NEURAL NETWORKS

Ron Michael Acda
Activity 9

Objectives

01

Introduce the mathematics and algorithm behind neural networks

02

Demonstrate NN learning using the MNIST Handwritten Digits Dataset

03

Build a star-galaxy classifier

Neural Network

Neural networks (NN) can be thought of as multi-layer perceptrons, with the first layer being the input layer. The nodes or neurons on the first layer are connected to the first hidden layer, then to 2^{nd}, and so on until the output layer.

Note:

1. For the classification of images, the number of nodes/neurons in the input layer is the number of pixels in the image.
2. The number of classes is the number of neurons on the output layer.

Neural networks learn by minimizing the error of the cost function; that is, it tries to minimize the error between the predicted output at the output layer and the actual output.

Neural Network Algorithm (from Nielsen)

1. **Initialization** of random weights and biases. Let l=1,2,…,L be the lth layer. Initialize the activation at the first layer, a^l**.**
2. Iterate through each training example x:
   1. **Feedforward:** For each l=2,3,…,L, compute $z^l = w^l a^{x,l-1} + b^l$ and the activation $a^{x,l} = \sigma(z^{x,l})$**.** Here, w^l is the weight matrix where $\mathrm{w}_{\mathrm{jk}}^{l}$ is the weight from the kth neuron in the (l-1)th layer to the jth neuron in the lth layer.
   2. **Output error:** $\delta^{x,L} = (a^L - y) \odot \sigma'(z^L)$ where y is the actual output, σ' is the derivative of the threshold function, and $\odot$ is the Hadamard product (element-wise matrix multiplication).
   3. **Backpropagation**: For each l=L-1,L-2,…,2 compute the error vector $\delta^{x,l} = \left(\left(w^{l+1}\right)^T \delta^{x,l+1}\right) \odot \sigma'(z^{x,l})$
3. **Gradient descent**: Perform the update rule:

$$w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} \left(a^{x,l-1}\right)^T$$

$$b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$$

Step 1: Initialization

```
class Network:

    def __init__(self, layers=5, neuron=5):
        self.layers = layers
        self.neuron = neuron

    def set_train(self, train_data, targets):
        self.x_train, self.y_train = train_data[0], train_data[1]
        self.train_data =  [(self.x_train[i,:,:], self.y_train[:,i]) for i in range(np.shape(self.x_train)[0])]

        # Initialize random weights. Returns a list containing:
        # Mx(neuron) 2D array in the first index, where M is the number of initial neurons, and (neuron)xN, where N is the number
            of neurons in the output layer, and (neuron)x(neuron) in the middle layers.
        self.w_ = [np.random.rand(self.neuron, self.neuron) for i in range(self.layers-1)]
        input_w, output_w = np.random.rand(self.neuron, len(self.x_train[0].flatten())), np.random.rand(targets, self.neuron)
        self.w_.insert(0, input_w)
        self.w_.append(output_w)

        # Initialize random column bias vectors. Returns a list containing:
        # 1xM row vector in the first index, where M is the number of initial neurons, and 1xN, where N is the number of neurons in
            the output layer, and 1x(neuron) in the middle layers.
        self.bias_ = [np.random.rand(self.neuron, 1) for i in range(self.layers)]
        input_b, output_b = np.zeros((len(self.x_train[0].flatten()),1) ), np.random.rand(targets,1)
        self.bias_.insert(0, input_b)
        self.bias_.append(output_b)

        return self
```

Step 2a: Feedforward

```
def feedforward(self, input):
    input = np.column_stack([np.array(input)])
    initial_act=  input + np.array(self.bias_[0])
    self.activations_ = [initial_act]
    self.z_ = [np.array(input)]
    for l in range(0,self.layers+1): #Loop over l = 0,1,2,...,L-1 where L-1 is the output layer
        z = np.matmul(self.w_[l], np.column_stack([np.array(self.activations_[l])])) + self.bias_[l+1]
        self.z_.append(z)
        self.activations_.append(self.threshold(z))
    return self
```

Step 2b: Output Error

```
def output_error(self, output):
    err = (self.activations_[-1] - np.column_stack([output]))*self.threshold_der(self.z_[-1])
    return err
```

Step 2c: Backpropagation

```
def backpropagate(self, output):
    self.error_array = [self.output_error(output)]
    for l in range(-1, -self.layers-1, -1):
        error_vector = np.dot(np.transpose(np.array(self.w_[l])),
        np.array(self.error_array[l]))*np.column_stack([np.array(self.threshold_der(self.z_[l-1]))])
        self.error_array.insert(0, error_vector)
    return self

def threshold(self, z): #Hyperbolic tangent
    return np.tanh(z)

def threshold_der(self, z): #Derivative of the activation function
    return 1 + self.threshold(z)**2
```

Step 3: Iteration through batches and update rule

```
def train(self, test, epochs = 10, eta = 0.01, batch_size=100):
        for iter in range(epochs+1):
            if test != None:
                print('Epoch {a}/{b}: {c}'.format(a=iter, b= epochs, c=self.evaluate(test)))
            np.random.shuffle(self.train_data)
            mini_batches = [self.train_data[k:k+batch_size] for k in range(0,len(self.train_data), batch_size)]
            for mini_batch in mini_batches:
                for (m,n) in mini_batch:
                    self.feedforward(np.array(m.flatten()))
                    self.backpropagate(n)
                    for l in range(-1, -self.layers-2, -1):
                        update_w = np.dot(self.error_array[l], self.activations_[l-1].T)
                        update_b = self.error_array[l]
                        self.w_[l] -= update_w*(eta/batch_size)
                        self.bias_[l] -= update_b*(eta/batch_size)
        return self.w_, self.bias_
```

Test: MNIST Handwritten Digits Dataset

A standard test for NN algorithms is to test it on the MNIST (Modified National Institute of Standards and Technology) Database, a database of 60000 handwritten digits from 0 to 9. Each image is 28x28, so the input layer should contain 784 nodes. The output layer should have 10 nodes.

Test: MNIST Handwritten Digits Dataset

We load the dataset

```
from keras.datasets import mnist
(train_X, train_y), (test_X, test_y) = mnist.load_data()
```

The output ranges from 0 to 9, but for our NN to work properly, we need to perform **one-hot encoding**. For example, we need to convert the output into column vectors like 1= 1000000000, 2= 0100000000, ..., 9 = 0000000001.

```
#Perform one-hot encoding
train_Y = np.zeros((train_y.max()+1, train_y.shape[0]))
train_Y[train_y, np.arange(train_y.shape[0])] = 1
test_Y = np.zeros((test_y.max()+1, len(test_y)))
test_Y[test_y, np.arange(len(test_y))]=1
```

Test: MNIST Handwritten Digits Dataset

Now we train the NN. Here, we have 1 hidden layer with 100 nodes. We print the accuracy (0 to 1) per epoch, tested against the test data per epoch.

```
net1 = Network(1, 100)
net1.set_train((train_X, train_Y), 10)
test_data = (test_X, test_y)
test_data_tuple= [(test_X[i], test_y[i]) for i in
range(len(test_data[0]))]
net1.train(test=test_data_tuple, epochs=20, eta=0.1,
batch_size=100)
```

Test: MNIST Handwritten Digits Dataset

Here's the output. Note that even after a single epoch, the NN shoots up to a fairly high accuracy. The initial epoch 0 has low accuracy as expected since the weights and biases are initially random.

```
Epoch 0/20: 0.098
Epoch 1/20: 0.8314
Epoch 2/20: 0.8467
Epoch 3/20: 0.8446
Epoch 4/20: 0.8421
Epoch 5/20: 0.8425
Epoch 6/20: 0.8381
Epoch 7/20: 0.8405
Epoch 8/20: 0.8542
Epoch 9/20: 0.838
Epoch 10/20: 0.8372
Epoch 11/20: 0.8359
Epoch 12/20: 0.8406
Epoch 13/20: 0.8411
Epoch 14/20: 0.844
Epoch 15/20: 0.8444
Epoch 16/20: 0.8446
Epoch 17/20: 0.8457
Epoch 18/20: 0.8275
Epoch 19/20: 0.8253
```

Conclusion: Our NN works!

Test: MNIST Handwritten Digits Dataset

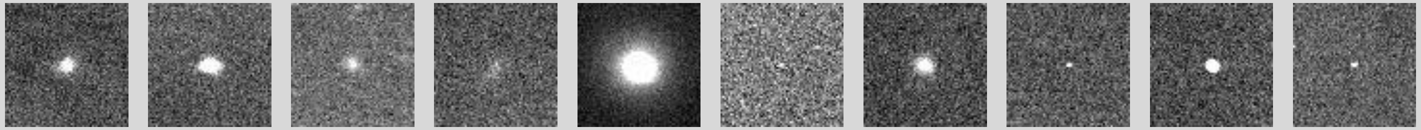
Modern architectures of NN can have accuracies as high as 96-99% in the MNIST Dataset. Our NN is quite primitive, but the accuracy is

```
Epoch 0/20: 0.098
Epoch 1/20: 0.8314
Epoch 2/20: 0.8467
Epoch 3/20: 0.8446
Epoch 4/20: 0.8421
Epoch 5/20: 0.8425
Epoch 6/20: 0.8381
Epoch 7/20: 0.8405
Epoch 8/20: 0.8542
Epoch 9/20: 0.838
Epoch 10/20: 0.8372
Epoch 11/20: 0.8359
Epoch 12/20: 0.8406
Epoch 13/20: 0.8411
Epoch 14/20: 0.844
Epoch 15/20: 0.8444
Epoch 16/20: 0.8446
Epoch 17/20: 0.8457
Epoch 18/20: 0.8275
Epoch 19/20: 0.8253
```

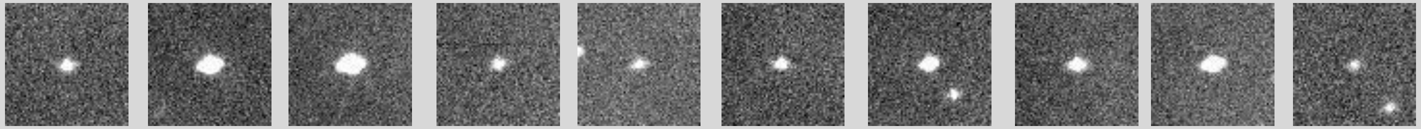
Test: Star-Galaxy Classifier

We perform classification of a star-galaxy dataset from Kaggle

Galaxy



Star



Test: Star-Galaxy Classifier

Result (eta = 10, epochs = 10, batch_size = 100, 2 hidden layers, 100 neurons each hidden layer).

Maximum accuracy: 76%

```
Epoch 0/20: 0.7632898696088265
Epoch 1/20: 0.5586760280842528
Epoch 2/20: 0.23671013039117353
Epoch 3/20: 0.23671013039117353
Epoch 4/20: 0.23671013039117353
Epoch 5/20: 0.2567703109327984
Epoch 6/20: 0.23771313941825475
Epoch 7/20: 0.6680040120361084
Epoch 8/20: 0.7622868605817452
Epoch 9/20: 0.4242728184553661
Epoch 10/20: 0.23671013039117353
Epoch 11/20: 0.7432296890672017
Epoch 12/20: 0.238716148445336
Epoch 13/20: 0.23671013039117353
Epoch 14/20: 0.7622868605817452
Epoch 15/20: 0.7642928786359077
Epoch 16/20: 0.23570712136409228
Epoch 17/20: 0.6820461384152458
Epoch 18/20: 0.23671013039117353
Epoch 19/20: 0.5767301905717152
Epoch 20/20: 0.23671013039117353
```

The erratic spikes in accuracy can be attributed to the very high learning rate (tendency to overshoot).

This is a general problem with unsupervised learning algorithms-- **hyperparameter tuning**: The hyperparameter for this NN architecture is the learning rate eta, the number of epochs, the number of hidden layers, neurons, and the batch size.

The higher the learning rate = the faster it converges to a local minimum, but the greater tendency to overshoot.

The lower the learning rate = the slower the convergence, but the lesser the chance to overshoot.

More neurons/layers = potentially can be trained for more complex classification tasks, but the harder it is to train.

Batch size = The NN architecture I implemented is **mini-batch stochastic gradient descent**, which is optimized for speed for larger datasets. Due to the stochastic nature of the algorithm, there is a chance for overshooting at each epoch.

Note that it is actually hard to train the NN because judging from the images, they actually look similar! Even a human would have a hard time classifying them.

Self-Reflection

Self-score: 105/100

This is by far the most interesting module I've taken. I've gained a deeper understanding of how NN works by understanding the mathematics behind it. I wrote the program **from scratch** just by analyzing the algorithm provided by Nielsen without looking at the code snippets provided in the textbook.

I also tested it in a standard dataset (MNIST) and applied it to a **real-world dataset from Kaggle** (star-galaxy classifier)

References

Nielsen, M.A. (2015). *Neural Networks and Deep Learning*. Determination Press. From http://neuralnetworksanddeeplearning.com/chap2.html

3blue1brown neural network playlist https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

Star-Galaxy dataset https://www.kaggle.com/datasets/divyansh22/dummy-astronomy-data