Main Project, Machine Learning

Båvenstrand, Erik
erikbav@kth.se

Berggren, Jakob
jaberggr@kth.se

December 2018

1 Initial study

In the pursuit of efficiency and productivity there is an ever growing need for automation in our everyday life. Whether it is when paying bills or making coffee we are always seeking ways of making things go fast, and preferably by it self. We will therefore explore the ways of making automated text analysis with machine learning, and more specifically within the domain of number recognition.

The project is centered around recognizing single numbers on a 28*28 pixel picture and outputting the correct value. We will use sigmoid neurons and use the MNIST data set for training and testing.

MNIST is a subset of the NIST data set. Our data set contains size-normalized and centered fixed-size images of numbers and their corresponding value.

The success of the project is determined by how well our neural network performs when trying to recognize the testing data and our own hand written numbers. This will be discussed further in a later section. Neural network is a term that will be used often so the abbreviation NN will be used.

1.1 Requirements

Non-functional	Functional
Performance	Recognition
Flexibility	Ease of use
Scalability	

1.1.1 Performance

Building a NN is a very CPU intensive process. Therefore we need to make sure that the building does not take more than a reasonable time. Since we are building this in Python it will not be the fastest of implementations. Due

to this the acceptable time frame for creating a network is around 10 minutes. This will of course vary based on how the layers are set up. We also want to be able to save a network for later use, without the need to compute it again.

1.1.2 Flexibility

A NN is a set of layers containing a different amount of neurons. The only restriction we have is that the first layer must have 28*28=784 neurons and the last layer must have 10 neurons. The part of the network that needs to be flexible is the layers between the first and the last. We want to be able to choose how many layers and how many neurons each layer has.

1.1.3 Scalability

The most important part of machine learning is the training data. We want to be able to change the data set if MNIST releases an updated version of their data.

1.1.4 Recognition

This is the most important functional requirement. The NN has to recognize numbers to a certain extent. If the numbers are from the MNIST data set we expect it to be at least a 90% recognition rate while if it is from our own data set we only require it to be above 50%. The reason why it is allowed to perform worse with our own test cases is because we do not center or size-normalize our numbers so the NN will not be specifically trained for that data set.

1.1.5 Ease of Use

The NN has to be easy to use and be designed intuitively. If additions to the code are needed the structure of it needs to be clear and consistent. The command line arguments also need to be easy to remember.

2 Design

2.1 Neural Network

A NN is a system that seeks to replicate the neurons in our brains. By having several layers of connected neurons we simulate the connections made by the real neurons in the brain. Our NN uses sigmoid neurons which is a very simple non-linear function that describes how bright the neuron should be. A non-linear neuron specifies that the neuron can takes a floating point value between 0 and 1 as input.

Our NN consist of 784 first layer neurons and 10 last layer neurons. In the first layer each neuron represents the gray scale value of that particular pixel. The last layer of neurons represents the different answers of which the NN can

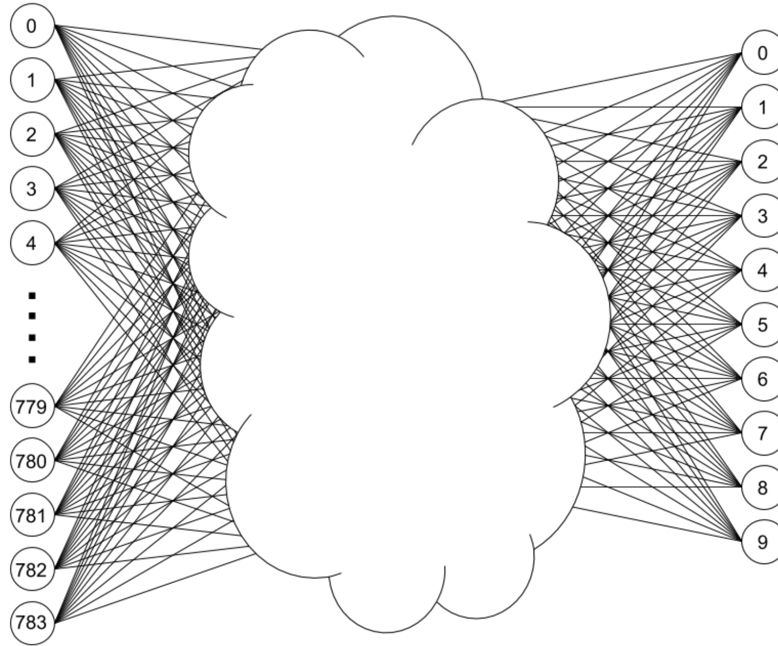

Figure 1: Our neural network with hidden layers

chose between, i.e all the ten digits in number base ten. The brightest neuron of the 10 is the answer.

A NN is essentially a set of weights and biases for each of the connected neurons. When we train the network we are in reality just changing these values a little bit at a time to try to find a local minimum in the extremely complex dimension that is the weights and biases.

2.1.1 Weights

When a NN is trained one part of the process is to tune the weights. Each line/connection in figure 1 has a weight associated with it. This weight represents how much the neuron trusts the decision of the neuron in the layer above. Weights are numerical values which defines the importance of that specific input to the neuron. Each neuron will have its own unique weights for each input from neurons in the layer above. For a single neuron, all the weights are combined with the values of their connecting neurons and put through an activation function. The one used in this project is the sigmoid function. That function will return a value which is the value of our neuron.

2.1.2 Biases

The second part of training is the tuning of biases. Each neuron has a bias which is essentially a constant that is added to each value of the inputs before the values are put through the sigmoid function. These represent patterns that do not necessarily emerge in all inputs. For example, think of what would happen if all the input neurons were set to 0. This would mean that without biases the output would be 0 on all neurons.

2.1.3 Training

When a NN is training it is slowly tuning both biases and weights to minimize a cost function which is essentially the outcome minus the desired outcome. We seek a local minimum in the extremely multi-dimensional solution space. We do this with stochastic gradient descent. This means that the NN will probably never find the global minimum but the local minimum it finds will be good enough.

3 Implementation

The project is written in Python 3.7 with the extensions: numpy 1.15.4, python-mnist 0.6 and pillow 5.3.0. Numpy is an extension that is used to describe matrices and perform dot multiplication etc. Python-mnist in an extension that reads the MNIST data set and returns a data structure that is easy to use in Python. Pillow is an imaging processing extension that is used to process our own test cases for the NN.

3.1 Initialization

When a NN is created there need to be some initial values. A list of layers is the only argument needed for the NN. The list specifies how many neurons there are in each layer of the NN. The biases and weights are randomly set to values between 1 and 0 initially. This is done to avoid converging to the same local minimum, which would happen if it always started from the same point. It also means that the NN could theoretically be perfect before any training is done but it is highly unlikely.

```
def __init__(self, layers):
    self.layers = layers
    self.size = len(layers)
    self.biases = [np.random.randn(y, 1)
                   for y in layers[1:]]
    self.weights = [np.random.randn(y, x)
                    for x, y in zip(layers[:-1],
                    layers[1:])]
```

3.2 Propagation

Propagation is the process of letting the NN decide on the input data. This is used when training and when testing the network. The variables b and w is the bias and weight of a specific input neuron. No in depth explanation is needed as the function is very basic.

```
def propagate(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a) + b)
    return a
```

3.3 Backpropagation

When the network is training this is the function that determines by how much the weights and biases should change. The change is named nabla which is the mathematical symbol for gradient. This is where the real stochastic gradient descent takes place.

```
def backPropagate(self, x, y):
    nablaB = [np.zeros(b.shape) for b in self.biases]
    nablaW = [np.zeros(w.shape) for w in self.weights]

    activation = x
    activations = [x]
    zs = []

    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation) + b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)

    delta = self.costDerivative(activations[-1], y) *
            sigmoid_prime(zs[-1])
    nablaB[-1] = delta
    nablaW[-1] = np.dot(delta,
                        activations[-2].transpose())

    for l in range(2, self.size):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l + 1].transpose(),
                delta) * sp
        nablaB[-l] = delta
        nablaW[-l] = np.dot(delta,
                     activations[-l - 1].transpose())
```

```
    return (nablaB, nablaW)
```

3.4 Stochastic Gradient Descent

In reality this the function that is called to train the network. The training data is a list of tuples containing a image as an array and the number written on that image. Epochs are the number of times the testing data as a whole is to be completely run through. When training a NN there is normally a need to run an entire epoch before any changes to the biases and weights can be made. This means that for large data sets it consumes an absurd amount of memory. It will also be very slow to change the biases and weights. To counter this batch size tells us how large a "batch" is. A batch is a random selection of a number of elements in the training data. After each batch is run through, changes will be made to the NN. The learning rate is just how much change is allowed in a single batch to the biases and weights. The testing data is data that the NN has not been trained with but has the same type of processing.

```
def stochasticGradientDescent(self, trainingData, epochs,
        batchSize, learningRate, testingData):
    nTest = len(testingData)
    n = len(trainingData)

    for j in range(epochs):
        shuffle(trainingData)
        batches = [trainingData[k:k + batchSize]
            for k in range(0, n, batchSize)]

        for batch in batches:
            self.updateBatch(batch, learningRate)
```

3.5 Updating

In this function we get a batch that contains a list of images and the numbers in a tuple. We use the function backpropagate to determine the needed change to the weights and biases and then change them based on the results of the NN and the batch.

```
def updateBatch(self, batch, learningRate):
    nablaB = [np.zeros(b.shape) for b in self.biases]
    nablaW = [np.zeros(w.shape) for w in self.weights]

    for x, y in batch:
        deltaNablaB, deltaNablaW = self.backPropagate(x, y)
        nablaB = [nb + dnb for nb, dnb in zip(nablaB,
                 deltaNablaB)]
```

```
        nablaW = [nw + dnw for nw, dnw in zip(nablaW,
                  deltaNablaW)]

    self.weights = [w - (learningRate / len(batch)) * nw
                    for w, nw in zip(self.weights, nablaW)]
    self.biases = [b - (learningRate / len(batch)) * nb
                    for b, nb in zip(self.biases, nablaB)]
```

3.6 Sigmoid function

This is the central function of the sigmoid neuron. Z is value * weight + bias. We then "squish" this number down to a floating point value between 0 and 1 which is the output of that neuron.

```
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))
```

4 Testing

To evaluate our project we chose to look at both the MNIST testing data set and draw our own numbers in a image processing program. We experimented with the hidden layers of the network and how many epochs we should run and came up with a pretty good solution. Our network is built of 784 first layer neurons, 100 second layer neurons and 10 last layer neurons. We ran the training data for 30 epochs and came up with a maximum of 96% correct guesses of the 10000 test cases that MNIST came with. For our own testing data we wrote the numbers 0 - 9 on a 28*28 image. The NN got 70% of them correct. One reason for the lower result could be that we did not center or resize the numbers as MNIST did. But our NN still guessed pretty close and "saw" a 3 instead of an 8 and a 2 instead of a 7. Those numbers are similar to each other in their appearances.

The network is fast to build and it only takes around 7 minutes to build our network on a 2018 MBP with standard specs. The hidden layers in the network can be configured to any size. The number of layers can also be configured to any size. Altough we reached the best results with a very simple setup there might be an even better design of the NN that we have not discovered.

5 Delivery

The first steps are to first download the project from github and after that install the necessary plugins. If you run python 3.7 as standard you can remove the 3 after pip.

```
git clone https://github.com/ErikBavenstrand/KTH-AI-PROJECT-ML
pip3 install -r requirements.txt
```

There are two parts of running the program. The first one is to build a NN. The syntax for that is as follows:

```
python3 main.py 784 <variable number of layers> 10 networkName.bin

*example of our NN*
python3 main.py 784 100 10 NN.bin
```

This will run until the NN is complete. The outputs from the program will tell you how many of the MNIST test cases the NN correctly guessed. If you would like to test the network with your own custom testing data (fileName.png) just enter the following commands:

```
python3 main.py <networkName.bin>
What file would you like to read? <fileName>
```