Part1-Step1: Introduction For MNIST

MNIST stands for Mixed National Institute of Standards and Technology, which has produced a handwritten digits dataset. This is one of the most researched datasets in machine learning, and is used to classify handwritten digits. This dataset is helpful for predictive analytics because of its sheer size, allowing deep learning to work its magic efficiently. This dataset contains 50000 examples and 784 dimensional feature vecto, formatted as 28 x 28 pixel monochrome images.

Part1-Step2: Training An SOM

In the process of training SOM, Gaussian distribution and Euclidean distance calculation are used.

```
class EuclideanDistance {
public:
    float EvaluateDistance(Eigen::VectorXd v1, Eigen::VectorXd v2){
        float KernelValue = (v1 - v2).norm();
        return KernelValue;
    }
};

class GaussianDistribution {
public:
    float Gaussian(float x, float sigma){
        float value = exp(-((float)x/(float)sigma));
        return value;
    }
}
```

And SOMLearningFunction, UpdateSOMLearningRate, UpdateSOMSigmaNeighbouring are constantly updated during the learning process.

```
class UpdateSOMLearningRate {
public:
    float UpdateLearningRate(float L0, float LearningRateFinal, float Iter, float l
        float value = (L0 - LearningRateFinal) * L0*exp(-((float)Iter/(float)lambda
        return value;
    }
};


class UpdateSOMSigmaNeighbouring {
public:
    float UpdateSigmaNeighbouring(float SigmaNeighbouringZero, float SigmaNeighbou
        float value = (SigmaNeighbouringZero -SigmaNeighbourhoodFinal) * exp(-((flo
        return value;
    }
};
```

The specific training process is as follows:

```
class SOMTrain :  public Initialize2DMap, public GaussianDistribution,
public SOMLearningFunction, public PrintSOM, public EuclideanDistance, public Upda
 public:
 int Iters;
 int BestXMap, BestYMap = 0;
 float Score, ScoreNow = 0;
    float LearningRateInitial, LearningRateFinal, SigmaNeighbouringInitial, SigmaN

 void Train(int Iters){
     // Initialize alpha
     SOMTrain::SigmaNeighbouringInitial;
     SOMTrain::SigmaNeighbourhoodFinal;
     SOMTrain::LearningRateInitial;
     SOMTrain::LearningRateFinal;
     SOMTrain::LearningRate=SOMTrain::LearningRateInitial;
     SOMTrain::SigmaNeighbouring=SOMTrain::SigmaNeighbouringInitial;
     std::cout <<  SOMTrain::SigmaNeighbouring <<std::endl;
     SOMTrain::lambda = (float)(Iters)/(float)(SOMTrain::xsize/2);

     Eigen::VectorXd BMU;

     for (int i=0; i<Iters; i++){
         int j = GenerateRandomNumber(SOMTrain::NumberInputVectors);
            // Find the BMU
             for (int RowMap=0;RowMap<SOMTrain::xsize;RowMap++){
                 for (int ColMap=0;ColMap<ysize;ColMap++){
                     ScoreNow = EvaluateDistance(Initialize2DMap::SOMMap(RowMap,Col
                     if (ScoreNow > SOMTrain::Score){SOMTrain::Score = ScoreNow; SO
                  }
             }
             BMU = Initialize2DMap::SOMMap(SOMTrain::BestXMap,SOMTrain::BestYMap);

             for (int RowMap=0;RowMap<SOMTrain::xsize;RowMap++){
                 for (int ColMap=0;ColMap<ysize;ColMap++){


                     int CoordX = RowMap;
                     int CoordY = ColMap;
                     if (CoordX >= (SOMTrain::xsize/2)){CoordX = (SOMTrain::xsize/2
                     if (CoordY >= (SOMTrain::ysize/2)){CoordY = (SOMTrain::ysize/2

                     Eigen::VectorXd BMUUpdated = LearnFunction(Initialize2DMap::SO
                     SOMTrain::BestXMap, SOMTrain::BestYMap, CoordX, CoordY, SOMTr
                     Initialize2DMap::SOMMap(RowMap,ColMap) = BMUUpdated;

                 }
```

```
                }

            SOMTrain::LearningRate = UpdateLearningRate(SOMTrain::LearningRateInitial ,
            SOMTrain::SigmaNeighbouring = UpdateSigmaNeighbouring(SOMTrain::SigmaNeig
i , SOMTrain::lambda);


    }
}
```

Part1-Step3: Parameter Adjustment

Through continuous experiment adjustment, the initial value of 2D lattices is finally adjusted to 80 X 80, while continuously adjusting NeighbouringInitial, NeighbouringFinal, and learningrate, the experimental process is as follows:

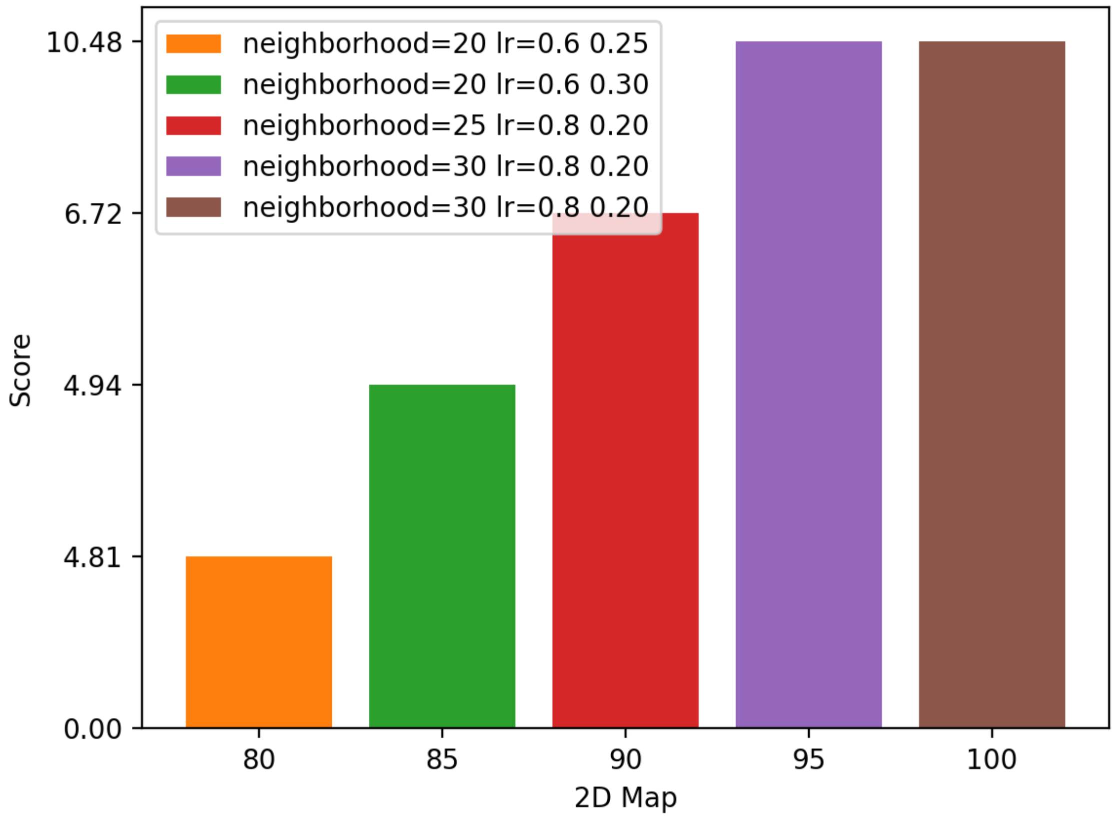

Figure 1: Parameter1

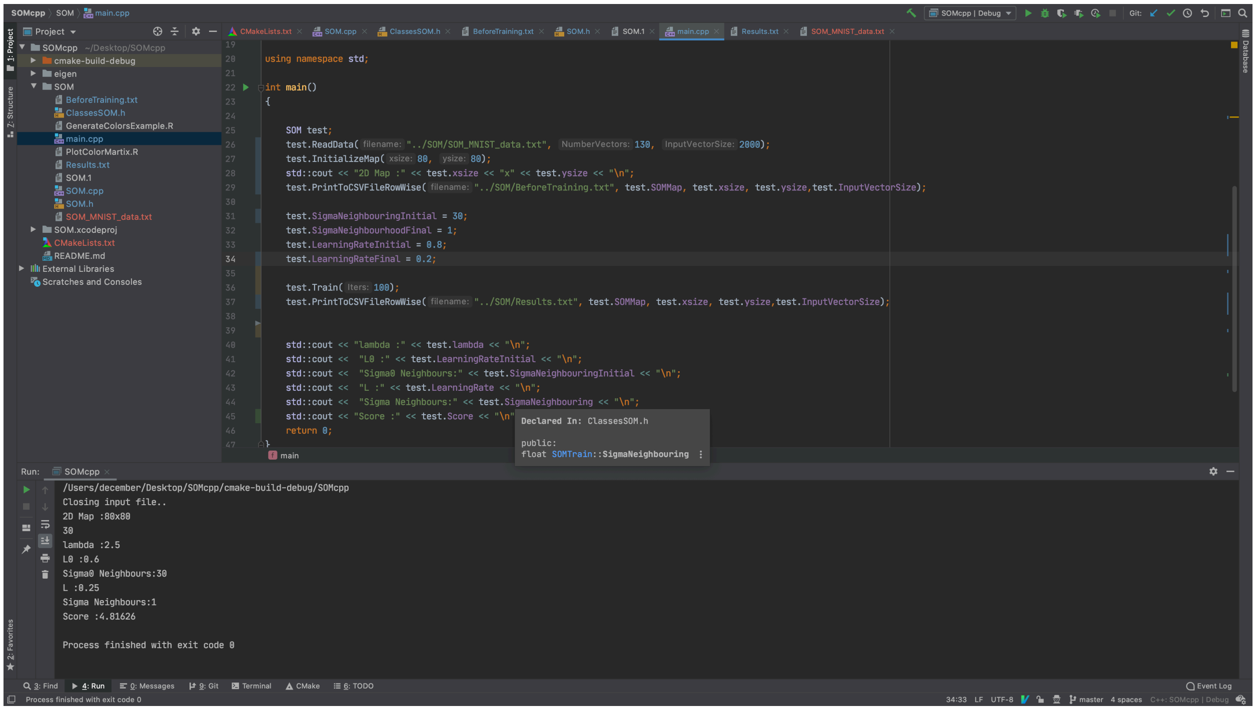

Figure 2: Parameter1

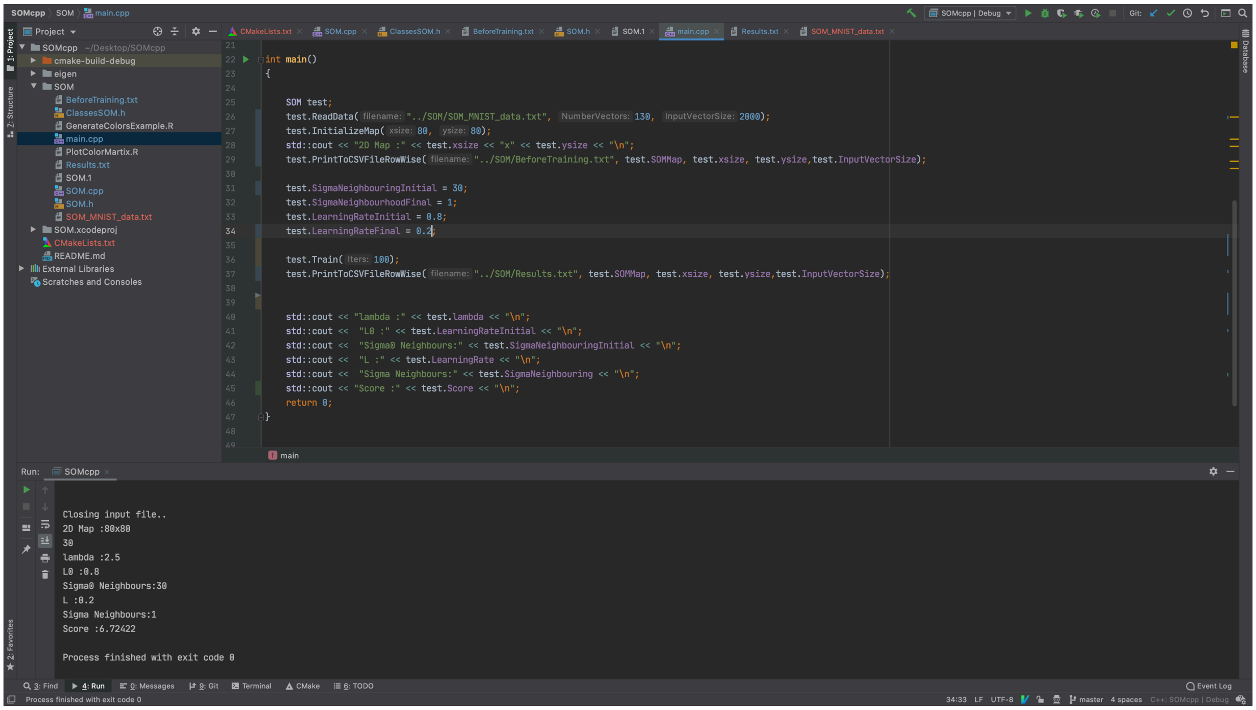

Figure 3: Parameter2

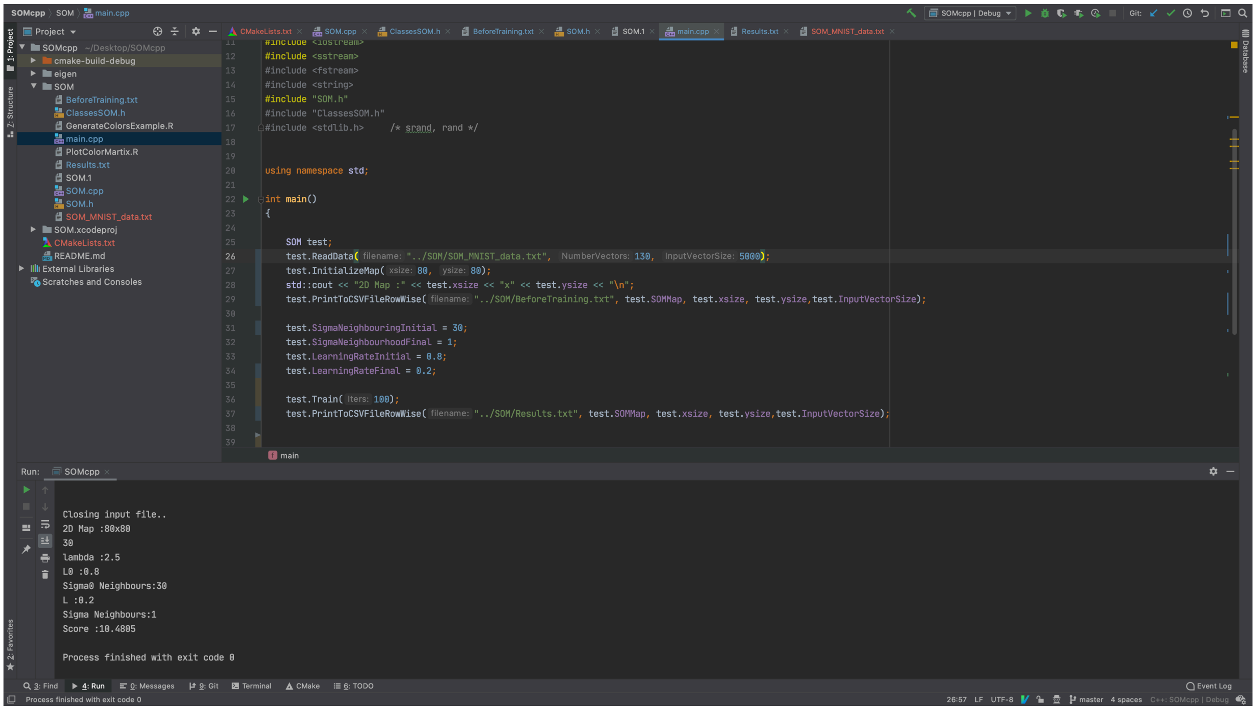

Figure 4: Parameter3

Part1-Step4: Result

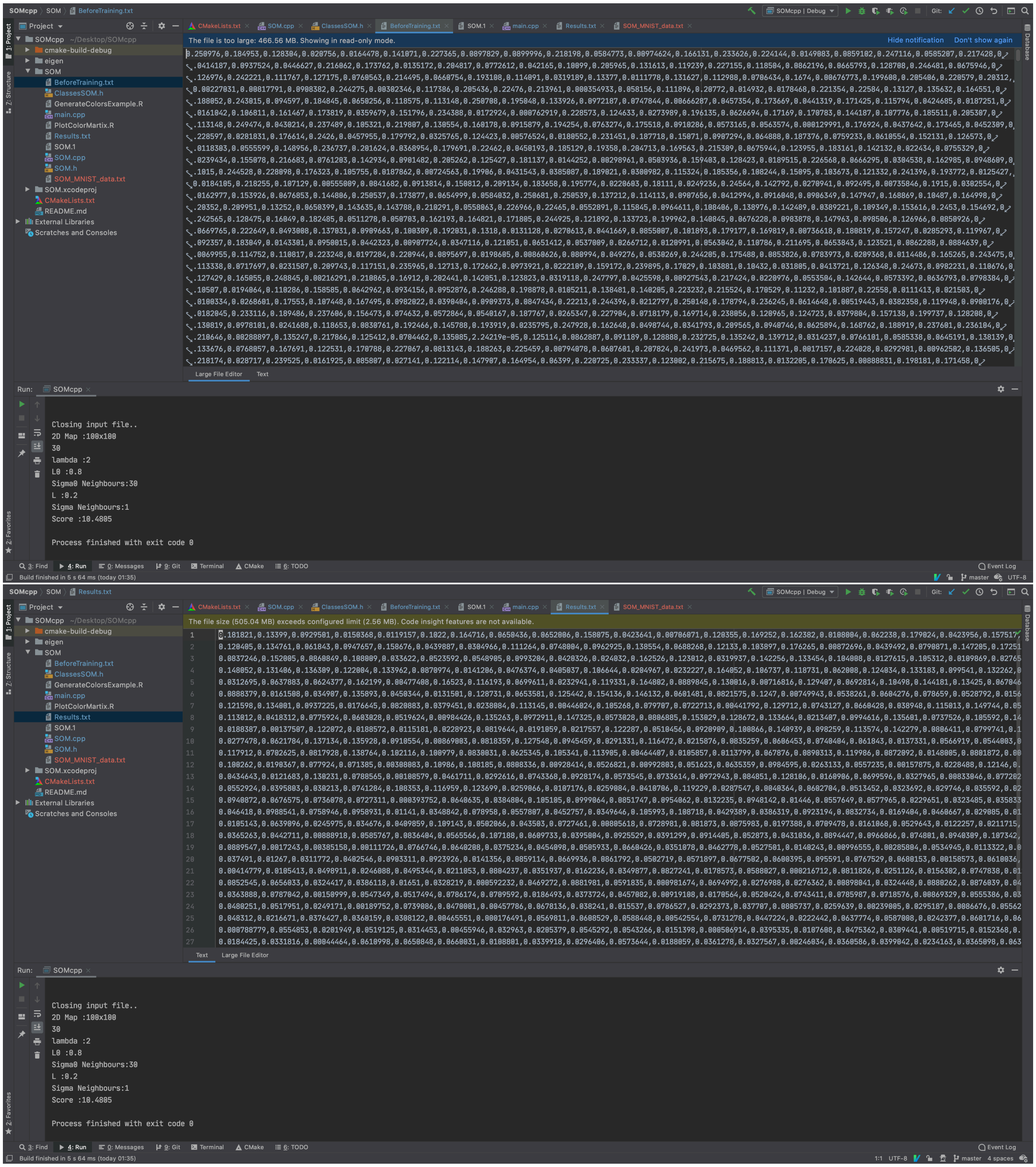

Figure 5: Output

Part2: Introduction On MNIST

The MNIST data is split into three parts: 55,000 data points of training data (mnist.train), 10,000 points of test data (mnist.test), and 5,000 points of validation data (mnist.validation). This split is very important: it's essential in machine learning that we have separate data which we don't learn from so that we can make sure that what we've learned actually generalizes!

the training images are mnist.train.images and the training labels are mnist.train.labels.

Each image is 28 pixels by 28 pixels. Each entry in the tensor is a pixel intensity between 0 and 1, for a particular pixel in a particular image.

Each image in MNIST has a corresponding label, a number between 0 and 9 representing the digit drawn in the image.

Part2: Multinomial Logistic Regression

```
learning_rate = 0.09
training_epochs = 20
batch_size = 80
display_step = 1

# mnist data image of shape 28*28=784
# 0 - 9 recognition -> 10 classes
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, 10])


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

# softmax model
pred = tf.nn.softmax(tf.matmul(x, W) + b)

# using cross entropy
# use gradient descent
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

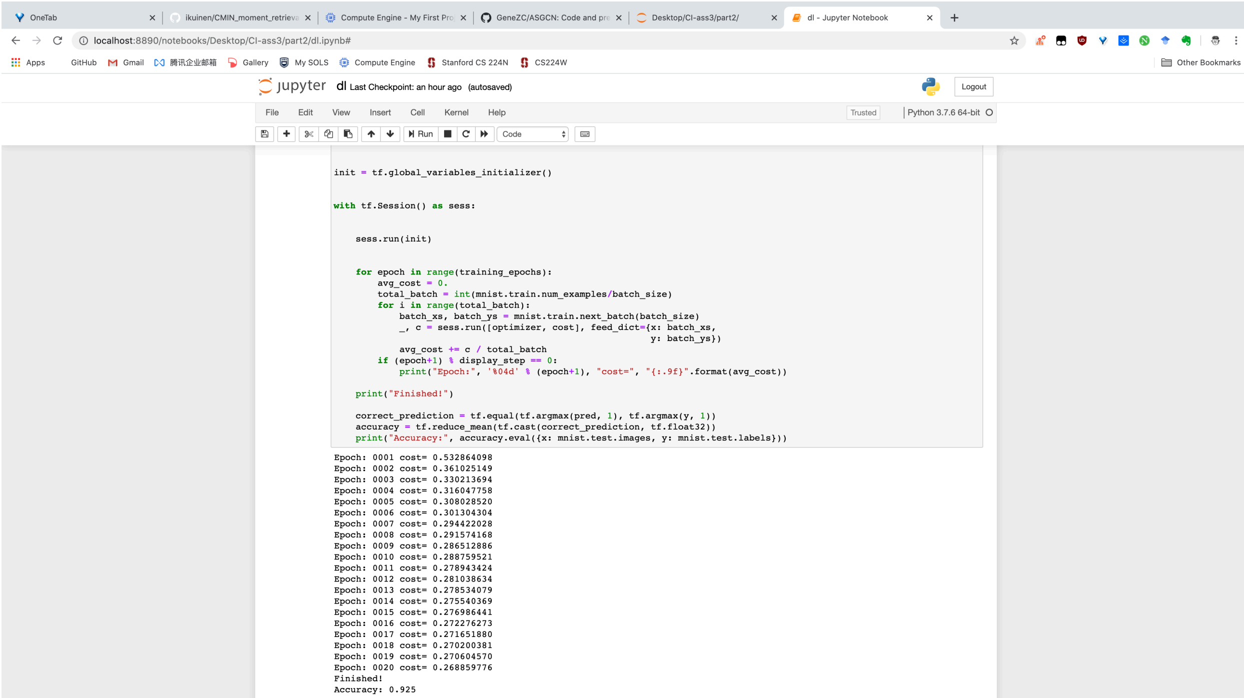

Figure 6: Multinomial Logistic Regression Output

Part2: CNN

```
learning_rate = 0.009
num_steps = 2000
batch_size = 128

num_input = 784
num_classes = 10
dropout = 0.25


# Create the neural network
def conv_net(x_dict, n_classes, dropout, reuse, is_training):
    with tf.variable_scope('ConvNet', reuse=reuse):
        x = x_dict['images']

        x = tf.reshape(x, shape=[-1, 28, 28, 1])

        # 32 filters and 5 kernels in Convolution Layer
        conv1 = tf.layers.conv2d(x, 32, 5, activation=tf.nn.relu)
        conv1 = tf.layers.max_pooling2d(conv1, 2, 2)

        # 64 filters and 3 kernels in Convolution Layer2
        conv2 = tf.layers.conv2d(conv1, 64, 3, activation=tf.nn.relu)
```

```
        conv2 = tf.layers.max_pooling2d(conv2, 2, 2)

        fc1 = tf.contrib.layers.flatten(conv2)

        fc1 = tf.layers.dense(fc1, 1024)
        fc1 = tf.layers.dropout(fc1, rate=dropout, training=is_training)

        out = tf.layers.dense(fc1, n_classes)

    return out
```

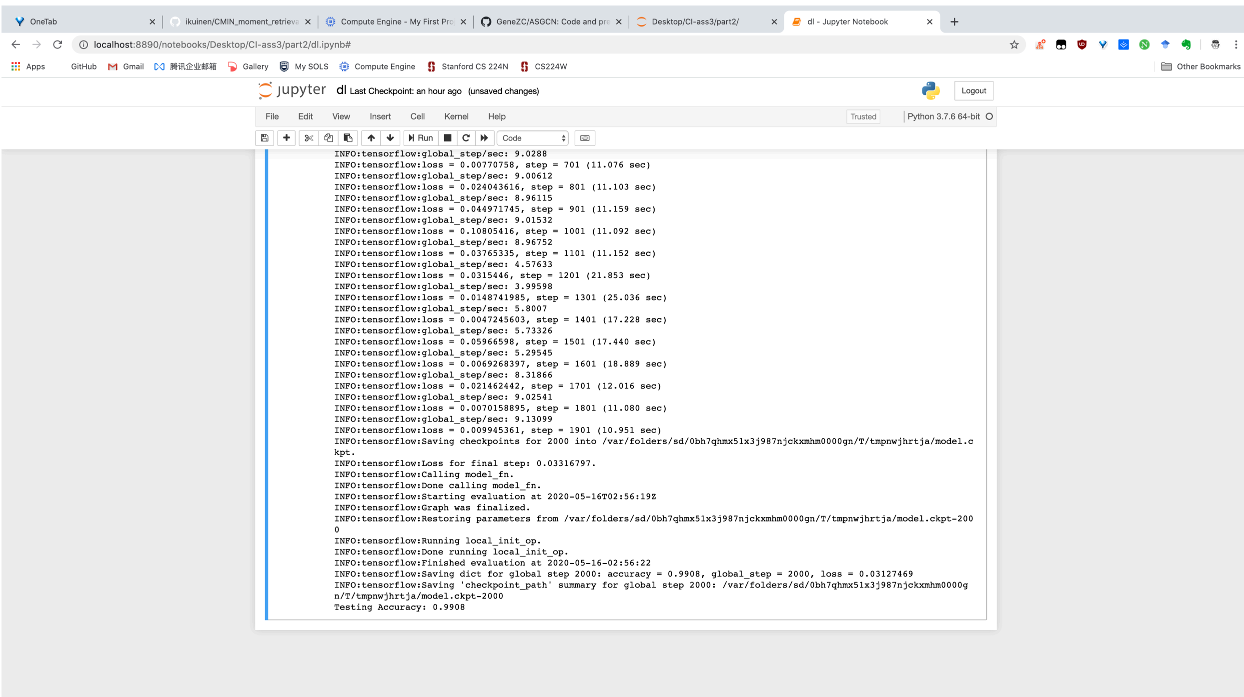

Figure 7: CNN Output

Part2: Cell Image Classification

```
training_set_size = 25998
test_set_size = 1560

model = keras.Sequential([
    keras.layers.Convolution2D(5,(3,3),input_shape=(img_size,img_size,training_set
    keras.layers.MaxPooling2D((2, 2), padding='same'),
    keras.layers.Convolution2D(10,(3,3),padding='same',data_format='channels_last',
    keras.layers.MaxPooling2D((2, 2), padding='same'),
    keras.layers.Flatten(),
   #keras.layers.Dense(10, activation = 'relu'), # if you have a good CPU/GPU :S
    keras.layers.Dense(1, activation = 'sigmoid')
```

```
])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()

epochs = 10

model.fit_generator(training_set,
                    steps_per_epoch = (training_set_size/batch_size),
                    epochs = epochs,
                    validation_data = test_set,
                    validation_steps = (test_set_size/batch_size))
```