UDC 004.93

YU.YA. TOMKA, M.V. TALAKH, V.V. DVORZHAK, O.G. USHENKO

IMPLEMENTATION OF A CONVOLUTIONAL NEURAL NETWORK USING TENSORFLOW MACHINE LEARNING PLATFORM

Yuriy Fedkovych Chernivtsi National University,
2 Kotsjubynskyi Str. Chernivtsi, Ukraine, e-mail: y.tomka@chnu.edu.ua

Анотація. Розглянута узагальнена алгоритміка реалізації типової архітектури згорткової нейронної мережі засобами бібліотеки машинного навчання TensorFlow. Проаналізовані особливості кодової імплементації згорткової нейронної мережі у задачі розпізнавання зображень на прикладі датасету MNIST.
Ключові слова: комп'ютерний зір, згорточна нейронна мережа, CNN, глибоке навчання, класифікація зображень, розуміння зображень

Abstract.The generalized algorithm of a typical convolutional neural network realization by means of TensorFlow machine learning library is considered. The peculiarities of the coding implementation of the convolutional neural network in the image recognition problem are analyzed with the example of the MNIST dataset.
Keywords: Computer Vision, Convolutional Neural Network, CNN, Deep Learning, Image Classification,Image Understanding

DOI: 10.31649/1681-7893-2022-44-2-55-65

INTRODUCTION

Convolutional artificial neural networks (ACNN) are a special type of multilayer neural networks, the main purpose of which is to recognize visual patterns in an image with minimal preprocessing of the latter. In recent years, the development of this type of neural networks has undergone intensive development, the result of which was the appearance of a number of different architectures of convolutional neural networks, each of which is characterized by its own features, quality and speed of work (Figure 1):

1. LeNet [1]. The first successful application of a convolutional neural network was made by Jan Lekun in the 1990s. The LeNet architecture was used to read postal codes, numbers, etc.
2. AlexNet [2]. This is the work of Alex Kryzhevsky, Ilya Sutzkever and Jeff Hinton, which played a significant role in popularizing CNN in the field of computer vision. The AlexNet architecture was presented at the ImageNet ILSVRC Challenge in 2012 and outperformed all competitors (16% errors versus 26% for the second-place architecture).
3. ZF Net [3]. The winner of ILSVRC 2013 was the convolutional neural network of Matthew Zeller and Rob Fergus, which is better known as ZF Net (abbreviation from Zeiler and Fergus). This architecture was an improved version of AlexNex: the size of the middle convolutional layers was increased and the pitch and size of the filter on the first layer were reduced.
4. GoogLeNet [4]. In 2015, the aforementioned competition was won by CNN developed by Sheged and other employees of the Google Corporation. The main merit of this architecture is the development and implementation of the input module (Inception Module), which made it possible to drastically reduce the number of parameters to 4 million from 60 million. The reduction of parameters also occurs due to the replacement of fully connected layers in the upper part of the network with layers of medium pooling.

© YU.YA. TOMKA, M.V. TALAKH, V.V. DVORZHAK, O.G. USHENKO, 2022

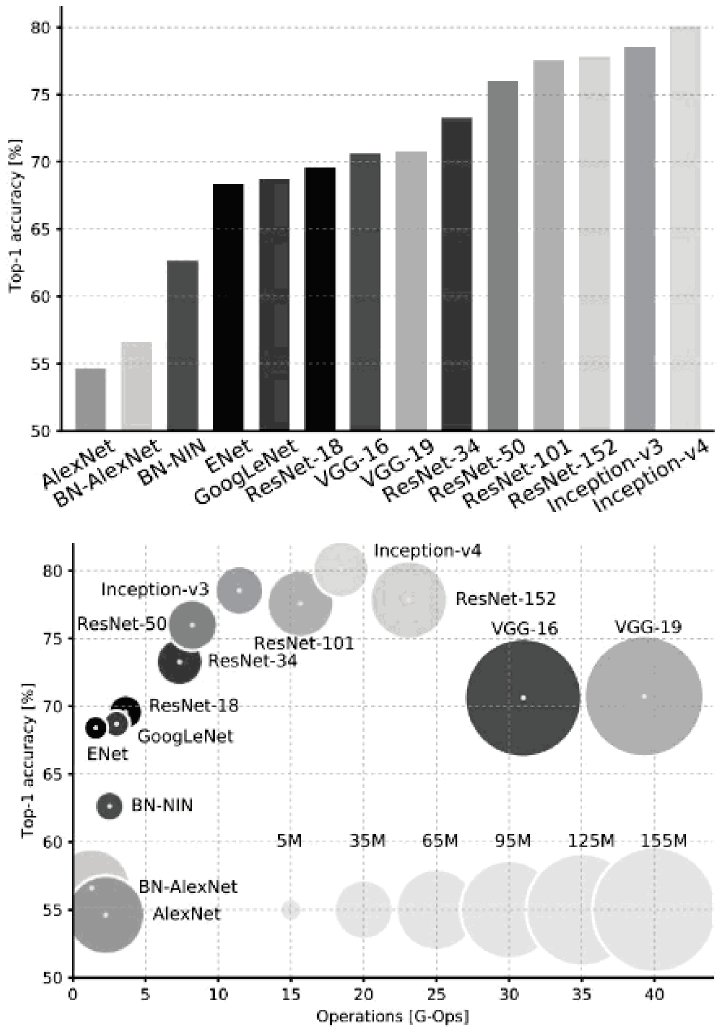


Figure 1 – Analysis of the quality of work and deep neural networks in 2017

5. VGGNet [5]. Just behind GoogLeNet at ILSVRC 2014 was the network of Karen Simonian and Andrew Zisserman, who managed to clearly demonstrate that depth is a key factor in performance. Their network contains 16 convolutional and fully connected layers and has an extremely uniform architecture that performs 3×3 convolution and 2×2 pooling end-to-end.

6. ResNet [6]. The Residual Network, developed by Kaiming He and others, was also the winner of ILSVRC 2015. The key features are the intensive use of packet normalization [7] and special skip connections. There are no fully connected layers at the end of the architecture. ResNet is currently used most often.

Unlike a multilayer perceptron, convolutional neural networks allow taking into account the topology of images and are invariant to shifts, scaling, and other distortions of the input image. As we can see, at the present time their evolution is taking place along the path of increasing the number of layers (depth), as a result of which deep convolutional neural networks (deep CNNs) appeared. Due to the deep architecture, such networks allow to achieve high accuracy of recognition of different types of images.

1. WHAT IS A CONVOLUTIONAL NEURAL NETWORK?

Let's consider the CNN organization in more detail in the context of the CIFAR-10 dataset image recognition and classification system [8].

Such a network is characterized by a typical architecture: INPUT – CONV – RELU – POOL – FC:

- INPUT (input data) [32×32×3] contains upstream information about the image (for the CIFAR-10 element of the dataset: 32 pixels – width, 32 pixels – height, 3 – color channels R, G, B).

- The CONV layer (convolution layer) multiplies the filter weights by the ascending values of the image pixels (elementwise multiplication), after which the results are summed. Each unique filter position on the image generates a number.
- The RELU block (linear rectification block) applies an element-wise activation function like f(x) = max(0,x), setting a zero threshold. In other words, RELU performs the following actions: if $x > 0$, then the volume remains the previous one ([32×32×12]), and if $x < 0$, then the unnecessary details in the channel are cut off by replacing with 0.
- The POOL layer (pooling layer) performs the operation of downsampling (subsampling) of spatial dimensions (width and height), as a result of which the volume can be reduced to [16×16×12]. In other words, at this step, a non-linear compression of the feature map is performed. The logic of the work is as follows: if some features were already detected in the previous convolution operation, then such a detailed image is no longer needed for further processing, and it is compressed to a less detailed one.
- The FC layer (fully connected layer) outputs an N-dimensional vector (N is the number of classes) to determine the required class to which the input image can be assigned. The work is organized by referring to the output of the previous layer (feature map) and determining the properties that are most characteristic of the defined class.

This is how the convolutional neural network transforms the ascending image layer by layer, starting with the ascending pixel values and ending with the class definition.

Layers do not necessarily have to contain parameters. In particular, the convolutional layer and the fully connected layer perform transformations that are a function not only of the input activation volume, but also of parameters (neuron weights and displacements).

On the other hand, the linear rectification block and the pooling layer implement a fixed function. The parameters in the CONV and FC layers will be trained using gradient descent, so the class determination by the convolutional neural network depends on the labels in the training set for each image.

So:

1. At its simplest, a CNN architecture is a set of layers that transform an image into an output image (such as class definition).
2. Each layer is responsible for a certain stage of the image processing process (the convolutional layer, the block of linear rectification, pooling and the fully connected layer form the most famous architectures of convolutional neural networks).
3. Each layer receives volumetric 3D information at the input and transforms it with the corresponding storage of 3D volume using a differentiable function.
4. A layer may not have parameters (CONV and FC do, RELU and POOL do not).
5. A layer may not have additional hyperparameters (for example, CONV, FC and POOL do, RELU does not).

2. THE PROGRAMMING MODEL AND BASIC CONCEPTS OF THE TENSORFLOW (TF) MACHINE LEARNING LIBRARY

In [9], an analysis of existing open source frameworks/libraries for deep learning was carried out (Table 1). The most popular and convenient today is TensorFlow. Let's consider the basic principles of working with it.

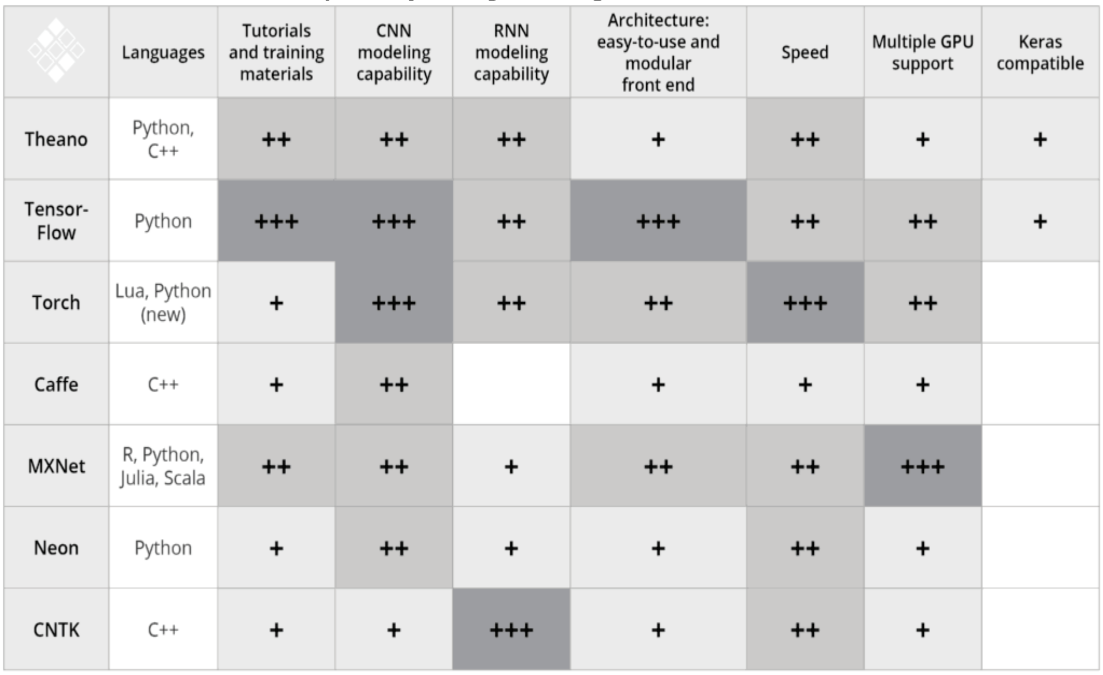
According to the official documentation [10], all work with the TF library is concentrated around the construction and execution of the calculation graph. In a TensorFlow graph, each node has zero or more inputs and is an instance of an operation. Nodes in such graphs implement arbitrary type mathematical operations or entry/exit points, while edges are multidimensional arrays of data (tensors, tf.Tensor objects) passing between nodes. Nodes can be attached to computing devices and run asynchronously, in parallel, while processing all tensors connected to them. The type of the base element of the tensor is specified or inferred when the graph is constructed.

A graph can also have special edges called control dependencies that indicate that the upstream node must complete execution before the sink node can start executing the control dependency.

Computational graphs are performed in sessions. The session object (tf.Session) hides the graph's execution context - necessary resources, auxiliary classes, address spaces. There are two types of sessions - regular sessions implemented in tf.Session and interactive sessions (tf.InteractiveSession). The difference between the two is that the interactive session is more suitable for running in the console and directly defines itself as the default session. The main effect is that the session object does not need to be passed to calculation functions as a parameter.

Table 1

Usability of major deep learning frameworks as of 2017



	Languages	Tutorials and training materials	CNN modeling capability	RNN modeling capability	Architecture: easy-to-use and modular front end	Speed	Multiple GPU support	Keras compatible
Theano	Python, C++	++	++	++	+	++	+	+
Tensor-Flow	Python	+++	+++	++	+++	++	++	+
Torch	Lua, Python (new)	+	+++	++	++	+++	++	
Caffe	C++	+	++		+	+	+	
MXNet	R, Python, Julia, Scala	++	++	+	++	++	+++	
Neon	Python	+	++	+	+	++	+	
CNTK	C++	+	+	+++	+	++	+	

To work with layers, the specified library provides the tf.layers object, which contains the following methods for creating three types of layers:

• conv2d – eceives the size of the filter and the indentation (parameter stride) of the window.

• reshape – constructs a fully connected layer. Gets the number of neurons and the activation function as arguments.

Each of the considered methods receives a tensor at the input and returns a transformed tensor at the output, which makes it possible to combine one layer with another.

3. IMPLEMENTATION OF CNN USING TENSORFLOW

Let us consider the generalized algorithm of CNN implementation in the TensorFlow software environment for the classical problem of classification of digits of the MNIST data set [11].

The input images will be images of digits from the 28x28 MNIST data set. All images are presented in grayscale. Next, 32.5×5 convolutional filters/channels plus ReLU (rectified linear unit) neuron activation function are created (Figure 2). After that, subsampling is performed using a 2×2 max pooling operation with an offset of 2 (padding). The second layer is characterized by a similar structure, but already contains 64 filters/channels and another subsampling with an offset of 2. This is followed by a fully connected layer with 3164 nodes, followed by another hidden layer with 1000 nodes. These layers will also use the ReLU activation function. At the end, a softmax classification layer is used to derive the probabilities of the 10 numbers that should be detected.

3.1. INPUT DATA AND PLACEHOLDERS

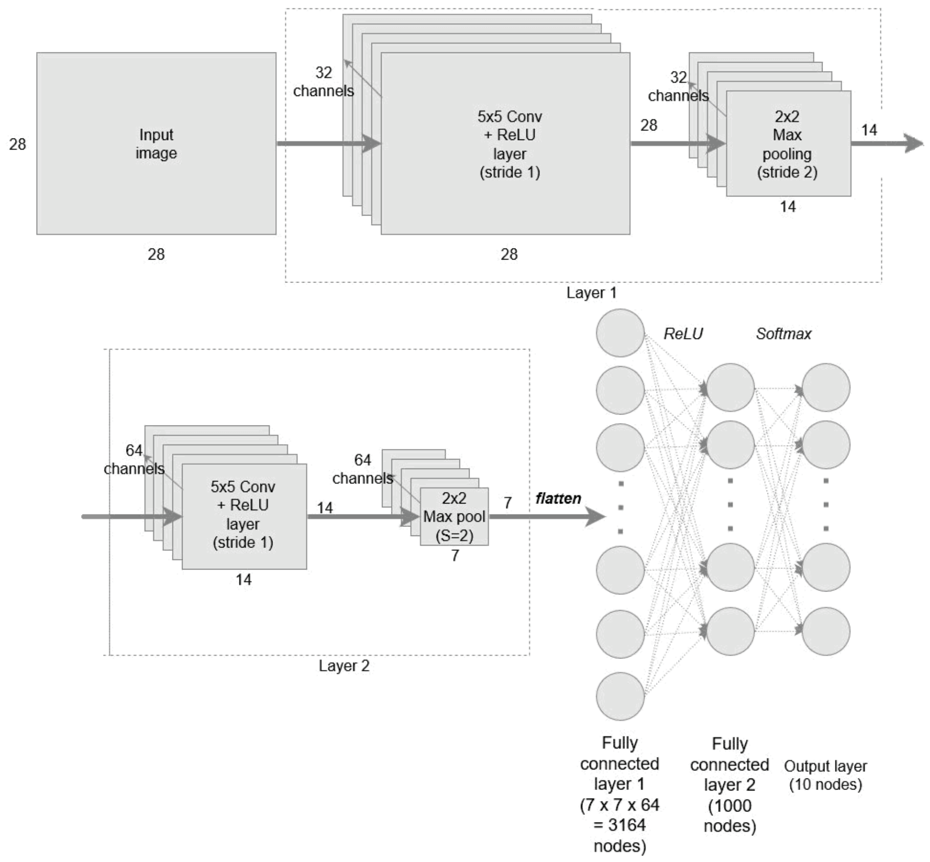
The following code initializes the input and placeholders for the classifier:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets ("MNIST_data", one_hot=True)

# Definition of hyperparameters
learning_rate = 0.0001
epochs = 10
batch_size = 50
```

```
# Placeholders for the training sample:
# Here x is an image of 28x28 pixels = 784 is a vectorized pixel-by-pixel image
x = tf.placeholder(tf.float32, [None, 784])
# processing of the input image
x_shaped = tf.reshape(x, [-1, 28, 28, 1])
# output placeholder declaration for 10 digits
y = tf.placeholder(tf.float32, [None, 10])
```

TensorFlow has a convenient loader for MNIST data, which is implemented in the first two lines, followed by a batch operation. Then the variables are initialized, on the basis of which the optimization behavior is determined (learning speed, batch size, etc.). Next, a placeholder for the input image, x, is declared. The input image will be obtained using the mnist.train.nextbatch function, which provides a 28 × 28 = 784 node single-channel grayscale image. However, before we can use the input data in the convolution and subsampling functions (conv2d and max_pool() respectively), we need to change their format because these functions work with 4D data.



Input image
28
28
32 channels
5x5 Conv + ReLU layer (stride 1)
28
28
32 channels
2x2 Max pooling (stride 2)
14
14
Layer 1
64 channels
5x5 Conv + ReLU layer (stride 1)
14
14
64 channels
2x2 Max pool (S=2)
7
7
flatten
Layer 2
ReLU
Softmax
Fully connected layer 1 (7 x 7 x 64 = 3164 nodes)
Fully connected layer 2 (1000 nodes)
Output layer (10 nodes)

Figure 2 – A typical scheme of a convolutional neural network for image classification of the MNIST dataset

The data format must follow the following structure: [i, j, k, l], where i is the number of training samples, j is the image height, k is the image width, and l is the channel number. Since the work is done with grayscale images, l will always be equal to 1 (in the case of RGB images – 3). Since the MNIST image is 28 x 28, both j and k will be equal to 28. When changing the dimension of the input data x in x_shaped, it is theoretically unknown what the dimension of x is, that is, we do not know exactly what i is equal to. tf.reshape() allows you to set the value of and equal to -1 and, as a result, the dimension will dynamically change depending on the dimension of the training images during the training phase. Thus, you can use [-1, 28, 28, 1] as the second argument to tf.reshape.

A placeholder for the output data is also required, which in fact should be a tensor with dimension [8, 10], where 10 is the number of possible digits to be classified. It is appropriate to use mnist.train.next_batch to obtain a one-dimensional vector of number labels – in other words, the number “3” will be represented by the pattern [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

3.2. DEFINITION OF CONVOLUTION LAYERS

Since it is necessary to create several layers of convolution, it is more appropriate to implement all this logic in the form of a function in order to avoid further duplication of code:

```
def create_new_conv_layer(input_data, num_input_channels, num_filters, filter_shape,
pool_shape, name):
   # set filter shape for tf.nn.conv_2d
   conv_filt_shape = [filter_shape[0], filter_shape[1], num_input_channels,
num_filters]

   # initialization of weights and displacements of the activation threshold of the
ReLU function
   weights = tf.Variable(tf.truncated_normal(conv_filt_shape,
stddev=0.03),name=name+'_W')
   bias = tf.Variable(tf.truncated_normal([num_filters]), name=name+'_b')

   # formation of the convolutional layer
   out_layer = tf.nn.conv2d(input_data, weights, [1, 1, 1, 1], padding='SAME')

   # adding an activation threshold bias
   out_layer += bias
   # application of non-lazy activation of the ReLU type
   out_layer = tf.nn.relu(out_layer)

   # application of the max-pooling algorithm
   ksize = [1, pool_shape[0], pool_shape[1], 1]
   strides = [1, 2, 2, 1]
   out_layer = tf.nn.max_pool(out_layer, ksize=ksize, strides=strides, padding='SAME')
return out_layer
```

Let's analyze in detail the presented code of the create_new_conv_layer function:

- conv_filt_shape – variable for storing the kernel weights, which will determine the behavior of the 5×5 convolutional filter. The format of the data that the conv2d() function receives for the filter is [filter_height, filter_width, in_channels, out_channels]. The height and width of the filter are determined by the filter shape variable –filter_shape (in our case [5, 5]). The number of input channels for the first convolutional layer is 1, which corresponds to a single-channel grayscale MNIST image. However, for the second convolutional layer, this number is 32, since this number corresponds to the number of outputs of the first convolutional layer. As shown in fig. 2. the number of output channels for the first convolutional layer is 32, for the second layer -¬ 64.
- the weght and bias variables are intended to store information about the weights and the activation threshold offset and randomly initialize the tensors.
- string

```
out_layer = tf.nn.conv2d(input_data, weights, [1, 1, 1, 1], padding='SAME')
```

Is responsible for initializing the operation of the convolutional filter, resulting in a convolutional layer. The input_data variable does not need an explanation like weights. The dimension of the weight tensor indicates the dimension of the filter itself. The next argument [1, 1, 1, 1] is the step parameter (offset from the current position of the filter - padding) required by the conv2d() function. In our case, it is necessary that the filter moves with a step of 1 in both x and directions. This information is passed in strides[1] and strides[2] - both values are equal to 1. The first and last values of strides are always equal to 1, otherwise the filter will move between training samples or between channels, which is not acceptable. The last parameter is the padding operation, which determines the output size of each channel, when this value is set to “SAME”, the sizes are determined as follows:

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width  = ceil(float(in_width) / float(strides[2]))
```

For the first convolutional layer, in_height = in_width = 28, and strides[1] = strides[2] = 1. In general, this procedure is designed to ensure that the filter has the ability to evenly fill the image plane as it moves, and expands the image if necessary zero columns/rows.

- in rows

```
out_layer += bias
out_layer = tf.nn.relu(out_layer)
```

a bias neuron is added for the convolutional layer, followed by the use of a non-linear ReLU activation function.

- the max_pool() function accepts a tensor (the first parameter of the function) on which the pooling operation will be performed. The next two arguments ksize and strides define the parameters of the pooling operation: ignoring the first and last values of these vectors (will always be equal to 1), the average values of ksize (pool_shape[0] and pool_shape[1]) define the shape of the window of the max-pooling algorithm in x and y directions As a rule, a 2×2 max-pooling window is used for convolutional neural networks.

The same applies to the strides vector - since our goal is to perform subsampling, we choose a pooling window shift equal to 2 for both the x and y directions (strides[1] and strides[2]). This will reduce the dimensionality of the input data (x,y).

The same approaches apply to the 'SAME' option as for the convolution function conv2d:

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width  = ceil(float(in_width) / float(strides[2]))
```

By substituting 2 in strides[1] and strides[2] for the first convolutional layer, we get the original size (14, 14). It's half the input size (28, 28) that we're looking for. TensorFlow organizes the padding in such a way that this original shape is achieved.

As a result of the execution of the function itself, the out_layer object is returned, which is a subgraph and encapsulates all operations and weight variables.

Next, two convolutional layers are created in the main program by calling the following commands:

```
# creation of convolutional layers
layer1 = create_new_conv_layer(x_shaped, 1, 32, [5, 5], [2, 2], name='layer1')
layer2 = create_new_conv_layer(layer1, 32, 64, [5, 5], [2, 2], name='layer2')
```

As you can see, the input parameter for layer1 is x_shaped, and the input for layer2 is the result of the create_new_conv_layer function, which is written to the layer1 variable. The next step is to create fully connected layers.

3.3. FULLY CONNECTED LAYERS

As discussed earlier, first it is necessary to bring the result of the work of the last layer into a one-dimensional vector (Fig. 2). It is a 7×7 grid with 64 channels, corresponding to 3136 nodes per training sample. To implement the task, you can use tf.reshape:

```
flattened = tf.reshape(layer2, [-1, 7 * 7 * 64])
```

Again, the dimension (set by specifying -1) of the first fully connected layer is dynamically calculated, corresponding to the number of input samples in the training sample. The next step is to install the first fully connected layer:

```
# в setting the weights and bias neurons for this layer with the subsequent application
of the ReLU function
wd1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1000], stddev=0.03), name='wd1')
bd1 = tf.Variable(tf.truncated_normal([1000], stddev=0.01), name='bd1')
dense_layer1 = tf.matmul(flattened, wd1) + bd1
dense_layer1 = tf.nn.relu(dense_layer1)
```

In accordance with the above code, there is a procedure for initializing the weights of the fully connected layer, multiplying them with the vectorized result of the second CNN layer, adding a displacement neuron and using the ReLU activation function.

The next layer is defined as follows:

```
# fully connected layer with softmax activations
wd2 = tf.Variable(tf.truncated_normal([1000, 10], stddev=0.03), name='wd2')

bd2 = tf.Variable(tf.truncated_normal([10], stddev=0.01), name='bd2')

dense_layer2 = tf.matmul(dense_layer1,wd2)+bd2
y_ = tf.nn.softmax(dense_layer2)
```

This layer is connected to the output, so in this case, the soft-max function is used to calculate the probabilistic output values. In other words, converts a vector of real numbers into a vector of probabilities (non-negative real numbers not exceeding 1).

3.4. CROSS-ENTROPY

In order to make the model better at classifying the input images, we must somehow change the weights for all layers of the network. For this, it is necessary to have information on how well the model works, which can be implemented by comparing the model's predicted output with the desired result.

Cross-entropy is a measure of efficiency used in classification and is a continuous function, always positive, and if the predicted result of the model exactly matches the desired result, it will take on the value 0. Therefore, the goal of optimization is to minimize cross-entropy: the closer to 0 , changing the variables of the network layers - the weights, the better the result of the network. TensorFlow has a built-in function for calculating cross-entropy:

```
cross_entropy =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=dense_layer2, labels=y))
```

The function softmax_cross_entropy_with_logits() accepts two arguments – the first (logits – non-normalized logarithmic probabilities) is the result of the matrix multiplication of the final layer (plus the bias neuron), and the second is the target training object vector. This function calculates the softmax inside itself, which obliges to use the result of the second convolutional layer as the first parameter. The result is the calculation of the cross-entropy per training sample, so we need to bring this tensor to a scalar (to a single value). For this, tf.reduce_mean() is used, which takes the average value of the tensor. In this way, we will get a measure of how poorly our classifier performs.

3.5. CONVOLUTIONAL NEURAL NETWORK TRAINING

The code below is responsible for the training procedure of a typical convolutional neural network. In general, the learning algorithm itself is described in detail in [12]. The mini-package method [13] will be used for the training itself.

The approximate algorithm is as follows:

- create an optimizer;
- create correct forecasting and accuracy assessment operations;
- initialize operations;
- determine the number of packet passes within one training epoch;
- for each era:
- for each package:
- get data from each package;
- run optimization and cross-entropy operations;
- add the average value of the cost function.
- calculate the current accuracy of the test;
- display intermediate results;
- display the final results.

```
# adding an optimizer
optimiser = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cross_entropy)

# definition of accuracy estimation operation
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# setting the initialization operator
init_op = tf.global_variables_initializer()

with tf.Session() as sess:
    # initialization of variables
    sess.run(init_op)
    total_batch = int(len(mnist.train.labels) / batch_size)
    for epoch in range(epochs):
        avg_cost = 0
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size=batch_size)
            _, c = sess.run([optimiser, cross_entropy],feed_dict={x: batch_x, y:
batch_y})
            avg_cost += c / total_batch
        test_acc = sess.run(accuracy, feed_dict={x: mnist.test.images, y:
mnist.test.labels})
    print("Epoch:", (epoch + 1), "cost = ", "{:.3f}".format(avg_cost), "Accuracy:
    {:.3f}".format(test_acc))

    print("\nTraining is complete!")
print(sess.run(accuracy, feed_dict={x: mnist.test.images, y: mnist.test.labels}))
```

The first line corresponds to the use of the gradient descent optimizer provided by the TensorFlow library. It must be initialized with the value of the learning speed, then indicate what exactly it is used for - to reduce the value of the cross-entropy, the function of which was previously described. This function will perform gradient descent [16, 17] and an error backpropagation algorithm [16, 18].

The prediction operation correct_prediction uses the tf.equal function of the TensorFlow library, which returns True and False depending on whether the arguments passed to it as parameters are equal or not. The tf.argmax function returns the index of the maximum value in the vector/tensor. Therefore, the operation correct_prediction returns a tensor of size (m×1) of True and False values, which detect the correctness of predictions of the neural network.

The next step is to calculate the mean accuracy value based on this tensor – first, we need to cast the correct_prediction boolean tensor to the float32 tensor in order to perform the reduce_mean operation. The result is a function that will evaluate the performance of our neural network.

As part of the session, the initialization operation and the calculation of the number of total_batch packages (parties) that will pass through each epoch are carried out.

As part of the iteration of the loop responsible for the procedure of passing through the epochs, the variable avg_cost is initialized to store the average value of the cross-ectropy costs for each epoch.

The next step is to get the randomized batches batch_x and batch_y, from the MNIST training dataset, encapsulated in the TensorFlow library itself along with the next_batch function, which makes it easy to get batches of data for training [19, 20].

The sess.run method is capable of receiving a list of operations to run as its first argument. In this case, the list [optimiser, cross_entropy] means that the specified operations will be performed, and the results of the execution will be stored in the _ and c variables. First of all, we are interested in the result of the cross_entropy operation stored in the c variable (the optimiser (and cross_entropy) operations are run on samples within the package). The following lines are responsible for calculating the average value of losses for each epoch.

CONCLUSION

The implemented algorithm demonstrated the following results of convolutional network training on the MNIST test dataset:

```
Epoch: 01 Cost = 0.739 Accuracy: 0.911
Epoch: 02 Cost = 0.169 Accuracy: 0.960
Epoch: 03 Cost = 0.100 Accuracy: 0.978
Epoch: 04 Cost = 0.074 Accuracy: 0.979
Epoch: 05 Cost = 0.057 Accuracy: 0.984
Epoch: 06 Cost = 0.047 Accuracy: 0.984
Epoch: 07 Cost = 0.040 Accuracy: 0.986
Epoch: 08 Cost = 0.034 Accuracy: 0.986
Epoch: 09 Cost = 0.029 Accuracy: 0.989
Epoch: 10 Cost = 0.025 Accuracy: 0.990
Training is complete! 0.9897
```

We can also reconstruct the graph of the dependence of quality on the epoch, using the TensorBoard toolkit (TensorFlow's visualization module) (Figure 3)

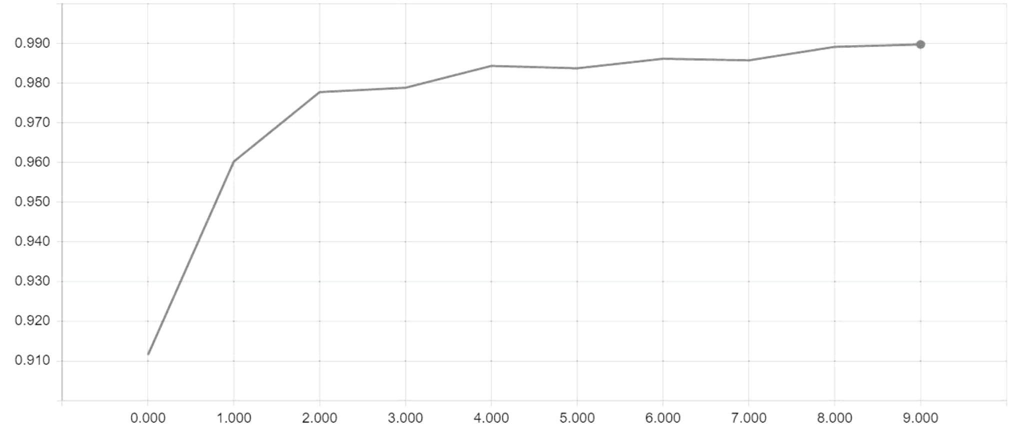


Figure 3 – Accuracy of convolutional neural network training on the MNIST dataset

After 10 epochs, the quality of image class recognition has reached 98.97%. This result was achieved without significant optimization of convolutional neural network parameters, as well as without using regularization [14]. Nevertheless, this result is better on average by 1-5%, compared to what is demonstrated by conventional multilayer neural networks.

It should also be noted that the difference will be much more noticeable when comparing the results that will be demonstrated by ordinary neural networks in comparison with convolutional networks of different architectures on more complex data sets, such as CIFAR [15].

REFERENCES

1. *Y. Lecun.* Gradient-based Learning Applied to Document Recognition [Електронний ресурс] / Y.Lecun, L. Bottou, Y. Bengio, P. Haffner. – 1998. – Access Mode: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.
2. *A Krizhevsky.* ImageNet Classification with Deep Convolutional Neural Networks [Електронний ресурс] / A. Krizhevsky, S. Sutskever, G. Hinton – Access Mode: http://papers.nips.cc/paper/4824-imagenet-classifica-tion-with-deep-convolutional-neural-networks.pdf.
3. *M. Zeiler.* Visualizing and Understanding Convolutional Networks [Електронний ресурс] / M. Zeiler, R. Fergus. – 2013. – Access Mode: https://arxiv.org/pdf/1311.2901v3.pdf.
4. *C. Szegedy.* Going Deeper with Convolutions [Електронний ресурс] / C. Szegedy, W. Liu, Y. Jia та ін. – 2014. – Access Mode: https://arxiv.org/pdf/1409.4842.pdf.
5. *K. Simonyan.* Very deep convolutional networks for large-scale image recognition [Електронний ресурс] / K. Simonyan, A. Zisserman. – 2015. – Access Mode: https://arxiv.org/pdf/1409.1556.pdf.
6. *K. He.* Deep Residual Learning for Image Recognition [Електронний ресурс] / K. He, X. Zhang, S. Ren, J. Sun. – 2015. – Access Mode: https://arxiv.org/pdf/ 1512.03385.pdf.
7. *Ioffe S.* Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift [Електронний ресурс] / S. Ioffe, C. Szegedy. – 2015. – Access Mode: https://arxiv.org/pdf/1502.03167.pdf.
8. An open-source software library for Machine Intelligence [Електронний ресурс]. – 2018. – Access Mode: https://www.tensorflow.org/
9. *LeCun Y.* The MNIST - Database of handwritten digits [Електронний ресурс] / Y. LeCun, C. Cortes, C. Burges – Access Mode: http://yann.lecun.com/exdb/mnist/.
10. Python TensorFlow Tutorial – Build a Neural Network [Електронний ресурс]. – 2017. – Access Mode: http://adventuresinmachinelearning.com/python-tensorflow- tutorial/.
11. Stochastic Gradient Descent – Mini-batch and more [Електронний ресурс]. – 2017. – Access Mode: http://adventuresinmachinelearning.com/stochastic-gradient-descent/.
12. Improve your neural networks – Part 1 [TIPS AND TRICKS] [Електронний ресурс]. – 2017. – Access Mode: http://adventuresinmachinelearning.com/ improve-neural-networks-part-1/.
13. The CIFAR-10 dataset / The CIFAR-100 dataset [Електронний ресурс] – Access Mode: https://www.cs.toronto.edu/~kriz/cifar.html.
14. Neural Networks Tutorial – A Pathway to Deep Learning [Електронний ресурс]. – 2017. – Access Mode: http://adventuresinmachinelearning.com/neural-networks-tutorial/.
15. Stochastic Gradient Descent – Mini-batch and more [Електронний ресурс]. – 2017. – Access Mode: http://adventuresinmachinelearning.com/stochastic-gradient-descent/.
16. TensorFlow. API Documentation. v.1.5. [Електронний ресурс]. – 2017. – Access Mode: https://www.tensorflow.org/api_guides/python/train.
17. Olexander N. Romanyuk, and etc. "A function-based approach to real-time visualization using graphics processing units", Proc. SPIE 11581, Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2020, 115810E (14 October 2020).
18. L.I. Timchenko, N.I. Kokriatskaia, S.V. Pavlov, and etc. "Q-processors for real-time image processing", Proc. SPIE 11581, Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2020, 115810F (14 October 2020).
19. Intellectual Technologies in Medical Diagnosis, Treatment and Rehabilitation: monograph / [S. In Pavlov, O.G. Avrunin, S.M. Zlepko, E.V. Bodyanskyi, etc.]; edited by S. Pavlov, O. Avrunin. - Vinnytsia: PP "TD "Edelveiss and K", 2019. -260 p. ISBN 978-617-7237-59-3
20. Intelligent Technologies of Computer Planning and Modeling in Medical Diagnosis, Treatment and Rehabilitation: monograph // edited by S.V. Pavlov, O.G. Avrunin, O.V. Hrushko - Zhytomyr: "Euro-Volyn" PE, 2021. - 202 p. ISBN 978-617-7992-15-7.

Надійшла до редакції 5.11.2022р.

TOMKA YURIY – Ph.D., assistant professor of Computer Science Department, Yuriy Fedkovich Chernivtsi National University, Chernivtsi, Ukraine, ***e-mail: y.tomka@chnu.edu.ua***

TALAKH MARIA – Ph.D., assistant professor of Computer Science Department, Yuriy Fedkovich Chernivtsi National University, Chernivtsi, Ukraine, ***e-mail: m.talah@chnu.edu.ua***

DVORZHAK VALENTINA – Ph.D., assistant professor of Computer Science Department, Yuriy Fedkovich Chernivtsi National University, Chernivtsi, Ukraine, ***e-mail: v.dvorzhak@chnu.edu.ua***

USHENKO OLEXANDER – D.Sc., Professor, Head of Optics and Publishing Department, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine, ***e-mail: o.ushenko@chnu.edu.ua***