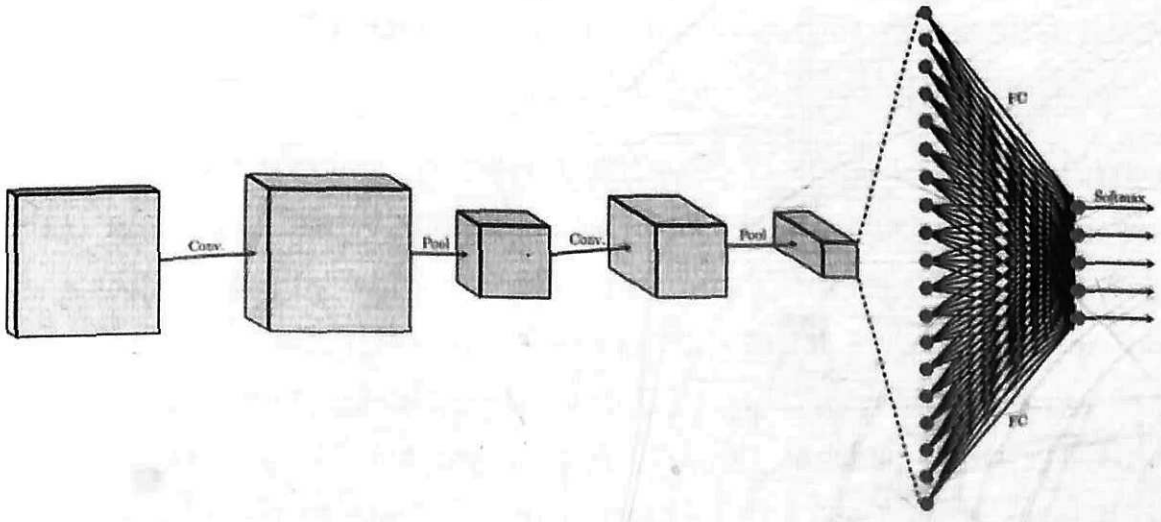

Figure 3.1: Convolution operation

For a better understanding of the most basic concepts, I recommend taking a look at this article. In it you will find a more detailed description of all these things. If some desired characteristic is present in the image fragment, the con-

volution operation will output a number with a relatively large value. If there is no characteristic, the output number will be small. The filter responsible for finding left-hand curves moves along the original image. When the fragment under consideration contains the desired curve, the summary of this convolution will be a big number (5500 in our case). However, when the filter moves to a position where there is no left-hand curve, the result of the convolution is a small number (in our case — zero). At the end of moving current filter along the image will be a matrix consisting of the results of single convolutions. It should be noted that the number of filter channels should correspond to the number of channels of the original image; only then the convolution operation will produce the proper effect. For example, if the source image consists of three channels (RGB: Red, Green, Blue), the filter must also have three channels.[3]

The filter can move along the input matrix in steps other than one. The amount of movement of the filter is called strydom (stride). Stride specifies the number of pixels you must move the filter in one sitting.

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

Figure 3.2: Allows you to calculate the number of output values after the rollup operation

In order for the training of weights enclosed in cores to be effective, some bias (bias) and nonlinearity should be introduced into the convolution results. Offset is a static value by which to "shift" the output values. At its core, this is a common operation of adding each element of the output matrix with the amount of displacement. If you explain very superficially, it is necessary in order to bring the neural network out of deadlocks, having purely mathematical reasons. The nonlinearity of the activation function. Thanks to it, the picture formed by the convolution operation receives some distortion, allowing the neural network to more clearly assess the situation. This need can be roughly compared to the need for people with low vision to wear contact lenses. In General,this need is due to the fact that the input data is nonlinear in nature, so you need to deliberately distort the intermediate results to the neural network response was appropriate.

Relu (Rectified Linear Unit) is often used as an activation function.

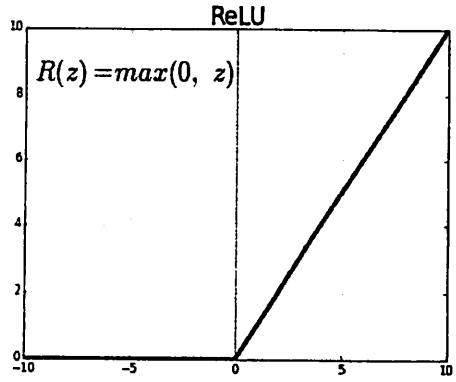

Figure 3.3: ReLU function

As should be obvious, this component is very basic. Info esteems not exactly or equivalent to zero are changed over to zero; values more prominent than zero are not changed. Typically, more than one filter is used in convolution layers. When this occurs, the results of each filter are collected along some axis, resulting in a three-dimensional matrix of output. Thanks to the NumPy library, programming a bundle is easy. At the top level there will be a for loop that will apply filters to the original image. Inside every cycle, two while circles will be utilized to move the channel along the picture (horizontally and vertically). Element product of the filter pixels every time and the image fragment (operator*) will be obtained. The resulting matrix will be collapsed by summation into a number to which the bias (bias) will be added.

The value of filt is initialized using normal distribution. Bias is a zero-filled vector. After one or two convolutional layers, the intermediate result is usually reduced. Look at the reduction as a decrease in the amount of data with the preservation of the General sense. In machine learning to determine the reduction is often used the term downsampling.

```
def convolution(image, filt, bias, s=1):

    (n_f, n_c_f, f, _) = filt.shape
    n_c, in_dim, _ = image.shape

    out_dim = int((in_dim - f)/s)+1
    assert n_c == n_c_f,

    out = np.zeros((n_f,out_dim,out_dim))


    for curr_f in range(n_f):
        curr_y = out_y = 0

        while curr_y + f <= in_dim:
            curr_x = out_x = 0

            while curr_x + f <= in_dim:

                out[curr_f, out_y, out_x] = np.sum(filt[curr_f] * image[:,curr_y:curr_y+f,
                                                curr_x:curr_x+f]) + bias[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return out
```

Figure 3.4: Convolves filt over images

3.2 Two-dimensional convolutions

Two-dimensional convolution is a operation, which is begins from a kernel, with a small size matrix with some weights. This kernel "moves" over the two-dimensional matrix resulting matrix "collapses" by summing its elements, forming a so-called "pixel".

Kernel repeats the described operation for each position occupied by it, turning the initial two-dimensional matrix of characteristics into some other — also two-dimensional — matrix. The obtained characteristics are actually the sums of the weighted input characteristics, where the weights are the elements of the kernel matrix, and the initial data are the elements of the matrix that are under the kernel at its given position.

If we draw an analogy, the input characteristics are "sifted" through the core, forming a matrix at the output, each element of which corresponds to a certain fragment of the original data. Occurs that core's size is directly affects the number of source elements that will be "sifted" and eventually turned into a pixel of the resulting matrix.

Now let's look at all this in the context of a fully connected layer of a conventional neural network. In the example above, there are 5x5=25 input and 4x4=16 output characteristics. If it would be a fully connected layer then the weight of matrix of 25x16=400 parameters should be occurred; the response of this layer

would be formed by summing the products of each input signal by some weight coefficient. (The activation function has been omitted for simplicity.) Convolutions allow to perform such operations only over sixteen parameters — with each output signal. The above means that folding is a way of obtaining an approximate version of the input data; and this contributes to a significant reduction in the number of calculations.

Keep this information in mind, as it is very important for further immersion in the topic of deep learning. Some commonly used techniques Before moving on, we should consider two basic techniques used to build convolutional layers: padding (filling) and stride (step).

Padding

Note how the edges of the original matrix are "cut" in the process of moving the kernel: the edge pixels never fall into the center of the kernel, due to the accepted rules of movement. This is not always acceptable, especially when the dimensions of the source and result matrices must be the same. This problem is solved by padding, which is the essence of "false" pixels, the value of which is usually zero (such padding is usually called "zero"). Due to the additional space around the original matrix, the kernel is able to bypass the edge pixels to the full extent. Note that the dimensions of the source and result matrices are the same.

Stride

Quite often, the task of the convolution layer is to reduce the dimension of the matrix of input signals. This is the basis of convolutional neural networks, in which the pulses entering the system are reduced at a rate directly proportional to the number of so-called channels. To gain that result, to use of a pudding (Assembly) layer, which in its essence is a conventional "sieve for sieving", which contributes to the reduction of the number of pulses.

The stride principle is based on the idea of kernel skipping a certain number of steps when it is moved. A strite with a value of 1 indicates that the kernel is shifted by one pixel, which leads us to the standard convolution. With a Strider with a value of 2, the kernel is shifted by two pixels, reducing the size of the resulting matrix by about half compared to the previous option e.t.c. In modern architectures of neural networks, such as ResNet, finally abandoned the pooling layers in favor of strides.

Multi-channel version

Everything we discussed above applies only to images with one input channel. But, as we know, images usually have three channels (RGB: Red, Green, Blue); the deeper a neural network is, the more channels it is capable of processing. It is easier to think of channels as some "components" of the image, each of which either suppresses or enhances certain features of the image. Look Neural network part. Now it is possible to trace the difference between the single-channel and multi-channel versions of the neural network. In the first case, when only one image channel is considered, the concepts of "filter" and "core" are interchangeable. In the second variant, when several channels are taken into account, these concepts acquire different meanings. In this case, each core from the collection is unique. Filters in convolution layer generates one output channel. The principle of operation of the filter is described below. Each of the cores included in the filter moves along the corresponding output channel, giving the processed version of the signals. Some kernel may have a stronger weight than the others, giving preference to specific input channels. Further, the results of each of the cores are summarized in order to obtain a single channel. Kernels generate their own versions of the channels, which in the complex is the resulting output channel. To complete the process, you must apply an offset (bias) to the result. In convolutional neural networks, the offset is applied to each output filter; this operation is the usual addition of the offset with each of the signals of the resulting (output) channel. The above sequence of actions is similar for any number of filters: each filter performs independent processing of the input image using its own system of cores and offsets described above, generating some response. Subsequently, these responses are summarized, resulting in a global response; the number of channels of this response corresponds to the number of filters. The global response can be passed through a nonlinear transducer and transferred to the next convolution layer, where the process is repeated.

Downsampling

In order to fasten learning process and decrese the consumption of computing resources, downsampling of initial and/or intermediate data is carried out, as mentioned in the previous section. There are several ways to do this. In this article we will consider the simplest and most common of them — the maximum Union (max pooling). The maximum join operation is to move the so-called sifting window along the data. From the pixels that fall into its field of view,

the maximum is selected and moved to the resulting matrix. The stride for this window can be different from one; it all depends on what size of the output matrix you want to get, and with what degree you need to "compress" the data. It clearly shows the operation of the maximum Union: a window of dimension 2x2 moves along the matrix; stride has a value of 2. As you remember, stride determines the step to which the window moves in one step. At each stage, the maximum value is selected. The original matrix is allegedly sieved through a sieve, which gives out only characteristic pixels.

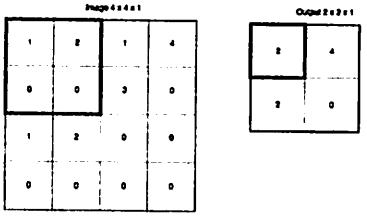

Figure 3.5: Normal convolution(step 1)

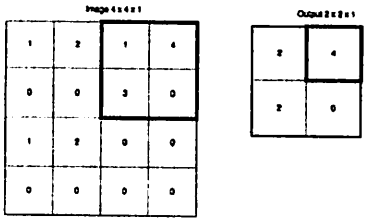

Figure 3.6: Normal convolution(step 2)

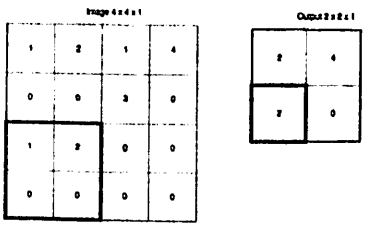

Figure 3.7: Normal convolution(step 3)

Thanks to the maximum combination, the number of pixels is reduced, which in turn leads to a reduction in the number of operations performed by the program, and, accordingly, to the savings in computing resources.

The operation of maximum unification has some other side effects of a positive nature. Thanks to it, the neural network is able to concentrate on the really weighty characteristics of the image, discarding insignificant details. Also,

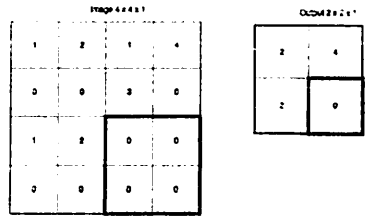

Figure 3.8: Normal convolution(step 4)

$$n_{out} = floor(\frac{n_{in} - f}{s}) + 1$$

Figure 3.9: Convolution outputs

the neural network becomes less susceptible to retraining, which often becomes a very time-consuming problem. The mechanism of convolutional layers is somewhat simple. But, no matter how hard you try, it is unlikely that you will be able to draw any significant analogies with the feed-forward layers (layers of direct distribution, or layers with direct connections; all this is the same). It is also unlikely that you will be able to get a clear rationale for why convolutional neural networks are so well suited for image processing. Something like that: it works — period! Imagine that you have a 4x4 input matrix and you want to transform it into a 2x2 matrix. If you were using a neural network with direct links, you would first have to expand the input matrix into a vector containing 16 signals. Then this vector should be passed through a system with 16 inputs and 4 outputs. Further, the matrix W of synaptic weights within layers of input and output and has been considered neural network is presented: The concept of cores used in

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} & w_{1,6} & w_{1,7} & w_{1,8} & w_{1,9} & w_{1,10} & w_{1,11} & w_{1,12} & w_{1,13} & w_{1,14} & w_{1,15} & w_{1,16} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} & w_{2,6} & w_{2,7} & w_{2,8} & w_{2,9} & w_{2,10} & w_{2,11} & w_{2,12} & w_{2,13} & w_{2,14} & w_{2,15} & w_{2,16} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} & w_{3,6} & w_{3,7} & w_{3,8} & w_{3,9} & w_{3,10} & w_{3,11} & w_{3,12} & w_{3,13} & w_{3,14} & w_{3,15} & w_{3,16} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & w_{4,5} & w_{4,6} & w_{4,7} & w_{4,8} & w_{4,9} & w_{4,10} & w_{4,11} & w_{4,12} & w_{4,13} & w_{4,14} & w_{4,15} & w_{4,16} \end{bmatrix}$$

Figure 3.10: Matrix of synaptic weights between input and output layers

convolutional layers may seem confusing at first glance. Although all kernel does is perform the same transformations that occur in direct distribution networks. The only difference is that in the first case the two-dimensional matrix of signals is used, and in the second — one-dimensional. If for our problem we used a kernel K of dimension 3x3 for the input matrix 4x4 in order to obtain a matrix 2x2 at the output, the corresponding weight matrix would take the following form: Although

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} & 0 \\ 0 & 0 & 0 & 0 & 0 & k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

Figure 3.11: Pseudometric weights generated by the movement of the kernel

the dimensions of the matrices are identical, they are formed using different approaches. In the first case, the matrix is a characteristic of multiple connections between one-dimensional layers, in the second case — a characteristic of the kernel in two-dimensional space.[1] We came to the conclusion that convolutions are also linear transformations, but at the same time they are different from those to which we are accustomed in "normal" neural networks. For the input matrix of 64 elements, there are 9 parameters that will be reused (kernel parameters). Each node of the output matrix is formed on the basis of only a certain number of input signals — those that fall into the field of view of the kernel. Multiple connections are destroyed by zero weights; this is seen in pic. 5. It is important to understand that the convolution operation is just a rough approximation of what happens when the input signals are "weighed" by weight coefficients. And yet in these two versions use some of the predefined parameters. The matrix of weight coefficients, as you know, characterizes connections between layers of a neural network. That's why connections determine how a neural network works. No wonder they say that all the power in the connections! Fully connected networks perform their tasks well. But they have serious performance issues. The number of links with the addition of new nodes is growing incredibly fast, which complicates both the direct passage of signals and the reverse propagation of the error. Convolution networks work much faster. There is no need to "train" all 64 parameters, as many of them are zero. The kernel has a dimension of 3x3, and therefore only 9 of its parameters will be amenable to training. This is important, especially when training data from MNIST with 784 inputs is replaced by real input images with dimensions of 224x224x3 (150528 inputs). Even if your fully connected system will divide input matrices in half in order to optimize, processing 75 264 signals in one approach, it will have more than 10 billion parameters! For comparison, the fully connected neural network ResNet-50 around 20 million options. It is clear that the transformation of some parameters of the system to zero — this is a great performance optimization. But how can we know if this is good for pattern recognition? The answer lies in the combinations of the charac-

teristics of a neural network. A well-chosen combination is the result of the use of an effective learning mechanism. If a convolutional neural network were used to analyze some other type of data, such as mobile app installation statistics, it will be a real disaster. And all because the data of this kind do not have local characteristics, which together form a complete picture. Therefore, you should pay special attention to the data that you plan to pass through the convolutional neural network. A response based on incompatible data may not be suitable for interpretation. To avoid such errors, you need to be well versed in the mechanism of the convolutions. Image pixels, on the other hand, are always a consistent structure. Here's the proof: the pixels surrounding the pixel in question have an effect on it; that is, if the surrounding pixels are red, then the pixel in question is likely to be red too. If this is not the case, there is an anomaly that can be interpreted as a special kind of characteristic. The conclusion is that the required characteristics can be found by mutual comparison of the pixel beam. For a space without explicit boundaries such as the sky, most pixels have values close to each other. Therefore, the kernel response for such plots will be zero or very close to it. For a space with vertical boundaries, there will be differences within some pixel bundles. Kernel detects that the difference of the pixels in such beams will not be equal to zero that indicates the border. The Sobel operator is based on a kernel of dimension 3x3. This makes it possible to find local anomalies. Moving this kernel along the entire image leads to finding all the boundaries that a particular model can detect. A very serious question is brewing: how to train the kernels so that they work as we need? To do this, you need to understand how detection of basic characteristics works: borders, lines, etc. There is a whole section of machine learning aimed at developing ways to interpret neural network responses. One of the most powerful tools used in this field is the visualization of characteristics through optimizations. Its idea is simple: you need to optimize the picture with some initial noise, until maximum effect of the filter is achieved. Intuitively, if the optimized picture will be full of edges, it is mean that an example of how you can visualize the filter itself. Using the described approach, you can look at trained filters. It should be noted that the collapsed images are still ordinary images. The output signals obtained from a small area somewhere in the upper left corner of the image are strictly fixed to their coordinates. Further, this data can be used to extract the characteristics of a deeper level. There is one "but". The deeper a

neural network digs, the more difficult it is to extract useful characteristics from the data without additional transformations of the data. For example, it is unlikely to be able to determine the face in the area of dimension 3x3 pixels. Here comes into force the idea of a receptive field.

3.3 Receptive field

An important architectural solution used in convolutional neural networks is that the matrix of input signals decreases with each layer, and the number of filters increases in the meantime. This effect is achieved thanks to the strides and pooling layers. The principle of common features, given in the previous section, determines the appearance of the matrix of output signals by "folding" the pixel beam. Receptive field — part of the matrix of input signals subjected to "coagulation". The density of the receptive field is the amount of information that can fit within the considered pixel beam. The process of collapsing is essentially compacting the amount of data stored in a single pixel. Imagine a situation in which you have a matrix of some pulses of dimension 3x3 (i.e. 9 pieces of information). Using a convolutional layer with a single kernel of dimension 2x2 will " collapse" the input pulses into a matrix of dimension 2x2, which will give 4 units of information. As you can see, the information previously stored in nine cells is now placed in a smaller space, namely four cells. But lossless is not enough. As you remember, stride is a fixed step of moving the kernel along the matrix of input signals. It is important to remember that stride is a good way of pooling signals (combining to reduce the dimension). If you look at the process of pooling at a special angle, you can see the assimilation of signals with their environment.

After convolution, the received signals are usually transformed non-linearly, and then they move to the next layer of the neural network, mainly convolutional. At this stage, the situation is heating up, because the deeper the data goes, the more interesting the results will be at the output. The receptive field is compacted and compacted. Even if we continue to use a 4x4 kernel, it will get more information with each compression level. To expand the field of view it makes no sense to increase its dimension. It's all about assimilation: a small area of "rolled" data will show the characteristics of a large number of assimilated signals in them Let's look at this issue in more detail. The response of the convolution layer still characterizes the original image, representing its compressed version. It is important to understand that we are not talking about croping. In our case, each pixel from the matrix of "collapsed" signals corresponds to a group of pixels of the original image. You can even approximate which group of source signals a given pixel corresponds to. Local output: the kernel of each subsequent

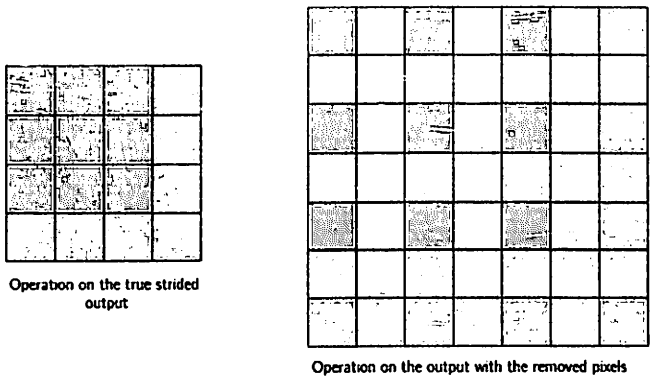

Figure 3.12: Assimilation of the pixel with its surroundings

layer interacts with signals, each of which is a generalized characteristic of a certain group of pixels of the original version of the image. Notice one thing. If you are familiar with dilated convolutions, you should understand that the above has nothing to do with them. Both of these approaches are aimed at compacting the receptive field. The difference is that the extensible convolution solves the problem by means of an additional layer of the neural network, whereas the approach described in this section is based on the use of strides as well as nonlinear transformations in the intervals. Seal receptive fields of convolutional layers gives the possibility to transform low-level features like lines and borders to feature more high-level. Using transaction pooling/staging, neural network as "immersion" generates increasingly sophisticated detectors, high-level features as fragments and patterns.

Imagine a stepwise reduction of the original image with the dimension 224x224 by a neural network with five convolutional blocks, which gives a matrix with the dimension 7x7. Each pixel of the resulting data is a characteristic of a 32x32 pixel grid corresponding to the original image.[1]

Layers in the early levels are activated when low-level elements, such as lines and boundaries, are detected. Layers of deeper levels deal with high-level elements and are able to detect, for example, a bird or a dog. Convolutional neural network usually begins with a small number of filters which are engaged in the detection of low-level characteristics, and gradually moves to a larger number of filters which have 1024 filters in the final stage of convolution, each of which is engaged in the search for some specific high-level images. All will end with a pooling layer that "collapses" each grid of 7x7 pixels into a single pixel. Compared to the capabilities of conventional neural networks of direct propagation,

the results of convolutional architectures are truly amazing. A direct distribution network would build abstract vectors of image characteristics, requiring significantly more computing power and an "unaffordable" amount of data for effective learning. Convolutional neural networks begin their work by trying to recognize low-level characteristics of the image. As the dive progresses, the receptive field thickens, allowing the network to combine low-level characteristics and recognize increasingly complex high-level things such as patterns and textures. It is difficult not to notice that convolutional neural network builds some visual hierarchy of the input image. In the process of learning these visual hierarchy are specified, accumulate and persist in the parameters of the kernels. Starting with the analysis of low-level characteristics and using them to build visual hierarchies and search for high-level target features, convolutional neural network eventually becomes able to distinguish complex visual concepts: faces, birds, trees, etc. This is what makes this architecture so effective in the analysis of graphic material.

Given the visual hierarchies built by convolutional neural networks, it is tempting to conclude that their way of evaluating visual images is close to human. Convolution networks are really good when it comes to images obtained naturally. But there are conditions in which they make banal mistakes, in which a person can not make a mistake exactly. The most common problem is adversarial attacks. The "attack" of this kind is that noise is deliberately introduced into the image, designed to confuse the neural network. Such "attacks" would not be a problem if they confused both man and machine. The trouble is that neural network models are susceptible to even the most minor distortions of the original image, where a person is not able to notice the catch. All this leads to situations in which the neural network blatantly lies, which is unacceptable for a wide range of machine vision tasks: self-driving cars, medicine, etc.

Programming "maximum unification"

The implementation of the maximum join operation is reduced to the application of the for loop at the top level, within which there will be two loops. "For" needed to iterate through all the channels of the picture. The task of while loops is to move the "sifting window" along the image. At each stage of the selection pixel of the captured fragment of the matrix will be applied the max function that is part of NumPy with the aim of finding the maximum value. Further, after several convolutional layers and a downsampling block, the three-dimensional

```
def maxpool(image, f=2, s=2):
    """
    Downsample input `image` using a kernel size of `f` and a stride of `s`
    """
    n_c, h_prev, w_prev = image.shape

    # calculate output dimensions after the maxpooling operation.
    h = int((h_prev - f)/s)+1
    w = int((w_prev - f)/s)+1

    # create a matrix to hold the values of the maxpooling operation.
    downsampled = np.zeros((n_c, h, w))

    # slide the window over every part of the image using stride s. Take the maximum value at each step.
    for i in range(n_c):
        curr_y = out_y = 0
        # slide the max pooling window vertically across the image
        while curr_y + f <= h_prev:
            curr_x = out_x = 0
            # slide the max pooling window horizontally across the image
            while curr_x + f <= w_prev:
                # choose the maximum value within the window at each step and store it to the output matrix
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+f, curr_x:curr_x+f])
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
    return downsampled
```

Figure 3.13: The function that executes the "maximum Association"

representation of the image will be expanded into a vector, which will then be transferred to a multilayer perceptron — a conventional fully connected neural network.

Fully cohesive layer

At this stage of data transformation the three-dimensional matrix of signals will be expanded into a vector which will be passed through a fully connected neural network. Its task is to tell us the probabilities of what digit the input image represents. deployment operation consists in "glueing" rows into a single — huge length — the number series. It will be a really big vector, which will need to be transformed with the help of a multi-layer network with full connections! If you do not know how a fully connected layer works, here is a simple description of the mechanism: each element of the vector is multiplied by the weight of the connection, these products are further summed together and with some displacement, after which the result is transformed by the activation function. For pic. 10 the above is presented in a visual form.

4. Algorithms

4.1 BackPropagation Algorithm

A multilayer perceptron is able to solve arbitrary problems. But for a long time its use was difficult due to the lack of an effective learning algorithm. With multiple layers of configurable weights, it is not clear which weights to adjust based on the output network error. The procedure of perceptron learning (section 1.3) at the same time stopped working. In 1986, Rumelhart, Hinton, and Williams proposed a so-called back propagation algorithm. Training of multilayer perceptron based on the minimization of the error function E(w). The error determines the deviation of the desired t_n network outputs from the resulting y_n. Typically, the function E(w) is determined by the least squares method: $E(w) = \frac{1}{2}\sum_{n=0}^{\infty}(t^n - y^n)^2$ The error function (2.2) is minimized by the gradient descent method known from the theory of numerical methods. The essence of the method is that moving in the direction opposite to the gradient of the function $\triangle w_{ij}$, in the end we approach the minimum, possibly local, function E(w):

$$\triangle w_{ij} = -\eta^{\frac{\partial E}{\partial w_{ij}}}$$

Here $0 < \eta < 1$ sets the learning rate; w_{ij} is the coupling coefficient i of the neuron layer $n-1$ with the j neuron of layer n. Find a way to calculate the gradient $\frac{\partial E}{\partial w}$. We will write down formally the scheme of work of multilayered persep-throne's. Each neuron calculates a weighted sum of its inputs:

$$a_i = \sum_{n=0}^{\infty} w_{ij} z_i \quad (2.1)$$

Here z_i is the inputs of the neuron and, accordingly, the outputs of the previous layer of neurons (Pic. 3). The output of neuron j is the transformation of sum - we a_i threshold function g: $z_j = g(a_j)$. Let us return to $\frac{\partial E}{\partial w_{ij}}$ and note that E(w) is a complex function for which $w = w(a_j)$ is a function of a_j. Hence, according to the rule of calculating the derivative of a complex function, we have

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E \partial a_j}{\partial a_j \partial w_{ji}}$$

By using (2.1) and consider the expression $\delta j = \frac{\partial E}{\partial a_j}$ is an error on layer j. As a result, you can see that to calculate the gradient on each layer i, the error value of the previous i, located on top of layer j, is calculated by the input value of the layer:

$$\frac{\partial E}{\partial w_{ji}} = \delta_j z_i$$

One run of error propagation is commonly called an epoch. The complexity of the error back propagation algorithm is defined as the value of the order $O(W)_N$. Here W is the number of configurable network values; N is the size of the training sample. The error function E(w), due to the fact that nonlinear threshold functions are used in the neural network, is a freely complex "gully" surface with a large number of legal minima. If at gradient descent to get to such a minimum, then, obviously, the network will not be configured for optimal productivity. Fortunately, the methods of optimizing the search for the minimum of functions are studied in detail in the theory of numerical methods, and some of them are transferred to the back propagation algorithm.

4.2 Improvement of back-propagation algorithm

1) the Simplest way is to use variable learning rate η. At the beginning of the algorithm, its value is a large value, close to 1, as the convergence η consistently decreases. This allows you to quickly approach the minimum, and then accurately get into it without unnecessary "Gating»; 2) "Gully" method. Takes into account trends in the surface before the addition of the moment of inertia μ : $\triangle w_n = -\eta\triangle E + \mu W_{n-1}$. Here W_n is a matrix of values of new weights; $W_{n^{\wedge}1}$ is a matrix of values of weights on the previous epoch; $\triangle E = \frac{\partial E}{\partial W_n}$. The idea is to jump through the sharp "dips" in the surface error is the local minima;[2] 3) conjugate gradient Method. Fletcher and Reeves proposed to choose the direction conjugated to the gradient, more accurately indicating it at least functions:

$$\triangle W_n = -\eta\triangle E + \beta W_{n-1}$$

, when $\beta = \triangle E_n|\frac{2}{|\triangle E_{n-1}|_2}$

4) the Most accurate solution – a solution that allows you to get the so-called methods of the second order. The General principle of operation is based on the use of the matrix of second derivatives of the Hessian $H = \triangle 2E$ The gradient may not always specify in the direction, opposite to the false minimum, and the value $H-1\triangle E$ the direction of Newton, will be always directed in the direction of the minimum. Hence, the change of weights occurs according to the rule $\triangle W = -\eta H - 1\triangle E$ But Hessian himself is quite difficult to calculate, so methods are used where the value of H is considered approximate. The best performance method is the Levenberg – Marquardt method, in which Hessian is approximated as H = JT_J+_J; J is a Jacobian which is much easier to compute. The weight correction procedure takes the form $\triangle W = -\eta(J_T J)-1(J_T)$. The complexity of this method is the value of the order $O(W_2)_N$. the Method is very effective and for a neural network of medium size – the minimum error is usually found in several epochs. A comparison of the performance of these training methods is given in. Multilayer perceptron is by far the most common and studied type of neural networks. It is widely used for classification tasks, building expert networks. One of the unsolved problems of multilayer perceptron, like most classes of neural networks, is the uncertainty of topology selection (number of layers, neurons in layers).

Example of using a multilayer perceptron Task: There is an encrypted message. Known cipher: each letter was put in accordance with the previous alphabetically letter. Each letter is in an area of 6×6 pixels. You want to decrypt the message. The training input and output vectors will be created as follows. If some pixel of the letter is colored, the corresponding component of the input or output vector takes the value 1, otherwise – 0. Since the input and output vectors are encoded by 36 bits, the network has 36 input and 36 output neurons. For training we use the algorithm of back propagation with the subsequent reduction by OBD method. Let there be 16 neurons in the hidden layer. After training, the process of excluding weights and neurons that have the least impact on the classification of images should be performed to increase the generalizing capacity of the network. As a result of training, in hidden layer there will be 8 neurons remain of the network, thus minimizing the impact of noise on the classification result. Examples of the trained network operation on some noisy symbols.

4.3 Monoplan

The algorithm which consist form input data, one output and only one layer. Algorithm could use any suitable method. At the training, each time when there occurs error, network creates new neuron(weight) and add to layer, retrain it till the error in output comes to minimal. Most suitable method is Backpropagation.

. /* train the hidden units */
h = h + 1; /* connect hidden unit h to the inputs */
learn the training set $\{\vec{\xi}^\mu, \tau_h^\mu\}$, $\mu = 1, \cdots, P$;
after learning, $\sigma_h^\mu = \text{sign}(\vec{w}_h \cdot \vec{\xi}^\mu)$, $\mu = 1, \cdots, P$;
if $h = 1$ /* for the first hidden neuron */

if $\sigma_1^\mu = \tau_1^\mu \; \forall \mu$ **then stop.** /* the training set is LS */;
else set $\tau_{h+1}^\mu = \sigma_h^\mu \tau^\mu$ for $\mu = 1, \cdots, P$; **go to 1;**

Figure 4.1: Monoplan

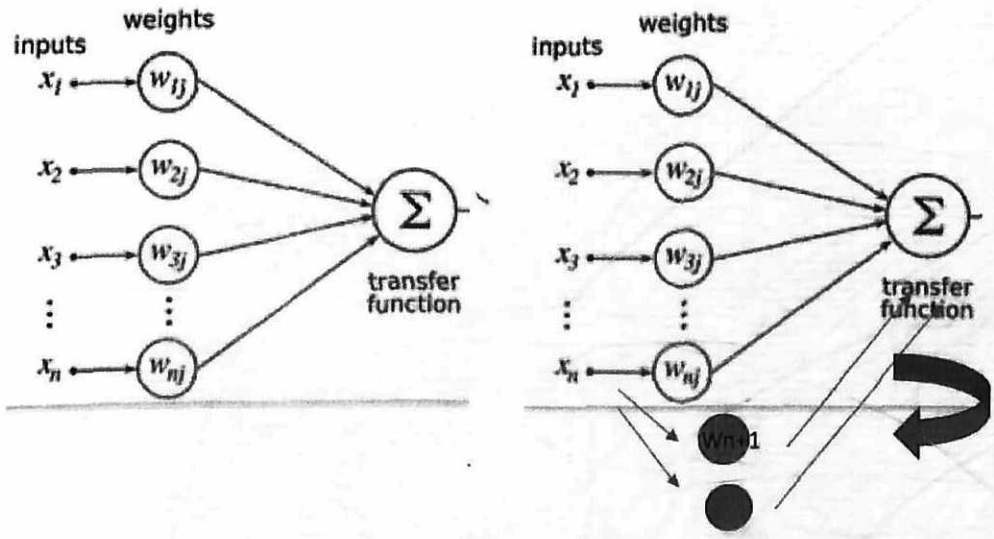

Figure 4.2: Monoplan

4.4 NetLines

During the training process, all neurons divides each hidden unit to classify output neurons and reduce the number which could appear as classification error. NetLines growth by using the simple perceptrons to be a part of training algorithm. $\varsigma_{(h)} = sign(\sum_{k=0}^{h})W_k\sigma(h) = sign[W(h) * \sigma(h)]$. The Adaptive Boosting method is similar to NetLines algorithm, but not flexible enough as NetLines.

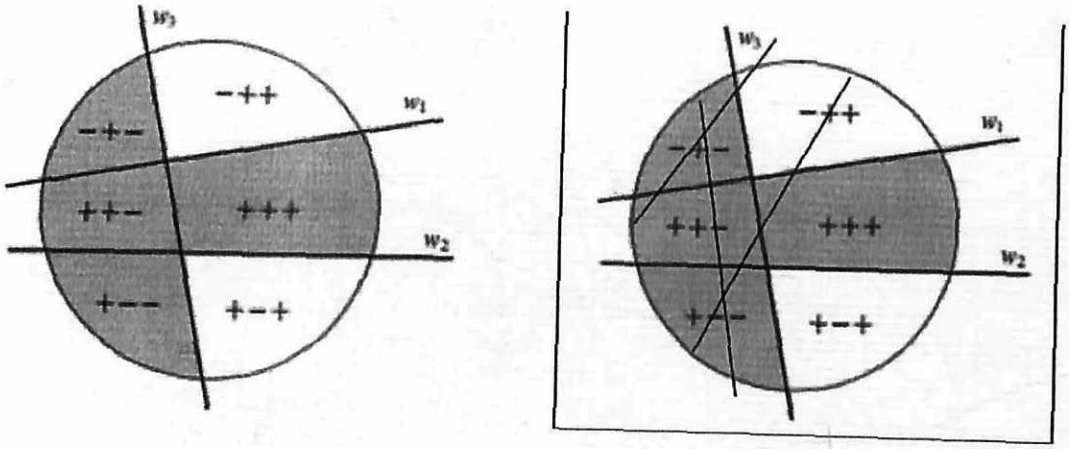

Figure 4.3: NetLines

4.5 NetSphere

The NetSphere algorithm only differs form NetLines algorithm, in the form of a threshold function $g(a) = |\sum_{i=1}^{N}(w_i - x_i)^2 - \Theta^2|$. Where is the radius of the circle, w_i are the neuron weights that define the center of the circle. Thus, the dividing surface will have a more complex appearance and significantly improve the results.

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                         input_shape=(32, 32, 3), activation='relu'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(X_train, Y_train,
              batch_size=batch_size,
              epochs=nb_epoch,
              validation_split=0.1,
              shuffle=True,
              verbose=2)

scores = model.evaluate(X_test, Y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Figure 4.4: CIFAR-10

5. Conclusion and Result

To make the realization of the algorithms, I decide to do classification of pictures, which I took from data CIFAR-10, to be able to make comparation with standard neural structure based on library called TensorFlow. The accuracy of standard structure gived us accuracy in 78.29 percent , while in algorithm of monoplan used with help of backpropagation method gave us 82.36 percent. It is not the best result but it had caused because of little data. CIFAR-10 got only 10000 picture to train algorithm, so with bigger data, the accuracy will definitely increase.

The algorithms is definitely worthy for usage, it does not need to much CP force and will not overfit the RAM of PC. It could be called "Budget version". Moreover, the accuracy is higher that in standard structures. However, it could need a little more time to make a compilation and a little bit harder to give new inputs.

References

[1] S.E. Fahlman and C. Lebiere. "The cascade-correlation learning architecture". In: *In Advances in Neural Information Processing II* 2 (1990), pp. 524–532. DOI: ftp://archive.cis.ohio-state.edu/pub/neuroprose/fahlman.cascor-tr.ps.gz.

[2] R. M. Goodman and P. Higgin. "Rule-based neural networks for classification and probability estimation ". In: *Neural network* CA 94143-0560 (1992). DOI: http://dx.doi.org/10.1002/andp.19053221004.

[3] Harris. *ISingle-layer learning revisited: a stepwise procedure for building and trainig a neural network.* Neurocomputing, algorithms, architectures and applicatons. Elsevier, 1990.

[4] Torres-Moreno. "Apprantissage et generalisation par des reseaux de neurones: etude de noveaux algorithmes constructifs". In: *Institut National Polytechique de Grenoble* 2.10 (1997), pp. 891–921. DOI: ftp://archive.cis.ohio-state.edu/pub/neuroprose/torres.thesis.ps.Z.

[5] В.В. Иванов, Б. Пурэвдорж, and И.В Пузырин. "Методы второго порядка для обучения многослойного перцептрона". In: 10.3 (1998), pp. 117–124.