Deep Learning Assignment 1: Neural Network Fundamentals

Huey Ning Lok

October 19, 2019

In this assignment, we will implement a neural network from scratch either in Numpy or in PyTorch but without using any of PyTorch's implementations for layers, optimizers, loss functions, callbacks etc - all significant logic relating to the neural network should be written by the student. We are only allowed to use PyTorch for Numpy-like GPU-accelerated tensor manipulation, dataset-related data structures, and optionally gradient calculations (autograd).

Requirements:

Implement a fully-connected (feed-forward) neural network from scratch in your library of choice. The NN must include at least dense layers, activations, and sigmoid/softmax in case of classification. You must also write your own optimizer and loss function (for example, stochastic gradient descent and binary cross-entropy for binary classification). You may use PyTorch's built-in gradient calculator (autograd) or you may write one yourself.

Finally, include runtime and results on a public dataset (MNIST? CIFAR10?).

Please document your code and write a brief summary which includes any interesting technical details and your results.

Extensions:

Implement one or more of the following, and include a comparison of the model's performance with/without each component:

- Manually calculate the gradients of each layer, and perform back propagation manually.
- More than 1 activation function - sigmoid, tanh, relu etc.
- More than 1 optimizer - SGD, Momentum, RMSProp, Adam etc.
- Regularization - L2/weight decay, dropout, possibly augmentations if image data etc.

Submission:

Please put all your code, results, and discussion in a jupyter notebook, and submit a pdf of the notebook for ease of grading. You may also make a secondary submission of the notebook (*.ipynb) or the code (*.py), but this is optional and might not be looked at.

Assignment Layout

The assignment will consist of two main parts:

1. Conceptual breakdown of the fundamentals of neural network architecture and the training process. This will allow me to grasp the fundamentals needed to code a neural network from scratch.

2. Coding of the neural network using Pytorch.

1 Conceptual Breakdown

The steps for training a neural network can be broken down as follows:

1.1 Initialize the network parameters.

1.1.1 Fixed Parameters

The fixed parameters are the number of neurons in each layer, aka the layer size. These are usually defined by the user and determined as according to our dataset needs.

Input size and hidden layer size The number of neurons in the input layer and hidden layer(s) can be defined arbitrarily or according to common best practices. Increasing the number of neurons (and number of layers) can increase the model capacity, i.e. the space of representable functions, which allows the modelling of more complex relations, but also makes the model prone to overfitting (Kaparthy, 2019). It is a general rule of thumb that the number of learnable parameters should not exceed the number of data points. That being said, Kaparthy argues that neural networks that are overfitting can and should be adjusted using regularization methods vs reducing the neuron count (2019).

Given the layer size(s) s, we can calculate the number of learnable parameters, p, from a fully-connected network with N number of layers as follows:

$p_{\text{weights}} = \sum_{i=0}^{N} s_i \cdot s_{i+1}$

$p_{\text{bias}} = \sum_{i=1}^{N} s_i$

$p = p_{\text{weights}} + p_{\text{bias}}$

In the equations above, $i = 0$ denotes the input layer and $i = N$ denotes the output layer.

Output size In cases of binary classification or regression, the final output size should be one. In multilabel classification, the final output size should correspond to the number of classes.

1.1.2 Learnable Parameters

While fixed parameters are only initialized once when defining the network, learnable parameters are continuously updated throughout the training process to minimize training loss and adjust the model's predicted outputs to be closer to the actual output.

The learnable parameters consist of the model weights and biases, and are usually randomly initialized.

Weights The weights determine the relationship between the neurons in a given layer to the neurons in the layer before and after it. The heavier the weight, the stronger the interneuron connection, and the stronger the implication that the neuron significantly affects our model's prediction ability. This is mostly relevant in regards to the input layer where the neurons may represent clear features of a given dataset. Neurons in the hidden layers are often less interpretable since they have undergone multiple transformations, though there have been relatively successful attempts at visualizing hidden layer features as shown in Olah, Mordvintsev, and Schubert (2017).

Bias The bias term allows us to adjust the displacement of the output functions of any given layer to achieve a better fit that reduces loss, i.e. the difference between predicted $\hat{y}$ and actual y.

1.2 Perform a feed-forward pass to compute the predicted output.

According to Karpathy (2016), "[t]he forward pass of a fully-connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function". The forward pass allows us to use the parameters that we have been training to predict an output based on our input. The steps are as follows:

For each layer in the network:

1.2.1 Multiply the layer inputs by their corresponding weights and add the bias offset.

Given a layer with weights $\vec{w}$, inputs $\vec{x}$, and bias term b, the output (before activation) can be calculated as follows:

$$o = \vec{w} \cdot \vec{x} + b$$

1.2.2 2.2 Apply an activation function.

Since the output is linear, we need to apply an activation function to model non-linearity within our layer outputs. Given an activation function $g(x)$, our output can then be written as:

$$o = g(\vec{w} \cdot \vec{x} + b)$$

Example activation functions include the sigmoid, softmax, rectified linear units (ReLU), etc.

1.3 Compute the loss.

The loss is defined as the difference between the predicted output $\hat{y}$ and the actual output y for an input $\vec{x}$. The loss is calculated using the loss function, which can take on different forms as called for by the given problem.

For regression problems, we typically use the mean squared error (MSE) as a loss function. For classification problems, we typically use cross-entropy loss (also called log loss).

1.4 Update parameters using gradient descent and backpropagation.

Given a MSE loss function $L(X) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2$, where X denotes a set of input-output pairs $X = \{(\vec{x_1}, y_1), ..., (\vec{x_n}, y_n)\}$, $\hat{y}$ denotes the predicted output, and y denotes the actual output, we see that when $\hat{y} = y$, $L(X) = 0$. As such, a natural objective is to minimize the loss function to achieve more accurate output. This can be done by adjusting the parameters that will affect the output, as shown in section 2.2.

1.4.1 Calculating loss function gradient w.r.t. the parameters using backpropagation.

The backpropagation algorithm uses the chain rule to calculate the gradient of the loss function w.r.t. to the parameters, i.e. the weights and biases. Given the final output of our neural network, y, and error E, we can calculate the gradient of the error w.r.t. the output as dE/dy. We then find the gradient of error w.r.t input, $dE/dx = dE/dy * dy/dx$, where dy/dx is the derivative of the activation function w.r.t. the output. After calculating dE/dx for each layer, we can then use this in our calculation of the error gradient w.r.t. the weights of each layer, dE/dw. We then use this error gradient to update our weights through gradient descent.

1.4.2 Minimizing loss and updating parameters using gradient descent.

Since we are trying to minimize the loss, this is an optimization problem that can be solved using gradient descent. We typically use stochastic gradient descent (SGD) whereby a parameter update is performed for each training example x_i and training label y_i (Ruder, 2018). Each iteration of gradient descent would be carried out as follows:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta L(\theta; x^{(i)}; y^{(i)})$$

where α represents the learning rate (usually user-defined and should follow industry best practices) and $\nabla_\theta L(\theta; x^{(i)}; y^{(i)})$ represents the gradient of the loss function w.r.t the parameters.

As seen from the update equation, each gradient descent step will update our parameters in the direction that decreases loss. If the loss term $(\alpha \cdot \nabla_\theta L(\theta; x^{(i)}; y^{(i)})$ is positive, then the parameter values are decreased, i.e. we want to decrease the significance of this parameter since it is increasing loss. If the loss term is negative, then the parameter values are increased, i.e. we want to increase the significance of this parameter since it is decreasing loss.

Through gradient descent, we iteratively update our parameters to achieve a local optima where loss is minimized.

2 Coding the Neural Network

The following Pytorch functionalities were implemented:

- torch.tensor() # to create tensors
- torch.randn() # to create tensors with random input
- torch.matmul() # to perform matrix multiplication
- torch.t() # to transpose matrices
- math operations: torch.exp(), torch.mean()

The autograd functionality was not utilized, so backpropagation was calculated manually.

```
In [0]: # import libraries
        import torch
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        sns.set()
```

2.1 Neural Network Class

The neural network class consists of a forward method to calculate the output, backward method to backpropagate and calculate the loss gradients w.r.t. the weights, optimize method to perform gradient descent and update the weights using the loss gradients, and finally a train method which performs the forward, backward, and optimize methods in order.[1]

The neural network class takes the following input parameters:

- the layer architecture (number of layers, size of layers, etc.)
- loss function
- activation function
- derivative of the activation function
- learning rate (α)

I have defined the loss functions and activation functions (and their derivatives) in separate cells.

```
In [0]: # define neural network class

        class NeuralNetwork:

          def __init__(self, layers, loss_func, active_func, active_func_prime, alpha = 0.01):

            '''
            Basic neural network class that can be trained on a given set of data using the
        self.train method.

            Inputs:

              layers (ARR)

                - Defines the architecture of the neural network.
                - The array should contain sub-arrays that represent the network layers, so that
                  number of sub-arrays == number of layers in network (including input and
        output layers).
```

[1]Link to the code gist can be found in the Appendix

```
          Ex: [[...],[...],[...]] would represent a network with 3 layers.
        - The sub-arrays should each contain 2 numbers, the first representing the
number of input neurons
          and the second representing the number of output neurons.
          Ex: [5,3] would represent a layer with 5 input neurons and 3 output neurons.
          The following layer must accordingly have 3 input neurons, e.g. [3, x]

      loss_func (FUNC)

        - Defines the loss function to be used in the neural network.

      active_func (FUNC)

        - Defines the activation function to be used in the neural network.
        - Only one type of activation function can be chosen, and this activation
function
          will be applied to each layer in the network, except the output layer.

      active_func_prime (FUNC)

        - The derivative of the chosen activation function.

      alpha (FLOAT)

        - Defines the learning rate to be used in the optimization method (SGD).
        - Default: 0.01

    '''

    self.weight_dct = {} # empty dict to store layer weights
    self.layer_output_dct = {} # empty dict to store layer outputs
    self.delta_dct = {} # empty dict to store loss gradients

    for l in range(len(layers)):
      # randomly initialize layer weights according to layer size
      self.weight_dct[l] = torch.randn(layers[l,0], layers[l,1])

    self.loss_func = loss_func # type of loss function
    self.active_func = active_func # type of activation function
    self.active_func_prime = active_func_prime # derivative of activation function
    self.alpha = alpha # learning rate


  def forward(self, layer_input):

    '''
    *** Forward-pass algorithm ***
    Multiplies the layer inputs by layer weights and applies the user-defined activation
function to every layer except the final output layer.

    Inputs:

      layer_input (ARR)
        The layer inputs.

    Outputs:

      layer_output (ARR)
        The output of the final layer.

    '''

    layer_output = 0 # initialize output value as 0

    for i in range(len(self.weight_dct)): # loop through each layer in the network

      layer_output = torch.matmul(layer_input, self.weight_dct[i]) # multiply the layer input by its weights
      if i != len(self.weight_dct) - 1: # apply activation function to all layers but the last

        layer_output = self.active_func(layer_output) # activation function

      self.layer_output_dct[i] = layer_output # add layer output to the layer output dict

      layer_input = layer_output # pass the output as the input to the next layer

    return layer_output # return final output of last layer in the network


  def backward(self, output, y_true):

    '''
    *** Backpropagation algorithm ***
    Loops through the network in reverse to calculate and sstore the
    error derivative w.r.t. the total input of a node, dE/dx.
    The dE/dx's are stored in delta_dct.

    Inputs:

      output (ARR)
        The predicted y values, i.e. final layer outputs.

      y_true (ARR)
        The true y values.

    '''

    err = self.loss_func(output, y_true) # output error, dE/dy

    # derivative of error w.r.t input, dE/dx = dE/dy * dy/dx,
    # where dy/dx is the derivative of the activation function w.r.t. the output
    delta = err * self.active_func_prime(output)

           # loop through the layers in reverse to backprop
    for i in reversed(range(1, len(self.weight_dct))):


      err = torch.matmul(delta, torch.t(self.weight_dct[i])) # dE/dy for the layer
      delta = err * self.active_func_prime(self.layer_output_dct[i]) # dE/dx for the layer


      self.delta_dct[i] = delta # save the deltas for calculating the loss gradient w.r.t. params, dE/dw



  def optimize(self, layer_input):

    '''
    *** Optimization step: Gradient Descent ***
    Loops through the network and calculates the the error derivative w.r.t. the weights
coming into the nodes of each layer, dE/dw. Ths loss gradient is then used to update the weights
by multiplying it with a learning_rate factor.

    Inputs:

      layer_input (ARR)
        The initial input of the first layer, aka the "X" data.

    '''
    # loop through all layers except the output layer which doesn't have params to optimize

    for i in range(len(self.weight_dct)-1):

      loss_gradient = torch.matmul(torch.t(layer_input), self.delta_dct[i+1]) #calculate dE/dw
      self.weight_dct[i] -= self.alpha * loss_gradient # update params using α * dE/dw
```

```
            # the input for the next layer, i + 1, will be the output from the layer before it, i
            layer_input = self.layer_output_dct[i]


        def train(self, layer_input, y_true):

          '''
          *** Train the Network ***
          Trains the neural network through one iteration of:
          1) Calculating the predicted y's using a forward pass on the x data,
          2) Using backprop to find the loss gradients,
          3) Optimizing weights using the loss gradients.

          Inputs:

            layer_input (MAT)
              The initial input of the first layer, aka the "X" data.

            y_true (MAT)
              The true y values.

          '''

          # forward pass
          y_pred = self.forward(layer_input)
          # backward pass
          self.backward(y_pred, y_true)
          # optimize
          self.optimize(layer_input)
```

2.2 Activation Functions and their Derivatives

The activation functions defined are the sigmoid and rectified linear units (ReLU) functions (and their derivatives).

```
In [0]: def sigmoid(s): # for binary classification
          return 1/(1 + torch.exp(-s))

        def sigmoid_prime(s):
          return s * (1-s)

        def relu(s):
          return s * (s > 0).float()

        def relu_prime(s):
          return 1. * (s > 0).float()
```

2.3 Loss Functions

The loss functions defined are the absolute error loss and square error loss. Due to time constraints, I have chosen to train and my neural network on a regression problem, and so did not include a classification-appropriate loss function, e.g. cross-entropy.

```
In [0]: def absolute_error(y_pred, y_true):
          return abs(y_pred - y_true)

        def square_error(y_pred, y_true):
          return (y_pred - y_true)**2
```

```
In [0]: def run_nn(NN, X_train, y_train, X_test, y_test, epochs = 1000, disp = False):
          '''
          Basic training function to train the neural network over a set amount of epochs and
        return the train and test loss.


          Inputs:

            NN (OBJ)
              The initialized neural network object.

            X_train (ARR)
              The training input.

            y_train (ARR)
              The true training output.

            X_test (ARR)
              The testing input.

            y_test (ARR)
              The true testing output.

            epochs (INT)
              Number of epochs to train the network.
              Default: 1000

            disp (BOOL)
              If true, prints the epoch, train loss, and test loss per 100 epochs.
              If false, does not print anything.
              Default: False


          Outputs:

            running_train_loss (ARR)
              The training loss at each epoch

            running_test_loss (ARR)
              The test loss at each epoch

          '''

          # keep track of train and test loss
          running_train_loss = []
          running_test_loss = []

          for i in range(epochs):

            # train network
            NN.train(X_train, y_train)
            train_loss = torch.mean(absolute_error(NN.forward(X_train), y_train))
            running_train_loss.append(train_loss)

            # test network
            test_loss = torch.mean(absolute_error(NN.forward(X_test), y_test))
            running_test_loss.append(test_loss)

            if i % 100 == 0 and disp == True:
              print(f"Epoch {i}")
              print ("Train Loss: " + str(train_loss))
              print ("Test Loss: " + str(test_loss))

          return running_train_loss, running_test_loss
```

2.4 Neural Network Test: Fake Data

A fake dataset is generated to train and test the neural network class.

2.4.1 Create the Fake Data

```
In [269]: np.random.seed(2)

          def f(x):
            return torch.mean(x, dim=1)**2 + torch.mean(x, dim=1) + np.random.normal(0, 0.05)

          # train set
          X_train = torch.randn(500, 5)
          y_train = f(X_train).reshape(-1, 1)

          # test set
          X_test = torch.randn(100, 5)
          y_test = f(X_test).reshape(-1, 1)

          print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

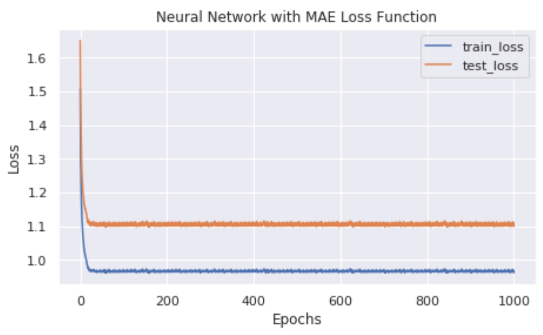
```
torch.Size([500, 5]) torch.Size([100, 5]) torch.Size([500, 1]) torch.Size([100, 1])
```

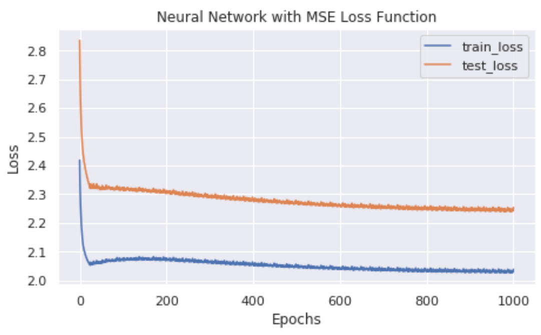
2.4.2 Run the Neural Network

```
In [0]: # define layer sizes
        input_s = X_train.size()[1]
        hidden_s = 10
        output_s = 1

        # initialize networks
        MAE_NN = NeuralNetwork(layers = torch.tensor([[input_s, hidden_s],[hidden_s,
        output_s]]),
                               loss_func = absolute_error,
                               active_func = relu,
                               active_func_prime = relu_prime,
                               alpha = 0.001)

        MSE_NN = NeuralNetwork(layers = torch.tensor([[input_s, hidden_s],[hidden_s,
        output_s]]),
                               loss_func = square_error,
                               active_func = relu,
                               active_func_prime = relu_prime,
                               alpha = 0.001)


        MAE_train_loss, MAE_test_loss = run_nn(MAE_NN, X_train, y_train, X_test, y_test, epochs
        = 1000)
        MSE_train_loss, MSE_test_loss = run_nn(MSE_NN, X_train, y_train, X_test, y_test, epochs
        = 1000)
```

```
In [276]: plt.figure(figsize=(16, 4))

          epochs = 1000
          plt.subplot(1,2,1)
          plt.title('Neural Network with MAE Loss Function')
          plt.plot(np.linspace(0,epochs,epochs), MAE_train_loss, label='train_loss')
          plt.plot(np.linspace(0,epochs,epochs), MAE_test_loss, label='test_loss')
          plt.xlabel('Epochs')
          plt.ylabel('Loss')
          plt.legend()

          plt.subplot(1,2,2)
          plt.title('Neural Network with MSE Loss Function')
          plt.plot(np.linspace(0,epochs,epochs), MSE_train_loss, label='train_loss')
          plt.plot(np.linspace(0,epochs,epochs), MSE_test_loss, label='test_loss')
          plt.xlabel('Epochs')
          plt.ylabel('Loss')
          plt.legend()
```

```
Out[276]: <matplotlib.legend.Legend at 0x7fedc389d5c0>
```





The results show that the neural network is decreasing loss at a higher rate when trained using the MAE loss function vs the MSE loss function. The "jitter" in the loss lines suggest that the network is quite unstable, i.e. it is unable to smoothly converge to the local optima and is instead taking steps towards and away from it repeatedly.

2.5 Neural Network Test: Public Dataset

In this section, I train and test my neural network on a public dataset, specifically the Boston Housing Prices dataset (Scikit-learn, 2019) which takes in the following inputs:

- CRIM per capita crime rate by town
- ZN proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS proportion of non-retail business acres per town
- CHAS Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX nitric oxides concentration (parts per 10 million)
- RM average number of rooms per dwelling
- AGE proportion of owner-occupied units built prior to 1940
- DIS weighted distances to five Boston employment centres
- RAD index of accessibility to radial highways
- TAX full-value property-tax rate per $10,000
- PTRATIO pupil-teacher ratio by town
- B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT % lower status of the population

To return the following output:

- MEDV Median value of owner-occupied homes in $1000's

2.5.1 Import Boston Housing Prices Dataset

```
In [160]: # load data
          from sklearn.datasets import load_boston
          import pandas as pd

          boston = load_boston()

          boston_data_df = pd.DataFrame(boston.data, columns=boston.feature_names)
          print(boston.data.shape)
          boston_data_df.describe().round()
```

```
(506, 13)
```

```
Out[160]:         CRIM     ZN  INDUS   CHAS    NOX  ...    RAD    TAX  PTRATIO      B  LSTAT
          count  506.0  506.0  506.0  506.0  506.0  ...  506.0  506.0    506.0  506.0  506.0
          mean     4.0   11.0   11.0    0.0    1.0  ...   10.0  408.0     18.0  357.0   13.0
          std      9.0   23.0    7.0    0.0    0.0  ...    9.0  169.0      2.0   91.0    7.0
          min      0.0    0.0    0.0    0.0    0.0  ...    1.0  187.0     13.0    0.0    2.0
          25%      0.0    0.0    5.0    0.0    0.0  ...    4.0  279.0     17.0  375.0    7.0
          50%      0.0    0.0   10.0    0.0    1.0  ...    5.0  330.0     19.0  391.0   11.0
          75%      4.0   12.0   18.0    0.0    1.0  ...   24.0  666.0     20.0  396.0   17.0
          max     89.0  100.0   28.0    1.0    1.0  ...   24.0  711.0     22.0  397.0   38.0

          [8 rows x 13 columns]
```

```
In [161]: boston_target_df = pd.DataFrame(boston.target, columns=['MEDIAN VALUE'])
          print(boston.target.shape)
          boston_target_df.describe().round()
```

```
(506,)
```

```
Out[161]:        MEDIAN VALUE
          count         506.0
          mean           23.0
          std             9.0
          min             5.0
          25%            17.0
          50%            21.0
          75%            25.0
          max            50.0
```

2.5.2 Transform and Preprocess Data

```
In [277]: # scale data
          from sklearn.preprocessing import StandardScaler
          scaler = StandardScaler()
          X = scaler.fit_transform(boston.data)
          y = scaler.fit_transform(boston.target.reshape(-1, 1))

          # split data into train and test
          from sklearn.model_selection import train_test_split
          X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
          random_state=42)

          print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(303, 13) (203, 13) (303, 1) (203, 1)
```

```
In [279]: # transform data into tensor with dtype = Float
          def transform_to_tensor(data):
            return torch.tensor(data, dtype=torch.float)

          X_train = transform_to_tensor(X_train)
          X_test = transform_to_tensor(X_test)
          y_train = transform_to_tensor(y_train)
          y_test = transform_to_tensor(y_test)

          print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

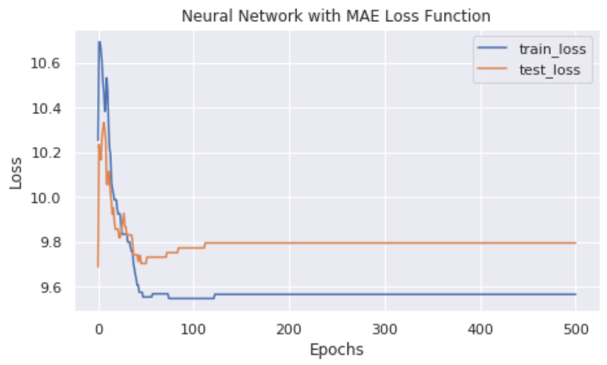
```
torch.Size([303, 13]) torch.Size([203, 13]) torch.Size([303, 1]) torch.Size([203, 1])
```

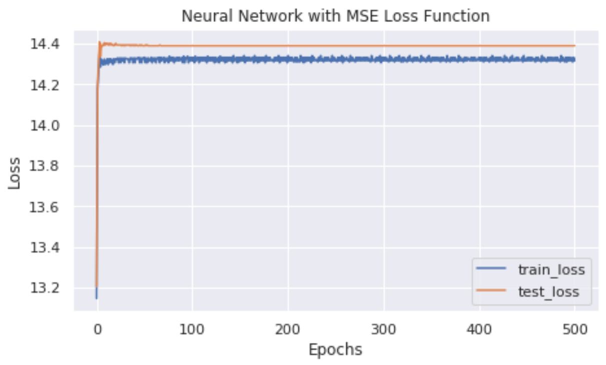
2.5.3 Run the Neural Network

```
In [0]: # define layer sizes
        input_s = X_train.shape[1]
        hidden_s = 30
        output_s = 1

        # initialize networks
        MAE_boston_NN = NeuralNetwork(layers = torch.tensor([[input_s, hidden_s],[hidden_s,
        output_s]]),
                           loss_func = absolute_error,
                           active_func = sigmoid,
                           active_func_prime = sigmoid_prime,
                           alpha = 0.001)

        MSE_boston_NN = NeuralNetwork(layers = torch.tensor([[input_s, hidden_s],[hidden_s,
        output_s]]),
                           loss_func = square_error,
                           active_func = sigmoid,
                           active_func_prime = sigmoid_prime,
                           alpha = 0.001)


        MAE_boston_train_loss, MAE_boston_test_loss = run_nn(MAE_boston_NN, X_train, y_train,
        X_test, y_test, epochs = 500)
        MSE_boston_train_loss, MSE_boston_test_loss = run_nn(MSE_boston_NN, X_train, y_train,
        X_test, y_test, epochs = 500)
```

```
In [346]: plt.figure(figsize=(16, 4))

          epochs = 500
          plt.subplot(1,2,1)
          plt.title('Neural Network with MAE Loss Function')
          plt.plot(np.linspace(0,epochs,epochs), MAE_boston_train_loss, label='train_loss')
          plt.plot(np.linspace(0,epochs,epochs), MAE_boston_test_loss, label='test_loss')
          plt.xlabel('Epochs')
          plt.ylabel('Loss')
          plt.legend()

          plt.subplot(1,2,2)
          plt.title('Neural Network with MSE Loss Function')
          plt.plot(np.linspace(0,epochs,epochs), MSE_boston_train_loss, label='train_loss')
          plt.plot(np.linspace(0,epochs,epochs), MSE_boston_test_loss, label='test_loss')
          plt.xlabel('Epochs')
          plt.ylabel('Loss')
          plt.legend()
```

```
Out[346]: <matplotlib.legend.Legend at 0x7fedc1e527f0>
```





The results show that the network training is much more unstable on the boston dataset vs my self-created fake dataset. With a MAE loss function, we see that the "pathway" to the local optima isn't as smooth a descend as the trend lines on the fake dataset. That being said, upon finding the local optima, the lines smooth out and there isn't any identifiable jitter.

However, with a MSE loss function, the neural network appears to be stuck at the wrong local optima and is unable to take any gradient descent steps to decrease the loss any further.

This is probably because the boston dataset has a more complex underlying function with 10+ features while my dataset only had 5 feature variables.

3 Summary

While my neural network class displays a decreasing loss trend on my fake dataset, it has a very unstable loss trend on the boston housing dataset (even post-scaling). As stated above, this is probably because the Boston dataset has a more complex underlying function vs the fake dataset I created. My neural network is very simple - it lacks a bias term which could help to map a displacement in the function, it lacks regularization methods such as momentum and weight decay (which could help to stabilize convergence to the optima and by extension, the decrease in loss), and I only implemented one hidden layer in each example.

4 References

Brilliant.org (2019). Feedforward Neural Networks. Retrieved October 14, 2019, from https://brilliant.org/wiki/feedforward-neural-networks/

Google Developers. (2019). Backpropagation Demo. Retrieved October 14, 2019, from https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/.

Karpathy, A. (2016). Cs231n convolutional neural networks for visual recognition. Neural networks, 1. Retrieved October 14, 2019, from http://cs231n.github.io/neural-networks-1/.

Olah, C., Mordvintsev, A., & Schubert, L. (2017). Feature visualization. Distill, 2(11), e7.

Ruder, S. (2018, November 29). An overview of gradient descent optimization algorithms. Retrieved October 14, 2019, from http://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent.

Scikit-learn. (2019). Boston House Prices Dataset. Retrieved October 19, 2019, from https://scikit-learn.org/stable/datasets/index.html#boston-house-prices-dataset.

5 Appendix

All code can be found here: `https://github.com/hueyning/DeepLearningTutorial/blob/master/DL_Assignment_1.ipynb`