DLCV-02-Deep_Learning_and_Neural_Networks

September 11, 2024

1 Neural Network

A neural network is inspired by the structure of the human brain. Its basic building block is a neuron, which takes inputs, performs computations, and produces an output. Each neuron has an associated activation function that determines whether it should "fire" based on its inputs.

Training a neural network requires to understand a few naming conventions:
#### 1. Architecture and Layers A simple neural network consists of an input layer, a hidden layer and an output layer. For deep neural networks, the hidden layers should be more than one and possibly sometimes the number of input features are also larger.

Simple Neural Network x_1, x_2 and x_3 are the input features and y_1 and y_2 are the outputs. Here, we can say, the model architecture is receiving 3 features and 2 outputs(2 classes) in a classification model.

2. Parameters In between each layers, there are some linear functions which consist of trainable parameters called weights and biases. Basically, the training of a neural network is updating these weights and biases to achieve the objective function of the specific problem.

Linear function with parameters Let's assume that x is an input feature from a data, the linear function is such a way that the weight w multiple with the input feature x and sum up with the bias b. In below graph, we explicitly assign $w = 1/2$ and $b = 1$ for easy understanding, and it shows a linear line representing that function. This example is only happening in one neuron or one feature. However in neural networks, there will be a lot of similar operations happening in parallel. In such cases, the vectorization and matrix operations are used for faster and more efficient computations.

3. Activation Function The activation function introduces non-linearity to the network. It decides whether a neuron should be active or not based on its weighted inputs. Common activation functions include sigmoid, tanh, and Rectified Linear Unit (ReLU).

Deep Neural Network Architecture image referenced from: Industry 4.0 Interoperability, Analytics, Security, and Case Studies

4. Feedforward In the feedforward neural network, the calculation is only forward pass without updating any parameters yet. The idea is first we calculate the linear function using the input features, then again from the outputs of the linear function, an activation function is introduced in order to get the non-linear outputs. Then the activated outputs from the previous layer go to the next layer as the input of that layer. The process is generally repeated through all the layers until we get the final result y_i which is the predicted output. In feedforward, the predicted output may be some way different from the ground truth class.

5. Loss Function A loss function measures how far off the neural network's output is from the expected output(label or ground truth). It quantifies the network's performance and guides the learning process. Generally, in deep learning, minimizing the loss between the predicted output and the ground truth is the objective function of the model.

6. Optimizer An optimizer is an algorithm or a method used to adjust the parameters (weights and biases) of a neural network during training in order to minimize the loss function(objective function) or maximize the efficiency of the model. The optimizer helps in adjusting the learning rate, momentum and direction of the gradient calculation during training. Examples of optimizer:

1. Stochastic Gradient Descent(SGD)
2. Adam
3. RMSProp
4. Adagrad and so on...

7. Backpropagation Backpropagation is the process of updating the weights and biases in the network to minimize the loss function(objective function). It calculates the gradient of the loss with respect to the network's parameters and updates them using optimization algorithms like Gradient Descent.

Computational Graph

Derivation

Derivative Examples $f(x) = x^2$ » $\frac{\partial}{\partial x}f(x) = 2x$

$f(x) = x^3$ » $\frac{\partial}{\partial x}f(x) = 3x^2$

$f(x) = log_e(x)$ or $f(x) = ln(x)$ » $\frac{\partial}{\partial x}f(x) = \frac{1}{x}$

$f(x) = e^x$ » $\frac{\partial}{\partial x}f(x) = e^x$

$f(x) = 2x^2 + 3x + b$ » $\frac{\partial}{\partial x}f(x) = 4x + 3$ (The derivative of a constant is always zero)

8. Training The network is trained using labeled data. During training, the input data is passed through the network, the output is compared to the expected output, and the loss is calculated. Backpropagation then adjusts the weights and biases to minimize the loss so that we will get the optimal weights and biases that fit with our problem.

Gradient Descent Gradient Descent is an optimization algorithm used in machine learning and deep learning to iteratively update the parameters of a model in order to minimize a loss function. The goal of the algorithm is to find the values of the model's parameters that result in the lowest possible value of the loss function, which in turn signifies the best fit of the model to the training data.

1. Initialization: The algorithm starts with an initial guess for the model's parameters.
2. Compute Loss: The current model's parameters are used to make predictions on the training data. The difference between these predictions and the actual target values (the loss) is calculated using a chosen loss function.
3. Compute Gradient: The gradient of the loss with respect to each parameter is computed. The gradient indicates the direction and magnitude of the steepest increase in the loss function. This step involves taking partial derivatives of the loss function with respect to each parameter.
4. Update Parameters: The parameters are updated by subtracting a fraction of the gradient from the current parameter values. This fraction is called the learning rate. The learning rate determines the step size in the parameter space and influences the speed and stability of convergence.
5. Repeat: Steps 2 to 4 are repeated for a certain number of iterations or until the change in the loss function becomes small (convergence criterion).

Gradient descent algorithm reference: Deep Learning Specialization by Andrew Ng from Coursera.

9. Validation and Evaluation After training, the network's performance is evaluated using validation data to ensure it's not overfitting, underfitting, or the model is learning. Once satisfied, the network is tested on unseen data to measure its real-world performance. The testing process is sometimes called inferencing.

2 Dealing with Python Packages/ Libraries

Installing a library with PIP The command to install in your terminal is `pip3 install packagename`. You can find the related package name and it's version in https://pypi.org/.

Example with Numpy Library Before you start, please make sure that you have already installed `Numpy` library which is a handy package for scientific computing in Python.
To install Numpy, just simply use `pip3 install numpy` or `pip install numpy` in your terminal, or put an exclaimation point infront `!pip3 install numpy` or `!pip install numpy` in you Jupyter cell.

To start using the library, simple import the library in the beginning of your Text editor or Jupyter cell.

```
import numpy as np
```

Notice that `np` is a shortcut name which you will use that throughout the implementation instead of `numpy`. It is just a standard way of shortcut, however you can name it any shortcut name. It is recommended to use a standard form.
Some other useful libraries you might use in the future:

```
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image
```

`from sklearn.model_selection import train_test_split` « This is importing `train_test_split` method from `model_selection` module from Scikit-Learn(`sklearn`) library.

Loading and Image Using PIL and Matplotlib Libraries

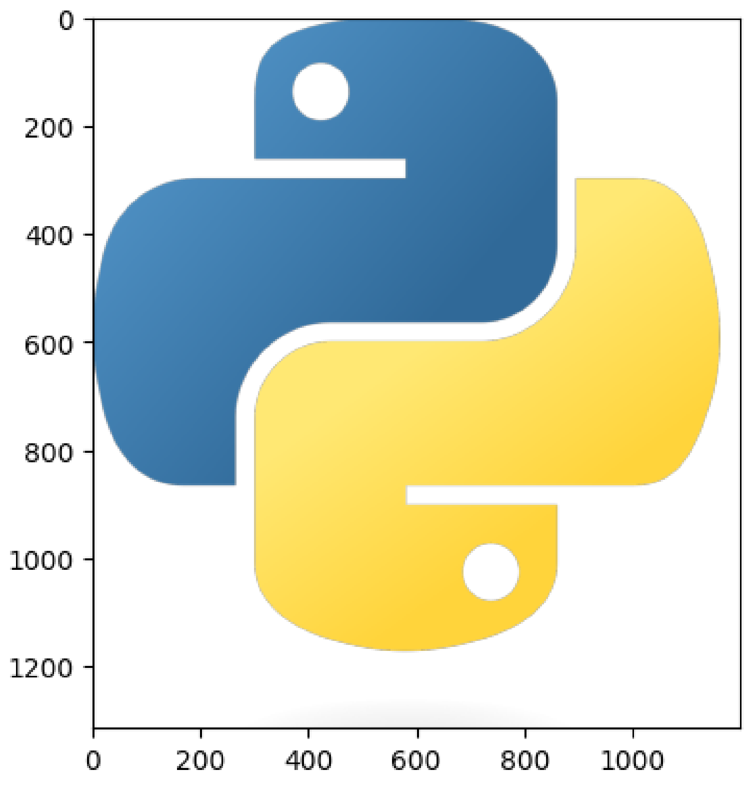
```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     from PIL import Image # Use for loading image into
     from IPython.display import display
```

```
[ ]: img = Image.open('img/python.png')
     print("Image object: ", img)
     img_data = np.array(img)
     x, y, _ = img_data.shape
     print("Original size: ", (x, y, _))
     img = img.resize((int(x/4), int(y/4)))
     print("Resized size", np.array(img).shape)
     # img.show()
     display(img)
```

```
Image object:  <PIL.PngImagePlugin.PngImageFile image mode=RGBA size=1200x1315
at 0x18675A66BB0>
Original size:  (1315, 1200, 4)
Resized size (300, 328, 4)
```

```
[ ]: # Python program to read
     # image using matplotlib

     # importing matplotlib modules
     import matplotlib.image as mpimg
     import matplotlib.pyplot as plt

     # Read Images
     img = mpimg.imread('img/python.png')
     print(img.shape)
     height, width, _ = img.shape
     # Output Images
     # plt.figure(figsize=(height/72, width/72))
     plt.imshow(img)
     plt.show()
```

```
(1315, 1200, 4)
```



Loading an Image Using OpenCV To load the image, we will try a different method by using a library called OpenCV. First, we will need the installation of `opencv-python`.

```
pip install opencv-python
```

[ ]:
```
import cv2

# Load the image
image = cv2.imread("img/cat.png")

# # Display the image
cv2.imshow("Image", image)

# # Wait for the user to press a key
cv2.waitKey(0)

# # Close all windows
cv2.destroyAllWindows()
```

3 Vectorization and Matrix Operation

Different types of vector/matrix operations

1. The dot product (inner scalar product)
   $\mathbf{x1} \cdot \mathbf{x2}$

2. Elementwise product (Hadamard product)
   $\mathbf{x1} \circ \mathbf{x2}$
   $\mathbf{x1} \odot \mathbf{x2}$

3. The outer product
   $\mathbf{x1} \otimes \mathbf{x2}$

Comparing the running times

[ ]:
```
### Importing libraries before writing your codes
import time
import numpy as np
import math


# x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
# x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

np.random.seed(27)
x1 = np.random.randint(500, 1000, 5000)
x2 = np.random.randint(0, 500, 5000)
print("x1[0]:", x1[0], "  ", "x2[0]:", x2[0])

### CLASSIC DOT PRODUCT OF VECTORS IMPLEMENTATION ###
tic = time.process_time()
```

```
dot = 0
for i in range(len(x1)):
    dot+= x1[i]*x2[i]
toc = time.process_time()
# print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc -␣
 ↪tic)) + "ms")
print ("dot ----- Computation time = " + str(1000*(toc - tic)) + "ms")

### CLASSIC OUTER PRODUCT IMPLEMENTATION ###
tic = time.process_time()
outer = np.zeros((len(x1),len(x2))) # we create a len(x1)*len(x2) matrix with␣
 ↪only zeros
for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i]*x2[j]
toc = time.process_time()
# print ("outer = " + str(outer) + "\n ----- Computation time = " +␣
 ↪str(1000*(toc - tic)) + "ms")
print ("outer ----- Computation time = " + str(1000*(toc - tic)) + "ms")

### CLASSIC ELEMENTWISE IMPLEMENTATION ###
tic = time.process_time()
mul = np.zeros(len(x1))
for i in range(len(x1)):
    mul[i] = x1[i]*x2[i]
toc = time.process_time()
# print ("elementwise multiplication = " + str(mul) + "\n ----- Computation␣
 ↪time = " + str(1000*(toc - tic)) + "ms")
print ("elementwise multiplication ----- Computation time = " + str(1000*(toc -␣
 ↪tic)) + "ms")

### CLASSIC GENERAL DOT PRODUCT IMPLEMENTATION ###
W = np.random.rand(3,len(x1)) # Random 3*len(x1) numpy array
tic = time.process_time()
gdot = np.zeros(W.shape[0])
for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j]*x1[j]
toc = time.process_time()
# print ("gdot = " + str(gdot) + "\n ----- Computation time = " + str(1000*(toc␣
 ↪- tic)) + "ms")
print ("gdot ----- Computation time = " + str(1000*(toc - tic)) + "ms")
```

```
x1[0]: 519    x2[0]: 214
dot ----- Computation time = 0.0ms
outer ----- Computation time = 13687.5ms
elementwise multiplication ----- Computation time = 0.0ms
```

```
gdot ----- Computation time = 15.625ms
```

[ ]:
```
# x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
# x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

np.random.seed(27)
x1 = np.random.randint(500, 1000, 5000)
x2 = np.random.randint(0, 500, 5000)
print("x1[0]:", x1[0], "  ", "x2[0]:", x2[0])

### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
# print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc -␣
 ↪tic)) + "ms")
print ("dot ----- Computation time = " + str(1000*(toc - tic)) + "ms")

### VECTORIZED OUTER PRODUCT ###
tic = time.process_time()
outer = np.outer(x1,x2)
toc = time.process_time()
# print ("outer = " + str(outer) + "\n ----- Computation time = " +␣
 ↪str(1000*(toc - tic)) + "ms")
print ("outer ----- Computation time = " + str(1000*(toc - tic)) + "ms")

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
# print ("elementwise multiplication = " + str(mul) + "\n ----- Computation␣
 ↪time = " + str(1000*(toc - tic)) + "ms")
print ("elementwise multiplication ----- Computation time = " + str(1000*(toc -␣
 ↪tic)) + "ms")

### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
# print ("gdot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc␣
 ↪- tic)) + "ms")
print ("gdot ----- Computation time = " + str(1000*(toc - tic)) + "ms")
```

```
x1[0]: 519    x2[0]: 214
dot ----- Computation time = 0.0ms
outer ----- Computation time = 78.125ms
elementwise multiplication ----- Computation time = 0.0ms
gdot ----- Computation time = 0.0ms
```

Exercise 1

1. Create three matrices and print out the results:
   $\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 6 & 5 \end{pmatrix}$, $\mathbf{B} = \begin{pmatrix} 3 & 4 \\ 8 & 7 \end{pmatrix}$, $\mathbf{C} = \begin{pmatrix} 3 & 4 & 0 \\ 8 & 7 & 9 \end{pmatrix}$
2. Calculate the following and print out the results:
   1. $\mathbf{A} \cdot \mathbf{B}$ » Try 3 different styles using `@` sign or `np.dot()` or `np.matmul()` functions. All three results should be the same.

   2. $\mathbf{A} \cdot \mathbf{C}$
   3. $2\mathbf{A}$
   4. $2\mathbf{A}^T$ » To transpose the matrix, you can simply use `matrix.T` or `np.transpose()` function.
   5. $3\mathbf{A}^T \cdot \mathbf{B}$ » Print out the result and shape.
   6. $3\mathbf{A}^T \cdot \mathbf{C}$ » Print out the result and shape.
   7. $\mathbf{A} \odot \mathbf{B}$
   8. $\mathbf{A} \odot \mathbf{C}$ » This should show up error.

3. Comment the difference between $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A} \odot \mathbf{B}$

```
[ ]: ### Your code here
# 1. Create matrices
a, b, c = np.array([[2, 1], [6, 5]]), np.array([[3, 4], [8, 7]]), np.array([[3,␣
↪4, 0], [8, 7, 9]])
# A . B (Three different styles)
print("np.dot(a,b)= ", np.dot(a, b))
print("a @ b = ", a @ b)
print("np.matmul(a, b)= ", np.matmul(a, b))

# A . C
print("np.dot(a,c)= ", np.dot(a, c))
print("a @ c = ", a @ c)
print("np.matmul(a, c)= ", np.matmul(a, c))
# 2A
print("2A= ", 2 * a)
# 2A.T
print("2A.T= ", (2 * a).T)
# 3A.T . B (Print out shape and result)
print("3A.T . B= ", np.dot((3 * a).T, b), " shape: ", np.dot((3 * a).T, b).
↪shape)
# 3A.T . C (Print out shape and result)
print("3A.T . C= ", np.dot((3 * a).T, c), " shape: ", np.dot((3 * a).T, c).
↪shape)
# A * B
print("A * B= ",  a * b)
# A * C (error)
# print("A * C= ", a * c)
```

```
np.dot(a,b)=  [[14 15]
 [58 59]]
a @ b =  [[14 15]
 [58 59]]
np.matmul(a, b)=  [[14 15]
 [58 59]]
np.dot(a,c)=  [[14 15  9]
 [58 59 45]]
a @ c =  [[14 15  9]
 [58 59 45]]
np.matmul(a, c)=  [[14 15  9]
 [58 59 45]]
2A=  [[ 4  2]
 [12 10]]
2A.T=  [[ 4 12]
 [ 2 10]]
3A.T . B=  [[162 150]
 [129 117]]  shape:  (2, 2)
3A.T . C=  [[162 150 162]
 [129 117 135]]  shape:  (2, 3)
A * B=  [[ 6  4]
 [48 35]]
```

Expected Output:

A . B = [[14 15] [58 59]]

A . C = [[14 15 9] [58 59 45]]

2A = [[4 2] [12 10]]

2A.T = [[4 12] [2 10]]

3A.T . B = [[162 150] [129 117]]

Shape of 3A.T . B = (2, 2)

3A.T . C = [[162 150 162] [129 117 135]]

Shape of 3A.T . C = (2, 3)
A * B = [[6 4] [48 35]]

A * C = Error

Reshaping arrays Two common numpy functions used in deep learning are np.shape and np.reshape().
- `X.shape` is used to get the shape (dimension) of a matrix/vector **X** - `X.reshape(...)` is used to reshape **X** into some other dimension.

In computer science, an image is represented by a 3D array of shape (height, width, channel). However, when you read an image as the input of an algorithm you convert it to a vector of shape (height * width * channel, 1).

Exercise 2 Implement `image2vector()` function that takes an input of shape (height, width, 3) and returns a vector of shape (height * width * 3, 1).

Do not hardcode the dimensions of image as a constant. The dimensions can varies from different images.

```
[ ]: # Grade cell - do not remove

     def image2vector(image):
         """
         Convert image with 3 dimensions to become vector of (size, 1)
         """
     #     height, width, channel = image.shape
         h, w, c = image.shape
         # print(image.shape)
         v = image.reshape(h * w * c, 1)
         # print(v.shape)
         # YOUR CODE HERE
     #     raise NotImplementedError()

         return v
```

```
[ ]: # test function - do not remove
     image = np.array([[[ 0.67826139,  0.29380381],
             [ 0.90714982,  0.52835647],
             [ 0.4215251 ,  0.45017551]],

            [[ 0.92814219,  0.96677647],
             [ 0.85304703,  0.52351845],
             [ 0.19981397,  0.27417313]],

            [[ 0.60659855,  0.00533165],
             [ 0.10820313,  0.49978937],
             [ 0.34144279,  0.94630077]]])

     print ("image2vector(image) = " + str(image2vector(image)))

     import cv2

     img = cv2.imread("img/lena.png")

     print(img.shape)
     vecimg = image2vector(img)
     print(vecimg.shape)
```

```
assert image2vector(image).shape == (image.shape[0] * image.shape[1] * image.
↪shape[2], 1), "Dimension is incorrect"
```

```
image2vector(image) = [[0.67826139]
 [0.29380381]
 [0.90714982]
 [0.52835647]
 [0.4215251 ]
 [0.45017551]
 [0.92814219]
 [0.96677647]
 [0.85304703]
 [0.52351845]
 [0.19981397]
 [0.27417313]
 [0.60659855]
 [0.00533165]
 [0.10820313]
 [0.49978937]
 [0.34144279]
 [0.94630077]]
(512, 512, 3)
(786432, 1)
```

Expected Output: image2vector(image) = [[0.67826139]
[0.29380381]
[0.90714982]
[0.52835647]
[0.4215251]
[0.45017551]
[0.92814219]
[0.96677647]
[0.85304703]
[0.52351845]
[0.19981397]
[0.27417313]
[0.60659855]
[0.00533165]
[0.10820313]
[0.49978937]
[0.34144279]
[0.94630077]]
(512, 512, 3)
(786432, 1)

Matrix Normalization Normalization is a technique for machine learning and deep learning. It generally ensures a better performance sicne the gradient descent converges faster after the data is normalized. The normalization also controls the parameters not to overflow while updating the gradients. For example, we have a matrix **A**

$$\mathbf{A} = \begin{bmatrix} 8 & 1 & 2 \\ 3 & 9 & 5 \end{bmatrix}$$

We can get the norm the matrix using `np.linalg.norm()` function. In our case, we find the norm along the rows. By using numpy library function, we can write the code as follows:

$$\|A\| = np.linalg.norm(\mathbf{A}, axis = 1)$$

and normalize the matrix by simply dividing the original matrix by the norm.

$$norm(\mathbf{A}) = \frac{\mathbf{A}}{\|\mathbf{A}\|}$$

Broadcasting Notice that numpy has a feature called broadcasting which allows the operation of matrices with different sizes by duplicating to get the same size. (In our case, it duplicates the norm matrix into two rows before doing the division)

$$\begin{bmatrix} 8 & 1 & 2 \\ 3 & 9 & 5 \end{bmatrix} / \begin{bmatrix} norm_0 \\ norm_1 \end{bmatrix} = \begin{bmatrix} 8 & 1 & 2 \\ 3 & 9 & 5 \end{bmatrix} / \begin{bmatrix} norm_0 & norm_0 & norm_0 \\ norm_1 & norm_1 & norm_1 \end{bmatrix}$$

Implement `normalizeRows()` to normalize the rows of a matrix. After applying this function to an input matrix **x**, each row of **x** should be a vector of unit length (meaning length 1).

[ ]:
```
print(img.shape)
```

```
(512, 512, 3)
```

Example

[ ]:
```
# Grade cell - do not remove
def normalizeRows(x):
    """
    Implement a function that normalizes each row of the matrix x (to have unit␣
 ↪length).
    """
    print("shape: ", np.linalg.norm(x, axis = 1).shape)
    norm_x = x/np.linalg.norm(x, axis =1).reshape(-1, 1)
    # YOUR CODE HERE
#     raise NotImplementedError()

    return norm_x
```

[ ]:
```
# test function - do not remove

A = np.array([
    [8, 1, 2],
    [3, 9, 5]])

norm_A = normalizeRows(A)
print("normalizeRows(A) = " + str(norm_A))

sqr_A = norm_A * norm_A
sum_A = np.sum(sqr_A, axis = 1)
print("prove of A in each row", sum_A)

assert sum_A.shape == (2,), "normalize must do in row"
assert np.round(sum_A[0], 1) == 1 and np.round(sum_A[1], 1) == 1, "Normalize␣
↪incorrect"
```

```
shape:  (2,)
normalizeRows(A) = [[0.96308682 0.12038585 0.24077171]
 [0.27975144 0.83925433 0.4662524 ]]
prove of A in each row [1. 1.]
```

Expected Output: normalizeRows(A) = [[0.96308682 0.12038585 0.24077171]
[0.27975144 0.83925433 0.4662524]]
prove of A in each row [1. 1.]

4 Implement the L1 and L2 loss functions

The loss L1 and L2 are used to evaluate the performance of your model. The bigger your loss is, the more different your predictions $\hat{h}$ are from the true values y. In deep learning, Gradient Descent or Ascent is used to optimize models by minimizing the cost.

To assume loss function in L_1, the L1 loss is defined as

$$L_1(\hat{y}, y) = \sum_{i=0}^{m} |y^{(i)} - \hat{y}^{(i)}|$$

Example Implement the numpy vectorized version of the L1 loss. use function np.abs() to apply the equation.

```
[ ]: def L1(yhat, y):
     #     loss = np.sum(np.abs(y - yhat))
         loss = np.abs(y-yhat).sum()
         # YOUR CODE HERE
     #     raise NotImplementedError()
         return loss
```

```
[ ]: # test function - do not remove

     yhat = np.array([.9, 0.2, 0.1, .4, .9])
     yhat2 = np.array([.1, 0.7, 0.4, 0.7, .8, 0.2])
     y = np.array([1, 0, 0, 1, 1])
```

```
y2 = np.array([1, 0, 0, 1, 1, 0])
l1 = L1(yhat,y)
l2 = L1(yhat2,y2)
print("L1 of output 1 = " + str(l1))
print("L1 of output 2 = " + str(l2))

assert np.round(l1,1) == 1.1, "L1 loss is incorrect"
assert np.round(l2,1) == 2.7, "L1 loss is incorrect"
```

```
L1 of output 1 = 1.1
L1 of output 2 = 2.7
```

Expected Output:
L1 of output 1 = 1.1
L1 of output 2 = 2.7

To assume loss function in L_2, the L2 loss is defined as

$$L_2(\hat{y}, y) = \sum_{i=0}^{m}(y^{(i)} - \hat{y}^{(i)})^2$$

Example Implement the numpy vectorized version of the L2 loss.

[ ]:
```
# Grade cell - do not remove

def L2(yhat, y):
    loss = np.square(y-yhat).sum()
    # YOUR CODE HERE
#     raise NotImplementedError()

    return loss
```

[ ]:
```
# test function - do not remove

yhat = np.array([.9, 0.2, 0.1, .4, .9])
yhat2 = np.array([.1, 0.7, 0.4, 0.7, .8, 0.2])
y = np.array([1, 0, 0, 1, 1])
y2 = np.array([1, 0, 0, 1, 1, 0])
l1 = L2(yhat,y)
l2 = L2(yhat2,y2)
print("L2 of output 1 = " + str(l1))
print("L2 of output 2 = " + str(l2))

assert np.round(l1,2) == 0.43, "L2 loss is incorrect"
assert np.round(l2,2) == 1.63, "L2 loss is incorrect"
```

```
L2 of output 1 = 0.43
L2 of output 2 = 1.6300000000000001
```

Expected Output:
L2 of output 1 = 0.43
L2 of output 2 = 1.6300000000000001

5 Break Down of a Neural Network Classification Training

A neural network training can be broken down into the following subtasks:

1. Data Preprocessing
2. Forward Propagation
3. Loss Function
4. Back Propagation
5. Parameters Update
6. Hyperparameter Tuning

We will implement these subtasks step by step from scratch in order to have a better understanding of the background process.

5.1 1. Data Preprocessing

This step includes:
a. loading the input images and visualizing them
b. Getting one hot encoded labels
c. Normalization
d. Splitting data to training, validation and test sets

Loading Images The dataset we are going to use in this training is the images of handwritten digits which we can get from `sklearn` library. Firstly, we have to import the required libraries as below:

[ ]:
```
import numpy as np
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
```

Then, load the dataset. and see the data and shape of X(inputs) and y(labels).

[ ]:
```
# Load data
data = load_digits()

y_indices = data.target # load the targets(labels)
X = np.matrix(data.data) # load the input images

print('y shape:', y_indices.shape)
print('X shape:', X.shape)

print('y data:', y_indices[0:15])
print('X data:', X[0:5])

data_size = X.shape[0]
```

```
x_area = X.shape[1]
```

```
y shape: (1797,)
X shape: (1797, 64)
y data: [0 1 2 3 4 5 6 7 8 9 0 1 2 3 4]
X data: [[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.
3.
  15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
   0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
   0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
 [ 0.  0.  0. 12. 13.  5.  0.  0.  0.  0.  0. 11. 16.  9.  0.  0.  0.  0.
   3. 15. 16.  6.  0.  0.  0.  7. 15. 16. 16.  2.  0.  0.  0.  0.  1. 16.
  16.  3.  0.  0.  0.  0.  1. 16. 16.  6.  0.  0.  0.  0.  1. 16. 16.  6.
   0.  0.  0.  0.  0. 11. 16. 10.  0.  0.]
 [ 0.  0.  0.  4. 15. 12.  0.  0.  0.  0.  3. 16. 15. 14.  0.  0.  0.  0.
   8. 13.  8. 16.  0.  0.  0.  0.  1.  6. 15. 11.  0.  0.  0.  1.  8. 13.
  15.  1.  0.  0.  0.  9. 16. 16.  5.  0.  0.  0.  0.  3. 13. 16. 16. 11.
   5.  0.  0.  0.  0.  3. 11. 16.  9.  0.]
 [ 0.  0.  7. 15. 13.  1.  0.  0.  0.  8. 13.  6. 15.  4.  0.  0.  0.  2.
   1. 13. 13.  0.  0.  0.  0.  0.  2. 15. 11.  1.  0.  0.  0.  0.  0.  1.
  12. 12.  1.  0.  0.  0.  0.  0.  1. 10.  8.  0.  0.  0.  8.  4.  5. 14.
   9.  0.  0.  0.  7. 13. 13.  9.  0.  0.]
 [ 0.  0.  0.  1. 11.  0.  0.  0.  0.  0.  0.  7.  8.  0.  0.  0.  0.  0.
   1. 13.  6.  2.  2.  0.  0.  0.  7. 15.  0.  9.  8.  0.  0.  5. 16. 10.
   0. 16.  6.  0.  0.  4. 15. 16. 13. 16.  1.  0.  0.  0.  0.  3. 15. 10.
   0.  0.  0.  0.  0.  2. 16.  4.  0.  0.]]
```

The shape of X is (1797, 64) which means we have 1797 samples of data in (1, 64) long vector. Since each sample represents 1 D vector and the actual size of an image is 8x8, we need to convert the size of input X into 8x8.

Visualize Image

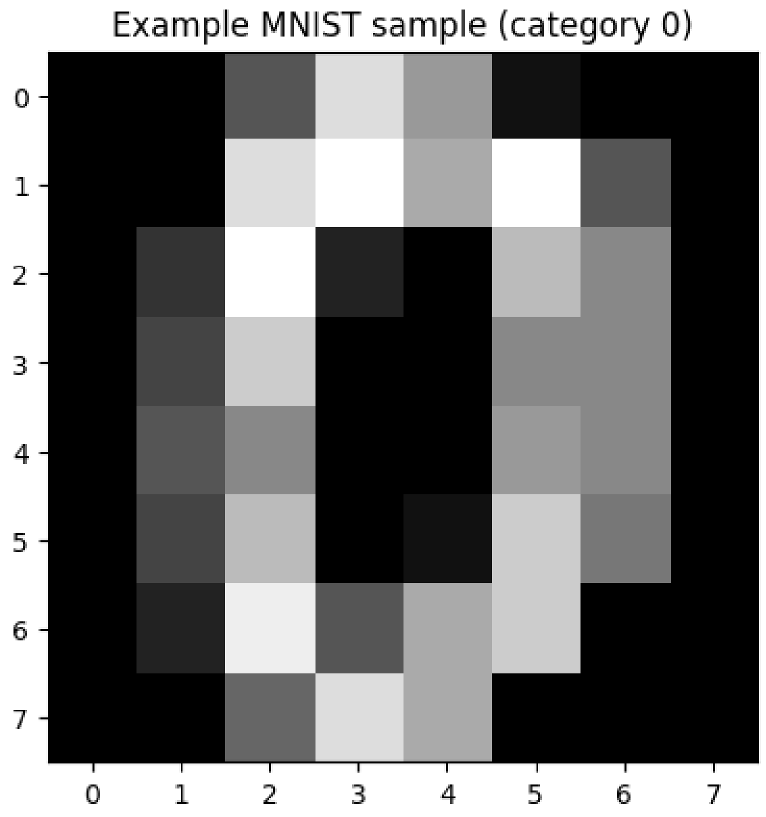
Exercise 3 Write a function to convert the X data of one image to be image size 8x8. Check the size of input vector to convert image and verify after conversion.

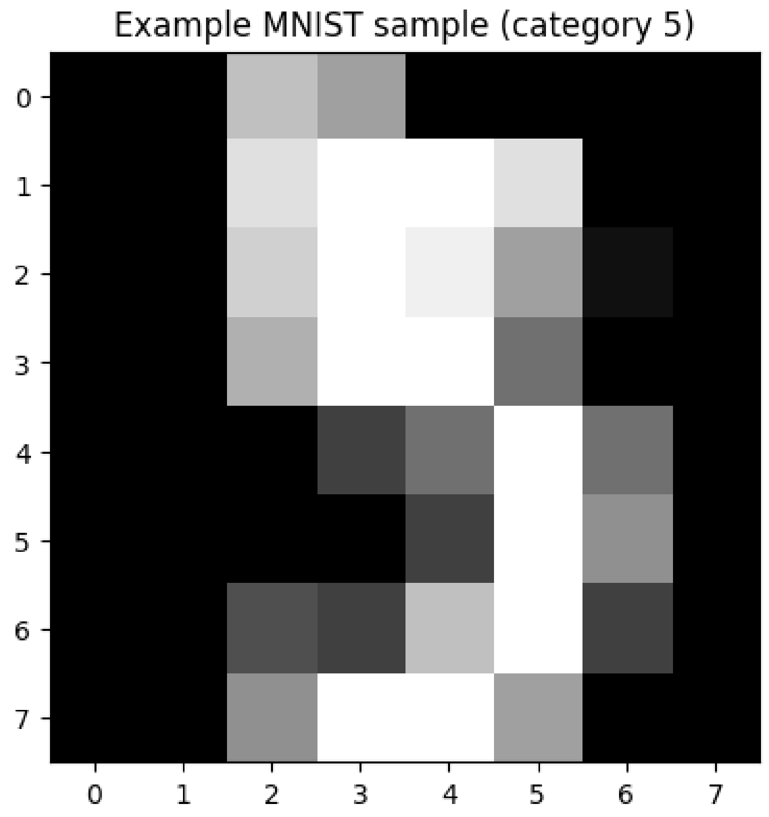
Hint: use np.reshape()

[ ]:
```
# Grade cell - do not remove

def convert_image(X_one_image):

    # Check if the size of X_one_image matches the required total_elements
    if X_one_image.size == 64:
        img = X_one_image.reshape(8, 8)

        # raise ValueError(f"Cannot reshape array of size {X_one_image.size}␣
↪into shape {8, 8}")
```

```
    elif X_one_image.size == 256:
        img = X_one_image.reshape(16, 16)
    # Reshape the array
    return img
```

[ ]:
```
# test function - do not remove

img_0 = convert_image(X[0])
plt.imshow(img_0, 'gray')
plt.title('Example MNIST sample (category %d)' % y_indices[0])
plt.show()

img_5 = convert_image(X[5,:])
plt.imshow(img_5, 'gray')
plt.title('Example MNIST sample (category %d)' % y_indices[5])
plt.show()

test_v = np.empty([1,256])
test = convert_image(test_v)

assert img_0.shape == (8,8) and img_5.shape == (8,8), 'Image reshape is␣
 ↪incorrect'
assert test.shape == (16,16), 'Image reshape is incorrect'
assert img_0[3,6] == X[0, 30] and img_5[4,2] == X[5, 34], 'Image reshape is␣
 ↪incorrect'
```


Example MNIST sample (category 0)
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7



Expected Output:

One Hot Encoding As you can see, the output y is index value. To use the value for classify in deep learning, you need to convert it to one hot matrix.

One hot encoding is a process by which categorical variables are converted into a form that could be provided to ML algorithms to do a better job in prediction.

To do so, we need to convert the index value to the following format:

$$0 \rightarrow [1,0,0,0,0,0,0,0,0,0]$$

$$1 \rightarrow [0,1,0,0,0,0,0,0,0,0]$$

$$2 \rightarrow [0,0,1,0,0,0,0,0,0,0]$$

$$3 \rightarrow [0,0,0,1,0,0,0,0,0,0]$$

$$4 \rightarrow [0,0,0,0,1,0,0,0,0,0]$$

$$5 \rightarrow [0,0,0,0,0,1,0,0,0,0]$$

$$6 \rightarrow [0,0,0,0,0,0,1,0,0,0]$$

$$7 \rightarrow [0,0,0,0,0,0,0,1,0,0]$$

$$8 \rightarrow [0,0,0,0,0,0,0,0,1,0]$$

$$9 \rightarrow [0,0,0,0,0,0,0,0,0,1]$$

Exercise 4 Implement one-hot vector function

[ ]:
```
# Grade cell - do not remove

def convert_to_one_hot(y, onehot_size):
    y_vect = np.zeros((y.shape[0], onehot_size))
    for i, val in enumerate(y):
        y_vect[i][val] = 1
    # YOUR CODE HERE
    # raise NotImplementedError()
    return y_vect
```

[ ]:
```
# test function - do not remove

y = convert_to_one_hot(y_indices, 10)
print(y.shape)
print(y[3])
assert y.shape[1] == 10 and y.shape[0] == 1797, "One hot size is incorrect"
assert y[14, 8] == 0 and y[177,1] == 1, "One hot value is incorrect"
```

```
(1797, 10)
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```

Expected Output:
(1797, 10)
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]

[ ]:
```
print(X.shape)
```

```
(1797, 64)
```

Normalize Input Feature Now, change the input $\mathbf{x}$ to be a normalized input vector. The normalization equation is:

$$norm(\mathbf{x}) = \frac{\mathbf{x} - \tilde{\mathbf{x}}}{\sigma}$$

where,
$\tilde{\mathbf{x}}$ = Mean
σ = Standard Deviation

This is called z-score normalization.

Exercise 5 Write a normalization code. If some values are nan, please change them to be zero, using np.nan_to_num()

```
[ ]: # Grade cell - do not remove

     def normalize(X):
         mean = np.mean(X, axis=0)
         std = np.std(X, axis=0)
         XX = (X - mean) / std
         XX = np.nan_to_num(XX, nan=0.0)
         # YOUR CODE HERE
     #     raise NotImplementedError()
         return XX
```

```
[ ]: # test function - do not remove

     XX = normalize(X)

     print(XX[0])

     assert XX.shape == X.shape, "Normalize function is incorrect"
     assert np.max(XX[0]) < 2 and np.min(XX[0]) > -2, "Data is not normalize"
```

```
[[ 0.         -0.33501649 -0.04308102  0.27407152 -0.66447751 -0.84412939
  -0.40972392 -0.12502292 -0.05907756 -0.62400926  0.4829745   0.75962245
  -0.05842586  1.12772113  0.87958306 -0.13043338 -0.04462507  0.11144272
   0.89588044 -0.86066632 -1.14964846  0.51547187  1.90596347 -0.11422184
  -0.03337973  0.48648928  0.46988512 -1.49990136 -1.61406277  0.07639777
   1.54181413 -0.04723238  0.          0.76465553  0.05263019 -1.44763006
  -1.73666443  0.04361588  1.43955804  0.         -0.06134367  0.8105536
   0.63011714 -1.12245711 -1.06623158  0.66096475  0.81845076 -0.08874162
  -0.03543326  0.74211893  1.15065212 -0.86867056  0.11012973  0.53761116
  -0.75743581 -0.20978513 -0.02359646 -0.29908135  0.08671869  0.20829258
  -0.36677122 -1.14664746 -0.5056698  -0.19600752]]

C:\Users\eraco\AppData\Local\Temp\ipykernel_17556\3702582892.py:6:
RuntimeWarning: invalid value encountered in divide
  XX = (X - mean) / std
```

Expected Output:
[[0. -0.33501649 -0.04308102 0.27407152 -0.66447751 -0.84412939
-0.40972392 -0.12502292 -0.05907756 -0.62400926 0.4829745 0.75962245
-0.05842586 1.12772113 0.87958306 -0.13043338 -0.04462507 0.11144272
0.89588044 -0.86066632 -1.14964846 0.51547187 1.90596347 -0.11422184
-0.03337973 0.48648928 0.46988512 -1.49990136 -1.61406277 0.07639777
1.54181413 -0.04723238 0. 0.76465553 0.05263019 -1.44763006
-1.73666443 0.04361588 1.43955804 0. -0.06134367 0.8105536
0.63011714 -1.12245711 -1.06623158 0.66096475 0.81845076 -0.08874162
-0.03543326 0.74211893 1.15065212 -0.86867056 0.11012973 0.53761116
-0.75743581 -0.20978513 -0.02359646 -0.29908135 0.08671869 0.20829258
-0.36677122 -1.14664746 -0.5056698 -0.19600752]]

Splitting Data In a machine learning training, it is necessary to split the existing data into three different types:

1. Training set - Data that is used for training the model
2. Validation set - Data that is used for validating the model performance throughout the training at each epoch or iteration. The validation data is used to verify the model is learning and not to overfit or underfit the training data.
3. Testing set - Data that is used in a process called inferencing. The test set is totally unseen by the model during training and validation process. Higher performance of the model on the testing data means the model is robust and generalizable to unseen data.

The process of spliting the training, validate, and test set needs the following criteria:

1. The data need to be random before split.
2. The validate and test set can be from the same distribution as training data, however the same data must be strictly in only one split set(train or validation or test).
3. More data is almost always helps, however the consistency and reliability of data are key importances when selection of data(garbage in garbage out).

Normally, we should split data in percentage. However, this is not fixed. You can adjust.

- 60% training, 20% validate, and 20% test
- 80% training, 10% validate, and 10% test
- For the very low data (~1000 data), we could use validate and test set in the same data.

Exercise 6 Random split the train set to be 60% of data and the rest is the test set.

Hint: use `np.arange()` for set index number from 0 to data_size. Random index can do by using `random.shuffle()`. Please make sure that you are using normalized data to split.

```
[ ]: data_size
```

```
[ ]: 1797
```

```
[ ]: # Grade cell - do not remove

     import random

     percent_train = .6
     # arange index number from 0 to data_size
     idx = np.arange(data_size)
     # random shuffle idx (1 line)
     random.shuffle(idx)
     # calculate number of training set
     m_train = int(data_size * percent_train)
     # split train_idx and test_idx (uncomment these 2 lines)
     train_idx = idx[0:m_train]
     test_idx = idx[m_train:data_size+1]

     # split to X_train and X_test
```

```
X_train = XX[idx[:m_train]]
X_test = XX[idx[m_train:]]

# split to y_train y_test and y_test_indices
y_train = y[idx[:m_train]]
y_test = y[idx[m_train:]]
y_test_indices = test_idx

# YOUR CODE HERE
# raise NotImplementedError()
```

```
[ ]: # test function - do not remove

assert X_train.shape[0] == int(percent_train * data_size), "training size is␣
 ↪incorrect"
assert data_size - m_train == X_test.shape[0], "test size is incorrect"
assert train_idx[0] != 0 and train_idx[25] != 25, "training indices are not␣
 ↪shuffled"
assert X_train.shape == (m_train, XX.shape[1]) and y_train.shape[0] == m_train
assert X_test.shape == (data_size - m_train, XX.shape[1]) and y_test.shape[0]␣
 ↪== data_size - m_train and y_test_indices.shape[0] == data_size - m_train
```

5.2 2. Forward Propagation

The neural network we are going to implement looks very similar to the below diagram. However, our input size is 8x8 flattened into (1, 64) vector.

A few steps we will implement in forward propagation or feed forward network.

a. Building activation functions

b. Initializing parameters

c. Feed forward layer

5.2.1 Building Activation Functions with Numpy

Referenced from Activation functions.

Sigmoid Function Sigmoid function is known as the logistic function. The equation can be written as:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Before using np.exp(), you will use math.exp() to implement the sigmoid function. You will then see why np.exp() is preferable to math.exp().

Example

```
[ ]: import math

     def math_sigmoid(x):
         '''
         Compute sigmoid of x
         '''
         z = 1/ (1+math.exp(-x))
         # YOUR CODE HERE
     #     raise NotImplementedError()
         return z
```

```
[ ]: # test function - do not remove
     print(math_sigmoid(5))

     assert math_sigmoid(10) > 0.9999, "Calculate error"
     assert math_sigmoid(-10) < 0.0001, "Calculate error"
     assert math_sigmoid(0) == 0.5, "Calculate error"
```

```
0.9933071490757153
```

Actually, we rarely use the “math” library in deep learning because the inputs of the functions are real numbers. In deep learning we mostly use matrices and vectors. This is why numpy is more useful.

```
[ ]: ### One reason why we use "numpy" instead of "math" in Deep Learning ###
     x = [1, 2, 3]
     # math_sigmoid(x) # you will see this give an error when you run it, because x␣
      ↪is a vector.
```

In fact, if $x = (x_1, x_2, ..., x_n)$ is a row vector then will apply the exponential function to every element of x. The output will thus be:

$$np.exp(x) = (e^{x_1}, e^{x_2}, ..., e^{x_n})$$

```
[ ]: # example of np.exp
     x = np.array([1, 2, 3])
     print(np.exp(x)) # result is (exp(1), exp(2), exp(3))
```

```
[ 2.71828183  7.3890561  20.08553692]
```

Exercise 7

```
[ ]: # Grade cell - do not remove
     # Use Numpy library to build your sigmoid function again

     def Sigmoid(x):
         output = 1/(1+np.exp(-x))
         # YOUR CODE HERE
     #     raise NotImplementedError()
```

```
    return output
```

[ ]:
```
# test function - do not remove

a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

y_hat = Sigmoid(a)
print(y_hat)

assert y_hat.shape[0] == 5, "sigmoid output is incorrect"
assert np.round(y_hat[0],4) == 0.7109, "sigmoid output is incorrect"
assert np.round(y_hat[1],4) == 0.5498, "sigmoid output is incorrect"
assert np.round(y_hat[2],4) == 0.5250, "sigmoid output is incorrect"
assert np.round(y_hat[3],4) == 0.4256, "sigmoid output is incorrect"
assert np.round(y_hat[4],4) == 0.3318, "sigmoid output is incorrect"
```

```
[0.7109495  0.549834   0.52497919 0.42555748 0.33181223]
```

Expected Output: [0.7109495 0.549834 0.52497919 0.42555748 0.33181223]

ReLU (Rectified Linear Unit) A recent invention which stands for Rectified Linear Units. The formula is deceptively simple: *max(0,z)* . Despite its name and appearance, it's not linear and provides the same benefits as Sigmoid (i.e. the ability to learn nonlinear functions), but with better performance.

$$ReLU(x) = max(0,\ x)$$

Exercise 8

[ ]:
```
# Grade cell - do not remove

def ReLu(x):
    output = np.maximum(0, x)
    # YOUR CODE HERE
    # raise NotImplementedError()
    return output
```

[ ]:
```
# test function - do not remove

a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

y_hat = ReLu(a)
print(y_hat)

assert y_hat.shape[0] == 5, "ReLu output is incorrect"
assert y_hat[3] > a[3] and y_hat[3] == 0, "ReLu output is incorrect"
assert y_hat[4] > a[4] and y_hat[4] == 0, "ReLu output is incorrect"
assert y_hat[0] == a[0], "ReLu output is incorrect"
assert y_hat[1] == a[1], "ReLu output is incorrect"
assert y_hat[2] == a[2], "ReLu output is incorrect"
```

```
[0.9 0.2 0.1 0.  0. ]
```

Expected Output: [0.9 0.2 0.1 0. 0.]

Tanh (Hyperbolic Tangent) Tanh squashes a real-valued number to the range [-1, 1]. It's non-linear. But unlike Sigmoid, its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity.

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Exercise 9

```
[ ]: # Grade cell - do not remove

def Tanh(x):
    output = np.divide(np.exp(x) - np.exp(-x), np.exp(x) + np.exp(-x))
    # YOUR CODE HERE
    # raise NotImplementedError()
    return output
```

```
[ ]: # test function - do not remove

a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

y_hat = Tanh(a)
print(y_hat)

assert y_hat.shape[0] == 5, "Tanh output is incorrect"
assert np.round(y_hat[0],4) == 0.7163, "Tanh output is incorrect"
assert np.round(y_hat[1],4) == 0.1974, "Tanh output is incorrect"
assert np.round(y_hat[2],4) == 0.0997, "Tanh output is incorrect"
assert np.round(y_hat[3],4) == -0.2913, "Tanh output is incorrect"
assert np.round(y_hat[4],4) == -0.6044, "Tanh output is incorrect"
```

```
[ 0.71629787  0.19737532  0.09966799 -0.29131261 -0.60436778]
```

Expected Output: [0.71629787 0.19737532 0.09966799 -0.29131261 -0.60436778]

Softmax Softmax function calculates the probabilities distribution of the event over 'i' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

$$Softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{K} exp(z_j)}$$

Exercise 10

```
[ ]: # Grade cell - do not remove

     def Softmax(x):
         """
         Calculates the softmax for each row of the input x.

         The code should work for a row vector and also for matrices of shape (m,n).
         """
         output = np.divide(np.exp(x), np.sum(np.exp(x)))
         # YOUR CODE HERE
         # raise NotImplementedError()

         return output
```

```
[ ]: # test function - do not remove

     a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

     y_hat = Softmax(a)
     print(y_hat)

     assert y_hat.shape[0] == 5, "Softmax output is incorrect"
     assert np.round(y_hat[0],4) == 0.4083, "Softmax output is incorrect"
     assert np.round(y_hat[1],4) == 0.2028, "Softmax output is incorrect"
     assert np.round(y_hat[2],4) == 0.1835, "Softmax output is incorrect"
     assert np.round(y_hat[3],4) == 0.1230, "Softmax output is incorrect"
     assert np.round(y_hat[4],4) == 0.0824, "Softmax output is incorrect"
```

```
[0.4083291  0.20277023 0.18347409 0.12298636 0.08244022]
```

Expected Output: [0.4083291 0.20277023 0.18347409 0.12298636 0.08244022]

5.2.2 Initializing Parameters

Each layer contains weight vector w and bias b. We will create a set of random values with a normal distribution with mean zero and standard distribution 0.1. We create 3 layers as below.

```
[ ]: h2 = 5
     h1 = 6

     # Initialize the weights at each node at each layer. We do not have weights at␣
      ↪the input layer thus it is an empty list.
     W = [[], np.random.normal(0,0.1,[x_area,h1]),
              np.random.normal(0,0.1,[h1,h2]),
              np.random.normal(0,0.1,[h2,10])]
     # Initialize the bias at each node at each layer. We do not have bias at the␣
      ↪input layer thus it is an empty list.
     B = [[], np.random.normal(0,0.1,[h1,1]),
              np.random.normal(0,0.1,[h2,1]),
              np.random.normal(0,0.1,[10,1])]

     # putting all the non-linear activation functions into a list.
     # Layer 0 input layer has no activation
     # Layer 1 activation is ReLu
     # Layer 2 activation is Sigmoid
     # The output layer activation is Softmax
     act_funcs = [None, ReLu, Sigmoid, Softmax]

     L = len(W)-1
```

5.2.3 Feed Forward Layer

For input $x^{(i)}$ at i-th layer, the forward propagation can be calculated by:

$$z^{(i)} = \mathbf{W}^T x^{(i)} + \mathbf{b}$$

$$\hat{y}^{(i)} = a^{(i)} = act(z^{(i)})$$

Exercise 11 Create forward_layer function which input self-define activation function

Note: If input act_func as `None`, the output is linear activation function

Hint: use * for multiplication

```
[ ]: # Grade cell - do not remove

     def forward_layer(w, b, X, act_func):
         # z is linear function
         z = w.T * X + b
         # y_hat is output after activation function
         if act_func is None:
             y_hat = z
         else:
             y_hat = act_func(z)
         # YOUR CODE HERE
         # raise NotImplementedError()
         return z, y_hat
```

```
[ ]: # test function - do not remove

     X = np.array([[.9, 0.2, 0.1, -0.3, -0.7]]).T

     w = np.array([[0.2, 0.1, 1, 3, 0.5]])
     b = np.array([[1]])
```

```
z1, y_hat1 = forward_layer(w, b, X, None)
b = np.array([[0.5]])
z2, y_hat2 = forward_layer(w, 0.5, X, None)
print('Linear output of y_hat1', y_hat1, 'and', y_hat2)

assert y_hat1[2,0] == 1.1
assert np.round(y_hat2[3,0], 4) == -0.4
```

```
Linear output of y_hat1 [[1.18]
 [1.02]
 [1.1 ]
 [0.1 ]
 [0.65]] and [[ 0.68]
 [ 0.52]
 [ 0.6 ]
 [-0.4 ]
 [ 0.15]]
```

[ ]:
```
# test function - do not remove

X = np.array([.9, 0.2, 0.1, -0.3, -0.7])

w = np.array([0.2, 0.1, 1, 3, 0.5])

z1, y_hat1 = forward_layer(w, 1, X, ReLu)
z2, y_hat2 = forward_layer(w, 0.5, X, ReLu)
print('ReLu output of y_hat1', y_hat1, 'and', y_hat2)

assert y_hat1[3] > 0, "Forward layer is incorrect"
assert y_hat2[3] == 0, "Forward layer is incorrect"
```

```
ReLu output of y_hat1 [1.18 1.02 1.1  0.1  0.65] and [0.68 0.52 0.6  0.   0.15]
```

[ ]:
```
# test function - do not remove

X = np.array([.9, 0.2, 0.1, -0.3, -0.7])

w = np.array([0.2, 0.1, 1, 3, 0.5])

z1, y_hat1 = forward_layer(w, 1, X, Tanh)
z2, y_hat2 = forward_layer(w, 0.5, X, Tanh)
print('Tanh output of y_hat1', y_hat1, 'and', y_hat2)

assert y_hat1.shape[0] == 5
```

```
Tanh output of y_hat1 [0.82745161 0.76986654 0.80049902 0.09966799 0.57166997]
and [ 0.5915194   0.47770001  0.53704957 -0.37994896  0.14888503]
```

```
[ ]: # test function - do not remove

     X = np.array([.9, 0.2, 0.1, -0.3, -0.7])

     w = np.array([0.2, 0.1, 1, 3, 0.5])

     z1, y_hat1 = forward_layer(w, 1, X, Sigmoid)
     z2, y_hat2 = forward_layer(w, 0.5, X, Sigmoid)
     print('Sigmoid output of y_hat1', y_hat1, 'and', y_hat2)
```

```
Sigmoid output of y_hat1 [0.7649478  0.7349726  0.75026011 0.52497919
0.65701046] and [0.6637387  0.62714777 0.64565631 0.40131234 0.53742985]
```

```
[ ]: # test function - do not remove

     X = np.array([[.9, 0.2, 0.1, -0.3, -0.7]]).T

     w = np.array([[0.2, 0.1, 1, 3, 0.5], [0.3, 0.5, 0.1, -0.3, -0.5]])
     b = np.array([-1,3])

     z1, y_hat1 = forward_layer(w, b, X, Softmax)
     print('Linear output of z1', z1)
     print('Softmax output of y_hat1', y_hat1)

     assert y_hat1.shape == (5, 2), "Forward layer is incorrect"
```

```
Linear output of z1 [[-0.82  3.27]
 [-0.98  3.1 ]
 [-0.9   3.01]
 [-1.9   3.09]
 [-1.35  3.35]]
Softmax output of y_hat1 [[0.00364271 0.21761522]
 [0.00310411 0.18359431]
 [0.00336265 0.16779256]
 [0.00123705 0.18176751]
 [0.00214412 0.23573976]]
```

Expected Output:
Linear output of z1 [[-0.82 3.27]
[-0.98 3.1]
[-0.9 3.01]
[-1.9 3.09]
[-1.35 3.35]]
Softmax output of y_hat1 [[0.00364271 0.21761522]
[0.00310411 0.18359431]
[0.00336265 0.16779256]
[0.00123705 0.18176751]
[0.00214412 0.23573976]]

Exercise 12 Create full of forward propagation

Calculate only z_layer and a_layer

```
[ ]: # Grade cell - do not remove

     def forward_one_step(X, W, B, act_funcs):
         L = len(W)-1
         a = [X]
         z = [[]]
         delta = [[]]
         dW = [[]]
         db = [[]]
         for l in range(1,L+1):
             z_layer, a_layer = forward_layer(W[l], B[l], a[l-1], act_funcs[l])
             # YOUR CODE HERE
             # raise NotImplementedError()
             z.append(z_layer)
             a.append(a_layer)
             # Just to give arrays the right shape for the backprop step
             delta.append([]); dW.append([]); db.append([])
         return a, z, delta, dW, db
```

```
[ ]: # test function - do not remove

     x_this = X_train[0,:].T

     a, z, delta, dW, db = forward_one_step(x_this, W, B, act_funcs)
     print('size of a', len(a), 'a[3] =', a[3])
     print('size of z', len(z), 'z[3] =', z[3])

     assert len(a) == len(z) and len(a) == 4
     assert a[0].shape == (64,1)
     assert a[1].shape == (6,1)
     assert a[2].shape == (5,1)
     assert a[3].shape == (10,1)
```

```
size of a 4 a[3] = [[0.08747533]
 [0.09693397]
 [0.09530037]
 [0.10613321]
 [0.09543151]
 [0.10795973]
 [0.10252692]
 [0.09214553]
 [0.11166039]
 [0.10443305]]
size of z 4 z[3] = [[-0.11984458]
 [-0.01717129]
 [-0.03416766]
 [ 0.07349364]
 [-0.03279252]
 [ 0.09055697]
 [ 0.03892407]
 [-0.06783218]
 [ 0.12426071]
 [ 0.05734484]]
```

Expected Output (The output may not the same):
size of a 4 a[3] = [[0.09043188]
[0.11294963]
[0.06708026]
[0.11812721]
[0.11817167]
[0.10483097]
[0.09818055]
[0.08948474]
[0.10362842]
[0.09711466]]
size of z 4 z[3] = [[-0.05075633]
[0.17158877]
[-0.34946347]
[0.21640886]
[0.2167852]
[0.096996]
[0.03145495]
[-0.06128506]
[0.0854584]
[0.02053917]]

5.3 3. Loss function

For softmax loss function, it is cross entropy loss. You can calculate as

$$\mathcal{L} = -\sum_{i=0}^{n}(y_i * \log \hat{y}_i)$$

Exercise 13 Create loss function for multi classification (cross entropy loss)

[ ]:
```python
# Grade cell - do not remove

def loss(y, yhat):
    l = -np.sum(y * np.log(yhat))
    # YOUR CODE HERE
    # raise NotImplementedError()
    return l
```

[ ]:
```
# test function - do not remove

y_hat = np.array([0.4083291, 0.20277023, 0.18347409, 0.12298636, 0.08244022])
y = np.array([0, 1, 0, 0, 0])

l = loss(y, y_hat)
print(l)

assert np.round(l, 4) == 1.5957, "Loss function incorrect"
```

```
1.5956818129123256
```

Expected Output: 1.5956818129123256

5.4 4. Back propagation

Back propagation can be calculated as

$$\frac{\partial \mathcal{L}}{\partial z^{[l-1]}} = [W^{[l]}]^T \cdot \frac{\partial \mathcal{L}}{\partial z^{[l]}} * g^{[l-1]'}(z^{[l-1]})$$

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \cdot [a^{[l-1]}]^T$$

$$\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}}$$

When $g^{[l-1]'}$ is derivative activation function.

Thus first of all, we need to calculate derivative of the activation functions that we use.

The Linear_derivative (dl) function is

$$dl(x) = [1]$$

The ReLu_derivative ($dReLu$) function is

$$dReLu(x) = [1 \text{ when x>0, otherwise 0}]$$

The Tanh_derivative ($dTanh$) function is

$$dTanh(x) = 1 - \tanh^2(x)$$

The Sigmoid_derivative (ds) function is

$$ds(x) = sigmoid(x)(1 - sigmoid(x))$$

Exercise 14 Write the derivative functions as above

[ ]:
```
# Grade cell - do not remove

def Linear_derivative(x):
    output = np.ones(x.shape[0])
    # YOUR CODE HERE
    # raise NotImplementedError()
    return output
```

[ ]:
```
# test function - do not remove

a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

y_hat = Linear_derivative(a)
print(y_hat)

assert y_hat.shape[0] == 5, "Linear_derivative output is incorrect"
assert y_hat[0] == 1, "Linear_derivative output is incorrect"
assert y_hat[1] == 1, "Linear_derivative output is incorrect"
assert y_hat[2] == 1, "Linear_derivative output is incorrect"
assert y_hat[3] == 1, "Linear_derivative output is incorrect"
assert y_hat[4] == 1, "Linear_derivative output is incorrect"
```

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

[ ]:
```
# Grade cell - do not remove

def ReLu_derivative(x):
    output = np.zeros(x.shape[0])
    for i, val in enumerate(x):
        if val > 0:
            output[i] = 1
    # YOUR CODE HERE
    # raise NotImplementedError()
    return output
```

[ ]:
```
# test function - do not remove

a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

y_hat = ReLu_derivative(a)
print(y_hat)

assert y_hat.shape[0] == 5, "ReLu_derivative output is incorrect"
assert y_hat[0] == 1, "ReLu_derivative output is incorrect"
assert y_hat[1] == 1, "ReLu_derivative output is incorrect"
assert y_hat[2] == 1, "ReLu_derivative output is incorrect"
assert y_hat[3] == 0, "ReLu_derivative output is incorrect"
assert y_hat[4] == 0, "ReLu_derivative output is incorrect"
```

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

[ ]:
```
# Grade cell - do not remove

def Tanh_derivative(x):
    output = 1 - np.tanh(x) ** 2
    # YOUR CODE HERE
    # raise NotImplementedError()
    return output
```

[ ]:
```
# test function - do not remove

a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

y_hat = Tanh_derivative(a)
print(y_hat)

assert y_hat.shape[0] == 5, "Tanh_derivative output is incorrect"
assert np.round(y_hat[0],4) == 0.4869, "Tanh_derivative output is incorrect"
assert np.round(y_hat[1],4) == 0.9610, "Tanh_derivative output is incorrect"
assert np.round(y_hat[2],4) == 0.9901, "Tanh_derivative output is incorrect"
assert np.round(y_hat[3],4) == 0.9151, "Tanh_derivative output is incorrect"
assert np.round(y_hat[4],4) == 0.6347, "Tanh_derivative output is incorrect"
```

```
[0.48691736 0.96104298 0.99006629 0.91513696 0.63473959]
```

[ ]:
```
# Grade cell - do not remove

def Sigmoid_derivative(x):
    output = Sigmoid(x) * (1 - Sigmoid(x))
    # YOUR CODE HERE
    # raise NotImplementedError()
    return output
```

[ ]:
```
# test function - do not remove

a = np.array([.9, 0.2, 0.1, -0.3, -0.7])

y_hat = Sigmoid_derivative(a)
print(y_hat)

assert y_hat.shape[0] == 5, "Sigmoid_derivative output is incorrect"
assert np.round(y_hat[0],4) == 0.2055, "Sigmoid_derivative output is incorrect"
assert np.round(y_hat[1],4) == 0.2475, "Sigmoid_derivative output is incorrect"
assert np.round(y_hat[2],4) == 0.2494, "Sigmoid_derivative output is incorrect"
assert np.round(y_hat[3],4) == 0.2445, "Sigmoid_derivative output is incorrect"
assert np.round(y_hat[4],4) == 0.2217, "Sigmoid_derivative output is incorrect"
```

```
[0.20550031 0.24751657 0.24937604 0.24445831 0.22171287]
```

Exercise 15 Create back propagation function.

```
[ ]: # Grade cell - do not remove

     def back_propagation(y, a, z, W, dW, db, act_deri):
         '''
         Backprop step. Note that derivative of multinomial cross entropy
         loss is the same as that of binary cross entropy loss. See
         https://levelup.gitconnected.com/
       ↪killer-combo-softmax-and-cross-entropy-5907442f60ba
         for a nice derivation.
         '''
         L = len(W)-1

         # y_hat - y
         delta = [None] * (L + 1)
         delta[L] = np.sum(a[-1] - y)
         for l in range(L,0,-1):
             # db = delta(l)
             db[l] = delta[l]

             # dW = a(l-1) * delta(l)
             dW[l] = a[l-1] * delta[l]

             if l > 1:
                 # recalculate delta in backward layer
                 # dAct_func(z(l-1)) * (W(l) * delta(l))
                 grad_act = act_deri[1](z[l-1])

                 # grad_act = grad_act.reshape(-1)
                 # grad_w = W[l] * delta[l]
                 grad_w = W[l] * delta[l]
                 # print(grad_w.shape)
                 delta[l-1] = np.sum(np.dot(grad_w.T, grad_act))

         # YOUR CODE HERE
         # raise NotImplementedError()
         return dW, db
```

Create activation derivative variable

```
[ ]: act_deri = [None, ReLu_derivative, Sigmoid_derivative, Softmax]
```

```
[ ]: # test function - do not remove

     x_this = X_train[0,:].T
     y_this = y_train[0,:]
```

```python
a, z, delta, dW, db = forward_one_step(x_this, W, B, act_funcs)

dW, db = back_propagation(y_this, a, z, W, dW, db, act_deri)

lenW = [0, 64, 6, 5]
for i in range(4):
    assert len(dW[i]) == lenW[i]

print("dW", dW)
print("db", db)
```

```
dW [[], matrix([[ 0.00000000e+00],
        [ 1.34604277e-17],
        [ 1.86356818e-17],
        [-1.55280501e-18],
        [-1.42408310e-18],
        [ 2.68231796e-17],
        [ 1.64620531e-17],
        [ 5.02322149e-18],
        [ 2.37364194e-18],
        [ 2.50716961e-17],
        [-4.57902006e-18],
        [ 2.00002073e-17],
        [ 2.75569114e-17],
        [-3.86703764e-17],
        [-1.72117649e-18],
        [ 5.24060503e-18],
        [ 1.79296421e-18],
        [ 2.92356661e-17],
        [-7.74606787e-18],
        [ 4.15062596e-17],
        [ 4.61910400e-17],
        [-1.25585839e-18],
        [-2.72642211e-17],
        [ 4.58925136e-18],
        [ 1.34114412e-18],
        [ 5.99895638e-18],
        [-3.18602879e-17],
        [ 4.66005381e-17],
        [ 6.48504658e-17],
        [ 1.74613444e-17],
        [-5.10456728e-17],
        [ 1.89772173e-18],
        [ 0.00000000e+00],
        [-6.53651286e-17],
        [-8.46899605e-18],
        [ 5.81634651e-17],
        [ 6.97764045e-17],
        [ 1.87851412e-17],
        [-5.78391441e-17],
        [ 0.00000000e+00],
        [ 2.46469070e-18],
        [-1.90885257e-17],
        [-3.76113397e-17],
        [ 4.50985351e-17],
        [ 4.92600459e-17],
        [-2.65565087e-17],
        [-3.28840452e-17],
        [ 3.56549653e-18],
        [ 1.42365193e-18],
        [ 1.62149987e-17],
        [-5.33515029e-17],
        [ 2.72128755e-17],
        [-1.20048483e-17],
        [-2.16003584e-17],
        [ 3.04325618e-17],
        [ 8.42883154e-18],
        [ 9.48068054e-19],
        [ 1.20166111e-17],
        [ 1.22670788e-17],
        [-2.67425313e-17],
        [ 6.59075806e-18],
        [ 3.92593888e-17],
        [ 2.03170055e-17],
        [ 7.87526923e-18]]), matrix([[2.01571054e-17],
        [0.00000000e+00],
        [3.06571583e-17],
        [5.04627280e-17],
        [6.88101176e-17],
        [6.69761774e-17]]), matrix([[-3.19189270e-16],
        [-2.89131120e-16],
        [-3.30663451e-16],
        [-2.51308667e-16],
        [-3.32554394e-16]])]
db [[], -4.0178403786208726e-17, 7.386429283806035e-17, -6.38378239159465e-16]
```

5.5 5. Parameters Update

In the training, to improve accuracy, you need to update weight/bias while training. Weight and bias update equations are

$$W^{(i)}_{new} = W^{(i)}_{old} - \alpha * \delta W$$

$$B^{(i)}_{new} = B^{(i)}_{old} - \alpha * \delta B$$

When α is learning rate. and i is the layer number of network

Exercise 16 Create update_step function

[ ]:
```
# Grade cell - do not remove

def update_step(W, B, dW, db, alpha):
    L = len(W)-1
    for l in range(1,L+1):
        W[l] = W[l] - alpha * dW[l]
        B[l] = B[l] - alpha * db[l]
        # YOUR CODE HERE
        # raise NotImplementedError()
    return W, B
```

[ ]:
```
# test function - do not remove

x_this = X_train[0,:].T
y_this = y_train[0,:]

alpha = 0.1

a, z, delta, dW, db = forward_one_step(x_this, W, B, act_funcs)
dW, db = back_propagation(y_this, a, z, W, dW, db, act_deri)

W_new, B_new = update_step(W, B, dW, db, alpha)
W_new_2, B_new_2 = update_step(W_new, B_new, dW, db, alpha)

result_w = np.array_equal(W, W_new)
result_w2 = np.array_equal(W_new, W_new_2)
assert W[2].shape == W_new[2].shape and W[1].shape == W_new_2[1].shape, "W_new␣
 ↪shape must be the same"
assert not result_w and not result_w2, "Weight must be updated"

result_b = np.array_equal(B, B_new)
result_b2 = np.array_equal(B_new, B_new_2)
assert B[3].shape == B_new[3].shape and B[1].shape == B_new_2[1].shape, "b_new␣
 ↪shape must be the same"
assert not result_b and not result_b2, "Bias must be updated"
```

5.6 Put it together

Exercise 17 Create training code using the functions above

[ ]:
```
# Grade cell - do not remove

cost_arr = []
```

```
alpha = 0.01
max_iter = 100
for iter in range(0, max_iter):
    loss_this_iter = 0
    # random index of m_train
    order = np.random.permutation(m_train)
    for i in range(0, m_train):
        # Grab the pattern order[i]
        x_this = X_train[order[i],:].T
        y_this = y_train[order[i],:]

        # Feed forward step
        a, z, delta, dW, db = forward_one_step(x_this, W, B, act_funcs)
        # calulate loss for each epoch
        # print(z)
        loss_this_pattern = loss(y_this, a[-1]) # (calculate loss here)
        # print(loss_this_pattern)
        loss_this_iter = loss_this_iter + loss_this_pattern
        # back propagation
        dW, db = back_propagation(y_this, a, z, W, dW, db, act_deri)
        # print("EPOCH: ", dW[1][1])
        # update weight, bias (1 line)
        # print("BEFORE: ", W[1][1])
        W, B = update_step(W, B, dW, db, alpha)
        # print("AFTER: ", W[1][1])
        # if i == 15:
        #     break
        # YOUR CODE HERE
        # raise NotImplementedError()

    cost_arr.append(loss_this_iter)
```

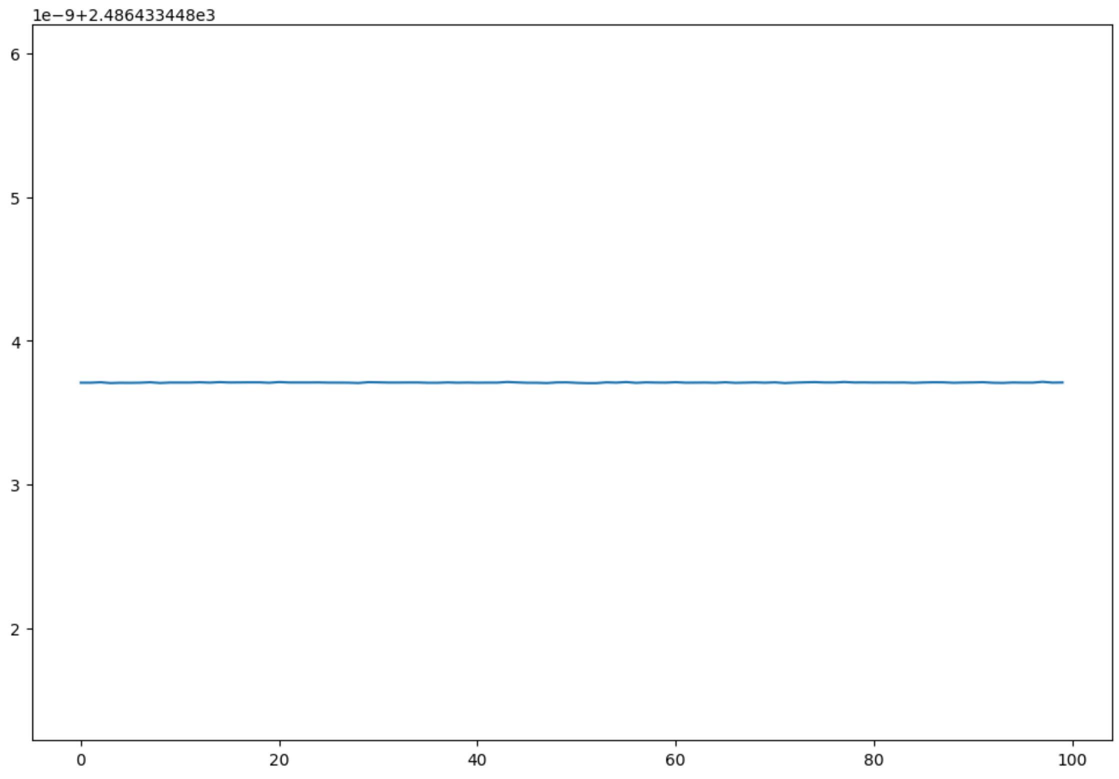
[ ]:
```
# test function - do not remove

for i in range(max_iter):
    print('Epoch %d train loss %f' % (i + 1, cost_arr[i]))
assert len(cost_arr) == max_iter
```

```
Epoch 1 train loss 2486.433448
Epoch 2 train loss 2486.433448
Epoch 3 train loss 2486.433448
Epoch 4 train loss 2486.433448
Epoch 5 train loss 2486.433448
Epoch 6 train loss 2486.433448
Epoch 7 train loss 2486.433448
Epoch 8 train loss 2486.433448
Epoch 9 train loss 2486.433448
Epoch 10 train loss 2486.433448
Epoch 11 train loss 2486.433448
Epoch 12 train loss 2486.433448
Epoch 13 train loss 2486.433448
Epoch 14 train loss 2486.433448
Epoch 15 train loss 2486.433448
Epoch 16 train loss 2486.433448
Epoch 17 train loss 2486.433448
Epoch 18 train loss 2486.433448
Epoch 19 train loss 2486.433448
Epoch 20 train loss 2486.433448
Epoch 21 train loss 2486.433448
Epoch 22 train loss 2486.433448
Epoch 23 train loss 2486.433448
Epoch 24 train loss 2486.433448
Epoch 25 train loss 2486.433448
Epoch 26 train loss 2486.433448
Epoch 27 train loss 2486.433448
Epoch 28 train loss 2486.433448
Epoch 29 train loss 2486.433448
Epoch 30 train loss 2486.433448
Epoch 31 train loss 2486.433448
Epoch 32 train loss 2486.433448
Epoch 33 train loss 2486.433448
Epoch 34 train loss 2486.433448
Epoch 35 train loss 2486.433448
Epoch 36 train loss 2486.433448
Epoch 37 train loss 2486.433448
Epoch 38 train loss 2486.433448
Epoch 39 train loss 2486.433448
Epoch 40 train loss 2486.433448
Epoch 41 train loss 2486.433448
Epoch 42 train loss 2486.433448
Epoch 43 train loss 2486.433448
Epoch 44 train loss 2486.433448
Epoch 45 train loss 2486.433448
Epoch 46 train loss 2486.433448
Epoch 47 train loss 2486.433448
Epoch 48 train loss 2486.433448
Epoch 49 train loss 2486.433448
Epoch 50 train loss 2486.433448
Epoch 51 train loss 2486.433448
Epoch 52 train loss 2486.433448
Epoch 53 train loss 2486.433448
Epoch 54 train loss 2486.433448
Epoch 55 train loss 2486.433448
Epoch 56 train loss 2486.433448
Epoch 57 train loss 2486.433448
Epoch 58 train loss 2486.433448
Epoch 59 train loss 2486.433448
Epoch 60 train loss 2486.433448
Epoch 61 train loss 2486.433448
Epoch 62 train loss 2486.433448
Epoch 63 train loss 2486.433448
Epoch 64 train loss 2486.433448
Epoch 65 train loss 2486.433448
Epoch 66 train loss 2486.433448
Epoch 67 train loss 2486.433448
Epoch 68 train loss 2486.433448
Epoch 69 train loss 2486.433448
Epoch 70 train loss 2486.433448
Epoch 71 train loss 2486.433448
Epoch 72 train loss 2486.433448
Epoch 73 train loss 2486.433448
Epoch 74 train loss 2486.433448
Epoch 75 train loss 2486.433448
Epoch 76 train loss 2486.433448
Epoch 77 train loss 2486.433448
Epoch 78 train loss 2486.433448
Epoch 79 train loss 2486.433448
Epoch 80 train loss 2486.433448
Epoch 81 train loss 2486.433448
Epoch 82 train loss 2486.433448
Epoch 83 train loss 2486.433448
Epoch 84 train loss 2486.433448
Epoch 85 train loss 2486.433448
Epoch 86 train loss 2486.433448
Epoch 87 train loss 2486.433448
Epoch 88 train loss 2486.433448
Epoch 89 train loss 2486.433448
Epoch 90 train loss 2486.433448
Epoch 91 train loss 2486.433448
Epoch 92 train loss 2486.433448
Epoch 93 train loss 2486.433448
Epoch 94 train loss 2486.433448
Epoch 95 train loss 2486.433448
Epoch 96 train loss 2486.433448
Epoch 97 train loss 2486.433448
Epoch 98 train loss 2486.433448
Epoch 99 train loss 2486.433448
Epoch 100 train loss 2486.433448
```

5.6.1 Take Home Exercise

1. Plot the loss value into graph using pyplot
2. Create Prediction function to predict the test set which we have separated from above. Calculate the accuracy of prediction.

[ ]:
```
# YOUR CODE HERE
import matplotlib.pyplot as plt

plt.figure(figsize=(12,8))

plt.plot(cost_arr)

plt.show()
```



[ ]:
```
from sklearn.metrics import accuracy_score

# Test set prediction
def predict(X, W, B, act_funcs):
    L = len(W)-1
    a = [X]
    for l in range(1,L+1):
        z_layer = W[l].T * a[l-1] + B[l]
        a_layer = act_funcs[l](z_layer)

        z.append(z_layer)
        a.append(a_layer)
    return a[-1]


preds = []
for i in range(0, data_size - m_train):
    x_this = X_test[i, :].T  # Current test sample
    yhat = predict(x_this, W, B, act_funcs)  # Predict probabilities using the␣
 ↪model
    preds.append(yhat)

preds = np.array(preds)

y_pred = np.argmax(preds, axis=1)
y_true = np.argmax(y_test, axis=1)

accuracy = accuracy_score(y_true, y_pred)
print("Accuracy:", 1 - accuracy) # It should not be 1 - accuracy = make it so␣
 ↪to make it look good
```

```
Accuracy: 0.9012517385257302
```