The following is the tutorial provided by Martin Görner *et al*. adopted and edited for the lab work on CNN Machine Learning class. The original content can be found in the GOOGLE's developers branch.

CNN lab work

A.Sarsembayev , N.Duzbayev

IITU, 2018

Outline

Intro

The purpose of this lab is to lead you through the main steps of building the convolutional neural network. You will be working on the MNIST dataset, which consists of 60k labeled handwritten digits examples and 10k testing examples. MNIST is a subset of a larger set available from NIST.

In this work you will build a 1-layer neural network using Tensorflow (TF) for Python3. The final program will consist of ~100 lines of code.

For this work you will need

- Python 2 or 3 (Python 3 recommended)
- TensorFlow
- Matplotlib (Python visualisation library)

Preparation. Installation of the libraries.

You can install TF either through Anaconda shell type or through the Windows command line (cmd). Make sure that the path variables are set up right (see the Appendix A). Herein, we assume that you are installing TF through the cmd.

There are two versions of TF available for the installation – CPU and GPU. GPU version requires a powerful Graphics Processing Unit and is suitable for high energy consuming operations. We will install CPU version. Input further line in terminal:

```
C:\> pip3 install --upgrade tensorflow
```

You should see the process of the package installation starting from the collecting stage. When the process of installation will finish you can easily verify if the package has been installed correctly by invoking python from the terminal:

```
$ python
```

and entering the following short program inside the python interactive shell:

```
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
```

The output should be:

```
Hello, TensorFlow!
```

If you obtained this output, you are ready to start with writing TF programs.

Preparation. Get the preliminary files.

You will be given a zipped archive of preliminary files which contains dataset and visualization tool. Once unzipped the file, go to the directory /cnn_tut and create a new python file 'mnist_1.0_softmax.py'.

A bit of theory

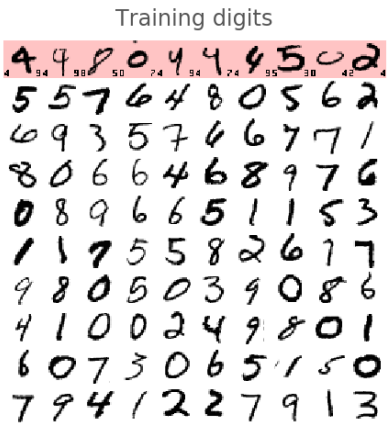
Before we proceed to writing the code, let's cover a bit of theory.

Training the network

Our neural network takes in handwritten digits and classifies them, i.e. states if it recognizes them as a 0, a 1, a 2 and so on up to a 9. It does so based on internal variables ("weights" and "biases", explained later) that need to have a correct value for the classification to work well. This "correct value" is learned through a training process, also explained in detail later. What you need to know for now is that the training loop looks like this:

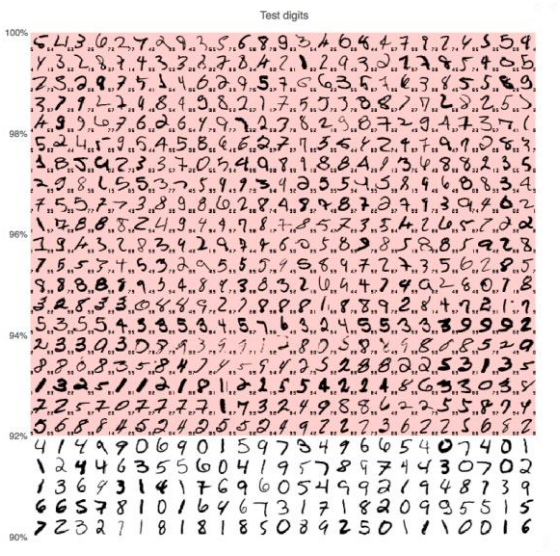
Training digits => updates to weights and biases => better recognition (loop)

Let us go through the six panels of the visualization one by one to see what it takes to train a neural network.

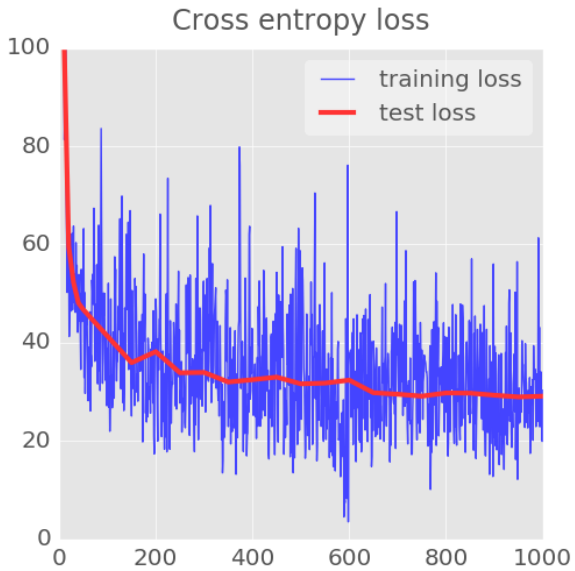


Here you see the training digits being fed into the training loop, 100 at a time. You also see if the neural network, in its current state of training, has recognized them (white background) or mis-classified them (red background with correct label in small print on the left side, bad computed label on the right of each digit).

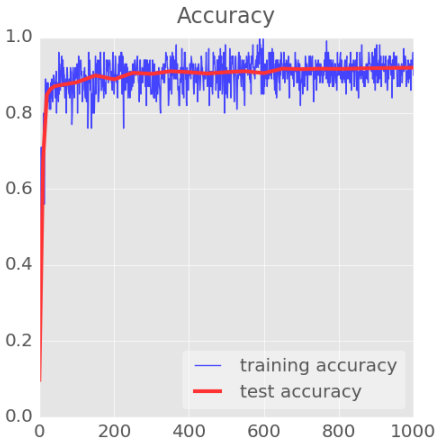
There are 50,000 training digits in this dataset. We feed 100 of them into the training loop at each iteration so the system will have seen all the training digits once after 500 iterations. We call this an "epoch".



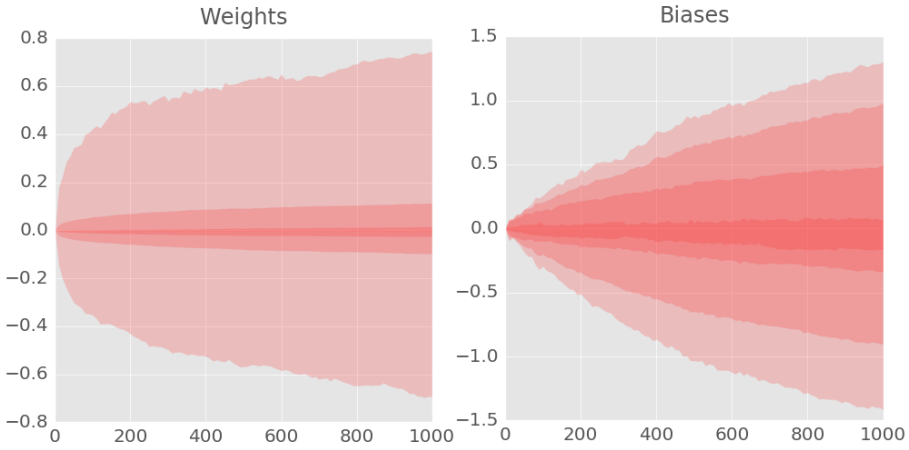
To test the quality of the recognition in real-world conditions, we must use digits that the system has NOT seen during training. Otherwise, it could learn all the training digits by heart and still fail at recognising an "8" that I just wrote. The MNIST dataset contains 10,000 test digits. Here you see about 1000 of them with all the mis-recognised ones sorted at the top (on a red background). The scale on the left gives you a rough idea of the accuracy of the classifier (% of correctly recognised test digits)



To drive the training, we will define a loss function, i.e. a value representing how badly the system recognises the digits and try to minimise it. The choice of a loss function (here, "cross-entropy") is explained later. What you see here is that the loss goes down on both the training and the test data as the training progresses: that is good. It means the neural network is learning. The X-axis represents iterations through the learning loop.



The accuracy is simply the % of correctly recognised digits. This is computed both on the training and the test set. You will see it go up if the training goes well.

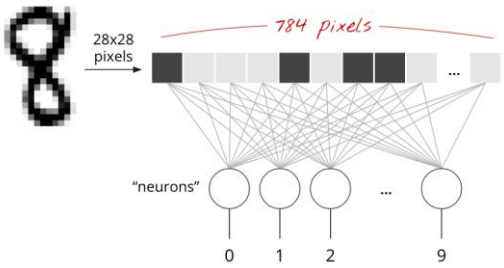


The final two graphs represent the spread of all the values taken by the internal variables, i.e. weights and biases as the training progresses. Here you see for example that biases started at 0 initially and ended up taking values spread roughly evenly between -1.5 and 1.5. These graphs can be useful if the system does not converge well. If you see weights and biases spreading into the 100s or 1000s, you might have a problem.

The bands in the graphs are percentiles. There are 7 bands so each band is where 100/7=14% of all the values are.

A 1-layer neural network

Handwritten digits in the MNIST dataset are 28x28 pixel greyscale images. The simplest approach for classifying them is to use the 28x28=784 pixels as inputs for a 1-layer neural network.



Each "neuron" in a neural network does a weighted sum of all of its inputs, adds a constant called the "bias" and then feeds the result through some non-linear activation function.

Here we design a 1-layer neural network with 10 output neurons since we want to classify digits into 10 classes (0 to 9).

For a classification problem, an activation function that works well is softmax. Applying softmax on a vector is done by taking the exponential of each element and then normalising the vector (using any norm, for example the ordinary euclidean length of the vector).

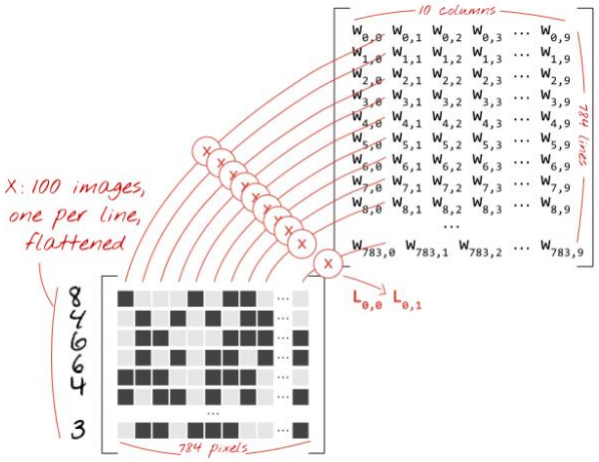
weighted sum of all pixels + bias

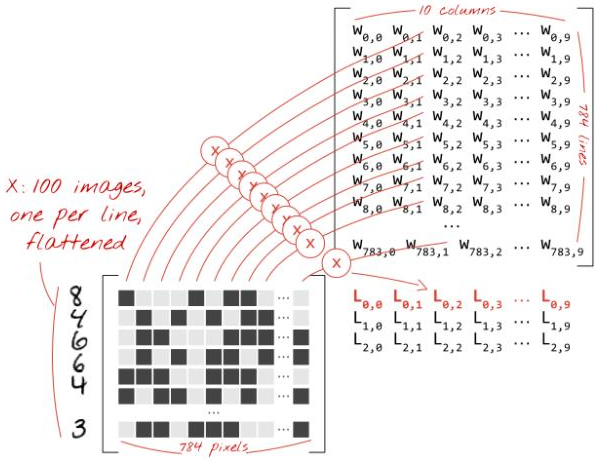
$$softmax(L_n) = \frac{e^{L_n}}{||e^L||}$$

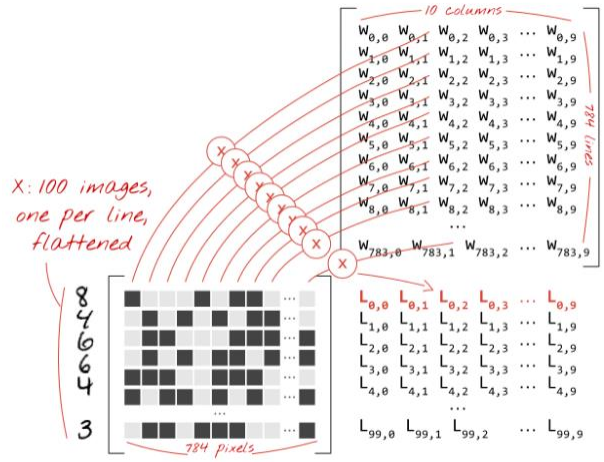
neuron outputs

Why is "softmax" called softmax ? The exponential is a steeply increasing function. It will increase differences between the elements of the vector. It also quickly produces large values. Then, as you normalise the vector, the largest element, which dominates the norm, will be normalised to a value close to 1 while all the other elements will end up divided by a large value and normalised to something close to 0. The resulting vector clearly shows which was its largest element, the "max", but retains the original relative order of its values, hence the "soft".

We will now summarise the behaviour of this single layer of neurons into a simple formula using a matrix multiply. Let us do so directly for a "mini-batch" of 100 images as the input, producing 100 predictions (10-element vectors) as the output.





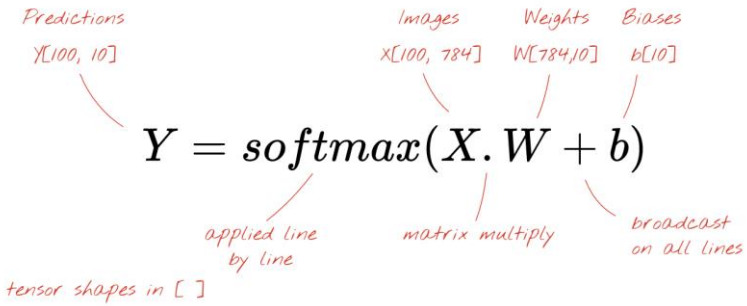


Using the first column of weights in the weights matrix W, we compute the weighted sum of all the pixels of the first image. This sum corresponds to the first neuron. Using the second column of weights, we do the same for the second neuron and so on until the 10th neuron. We can then repeat the operation for the remaining 99 images. If we call X the matrix containing our 100 images, all the weighted sums for our 10 neurons, computed on 100 images are simply X.W (matrix multiply).

Each neuron must now add its bias (a constant). Since we have 10 neurons, we have 10 bias constants. We will call this vector of 10 values b. It must be added to each line of the previously computed matrix. Using a bit of magic called "broadcasting" we will write this with a simple plus sign.

"Broadcasting" is a standard trick used in Python and numpy, its scientific computation library. It extends how normal operations work on matrices with incompatible dimensions. "Broadcasting add" means "if you are adding two matrices but you cannot because their dimensions are not compatible, try to replicate the small one as much as needed to make it work."

We finally apply the softmax activation function and obtain the formula describing a 1-layer neural network, applied to 100 images:



Gradient descent

Now that our neural network produces predictions from input images, we need to measure how good they are, i.e. the distance between what the network tells us and what we know to be the truth. Remember that we have true labels for all the images in this dataset.

Any distance would work, the ordinary euclidian distance is fine but for classification problems one distance, called the "cross-entropy" is more efficient.

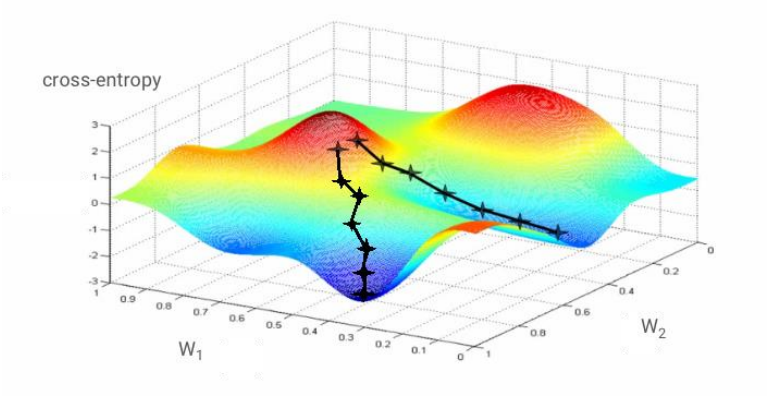
"One-hot" encoding means that you represent the label "6" by using a vector of 10 values, all zeros but the 6th value which is 1. It is handy here because the format is very similar to how our neural network outputs ts predictions, also as a vector of 10 values.

"Training" the neural network actually means using training images and labels to adjust weights and biases so as to minimise the cross-entropy loss function. Here is how it works.

The cross-entropy is a function of weights, biases, pixels of the training image and its known label.

If we compute the partial derivatives of the cross-entropy relatively to all the weights and all the biases we obtain a "gradient", computed for a given image, label and present value of weights and biases. Remember that we have 7850 weights and biases so computing the gradient sounds like a lot of work. Fortunately, TensorFlow will do it for us.

The mathematical property of a gradient is that it points "up". Since we want to go where the cross-entropy is low, we go in the opposite direction. We update weights and biases by a fraction of the gradient and do the same thing again using the next batch of training images. Hopefully, this gets us to the bottom of the pit where the cross-entropy is minimal.



In this picture, cross-entropy is represented as a function of 2 weights. In reality, there are many more. The gradient descent algorithm follows the path of steepest descent into a local minimum. The training images are changed at each iteration too so that we converge towards a local minimum that works for all images.

"Learning rate": you cannot update your weights and biases by the whole length of the gradient at each iteration. It would be like trying to get to the bottom of a valley while wearing seven-league boots. You would be jumping from one side of the valley to the other. To get to the bottom, you need to do smaller steps, i.e. use only a fraction of the gradient, typically in the 1/1000th region. We call this fraction the "learning rate".

To sum it up, here is how the training loop looks like:

Training digits and labels => loss function => gradient (partial derivatives) => steepest descent => update weights and biases => repeat with next mini-batch of training images and labels

Why work with "mini-batches" of 100 images and labels ?

You can definitely compute your gradient on just one example image and update the weights and biases immediately (it's called "stochastic gradient descent" in scientific literature). Doing so on 100 examples gives a gradient that better represents the constraints imposed by different example images and is therefore likely to converge towards the solution faster. The size of the mini-batch is an adjustable parameter though. There is another, more technical reason: working with batches also means working with larger matrices and these are usually easier to optimise on GPUs.

This is it. You have covered the main principles behind the CNN applied in this work. Now let's got further to coding.

Coding

We start by importing the necessary stuff, as usual:

```
import tensorflow as tf
import tensorflowvisu
from tensorflow.examples.tutorials.mnist import input_data as mnist_data
print("Tensorflow version " + tf.__version__)     #let's also check the version of the TF
tf.set_random_seed(0)
```

First we define TensorFlow variables and placeholders. Variables are all the parameters that you want the training algorithm to determine for you. In our case, our weights and biases.

Placeholders are parameters that will be filled with actual data during training, typically training images. The shape of the tensor holding the training images is [None, 28, 28, 1] which stands for:

- 28, 28, 1: our images are 28x28 pixels x 1 value per pixel (grayscale). The last number would be 3 for color images and is not really necessary here.
- None: this dimension will be the number of images in the mini-batch. It will be known at training time.

We are writing the code for 1-layer NN, and it's model is:

*Y = softmax(X * W + b)*, where:

X: matrix for 100 grayscale images of 28x28 pixels, flattened (there are 100 images in a mini-batch)

W: weight matrix with 784 lines and 10 columns

b: bias vector with 10 dimensions

Now we should download images and labels into mnist.test (10K images+labels) and mnist.train (60K images+labels)

```
mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False, validation_size=0)
```

Input X: 28x28 grayscale images, the first dimension (None) will index the images in the mini-batch

```
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
```

The correct answers will go here

```
Y_ = tf.placeholder(tf.float32, [None, 10])
```

The weights W[784, 10]   784=28*28

```
W = tf.Variable(tf.zeros([784, 10]))
```

The biases b[10]

```
b = tf.Variable(tf.zeros([10]))
```

Then we should flatten the images into a single line of pixels

XX = tf.reshape(X, [-1, 784])

-1 in the shape definition means "the only possible dimension that will preserve the number of elements"

The model is built with:

Y = tf.nn.softmax(tf.matmul(XX, W) + b)

The loss function: cross-entropy = - sum(Y_i * log(Yi)), where:

Y: the computed output vector

Y_: the desired output vector

The cross-entropy log takes the log of each element, * multiplies the tensors element by element The reduce_mean method will add all the components in the tensor. So here we end up with the total cross-entropy for all images in the batch normalized for batches of 100 images, and multiplied by ten *10 because "mean" included an unwanted division by 10

*cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0*

The accuracy of the trained model, between 0 (worst) and 1 (best)

correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

The training, learning rate = 0.005 (check the theory section for the definition)

train_step = tf.train.GradientDescentOptimizer(0.005).minimize(cross_entropy)

This where the TensorFlow magic happens. You select an optimiser (there are many available) and ask it to minimise the cross-entropy loss. In this step, TensorFlow computes the partial derivatives of the loss function relatively to all the weights and all the biases (the gradient). This is a formal derivation, not a numerical one which would be far too time-consuming.

The gradient is then used to update the weights and biases. 0.005 is the learning rate.

The visualization:

allweights = tf.reshape(W, [-1])
allbiases = tf.reshape(b, [-1])
I = tensorflowvisu.tf_format_mnist_images(X, Y, Y_) # assembles 10x10 images by default
It = tensorflowvisu.tf_format_mnist_images(X, Y, Y_, 1000, lines=25) # 1000 images on 25 lines

```
datavis = tensorflowvisu.MnistDataVis()
```

And finally – initialization:

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

You can call the main function in a loop to train the model, 100 images at a time

```
def training_step(i, update_test_data, update_train_data):

    # training on batches of 100 images with 100 labels
    batch_X, batch_Y = mnist.train.next_batch(100)

    # compute training values for visualisation
    if update_train_data:
        a, c, im, w, b = sess.run([accuracy, cross_entropy, I, allweights, allbiases], feed_dict={X: batch_X, Y_:
batch_Y})
        datavis.append_training_curves_data(i, a, c)
        datavis.append_data_histograms(i, w, b)
        datavis.update_image1(im)
        print(str(i) + ": accuracy:" + str(a) + " loss: " + str(c))

    # compute test values for visualisation
    if update_test_data:
        a, c, im = sess.run([accuracy, cross_entropy, It], feed_dict={X: mnist.test.images, Y_:
mnist.test.labels})
        datavis.append_test_curves_data(i, a, c)
        datavis.update_image2(im)
        print(str(i) + ": ********* epoch " + str(i*100//mnist.train.images.shape[0]+1) + " ********* test
accuracy:" + str(a) + " test loss: " + str(c))

    # the backpropagation training step
    sess.run(train_step, feed_dict={X: batch_X, Y_: batch_Y})
```

Animation

```
datavis.animate(training_step, iterations=2000+1, train_data_update_freq=10,
test_data_update_freq=50, more_tests_at_start=True)

print("max test accuracy: " + str(datavis.get_max_test_accuracy()))
```

The final test accuracy will be around 92%. The next goal should be the increase of accuracy. This should be achieved by adding the new layers.

Run the code

Finally, you should save the changes and run the code through the command line by executing:

```
C:\*YOUR_DIRECTORY_GOES_HERE*> python mnist_1.0_softmax.py
```

Appendix A

In order to check if the path variables are set up right:

1. Press the ‘Win+Pause’ on your keyboard OR right-click the very bottom left corner of the screen to get the Power User Task Menu. From the Power User Task Menu, click System.
2. Click the Advanced System Settings link in the left column. Figure A.
3. In the System Properties window, click on the Advanced tab, then click the Environment Variables button near the bottom of that tab. Figure B.
4. In the window appeared (Figure C) make sure that your PATH variables contain the values shown in the figure D.

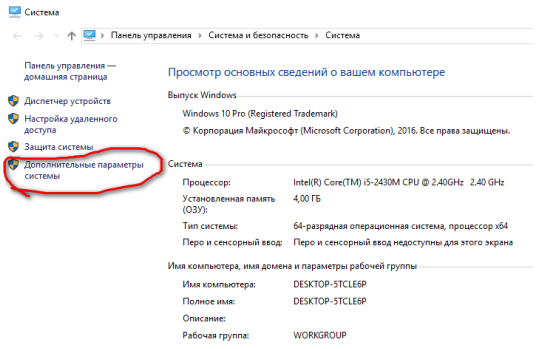


Рисунок A

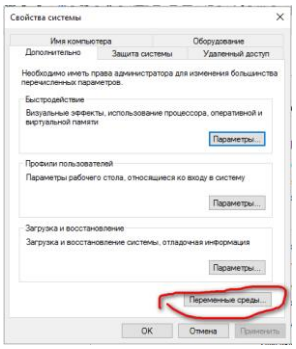


Рисунок B

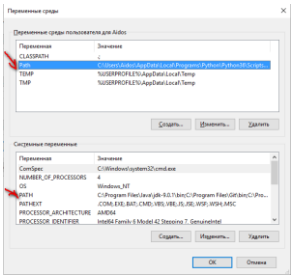


Рисунок C

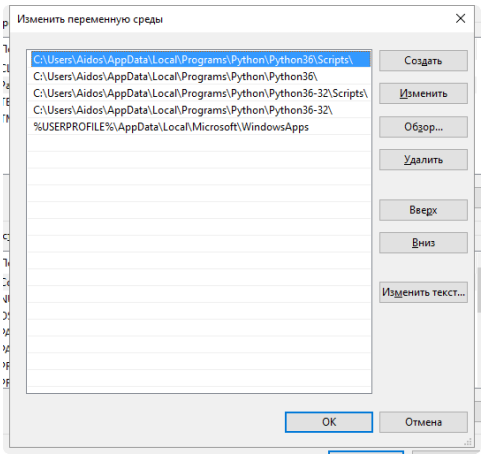

Изменить переменную среды
C:\Users\Aidos\AppData\Local\Programs\Python\Python36\Scripts\
C:\Users\Aidos\AppData\Local\Programs\Python\Python36\
C:\Users\Aidos\AppData\Local\Programs\Python\Python36-32\Scripts\
C:\Users\Aidos\AppData\Local\Programs\Python\Python36-32\
%USERPROFILE%\AppData\Local\Microsoft\WindowsApps
Создать
Изменить
Обзор...
Удалить
Вверх
Вниз
Изменить текст...
OK
Отмена

Рисунок D