

5_Neural_Networks_students

July 13, 2019

1 HaMLeT

1.1 Session 5: Neural Networks

by Raphael Kolk

1.1.1 Goal of this Session

In this session you will, step by step, implement a neural network yourself without using any deep learning libraries. You should already be familiar with Python as well as NumPy (a package for scientific computing with Python).

1.1.2 Tasks

Task 0: Please execute the following cell which is a workaround for data loading and ignore it until further notice.

```
In [9]: from six.moves import urllib
        from sklearn.datasets import fetch_mldata
        class MNIST:
            def __init__(self):
                # Alternative method to load MNIST, if mldata.org is down
                from scipy.io import loadmat
                mnist_alternative_url = "https://github.com/amplab/datascience-sp14/raw/master"
                mnist_path = "./mnist-original.mat"
                response = urllib.request.urlopen(mnist_alternative_url)
                with open(mnist_path, "wb") as f:
                    content = response.read()
                    f.write(content)
                mnist_raw = loadmat(mnist_path)
                self.data = mnist_raw["data"].T
                self.target = mnist_raw["label"][0]
```

Task 1a: Implement the sigmoid function! Make sure it can handle a vector of values.

```
In [10]: import numpy as np

        def sigmoid(z):
```

```

sig_val = 1/(1+np.exp(-z))
return sig_val
# ----- Add code in between these comments -----
# -----

```

Task 1b: For later use in the backpropagation algorithm calculate the first derivative of the sigmoid function and implement it.

```

In [11]: def sigmoid_derivative(z):
        val = sigmoid(z) * (1-sigmoid(z))
        return val
        # ----- Add code in between these comments -----
        # -----

```

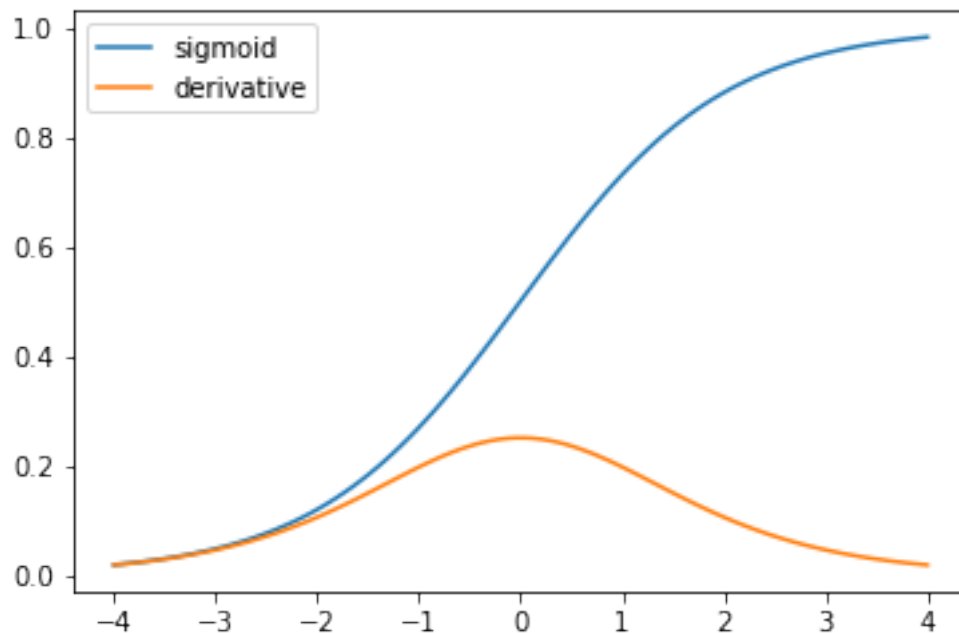
Task 1c: Plot the implemented sigmoid function and its derivative by executing the next cell to ensure that it works properly!

```

In [12]: %matplotlib inline
import matplotlib.pyplot as plt

x = np.linspace(-4, 4, num=100)
plt.plot(x, sigmoid(x))
plt.plot(x, sigmoid_derivative(x))
plt.legend(("sigmoid", "derivative"))
plt.show()

```



The centerpiece is the `Network` class which we use to represent a neural network. The following cell starts to implement the network object by defining the `__init__(self, ...)` method that is the constructor of the class and initializes the member variables. The biases and weights in the `Network` object are all initialized randomly using a Gaussian distribution with mean 0 and standard deviation 1. This random initialization gives our stochastic gradient descent algorithm a place to start from.

The list `sizes`, which contains the number of neurons in the respective layers of the network, is given as a parameter. For example, if the list was `[2, 3, 1]`, then it would be a three-layer network with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron.

After initialization the class instance should contain the following four member variables:

- `self.num_layers` is the total number of layers in the network.
- `self.sizes` is the same as the parameter `sizes`.
- `self.biases` is a list of NumPy vectors. The biases for each hidden and output layer are stored in the respective vector of the list. The biases for the network are initialized randomly, using a Gaussian distribution with mean 0 and variance 1. Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from preceding layers.
- `self.weights` is a list of NumPy matrices, each storing the weights of the connection between layers. So for example `self.weights[1]` is a NumPy matrix storing the weights connecting the second and third layers of neurons. (It's not the first and second layers, since Python's list indexing starts at 0.) Since `self.weights[1]` is rather verbose, let's just denote that matrix w . It's a matrix such that w_{jk} is the weight for the connection between the k -th neuron in the second layer, and the j -th neuron in the third layer.

To generate random values you are supposed to use the NumPy function `np.random.randn()`. Look up the NumPy documentation for further information on the usage.

Task 2a: Implement the code initializing the four member variables of the network!

```
In [13]: import jdc # import for jupyter magic command "%add_to".
import random

class Network(object):

    def __init__(self, sizes):
        """Initialize network with random weights and biases."""
        #self.num_layers = len(sizes)
        #self.sizes = sizes

        #self.biases = np.random.rand(len(sizes) - 1,1)
        #self.weights = np.array(self.layers-1, None, None)
        #self.w = []
        #self.weights=[np.random.rand(self.sizes[i],self.sizes[i+1]) for i in np.arange(1, len(sizes))]

        self.biases = [np.random.normal(0, 1, (num_neu, 1)) for num_neu in sizes[1:]]
```

```

self.sizes = sizes
self.num_layers = len(sizes)
self.weights = [np.random.normal(0, 1, (sizes[i+1], sizes[i])) for i in range

#self.w1=np.random.rand(self.sizes[0],self.sizes[1])
#self.w2=np.random.rand(self.sizes[1],self.sizes[2])
#self.w3=np.random.rand(self.sizes[2],self.sizes[3])
#self.weights = np.array([np.array([np.random.rand(self.sizes[0],self.sizes[1]
#self.weights=[self.w1,self.w2,self.w3]

def __str__(self):
    """String representation of class"""
    return "Number of layers: " + str(self.num_layers) + "\n" + \
        "Sizes of layers: " + str(self.sizes) + "\n" + \
        "Biases:" + "\n" + "\n".join([str(s) for s in self.biases]) + "\n" + \
        "Weights:" + "\n" + "\n".join([str(s) for s in self.weights])

```

Task 2b: In order to verify your implementation, initialize a new Network with an input layer of 3 neurons, two hidden layers with 4 neurons each and an output layer of 2 neurons.

```

In [14]: net = Network([3,4,4,2])
         print(net)

         print("Bias shape",net.biases.shape)
         print("Bias Type",net.biases.dtype)
         print("weights",net.weights[0].shape)

Number of layers: 4
Sizes of layers: [3, 4, 4, 2]
Biases:
[ 0.87555063]
[ 0.97887214]
[ 0.80012319]
Weights:
[[ 0.80370117  0.30615041  0.39916758  0.54097387]
 [ 0.35118877  0.64506271  0.99574283  0.5361292 ]
 [ 0.96524625  0.38240361  0.59352      0.15339779]]
[[ 0.07192149  0.8859335   0.24003605  0.00666043]
 [ 0.30454755  0.66931365  0.87776091  0.43139234]
 [ 0.90970345  0.44218939  0.82025021  0.2551223 ]
 [ 0.78610082  0.31841182  0.44230508  0.39755    ]]
Bias shape (3, 1)
Bias Type float64
weights (3, 4)

```

As mentioned above the weight matrix `self.weights[1]` denoted as w stores the weights such that w_{jk} is the weight for the connection between the k -th neuron in the second layer and the j -th neuron in the third layer This ordering of the j and k indices may seem strange - surely it would

make more sense to swap the j and k indices around? The big advantage of using this ordering is that it means the vector of activations of the third layer of neurons is:

$$a' = \sigma(wa + b)$$

Task 3a: Using the earlier implemented `sigmoid()` function implement the `feed_forward()` method, which applies the equation stated above layer by layer to a given input vector `a`.

Using the built-in `zip()` iterator helps you keep your code cleaner and thereby easier to understand.

```
In [20]: %%add_to Network
def feed_forward(self, a):
    """Return the output of the network if ``a`` is input."""
    #a_dash = 0

    #for i in np.arange(0,len(self.biases)):
    #    #print(i)
    #    #print("a_dash",sigmoid(np.sum(np.matmul(a.T, self.weights[i]))))
    # L1=sigmoid(np.sum(np.matmul(a.T, self.weights[0])))
    #print(L1)
    #L2=sigmoid(np.sum(np.matmul(L1, self.weights[1]))+self.biases[0])
    #print(L2)
    #L3=sigmoid(np.sum(np.matmul(L2, self.weights[2]))+self.biases[1])
    #print(L2)

    #a_dash = a_dash + sigmoid(np.sum(np.matmul(a_dash, self.weights[i])+ self.biases[i]))
    # ...

    self.z = np.dot(a.T,self.weights[0])
    self.z2=sigmoid(self.z)
    self.z3=np.dot(self.z2,self.weights[1])
    self.z4=sigmoid(self.z3)
    self.z5= np.dot(self.z4,self.weights[2])
    a_dash = sigmoid(self.z5)
    # ...

    for (w, b) in zip(self.weights, self.biases):
        a = sigmoid(np.matmul(w,a)+b)
    return a

# return a_dash
```

Task 3b: In order to verify your implementation feed the randomly initialized network from task 2b the input vector `[1.2, 3.4, 5.6]`.

```
In [21]: input = np.array([1.2, 3.4, 5.6]).reshape((-1, 1))
         print(net.feed_forward(input))
```

ValueError

Traceback (most recent call last)

```
<ipython-input-21-9ba9293946c6> in <module>()
    1 input = np.array([1.2, 3.4, 5.6]).reshape((-1, 1))
----> 2 print(net.feed_forward(input))
```

```
<string> in feed_forward(self, a)
```

ValueError: shapes (3,4) and (3,1) not aligned: 4 (dim 1) != 3 (dim 0)

Now we are having a look at the actual training and validation dataset, which was introduced in the preparation material.

Using scikit-learn we can easily import the MNIST dataset. After loading, images and labels are accessible in two numpy.ndarrays.

- Each of the 70000 rows in `mnist.data` stores one image (28 x 28 pixel) of a handwritten digit as a vector with a length of 784.
- `mnist.target` stores the corresponding labels as an integer number.

Task 4a: To familiarize yourself with the data, extract and plot ten different images of each digit (0 - 9).

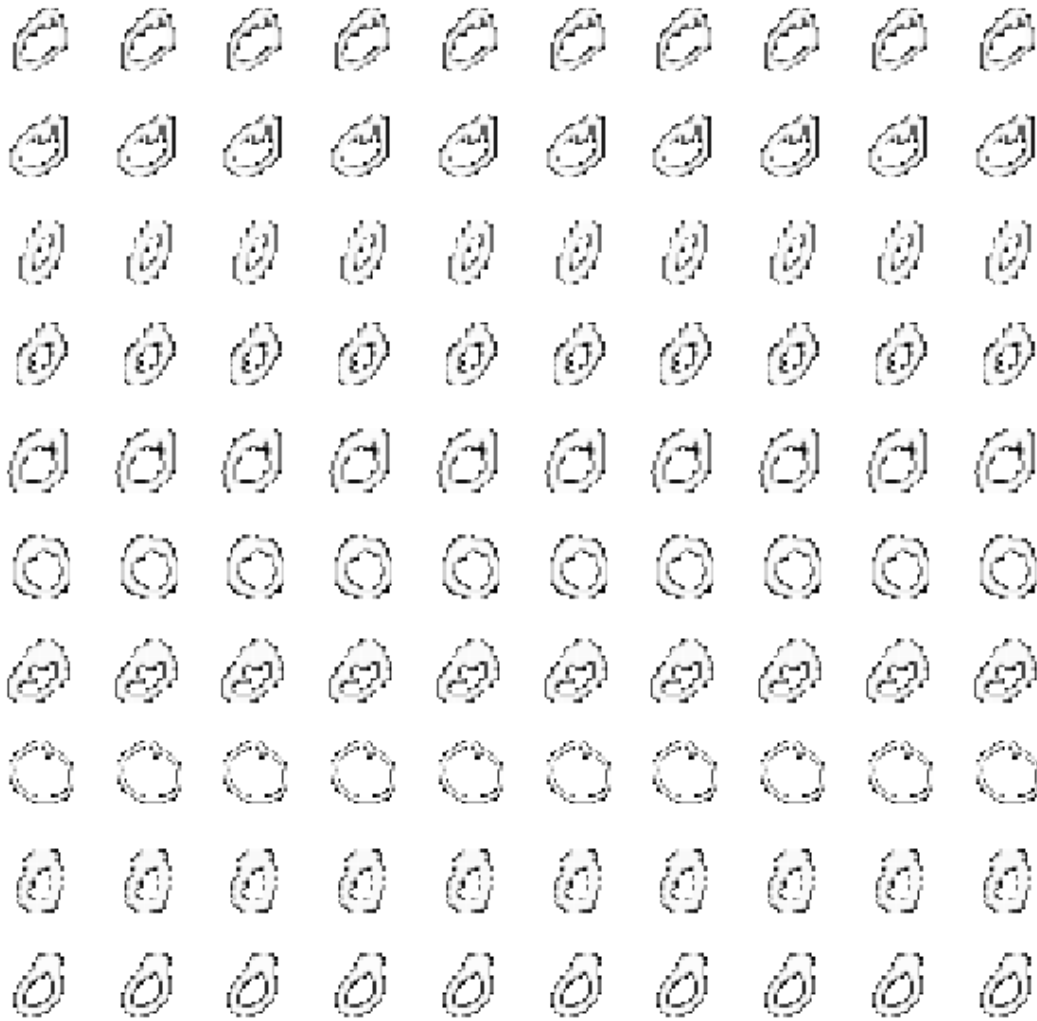
```
In [22]: from sklearn.datasets import fetch_mldata
mnist = MNIST()
print(mnist.target [ :10])

%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10, 10))

for i in range(0, 10):
    for j in range (0, 10):
        image = mnist.data[i]
        image = (np.reshape(image, (28, 28)) * 255).astype(np.uint8)
        plt.subplot(10, 10, 10*i+j+1)
        plt.imshow(image, cmap='gist_yarg')
        fig.axes[10*i+j].set_axis_off()

plt.show()

[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```



Since we want to design our network with 10 output neurons and train it in a way that the output activation for the digit j becomes 1.0 at the j -th neuron, we need to vectorize our label.

Task 4b: Implement the function `vectorize()` which accepts an integer j between 0 and 9 and returns a 10-dimensional unit vector with a 1.0 in the j -th position and zeroes elsewhere. This is used to convert a digit (0...9) into a corresponding desired output from the neural network.

```
In [25]: def vectorize(j):
          # ----- Add code in between these comments -----
          vect = np.zeros((10))
          vect[j] = 1
          return vect
          # -----

          print(vectorize(9))
```

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
```

We need to load and prepare the MNIST dataset to be used in the training of our network. Therefore the following steps need to be taken:

1. Load the data as we did in task 4a and convert images and labels into lists of samples. This is already implemented in the function below.
2. The images in the dataset are integer-valued with a maximum intensity of 255. Before feeding them to the input neuron we have to ensure that their pixel values are between 0.0 and 1.0.
3. The labels in the dataset are integers, so we need to vectorize them using the function implemented above.
4. We have to merge the list of images and the list of labels so that we end up with one list of tuples. Each tuple contains one image and one label.
5. The list of tuples needs to be split into `training_data` (60000 samples) and `validation_data` (10000 samples).

Task 4c: Implement steps 2 to 5 in the function below.

```
In [26]: from sklearn.datasets import fetch_mldata
         from sklearn.preprocessing import StandardScaler

         def load_data():
             mnist = MNIST()
             images = [i.reshape((784,1)) for i in mnist.data]

             labels = [int(l) for l in mnist.target]
             print("labels",labels[:3])
             images = [i / max(i) for i in images]
             # print("images",images[:3])
             labels = [vectorize(i) for i in labels]
             print("labels",labels[:3])
             dset = np.array([images,labels])
             training_data = dset[:60000,:]
             validation_data = dset[60001:70000,:]

             # ----- Add code in between these comments -----
             # -----

             return training_data, validation_data

         training_data, validation_data = load_data()
```



```
labels [0, 0, 0]
labels [array([ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]), array([ 1.,  0.,  0.,  0.,
```

After each training epoch the performance of the network is evaluated by feeding the validation data into the network and comparing the output to the ground truth label. The neural network's output is assumed to be the index of the neuron with the highest activation in the output layer.

Task 5a: Implement the class method `evaluate()`, which feeds the network the validation data sample by sample using the `feed_forward()` method and return the number of correct classifications.

```
In [28]: %%add_to Network
def evaluate(self, validation_data):
    a_dash=[net.feed_forward(i) for i in validation_data[:784]]
    count = 0
    if a_dash == validation_data[784:]:
        count+=1
    return count
```

Task 5b: Test your implemented code by applying the method to a randomly initialized and untrained network. Since the odds for guessing correctly are 1 to 10 the evaluation output should be roughly 10%. You can execute the code several times to test the assumption.

```
In [29]: training_data, validation_data = load_data()
net = Network([784, 30, 10])
print("Correct classifications of randomly initialized network: {0} of {1}".format(net.evaluate(validation_data), len(validation_data)))
```

```
labels [0, 0, 0]
labels [array([ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]), array([ 1.,  0.,  0.,  0.,
```

IndexError

Traceback (most recent call last)

```
<ipython-input-29-7e7b4c72a498> in <module>()
    1 training_data, validation_data = load_data()
----> 2 net = Network([784, 30, 10])
    3 print("Correct classifications of randomly initialized network: {0} of {1}".format(net.evaluate(validation_data), len(validation_data)))

<ipython-input-13-4581c8839ffe> in __init__(self, sizes)
    11         #self.weights = np.array(self.layers-1, None, None)
    12         #self.w = []
---> 13         self.weights=[np.random.rand(self.sizes[i],self.sizes[i+1]) for i in np.arange(len(sizes)-1)]
```

```

14
15         #self.w1=np.random.rand(self.sizes[0],self.sizes[1])

<ipython-input-13-4581c8839ffe> in <listcomp>(.0)
11         #self.weights = np.array(self.layers-1, None, None)
12         #self.w = []
--> 13         self.weights=[np.random.rand(self.sizes[i],self.sizes[i+1]) for i in np.ar
14
15         #self.w1=np.random.rand(self.sizes[0],self.sizes[1])

```

IndexError: list index out of range

The `train()` method, as its name implies, trains our network. The modification of the weights and biases itself is done by the already implemented function `update_mini_batch()`, which again uses the backpropagation algorithm to calculate the gradient.

However, the `train()` method implements the stochastic gradient descent, which means that the gradient is calculated for a subset of training samples, a so called mini batch, rather than for the whole training set. Besides having other advantages this is a lot faster and shortens the time needed for training. After the network has been fed the training set, split into mini batches, and the parameters have been updated accordingly, the process is repeated in the next epoch, thereby iteratively tuning the network parameters to a near-optimal state.

In our implementation of the `train()` method the following steps need to be taken: - The training and validation data is given formatted as described and implemented in task 4c. - For every epoch (the number of epochs is given by `num_epochs`): - The order of the training data is randomly shuffled. - The training dataset is split into mini batches of the size `mini_batch_size`. - The `update_mini_batch()` method is applied to each mini batch. - The network's performance is evaluated and displayed.

Task 6a: Implement the missing code in the `train()` method as described above.

```

In [ ]: %%add_to Network
def train(self, training_data, validation_data, num_epochs, mini_batch_size, learning_rate):
    for j in range(num_epochs):
        # ----- Add code in between these comments -----
        # -----

def update_mini_batch(self, mini_batch, learning_rate):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backpropagation(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(learning_rate/len(mini_batch))*nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(learning_rate/len(mini_batch))*nb for b, nb in zip(self.biases, nabla_b)]

```

```

def backpropagation(self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = (activations[-1] - y) * sigmoid_derivative(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_derivative(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)

```

Using all the implemented functionality above you can finally train a network on the MNIST dataset. Therefore: - Load the training and validation data. - Instantiate a new network with - 784 neurons in the input layer, - 30 neurons in the first and only hidden layer and - 10 neurons in the output layer. - Train the network using the training and validation data - for 30 epochs, - with a mini batch size of 10 and - a learning rate of 3.0.

Task 6b: Implement the training process as stated above.

```

In [ ]: # ----- Add code in between these comments -----
        # -----

```

Task 7: Have fun changing the training parameters or the design of the network and observing the resulting classification rate.

```

In [ ]: # ----- Add code in between these comments -----
        # -----

```

1.1.3 Feedback

Aaaaaand we're done

If you have any suggestions on how we could improve this session, please let us know in the following cell. What did you particularly like or dislike? Did you miss any contents?

1.1.4 Additional Tasks

Task 8: Expand your train function by a learning rate decay / scheduler.

```
In [ ]: # ----- Add code in between these comments -----  
        # -----
```