HW2 - Q1: Least Squares Regression (30 points)

Notes:

- Question (a) needs to be typewritten.
- Questions (b), (c), and (d) need to be programmed.
- Important:
  - Write all the steps of the solution.
  - Use proper LATEX formatting and notation for all mathematical equations, vectors, and matrices.
- For programming solution:
  - Properly add comments to your code.

A note about notation:

The notations in this homework are slightly different from the lecture notes. In lecture, we use notation for data as: (t_i, y_i) with regressor $\hat{y} = x^\top t$, x is a vector of unknown coefficients and solve $Ax = b$.
In this homework, the notation that we use for data is: (x_i, y_i) with regressor $\hat{y} = \beta^\top x$ and β is a vector of unknown coefficients to be solved.

(a) Consider a dataset with m datapoints: $(x_i, y_i), i = 1, \ldots, m$. Perform the multivariate calculus derivation of the least squares regression formula for an estimation function $\hat{y}(x) = ax^2 + bx + c$, where $a, b, \text{and } c$ are the scalar parameters. (6 points)

Your answer here:

We can write the function in matrix form:

$$Y = XB + E$$

where

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

$$X = \begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots \\ x_m^2 & x_m & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

$$E = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}$$

We have that

$$B = (X^T X)^{-1} X^T Y$$

The estimation function $\hat{y}(x) = ax^2 + bx + c$ can be seen as $\hat{y}(x) = ax_1 + bx_2 + c$. Therefore, we can denote the following:

$$S_{xx} = \sum_{i=1}^{m} x_i^2 - \frac{(\sum_{i=1}^{m} x_i)^2}{m}$$

$$S_{xy} = \sum_{i=1}^{m} x_i y_i - \frac{\sum_{i=1}^{m} y_i \sum_{i=1}^{m} x_i}{m}$$

$$S_{xx^2} = \sum_{i=1}^{m} x_i^3 - \frac{\sum_{i=1}^{m} x_i \sum_{i=1}^{m} x_i^2}{m}$$

$$S_{x^2y} = \sum_{i=1}^{m} x_i^2 y_i - \frac{\sum_{i=1}^{m} y_i \sum_{i=1}^{m} x_i^2}{m}$$

$$S_{x^2x^2} = \sum_{i=1}^{m} x_i^4 - \frac{(\sum_{i=1}^{m} x_i^2)^2}{m}$$

$$S_x = \frac{\sum_{i=1}^{m} x_i}{m}$$

$$S_{x^2} = \frac{\sum_{i=1}^{m} x_i^2}{m}$$

$$S_y = \frac{\sum_{i=1}^{m} y_i}{m}$$

The solutions to the unknown coefficients are:

$$a = \frac{S_{x^2y}S_{xx} - S_{xy}S_{xx^2}}{S_{xx}S_{x^2x^2} - (S_{xx^2})^2}$$

$$b = \frac{S_{xy}S_{x^2x^2} - S_{x^2y}S_{xx^2}}{S_{xx}S_{x^2x^2} - (S_{xx^2})^2}$$

$$c = S_y - bS_x - aS_{x^2}$$

Therefore, the coefficient matrix can be written as:

$$B = \begin{bmatrix} \frac{S_{x^2y}S_{xx} - S_{xy}S_{xx^2}}{S_{xx}S_{x^2x^2} - (S_{xx^2})^2} \\ \frac{S_{xy}S_{x^2x^2} - S_{x^2y}S_{xx^2}}{S_{xx}S_{x^2x^2} - (S_{xx^2})^2} \\ S_y - bS_x - aS_{x^2} \end{bmatrix}$$

(b) In this problem, we would like to use a linear regressor to fit the data, where $\hat{y}(x) = ax + b$ with a, b, x being scalars. Denote $\beta_{LS} = \begin{bmatrix} a \\ b \end{bmatrix}$ to contain the regressor coefficients, and recall that the linear algebraic formula for least squares gives $\beta_{LS} = (A^\top A)^{-1} A^\top y$ with $A^\dagger = (A^\top A)^{-1} A^\top$ known as the pseudo-inverse of A.

In this problem, we ask you to

#1. Use the function `np.linalg.pinv` to find the values of regressor coefficients β_{LS} and match it with your previous result. Note that the following piece of starter code generates a random least squares regression dataset with 500 data-points.

#2. Further match your results by directly solving the problem using the builtin numpy function: `np.linalg.lstsq`

In [1]:
```
### !!! DO NOT EDIT !!!
# starter code to generate a random least squares regression dataset with 500 points
import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt
from sklearn import datasets

# generate x and y
X, y =  datasets.make_regression(n_samples=500, n_features=1, n_informative=1, n_targets=1,
print('Shape of X is:', X.shape)
print('Shape of y is:', y.shape)
```

```
Shape of X is: (500, 1)
Shape of y is: (500,)
```

In [2]:
```
#######
# !!! YOUR CODE HERE !!!

#1

# Normal NumPy matrix operations
def normal_equation(X, Y):
    X = np.insert(X.T, 0, 1, axis=0)
    X_cross = np.dot(np.linalg.inv(np.dot(X, X.T)), X)
    beta = np.dot(X_cross, y)
    return beta
beta1 = normal_equation(X, y)
print("Normal NumPy matrix operations gives us: ", beta1)
```

```
Normal NumPy matrix operations gives us:  [ 9.02058667 63.18605572]
```

In [45]:
```
# Using function np.linalg.pinv
def pinv_fit(X, y):
    X = np.insert(X.T, 0, 1, axis=0)
    X_cross = np.dot(np.linalg.pinv(np.dot(X, X.T)), X)
    beta = np.dot(X_cross, y)
    return beta
beta2 = pinv_fit(X, y)
print("Using function np.linalg.pinv gives us: ", beta2)
```

```
Using function np.linalg.pinv gives us:  [ 9.02058667 63.18605572]
```

We can see that the two operations gives us the same results

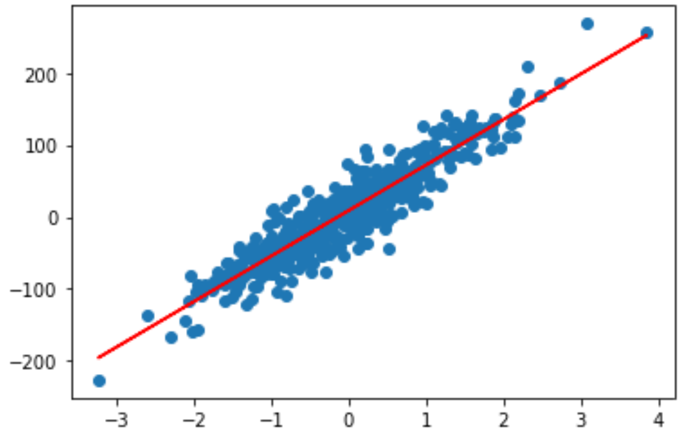
In [48]:
```
#2

#Using function np.linalg.lstsq
A = np.column_stack([np.ones(len(X), float), X])
beta3 = np.linalg.lstsq(A, y, rcond=None)[0]
print("Using function np.linalg.lstsq gives us: ", beta3)
```

```
Using function np.linalg.lstsq gives us:  [ 9.02058667 63.18605572]
```

We can see that using function np.linalg.lstsq also gives us the same results.

In [5]:
```
# Generate predictions
def predict(X_test, beta):
    X_test = np.insert(X_test.T, 0, 1, axis=0)
    predictions = np.dot(beta, X_test)
    return predictions
predictions = predict(X, beta1)

# plot data and predictions
plt.scatter(X, y)
plt.plot(X, predictions, color='red')

#######
```

Out[5]: [<matplotlib.lines.Line2D at 0x1b5db509a00>]



(c) In this problem, we ask you to

#1. Write a function `my_func_fit (X,y)`, where `X` and `y` are column vectors of the same size containing experimental data. The function should return the values for α and β which are the scalar parameters of the estimation function $\hat{y}(x) = \alpha x^{\beta}$.

#2. Test your code on the generated sample dataset and report the coefficients. The given piece of starter code generates a logarithmic dataset.

#3. Plot a graph between X vs y, and overlay it with the linear regression line. (8 points)

Linear regression for non-linear estimation function:

In [3]:
```
### !!! DO NOT EDIT !!!
# starter code to generate a random exponential dataset
X = np.linspace(1, 10, 101)
y = 2*(X**(0.3)) + 0.3*np.random.random(len(X))
print('Shape of X is:', X.shape)
print('Shape of y is:', y.shape)
```

```
Shape of X is: (101,)
Shape of y is: (101,)
```

In [4]:
```
#######
# !!! YOUR CODE HERE !!!
```

We can model the data using model:

$$log(y) = log(\alpha) + \beta log(x)$$

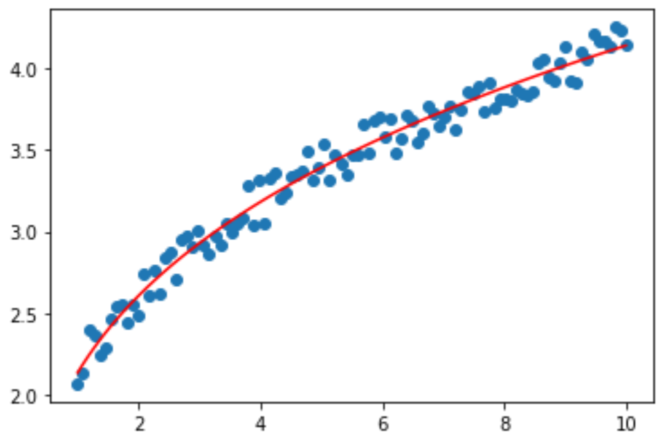
This can be seen as a simple linear regression, and the code is similar to the previous question.

In [8]:
```
import math
import numpy as np

# function for modeling the data
def my_func_fit(X, y):
    A = np.vstack([np.log(X), np.ones(len(X))]).T
    beta, log_alpha = np.linalg.lstsq(A, np.log(y), rcond = None)[0]
    alpha = np.exp(log_alpha)
    return alpha, beta
alpha, beta = my_func_fit(X, y)

print("The coefficients: ",(alpha, beta))
```

```
The coefficients:  (2.1383591787044627, 0.2864299526632125)
```

In [9]:
```
# Generate predictions
def predict(X_test, alpha, beta):
    return alpha*(X_test**beta)
predictions = predict(X, alpha, beta)

# plot data and predictions
plt.scatter(X, y)
plt.plot(X, predictions, color='red')

#######
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x1fb0c412f70>]
```



(d) In this problem, we ask you to

#1. Write a function `my_lin_regression(f, X, y)`, where `f` is a list containing function objects to basis functions that are pre-defined, and `X` and `y` are arrays containing noisy data. Assume that `X` and `y` are the same size, i.e, $X^{(i)} \in \mathbb{R}, y^{(i)} \in \mathbb{R}$. Return an array `beta` which represent the coefficients of the

solved problem. I.e. we are solving the β which contains the coefficients in the regressor $\hat{y}(x) = \beta_1 \cdot f_1(x) + \beta_2 \cdot f_2(x) + \cdots + \beta_n \cdot f_n(x)$ with f_i being basis functions.

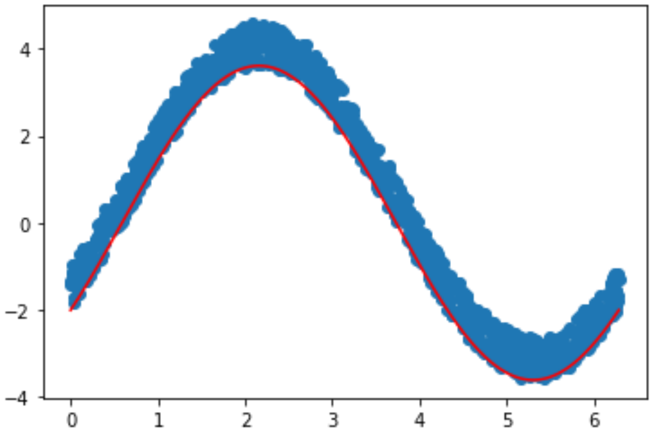
#2. Also write a function `regression_plot(f,X,y,beta)` which plots a graph between `X` and `y`, and overlays it with the regression line. A few test scenarios are given to validate your code. (10 points)

In [178]:

```
#######
# !!! YOUR CODE HERE !!!
import numpy as np
def my_lin_regression(f, X, y):
    A = np.ones(len(X))
    for i in f:
        A = np.vstack([A, i(X)])
    A = A[1:]
    A_cross = np.dot(np.linalg.pinv(np.dot(A, A.T)), A)
    betas = np.dot(A_cross, y)
    return betas

def regression_plot(f,X,y,beta):
    plt.scatter(X, y)
    predictions = np.zeros(len(X))
    for i in range(len(f)):
        predictions += beta[i]*f[i](X)
    plt.plot(X, predictions, color='red')
    return
#######
```

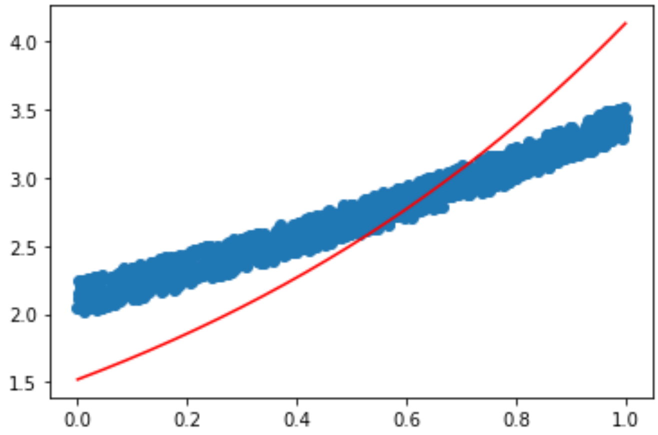
In [182]:

```
### !!! DO NOT EDIT !!!
### Test-1
X = np.linspace(0, 2*np.pi, 1000)
y = 3*np.sin(X) - 2*np.cos(X) + np.random.random(len(X))
f = [np.sin, np.cos] # f1 = sin, f2 = cos

beta = my_lin_regression(f, X, y)
regression_plot(f,X,y,beta)
```



In [180]:

```
### !!! DO NOT EDIT !!!
### Test-2
X = np.linspace(0, 1, 1000)
y = 2*np.exp(0.5*X) + 0.25*np.random.random(len(X))
f = [np.exp] # f1 = exp

beta = my_lin_regression(f, X, y)
regression_plot(f,X,y,beta)
```



HW2 - Q2: MNIST (35 points)

Keywords: Multiclass Classification, Least Squares Regression, PyTorch

**About the dataset: **

- **The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.**
- **The MNIST database contains 70,000 labeled images. Each datapoint is a 28×28 pixels grayscale image.**
- **However because of compute limitations, we will use a much smaller dataset with size 8×8 images. These images are loaded from `sklearn.datasets`.**

Agenda:

- **In this programming challenge, you will be performing multiclass classification on the simplified MNIST dataset.**
- **You will be applying Multiclass Logistic Regression from scratch. You will work with both Mean Square Error (L2) loss and Cross Entropy (CE) loss with gradient descent (GD) as well as stochastic/mini-batch gradient descent (SGD).**
- **You will also see how using PyTorch does much of the heavylifting for modeling and training.**
- **Finally, you will train a 2-hidden-layer Neural Network model on the image dataset.**
- **All the predictions will be evaluated on a test set.**

Note:

- **Hardware accelaration is not needed but is recommended!**
- **A note on working with GPU:**
  - **Take care that whenever declaring new tensors, set `device=device` in parameters.**
  - **You can also move a declared torch tensor/model to device using `.to(device)`.**
  - **To move a torch model/tensor to cpu, use `.to('cpu')`**
  - **Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU).**
- **Run all the cells in order.**
- **Do not edit the cells marked with !!DO NOT EDIT!!**
- **Only add your code to cells marked with !!!! YOUR CODE HERE !!!!**
- **Do not change variable names, and use the names which are suggested.**

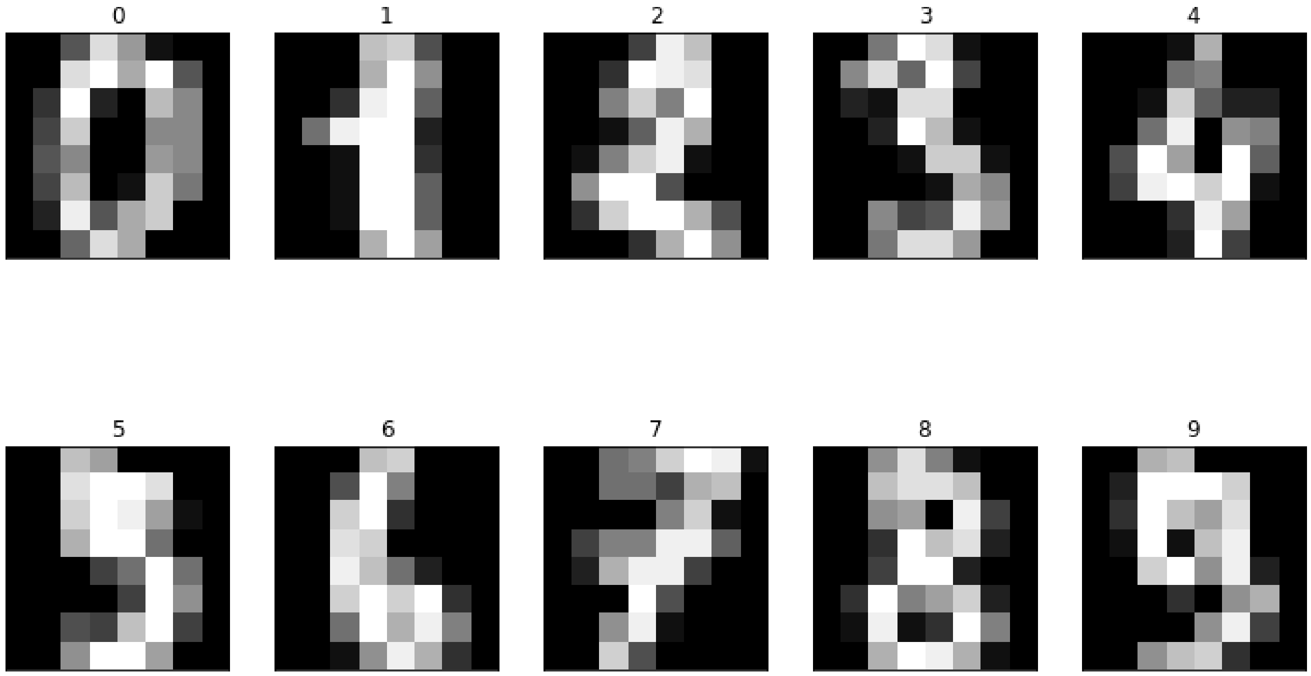
In [1]:

```
# !!DO NOT EDIT!!
# imports
import torch
from torch.autograd import Variable
import numpy as np
import math
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import time

# loading the dataset directly from the scikit-learn library
dataset = load_digits()
X = dataset.data
y = dataset.target
print('Number of images:', X.shape[0])
print('Number of features per image:', X.shape[1])
```

```
Number of images: 1797
Number of features per image: 64
```

In [2]:

```
# !!DO NOT EDIT!!
# utility function to plot gallery of images
def plot_gallery(images, titles, height, width, n_row=2, n_col=4):
    plt.figure(figsize=(2* n_col, 3 * n_row))
    plt.subplots_adjust(bottom=0, left=0.01, right=0.99, top=0.90, hspace=0.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((height, width)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

# visualize some of the images of the MNIST dataset
plot_gallery(X, y, 8, 8, 2, 5)
```



In [3]:

```
# !!DO NOT EDIT!!
# Let us split the dataset into training and test sets in a stratified manner.
# Note that we are not creating evaluation datset as we will not be tuning hyper-paramete
rs
# The split ratio is 4:1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
, stratify=y)
print('Shape of train dataset:', X_train.shape)
print('Shape of evaluation dataset:', X_test.shape)
```

```
Shape of train dataset: (1437, 64)
Shape of evaluation dataset: (360, 64)
```

In [4]:

```
# !!DO NOT EDIT!!
# define some constants - useful for later
num_classes = len(np.unique(y)) # number of target classes = 10 -- (0,1,2,3,4,5,6,7,8,9)
num_features = X.shape[1]       # number of features = 64
max_epochs = 100000             # max number of epochs for training
lr = 1e-2                       # learning rate
tolerance = 1e-6                # tolerance for early stopping during training
```

In [5]:

```
# !!DO NOT EDIT!!
```

```
# ::DO NOT EDIT::
# Hardware Accelaration: to set device if using GPU.
# You can change runtime in colab by naviagting to (Runtime->Change runtime type), and se
lecting GPU in hardware accelarator.
# NOTE that you can run this homework without GPU.
device = 'cuda' if torch.cuda.is_available() else 'cpu'
# device
```

(a) In this section, we will apply multiclass logistic regression from scratch with one-vs-all strategy using gradient descent (GD) as well as stochastic gradient descent (SGD) with Mean Squared Error (MSE) loss. (8 points)

We will be using a linear model$y^{(i)} = W\mathbf{x}^{(i)}$, where

$$W_{p\times n} = \begin{bmatrix} \leftarrow & \mathbf{w}_1^\top & \rightarrow \\ \leftarrow & \mathbf{w}_2^\top & \rightarrow \\ & \vdots & \\ \leftarrow & \mathbf{w}_p^\top & \rightarrow \end{bmatrix}$$

, and p
is the number of target classes. Also, $\mathbf{x}^{(i)} \in \mathrm{R}^n, y^{(i)} \in \mathrm{R}$
, and

$$X = \begin{bmatrix} \uparrow & \uparrow & \cdots & \uparrow \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ \downarrow & \downarrow & \cdots & \downarrow \end{bmatrix}, Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

, where m
is the number of datapoints.

#1. Follow the steps outlined below:

In [6]:

```
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from collections import defaultdict
plt.rcParams["figure.figsize"] = (8,6)
```

In [7]:

```
# 1. Scale the features between 0 and 1
# To scale, you can directly use the MinMaxScaler from sklearn.
#######
# !!!! YOUR CODE HERE !!!!

scalar = MinMaxScaler()
scalar.fit(X_train)

X_train = scalar.transform(X_train)
X_test = scalar.transform(X_test)

accuracy_tracker = defaultdict(lambda: defaultdict(dict))

# output variable names -  X_train, X_test
#######
```

In [8]:

```
X_train.shape
# y_train
# y_train.shape
```

Out[8]:

```
(1437, 64)
```

In [9]:

```
# 2. One-Hot encode the target labels
# To one-hot encode, you can use the OneHotEncoder from sklearn
#######
# !!!! YOUR CODE HERE !!!!

encoder = OneHotEncoder()

y_train_one =  encoder.fit_transform(y_train.reshape(-1,1)).toarray()
y_test_one =  encoder.fit_transform(y_test.reshape(-1,1)).toarray()

# output variable names -  y_train_one, y_test_one
#######
print('Shape of y_train_one:',y_train_one.shape)
print('Shape of y_test_one:',y_test_one.shape)

# y_test_one[3].toarray()
```

```
Shape of y_train_one: (1437, 10)
Shape of y_test_one: (360, 10)
```

Note: Here we need to define the model prediction. The input matrix is $X_{n \times m}$
where m
is the number of examples, and n
is the number of features. The linear predictions can be given by: $Y = WX + b$
where W
is a $p \times n$
weight matrix and b
is a p
size bias vector. p
is the number of target labels.

#2. Define a function `linear_model` that takes as input a weight matrix (`W`), bias vector (`b`), and input data matrix of size $m \times n$ (`XT`). This function should return the predictions $\hat{y}$.

In [10]:

```
#######
# !!!! YOUR CODE HERE !!!!

def linear_model(W, b, XT):
    return  XT @ W.T + b
#######
```

Note: The loss function that we would be using is the Mean Square Error (L2) Loss: $MSE = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$
, where m
is the number of examples, $\hat{y}^{(i)}$
is the predicted value and $y^{(i)}$
is the ground truth.

#3.Define a function `mse_loss` that takes as input prediction (`y_pred`) and actual labels (`y`), and returns the MSE loss.

In [11]:

```
#######
# !!!! YOUR CODE HERE !!!!

def mse_loss(y_pred, y):
    diff = y - y_pred
    return torch.sum(diff*diff) / len(diff)

#######
```

In the following part, we will do some setup required for training such as initializing weights and biases moving everything to torch tensors.

#4. Define a function: `initializeWeightsAndBiases` that returns tuple `(W, b)`, where `W` is a randomly generated torch tensor of size `num classes x num_features`, and `b` is a randomly generated torch vector of size `num_classes`. For both the tensors, set `requires_grad=True` in parameters.

Move all training and testing data to torch tensors with `dtype=float32`. Remember to set `device=device` in parameters.

In [12]:

```
#######
# !!!! YOUR CODE HERE !!!!

def initializeWeightAndBiases():
    W = torch.rand((num_classes, num_features), requires_grad=True, dtype = torch.float3
2, device=device )
    b = torch.rand(num_classes, requires_grad=True, dtype = torch.float32,  device = dev
ice)
    return W, b


X_train_torch = torch.tensor(X_train, dtype = torch.float32, device = device)
y_train_one_torch = torch.tensor(y_train_one, dtype = torch.float32, device = device)
X_test_torch = torch.tensor(X_test, dtype = torch.float32, device = device)
y_test_one_torch = torch.tensor(y_test_one, dtype = torch.float32, device = device)


# output variable names -  X_train_torch, X_test_torch, y_train_one_torch, y_test_one_tor
ch
#######
```

#5. In this part we will implement the code for training. Given below is a function: `train linear_regression model` that takes as input max number of epochs (`max epochs`), batch size (`batch_size`), Weights (`W`), Biases (`b`), training data (`X_train, y train`), learning rate (`lr`), tolerance for stopping (`tolerance`). It return a tuple `(W,b,losses)` where `W,b` are the trained weigths and biases respectively, and `losses` is a list of tuples of loss logged every 100^{th} epoch.

Complete each of the steps outlines below. You can go through [this] article for reference.

In [13]:

```
# Define a function train_linear_regression_model
def train_linear_regression_model(max_epochs, batch_size, W, b, X_train, y_train, lr, to
lerance):
  losses = []
  prev_loss = float('inf')
```

```
    number_of_batches = math.ceil(len(X_train)/batch_size)

    # optimizer = torch.optim.SGD(params = [W,b], lr=lr)

    for epoch in tqdm(range(max_epochs)):
      for i in range(number_of_batches):
        X_train_batch = X_train[i*batch_size: (i+1)*batch_size]
        y_train_batch = y_train[i*batch_size: (i+1)*batch_size]

        #######
        # !!!! YOUR CODE HERE !!!!
        # 7. do prediction
        y_pred = linear_model(W,b, X_train_batch)

        # 8. get the loss
        loss = mse_loss(y_pred= y_pred, y = y_train_batch)

        # 9. backpropagate loss
        loss.backward()

        # 10. update the weights and biasees
        with torch.no_grad():
          W -= lr*W.grad
          b -= lr*b.grad

          # 11. set the gradients to zero
          W.grad.zero_()
          b.grad.zero_()

        #######

      # log loss every 100th epoch and print every 5000th epoch:
      if epoch%100==0:
        losses.append((epoch, loss.item()))
        if epoch%5000==0:
          print('Epoch: {}, Loss: {}'.format(epoch, loss.item()))

      # break if decrease in loss is less than threshold
      if abs(prev_loss-loss)<=tolerance:
        break
      else:
        prev_loss=loss

    # return updated weights, biases, and logged losses
    return W, b, losses
```

#6. Initialize weights and biases using the `initializeWeightsAndBiases` function that you defined earlier, and train your model using function `train_linear_regression_model` defined above. Use full batch (set `batch_size=len(X_train)` for training (Gradient Descent). Also plot the graph of loss vs number of epochs (Recall that values for learning rate (`lr`) and tolerance (`tolerance`) are already defined above).

In [14]:

```
W, b = initializeWeightAndBiases()
W.shape, b.shape
```

Out[14]:

```
(torch.Size([10, 64]), torch.Size([10]))
```

In [15]:

```
#######
# !!!! YOUR CODE HERE !!!!

start = time.time()
W, b = initializeWeightAndBiases()
W, b, losses = train_linear_regression_model(max_epochs=max_epochs,batch_size=len(X_trai
```

```
n),W = W, b = b, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, toleranc
e=tolerance );
end = time.time()
print(f"Time taken for full gradient descent {end-start}",)
#######
```

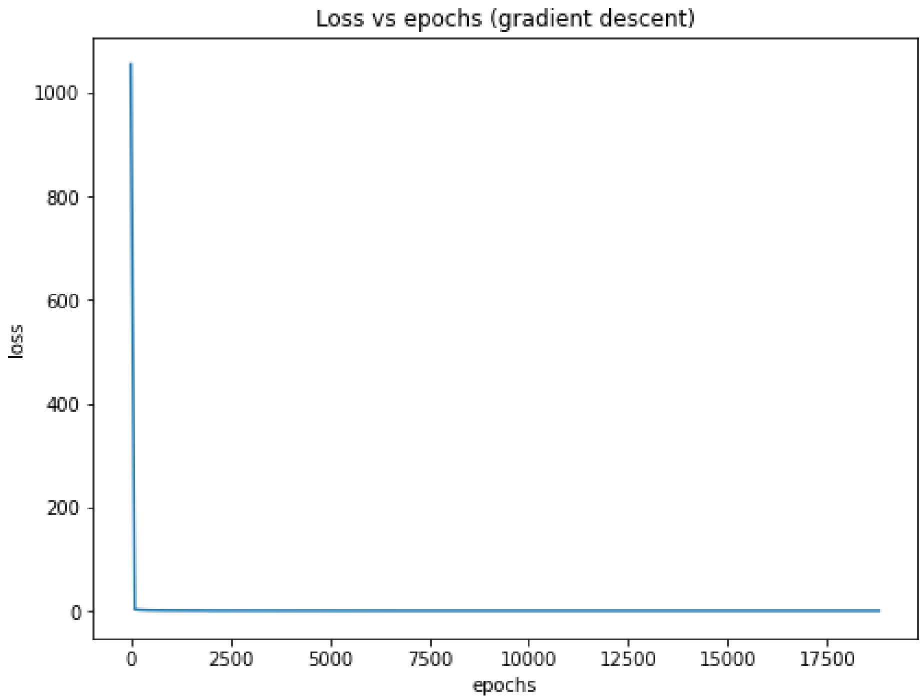
```
Epoch: 0, Loss: 1054.0625
Epoch: 5000, Loss: 0.3990848660469055
Epoch: 10000, Loss: 0.34125944497203827
Epoch: 15000, Loss: 0.32514601945877075
Time taken for full gradient descent 14.340976238250732
```

In [16]:

```
# Plot loss vs epochs

plt.title("Loss vs epochs (gradient descent)")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
# losses
```

Out[16]:

```
[<matplotlib.lines.Line2D at 0x7ffa7a317050>]
```



In [17]:

```
accuracy_tracker["lm_scratch_full_gd"]
```

Out[17]:

```
defaultdict(dict, {})
```

In [18]:

```
# !!DO NOT EDIT!!
# print accuracies of model
predictions_train = linear_model(W,b,X_train_torch).to('cpu')
predictions_test = linear_model(W,b,X_test_torch).to('cpu')
y_train_pred = torch.argmax(predictions_train, dim=1).numpy()

y_test_pred = torch.argmax(predictions_test, dim=1).numpy()
print("Train accuracy:",accuracy_score(y_train_pred, np.asarray(y_train, dtype=np.float3
2)))
print("Test accuracy:",accuracy_score(y_test_pred, np.asarray(y_test, dtype=np.float32))
```

```
)


accuracy_tracker["lm_scratch_full_gd"]["train"] = accuracy_score(y_train_pred, np.asarra
y(y_train, dtype=np.float32))
accuracy_tracker["lm_scratch_full_gd"]["test"] = accuracy_score(y_test_pred, np.asarray(
y_test, dtype=np.float32))
```

```
Train accuracy: 0.9478079331941545
Test accuracy: 0.9361111111111111
```

#7. Now, retrain the above model with `batch_size=64` (Stochastic/Mini-batch Gradient Descent) keeping else everything same. Like before, plot the graph between loss and number of epochs.

In [19]:

```
W, b = initializeWeightAndBiases()

start = time.time()
W, b, losses = train_linear_regression_model(max_epochs=max_epochs,batch_size=64,W = W,
b = b, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, tolerance=toleranc
e );
end = time.time()
print(f"Time taken for mini-batch gradient descent {end-start}",)
```

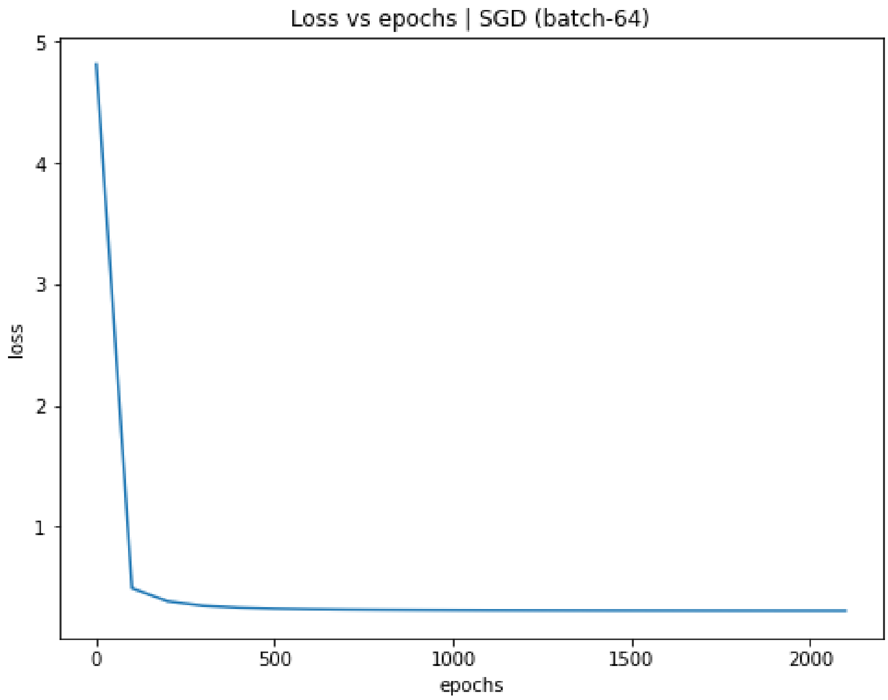
```
Epoch: 0, Loss: 4.815925121307373
Time taken for mini-batch gradient descent 30.465348482131958
```

In [20]:

```
# Plot loss vs epochs

plt.title("Loss vs epochs | SGD (batch-64)")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
# losses
```

Out[20]:

```
[<matplotlib.lines.Line2D at 0x7ffa71b79e50>]
```



In [21]:

```
# !!DO NOT EDIT!!
# print accuracies of model
predictions_train = linear_model(W,b,X_train_torch).to('cpu')
predictions_test = linear_model(W,b,X_test_torch).to('cpu')
y_train_pred = torch.argmax(predictions_train, dim=1).numpy()

y_test_pred = torch.argmax(predictions_test, dim=1).numpy()
print("Train accuracy:",accuracy_score(y_train_pred, np.asarray(y_train, dtype=np.float3
2)))
print("Test accuracy:",accuracy_score(y_test_pred, np.asarray(y_test, dtype=np.float32))
)

accuracy_tracker["lm_scratch_sgd"]["train"] = accuracy_score(y_train_pred, np.asarray(y_
train, dtype=np.float32))
accuracy_tracker["lm_scratch_sgd"]["test"] = accuracy_score(y_test_pred, np.asarray(y_te
st, dtype=np.float32))
```

```
Train accuracy: 0.9464161447459986
Test accuracy: 0.9388888888888889
```

(b) In the previous question, we defined the model, loss, and even the gradient update step. We also had to manully set the grad to zero. In this question, we will re-implement the linear model and see how we can directly use Pytorch to do all this for us in a few simple steps. (6 points)

In [22]:

```
# !! DO NOT EDIT !!
# common utility function to print accuracies
def print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test):
  predictions_train = model(X_train_torch).to('cpu')
  predictions_test = model(X_test_torch).to('cpu')
  y_train_pred = torch.argmax(predictions_train, dim=1).numpy()
  y_test_pred = torch.argmax(predictions_test, dim=1).numpy()

  train_acc = accuracy_score(y_train_pred, np.asarray(y_train, dtype=np.float32))
  test_acc = accuracy_score(y_test_pred, np.asarray(y_test, dtype=np.float32))

  print("Train accuracy:", train_acc)
  print("Test accuracy:",test_acc)

  return train_acc, test_acc
```

In [23]:

```
W, b = initializeWeightAndBiases()
```

#1. Define the linear model using PyTorch

In [24]:

```
#######
# !!!! YOUR CODE HERE !!!!
# Define a model class using torch.nn
class Linear_Model(torch.nn.Module):
  def __init__(self):
    super(Linear_Model, self).__init__()
    # Initalize various layers of model as instructed below
    # 1. initialze one linear layer: num_features -> num_targets
    self.linear = torch.nn.Linear(num_features, num_classes, bias=True)


  def forward(self, X):
    # 2. define the feedforward algorithm of the model and return the final output
```

```
    output =  self.linear(X)
    return output

#######
```

#2. In this part we will implement a general function for training a PyTorch model. Define a general training function: `train torch model` that takes as input an initialized torch model (`model`), batch size (`batch_size`), initialized loss (`criterion`), max number of epochs (`max_epochs`), training data (`X_train, y_train`), learning rate (`lr`), tolerance for stopping (`tolerance`). This function will return a tuple `(model, losses)`, where `model` is the trained model, and `losses` is a list of tuples of loss logged every 100^{th} epoch. Complete each of the steps outlines below. You can go through this article for reference. You can also refer Q3-(d) from HW1.

In [25]:

```
# Define a function train_torch_model
def train_torch_model(model, batch_size, criterion, max_epochs, X_train, y_train, lr, to
lerance):
  losses = []
  prev_loss = float('inf')
  number_of_batches = math.ceil(len(X_train)/batch_size)

  #######
  # !!!! YOUR CODE HERE !!!!
  # 3. move model to device
  model = model.to(device)

  # 4. define optimizer (use torch.optim.SGD (Stochastic Gradient Descent))
  # Set learning rate to lr and also set model parameters
  optimizer = torch.optim.SGD(params = model.parameters(), lr=lr)

  for epoch in tqdm(range(max_epochs)):
    for i in range(number_of_batches):
      X_train_batch = X_train[i*batch_size: (i+1)*batch_size]
      y_train_batch = y_train[i*batch_size: (i+1)*batch_size]

      # 5. reset gradients
      optimizer.zero_grad()

      # 6. prediction
      y_pred = model(X_train_batch)

      # 7. calculate loss
      loss = criterion(y_pred,y_train_batch)

      # 8. backpropagate loss
      loss.backward()

      # 9. perform a single gradient update step
      optimizer.step()

  #######

    # log loss every 100th epoch and print every 5000th epoch:
    if epoch%100==0:
      losses.append((epoch, loss.item()))
      if epoch%5000==0:
        print('Epoch: {}, Loss: {}'.format(epoch, loss.item()))

    # break if decrease in loss is less than threshold
    if abs(prev_loss-loss)<=tolerance:
      break
    else:
      prev_loss=loss

  # return updated model and logged losses
  return model, losses
```

#3. Initialize your model and loss function. Use `nn.MSELoss`. Use full batch for training (Gradient Descent). Also plot the graph of loss vs number of epochs.

In [26]:

```
#######
# !!!! YOUR CODE HERE !!!!
model = Linear_Model()
criterion = torch.nn.MSELoss()

start=time.time()
model, losses = train_torch_model(model, batch_size=len(X_train), criterion = criterion,
max_epochs= max_epochs, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, t
olerance=tolerance )
end=time.time()
print(f"Time taken for full gradient descent (MSE Loss) - {end-start} s")

# losses
#######
```

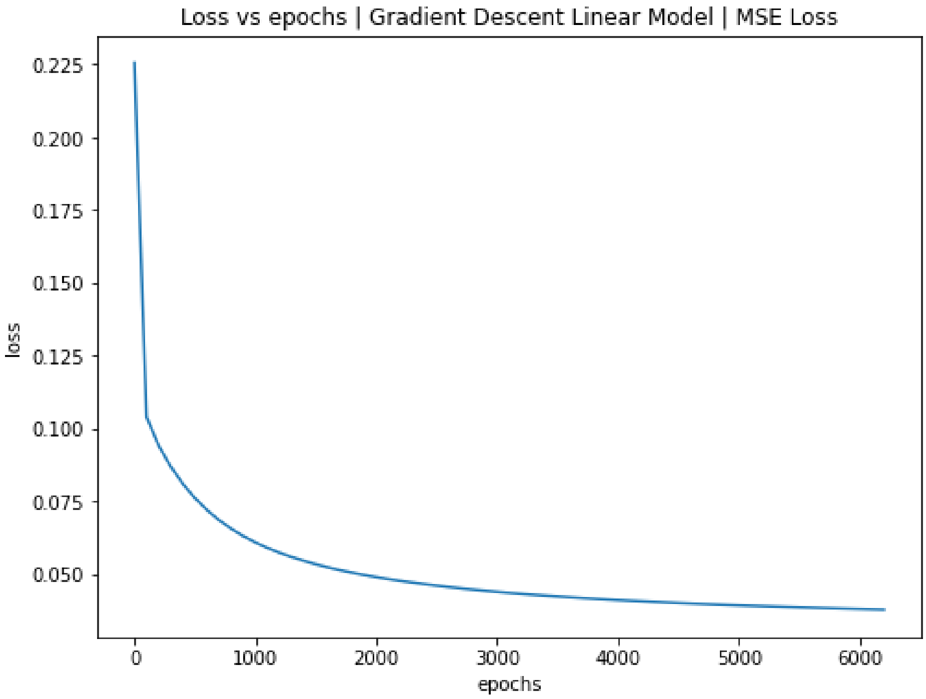
```
Epoch: 0, Loss: 0.22564804553985596
Epoch: 5000, Loss: 0.0392778255045414
Time taken for full gradient descent (MSE Loss) - 4.749436616897583 s
```

In [27]:

```
plt.title("Loss vs epochs | Gradient Descent Linear Model | MSE Loss")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
```

Out[27]:

```
[<matplotlib.lines.Line2D at 0x7ffa71b0d390>]
```



In [28]:

```
# !!DO NOT EDIT!!
# print accuracies of model
train_acc, test_acc = print_accuracies_torch(model, X_train_torch, X_test_torch, y_train
, y_test)

accuracy_tracker["lm_torch_full_gd_mse"]["train"] = train_acc
accuracy_tracker["lm_torch_full_gd_mse"]["test"] = test_acc
```

```
Train accuracy: 0.9262352122477383
Test accuracy: 0.9083333333333333
```

#4. Now, retrain the above model with `batch_size=64` (Stochastic/Mini-batch Gradient Descent) keeping else everything same. Like before, plot the graph between loss and number of epochs.

In [29]:

```
#######
# !!!! YOUR CODE HERE !!!!

model = Linear_Model()
criterion = torch.nn.MSELoss()

start = time.time()
model, losses = train_torch_model(model, batch_size=64, criterion = criterion, max_epoch
s= max_epochs, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, tolerance=
tolerance )
end=time.time()
print(f"Time taken for SGD-64 (MSE Loss) - {end-start} s")

#######
```

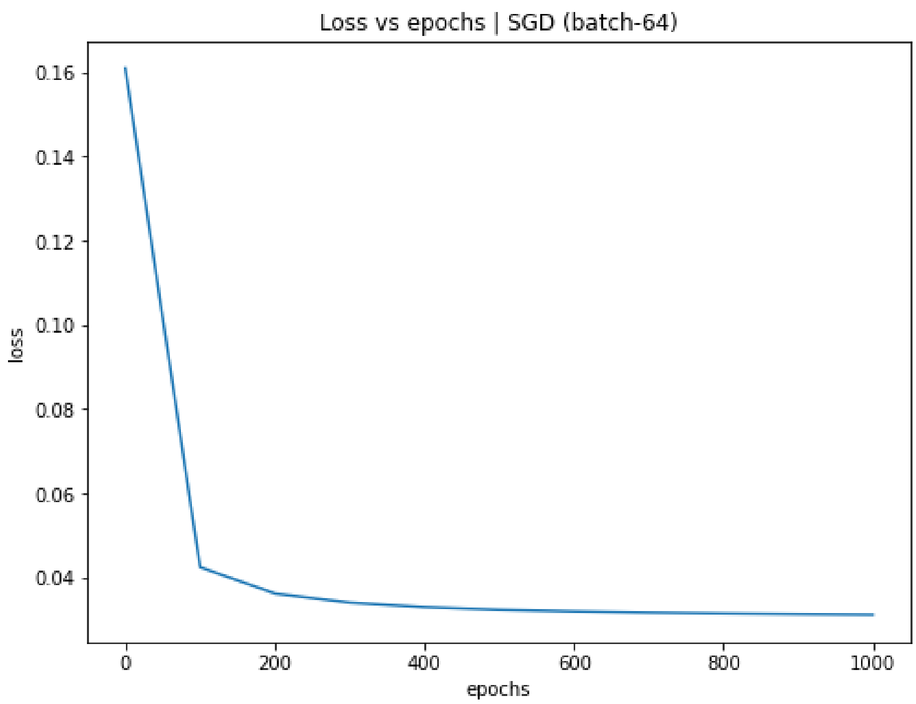
```
Epoch: 0, Loss: 0.1609172224998474
Time taken for SGD-64 (MSE Loss) - 14.869995355606079 s
```

In [30]:

```
plt.title("Loss vs epochs | SGD (batch-64)")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
```

Out[30]:

```
[<matplotlib.lines.Line2D at 0x7ffa71a8bf50>]
```



In [31]:

```
# !!DO NOT EDIT!!
# print accuracies of model
train_acc, test_acc = print_accuracies_torch(model, X_train_torch, X_test_torch, y_train
```

```
, y_test)

accuracy_tracker["lm_torch_full_sgd_mse"]["train"] = train_acc
accuracy_tracker["lm_torch_full_sgd_mse"]["test"] = test_acc
```

```
Train accuracy: 0.941544885177453
Test accuracy: 0.9416666666666667
```

(c) Now, instead of using MSELoss, we will use a much more natural loss function for logistic regression task which is the Cross Entropy Loss. (8 points)

Note: The [Cross Entropy Loss](#) for multiclass calssification is the mean of the negative log likelihood of the output logits after softmax: $L = \underbrace{\underbrace{\frac{1}{m}\sum_{i=1}^{m}\underbrace{-y^{(i)}\underbrace{log\underbrace{\frac{e^{\hat{y}^{(i)}}}{\sum_{j=1}^{p} e^{\hat{y}^{(j)}}}}_{\text{Softmax}}}_{\text{LogSoftmax}}}}_{\text{Negative Log Likelihood (NLL)}}}_{\text{Cross Entropy (CE) Loss}}$

,

where $y^{(i)}$ **is the ground truth, and** $\hat{y}^{(k)}$ **(also called as *logits*) represent the outputs of the last linear layer of the model.**

#1. Instead of `nn.MSELoss`, train the above model with `nn.CrossEntropyLoss`. Use full-batch. Also plot the graph between loss and number of epochs.

In [32]:

```
#######
# !!!! YOUR CODE HERE !!!!
model = Linear_Model()
criterion = torch.nn.CrossEntropyLoss()

start = time.time()
model, losses = train_torch_model(model, batch_size=len(X_train), criterion = criterion,
max_epochs= max_epochs, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, t
olerance=tolerance )
end=time.time()
print(f"Time taken for Full-GD (Cross Entropy Loss) - {end-start} s")
#######
```

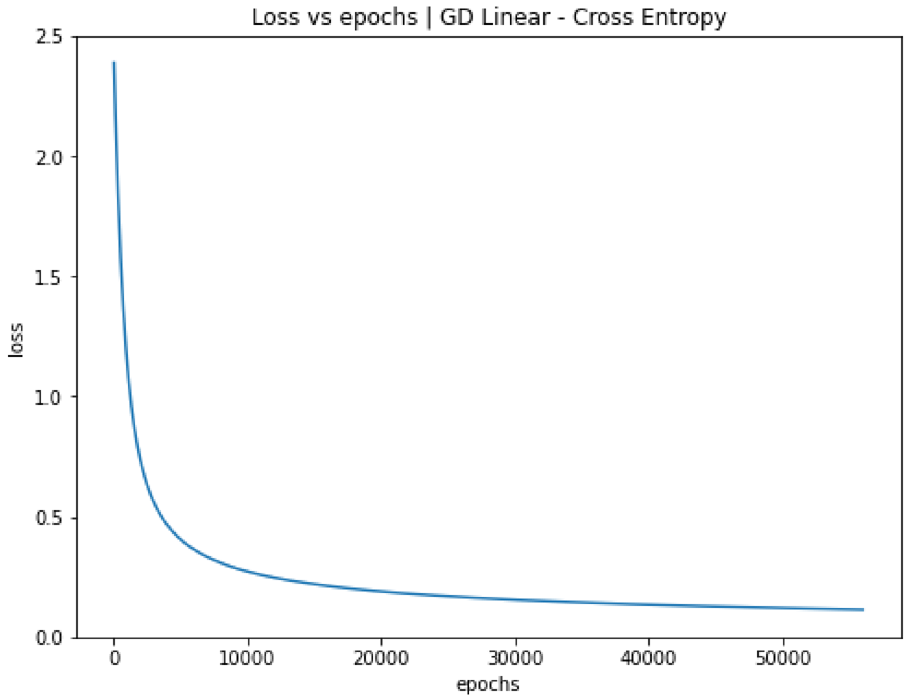
```
Epoch: 0, Loss: 2.388305902481079
Epoch: 5000, Loss: 0.4048900008201599
Epoch: 10000, Loss: 0.27198126912117004
Epoch: 15000, Loss: 0.21928532421588898
Epoch: 20000, Loss: 0.18934257328510284
Epoch: 25000, Loss: 0.16937997937202454
Epoch: 30000, Loss: 0.15479661524295807
Epoch: 35000, Loss: 0.14349493384361267
Epoch: 40000, Loss: 0.13436894118785858
Epoch: 45000, Loss: 0.12677443027496338
Epoch: 50000, Loss: 0.1203075498342514
Epoch: 55000, Loss: 0.1147010400891304
Time taken for Full-GD (Cross Entropy Loss) - 46.67390465736389 s
```

In [33]:

```
plt.title("Loss vs epochs | GD Linear - Cross Entropy")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
```

Out[33]:

```
[<matplotlib.lines.Line2D at 0x7ffa71a1ffd0>]
```



In [34]:

```
# !!DO NOT EDIT!!
# print accuracies of model
train_acc, test_acc = print_accuracies_torch(model, X_train_torch, X_test_torch, y_train
, y_test)
accuracy_tracker["lm_torch_full_gd_ce"]["train"] = train_acc
accuracy_tracker["lm_torch_full_gd_ce"]["test"] = test_acc
```

```
Train accuracy: 0.9798190675017397
Test accuracy: 0.9611111111111111
```

#2. Perform the same task above with `batch_size=64`. Also plot the graph of loss vs epochs.

In [35]:

```
#######
# !!!! YOUR CODE HERE !!!!
model = Linear_Model()
criterion = torch.nn.CrossEntropyLoss()

start = time.time()
model, losses = train_torch_model(model, batch_size=64, criterion = criterion, max_epoch
s= max_epochs, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, tolerance=
tolerance )
end=time.time()
print(f"Time taken for SGD-64 (Cross Entropy Loss) - {end-start} s")
#######
```

```
Epoch: 0, Loss: 2.261371374130249
Epoch: 5000, Loss: 0.08763083070516586
Epoch: 10000, Loss: 0.06120011582970619
Epoch: 15000, Loss: 0.04839373379945755
Epoch: 20000, Loss: 0.040525201708078384
Time taken for SGD-64 (Cross Entropy Loss) - 376.76169872283936 s
```

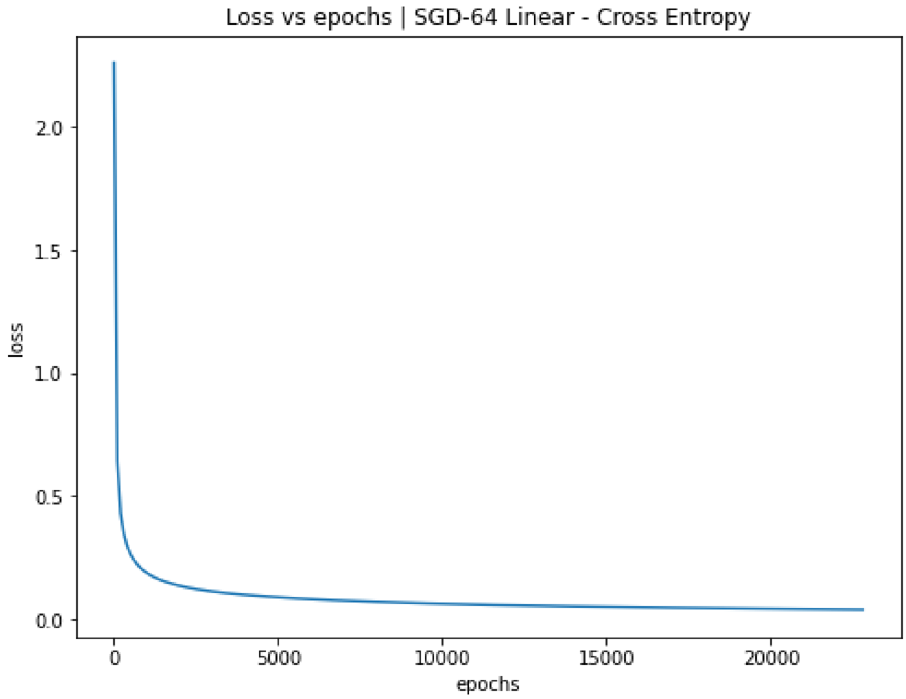
```
Time taken for SGD-64 (Cross Entropy Loss)    376.76169872283936 s
```

In [36]:

```python
plt.title("Loss vs epochs | SGD-64 Linear - Cross Entropy")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
```

Out[36]:

```
[<matplotlib.lines.Line2D at 0x7ffa719374d0>]
```



In [37]:

```python
# !!DO NOT EDIT!!
# print accuracies of model
train_acc, test_acc = print_accuracies_torch(model, X_train_torch, X_test_torch, y_train
, y_test)
accuracy_tracker["lm_torch_full_sgd_ce"]["train"] = train_acc
accuracy_tracker["lm_torch_full_sgd_ce"]["test"] = test_acc
```

```
Train accuracy: 0.9979123173277662
Test accuracy: 0.9694444444444444
```

(d) Now, we will train a neural network in pytorch with two hidden layers of sizes 32 and 16 neurons. We will use non-linear ReLU activations thus effectively making this a non-linear model. We will use this neural network model for multi-class classification with Cross Entropy Loss. (6 points)

Note: The neural network model output can be represented mathematically as below:
$\hat{y}^{(i)}_{10\times 1} = W^{(3)}_{10\times 16}\sigma(W^{(2)}_{16\times 32}\sigma(W^{(1)}_{32\times 64}\mathbf{x}^{(i)}_{64\times 1} + \mathbf{b}^{(1)}_{32\times 1}) + \mathbf{b}^{(2)}_{16\times 1}) + \mathbf{b}^{(3)}_{10\times 1}$
, \ where σ
represents ReLU activation, $W^{(i)}$
is the weight of the i^{th}
linear layer, and b$^{(i)}$
is the layer's bias. We use the subscript to denote the dimension for clarity.

#1. Define the 2-hidden layer neural network model below.

In [38]:

```
#######
# !!!! YOUR CODE HERE !!!!
# Define a neural network model class using torch.nn
class NN_Model(torch.nn.Module):
  def __init__(self):
    super(NN_Model, self).__init__()
    # Initalize various layers of model as instructed below
    # 1. initialize three linear layers: num_features -> 32, 32 -> 16, 16 -> num_targets
    self.linear1 = torch.nn.Linear(num_features, 32, bias=True)
    self.linear2 = torch.nn.Linear(32, 16, bias=True)
    self.linear3 = torch.nn.Linear(16, num_classes, bias=True)
    self.relu = torch.nn.ReLU()

    # 2. initialize RELU

  def forward(self, X):
    # 3. define the feedforward algorithm of the model and return the final output
    # Apply non-linear ReLU activation between subsequent layers
    out1 = self.relu(self.linear1(X))
    out2 = self.relu(self.linear2(out1))
    out3 = self.linear3(out2)
    return out3


#######
```

#2. Train the newly defined Neural Network two hidden layer model with Cross Entropy Loss. Use full-batch and plot the graph of loss vs number of epochs. Note that you can re-use the training function `train_torch_model` (from part (b)).

In [39]:

```
#######
# !!!! YOUR CODE HERE !!!!
model = NN_Model()
criterion = torch.nn.CrossEntropyLoss()

start = time.time()
model, losses = train_torch_model(model, batch_size=len(X_train), criterion = criterion,
max_epochs= max_epochs, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, t
olerance=tolerance )
end = time.time()
print(f"Time taken for NN-Full GD (Cross Entropy Loss) - {end-start} s")

#######
```

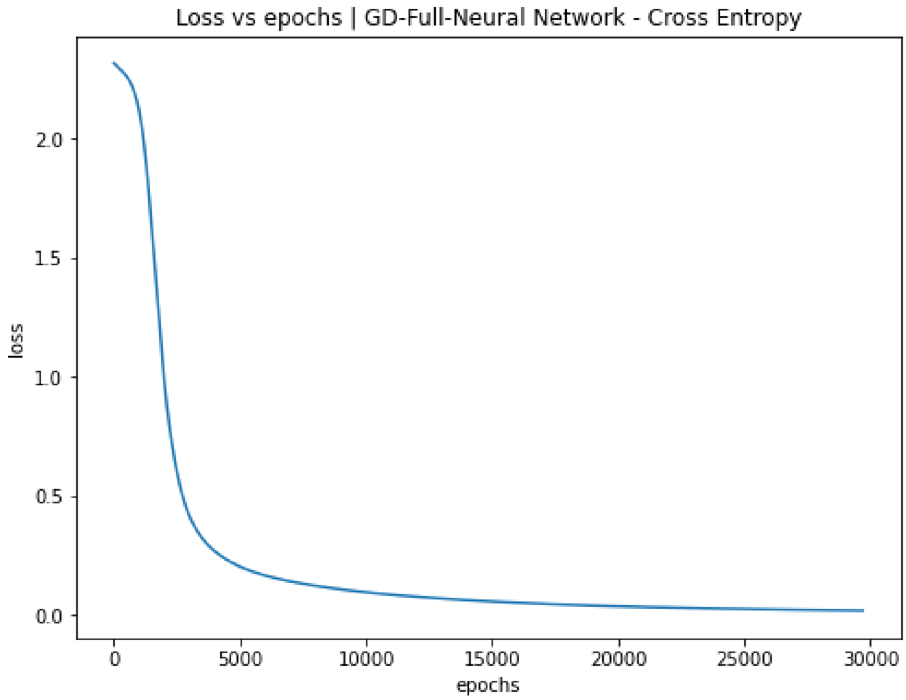
```
Epoch: 0, Loss: 2.316016435623169
Epoch: 5000, Loss: 0.2013154774904251
Epoch: 10000, Loss: 0.09385070949792862
Epoch: 15000, Loss: 0.05498850345611572
Epoch: 20000, Loss: 0.0345892459154129
Epoch: 25000, Loss: 0.023184722289443016
Time taken for NN-Full GD (Cross Entropy Loss) - 42.95395517349243 s
```

In [40]:

```
plt.title("Loss vs epochs | GD-Full-Neural Network - Cross Entropy")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
```

Out[40]:

```
[<matplotlib.lines.Line2D at 0x7ffa7192f4d0>]
```



In [41]:

```
# !!DO NOT EDIT!!
# print accuracies of model
train_acc, test_acc = print_accuracies_torch(model, X_train_torch, X_test_torch, y_train
, y_test)
accuracy_tracker["nn_torch_full_gd_ce"]["train"] = train_acc
accuracy_tracker["nn_torch_full_gd_ce"]["test"] = test_acc
```

```
Train accuracy: 0.9986082115518441
Test accuracy: 0.9638888888888889
```

#3. Re-train the above model with `batch_size=64`. Also plot the graph of loss vs epochs.

In [42]:

```
#######
# !!!! YOUR CODE HERE !!!!
model = NN_Model()
criterion = torch.nn.CrossEntropyLoss()

start = time.time()
model, losses = train_torch_model(model, batch_size=64, criterion = criterion, max_epoch
s= max_epochs, X_train = X_train_torch, y_train = y_train_one_torch, lr = lr, tolerance=
tolerance )
end = time.time()
print(f"Time taken for NN-SGD-64 (Cross Entropy Loss) - {end-start} s")
#######
```

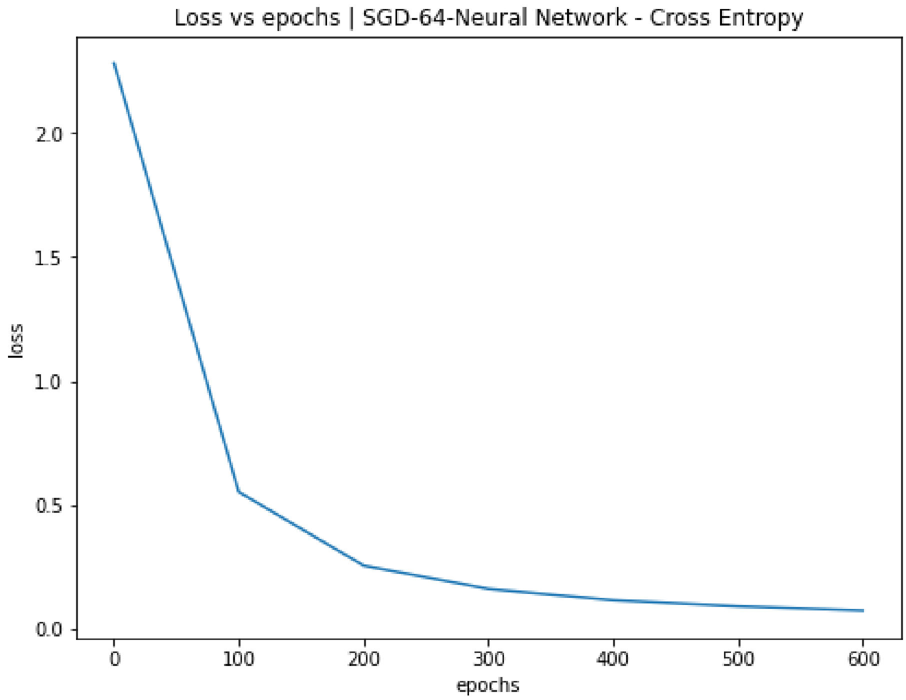
```
Epoch: 0, Loss: 2.2808258533477783
Time taken for NN-SGD-64 (Cross Entropy Loss) - 15.643309831619263 s
```

In [43]:

```
plt.title("Loss vs epochs | SGD-64-Neural Network - Cross Entropy")
plt.ylabel("loss")
plt.xlabel("epochs")

x, y = zip(*losses)
plt.plot(x,y)
```

Out[43]:

```
[<matplotlib.lines.Line2D at 0x7ffa718a7690>]
```



In [44]:

```
# !!DO NOT EDIT!!
# print accuracies of model
train_acc, test_acc = print_accuracies_torch(model, X_train_torch, X_test_torch, y_train
, y_test)
accuracy_tracker["nn_torch_full_sgd_ce"]["train"] = train_acc
accuracy_tracker["nn_torch_full_sgd_ce"]["test"] = test_acc
```

```
Train accuracy: 0.9860821155184412
Test accuracy: 0.9555555555555556
```

(e) In the above few problems, you performed several experiments with different batch size and loss functions. Write down an analysis of your observations from the results. (5 points)

Some points that you could cover are:

- **Effect of using full vs. batch gradient descent.**
- **Effect of different loss strategy on performance.**
- **Effect of using linear vs. non-linear models.**
- **Training time per epoch in different cases.**

Also, plot a line graph of accuracy vs. model for both train and test sets. Recall that you trained the following models in this question:

1. **Linear Model - Scratch + MSE Loss + Full Batch**
2. **Linear Model - Scratch + MSE Loss + Mini Batch**
3. **Linear Model - PyTorch + MSE Loss + Full Batch**
4. **Linear Model - PyTorch + MSE Loss + Mini Batch**
5. **Linear Model - PyTorch + CE Loss + Full Batch**
6. **Linear Model - PyTorch + CE Loss + Mini Batch**
7. **NN Model - PyTorch + CE Loss + Full Batch**
8. **NN Model - PyTorch + CE Loss + Mini Batch**

Your answer here:

Effect of using Full gradient descent vs Batch Gradient Descent

- **We see that batch gradient descent takes more time for converging. However, the accuracy is much higher.**
- **However the time to compute one gradient is much higher in full gradient descent as compared to**

stochastic/batch gradient descent.
- The reason for longer convergence is that once the parameters reach in the confusion region, the stochasticity is much higher.

Effect of different loss strategy on performance.

- We see that time taken while using Cross Entropy loss is much higher as compared to MSE Loss
- However, the accuracy is much higher with Cross Entropy loss as compared to MSE Loss.

Effect of using linear vs. non-linear models.

- We see non-linear models have higher accuracy as compared to linear models.
- The time taken by non-linear models considerably lower as compared to their linear counterparts.

Training time per epoch in different cases.

- Training time per update for full GD is higher as compared to SGD as the number of gradients to be computed are less. Training time per epoch however would be roughly around the same.
- Training time per epoch is much higher when using cross entropy loss as compared to MSE loss.
- Trainig time per epoch in non-linear models is comparatively lower as compared to their linear counterparts.

In [45]:

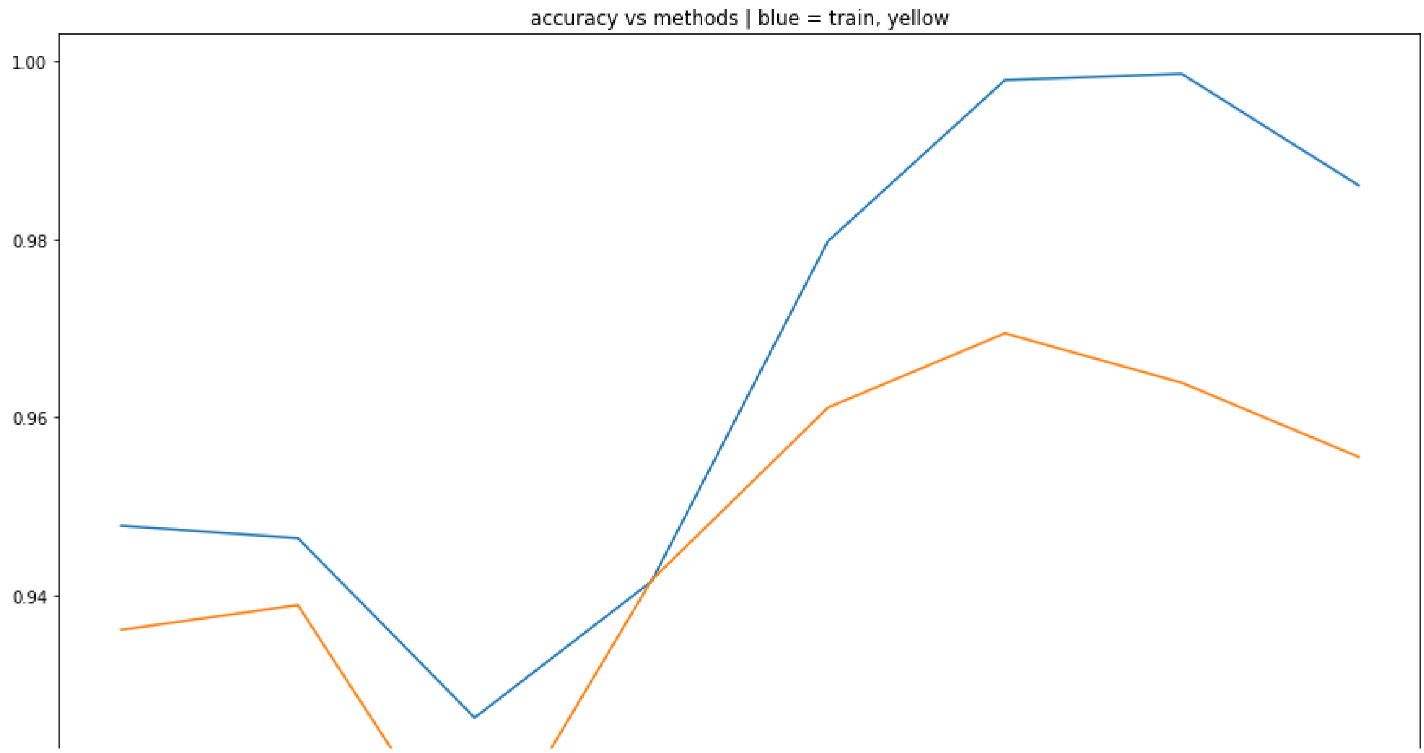
```
train_acc = []
test_acc = []
labels= []
for k,v in accuracy_tracker.items():
  labels.append(k)
  train_acc.append(v["train"])
  test_acc.append(v["test"])
```

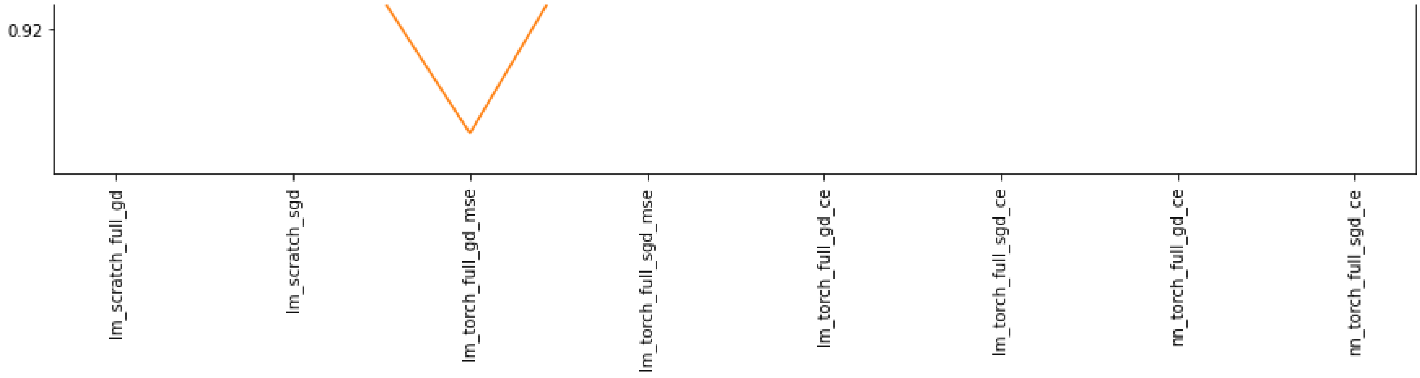
In [46]:

```
plt.rcParams["figure.figsize"] = (15,10)
plt.title("accuracy vs methods | blue = train, yellow = test")
plt.plot(labels, train_acc, label = "train")
plt.plot(labels, test_acc, label = "test")
plt.xticks(rotation=90)
```

Out[46]:

```
([0, 1, 2, 3, 4, 5, 6, 7], <a list of 8 Text major ticklabel objects>)
```





0.92
lm_scratch_full_gd
lm_scratch_sgd
lm_torch_full_gd_mse
lm_torch_full_sgd_mse
lm_torch_full_gd_ce
lm_torch_full_sgd_ce
nn_torch_full_gd_ce
nn_torch_full_sgd_ce

HW2 - Q3: Evaluating Robustness of Neural Networks (35 points)

Keywords: Adversarial Robustness, FGSM/PGD Attack, Certification

About the dataset:
The [MNIST (https://en.wikipedia.org/wiki/MNIST_database)](https://en.wikipedia.org/wiki/MNIST_database) database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.
The MNIST database contains 70,000 labeled images. Each datapoint is a 28×28 pixels grayscale image.
Here we will be starting off with a pre-trained 2-hidden-layer model on the full MNIST dataset.

Agenda:

- In this programming challenge, you will implement adversarial attack on an MNIST neural network model as well as visualize those attacks.
- You will do this by solving the inner maximization problem using FGSM (Fast Gradient Sign Method) and PGD (Projected Gradient Descent).
- You will then perform verification of the model using Interval-Bound -Propagation (IBP).

Note:

- It is important that you use **GPU accelaration** for this Question.
- A note on working with GPU:
    - Take care that whenever declaring new tensors, set `device=device` in parameters.
    - You can also move a declared torch tensor/model to device using `.to(device)` .
    - To move a torch model/tensor to cpu, use `.to('cpu')`
    - Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU).
- Run all the cells in order.
- **Do not edit** the cells marked with !!DO NOT EDIT!!
- Only **add your code** to cells marked with !!!! YOUR CODE HERE !!!!
- Do not change variable names, and use the names which are suggested.

Preprocessing

In [1]:
```
# install this library
!pip install gdown
```

```
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) https://us-python.p
kg.dev/colab-wheels/public/simple/ (https://us-python.pkg.dev/colab-wheels/public/simple/)
Requirement already satisfied: gdown in /usr/local/lib/python3.7/dist-packages (4.4.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from gdown)
(4.64.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages (from gdo
wn) (3.7.0)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from gdown)
(1.15.0)
```

- We will be using a pre-trained 2-hidden layer neural network model (`nn_model`) that takes as input features vectors of size 784, and ouputs logits vector of size 10. Each of the two hidden layers are of size 1024.
- This is a highly accurate model with train accuracy of approx 99.88% and test accuracy of approx 98.14%.
- We will also be loading and initializing a dummy model (`test_model`) for unit testing code implementation.

In [3]:
```python
# !!DO NOT EDIT!!
# imports
import torch
import torch.nn as nn
import numpy as np
import requests
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
import gdown
from zipfile import ZipFile

# set hardware device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Loading the dataset full MNIST dataset
mnist_train = datasets.MNIST("./data", train=True, download=True, transform=transforms.ToTen
mnist_test = datasets.MNIST("./data", train=False, download=True, transform=transforms.ToTen

mnist_train.data = mnist_train.data.to(device)
mnist_test.data = mnist_test.data.to(device)

mnist_train.targets = mnist_train.targets.to(device)
mnist_test.targets = mnist_test.targets.to(device)

# number of target classes
num_classes = 10
num_classes_test = 2

# reshape and min-max scale
X_train =  (mnist_train.data.reshape((mnist_train.data.shape[0], -1))/255).to(device)
y_train = mnist_train.targets
X_test = (mnist_test.data.reshape((mnist_test.data.shape[0], -1))/255).to(device)
y_test = mnist_test.targets


# Load pretrained and dummy model
url_nn_model = 'https://bit.ly/3sKvyOs'
url_models   = 'https://bit.ly/3lsVcDn'
gdown.download(url_nn_model, 'nn_model.pt')
gdown.download(url_models, 'models.zip')
ZipFile("models.zip").extractall("./")

from model import NN_Model
from test_model import Test_Model
nn_model = torch.load("./nn_model.pt").to(device)
print('Pretrained model (nn_model):', nn_model)

test_model = Test_Model()
```

```
print('Dummy model (test_model):', test_model)
```

```
Downloading...
From: https://bit.ly/3sKvyOs (https://bit.ly/3sKvyOs)
To: /content/nn_model.pt
100%|██████████| 7.46M/7.46M [00:00<00:00, 155MB/s]
Downloading...
From: https://bit.ly/3lsVcDn (https://bit.ly/3lsVcDn)
To: /content/models.zip
100%|██████████| 1.55k/1.55k [00:00<00:00, 4.44MB/s]

Pretrained model (nn_model): NN_Model(
  (l1): Linear(in_features=784, out_features=1024, bias=True)
  (l2): Linear(in_features=1024, out_features=1024, bias=True)
  (l3): Linear(in_features=1024, out_features=10, bias=True)
)
Dummy model (test_model): Test_Model(
  (l1): Linear(in_features=2, out_features=3, bias=True)
  (l2): Linear(in_features=3, out_features=3, bias=True)
  (l3): Linear(in_features=3, out_features=2, bias=True)
)
```

In this problem set you need to access the individual layers of the neural network. The below piece of code creates a list of ordered layers for each of the neural network models for easy access.

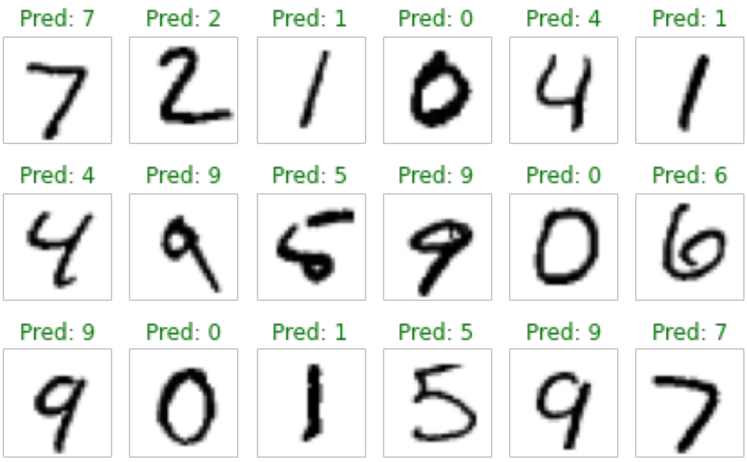
In [4]:

```
# This will save the linear layers of the neural network model in a ordered list
# Eg:
# to access weight of first layer: model_layers[0].weight
# to access bias of first layer: model_layers[0].bias
model_layers = [layer for layer in nn_model.children()] # for nn_model
test_model_layers = [layer for layer in test_model.children()] # for dummy model
```

In [5]:

```
# !!DO NOT EDIT!!
# utility function to plot the images
def plot_images(X,y,yp,M,N):
  f,ax = plt.subplots(M,N, sharex=True, sharey=True, figsize=(N,M*1.3))
  for i in range(M):
    for j in range(N):
      ax[i][j].imshow(1-X[i*N+j].cpu().detach().numpy(), cmap="gray")
      title = ax[i][j].set_title("Pred: {}".format(yp[i*N+j].max(dim=0)[1]))
      plt.setp(title, color=('g' if yp[i*N+j].max(dim=0)[1] == y[i*N+j] else 'r'))
      ax[i][j].set_axis_off()
    plt.tight_layout()
```

In [6]:
```python
# !!DO NOT EDIT!!
# Let us visualize a few test examples
example_data = mnist_test.data[:18]/255
example_data_flattened  = example_data.view((example_data.shape[0], -1)).to(device) # needed
example_labels = mnist_test.targets[:18].to(device)
plot_images(example_data, example_labels, nn_model(example_data_flattened), 3, 6)
```



(a) FGSM attack: In this part you will create a few adversarial examples using FGSM attack. Use an attack budget $\epsilon = 0.05$. (5 points)

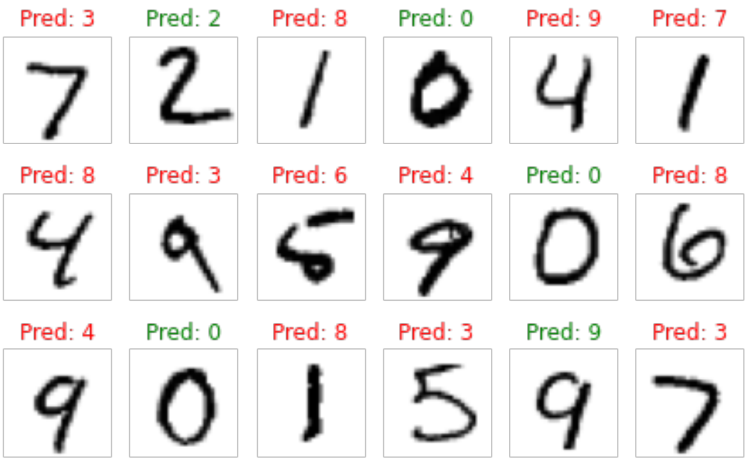
In the Fast Gradient Sign Method (FGSM), the perturbation δ on an input example (e.g. input image) X is given by $\epsilon \cdot sign(g)$, where g is the gradient of the loss function $g := \nabla_\delta \ell(h_\theta(x + \delta), y)$, and ℓ is the loss function, more precisely `nn.CrossEntropyLoss`. In the first timestep, this value of δ is 0.

#1. Define a function `fgsm` which takes as input the neural network model (`model`), test examples (`X`), target labels (`y`), and the attack budget (`epsilon`). Return the value of the perturbation (δ) after one gradient descent step.

In [7]:
```python
#######
# !!! YOUR CODE HERE !!!

def fgsm(model, X, y, epsilon):
    delta = torch.zeros_like(X, requires_grad=True)
    loss = nn.CrossEntropyLoss()(model(X + delta), y)
    loss.backward()
    return epsilon * delta.grad.detach().sign()

#######
```

#2. Now, consider the first few examples from the training dataset which are already defined above as `example_data_flattened` and `example_labels`. Using the function `fgsm`, get the value of `delta` for these examples. Perform prediction on the modified dataset (`example_data_flattened + delta`), and construct a similar plot of images as above. You may reuse the `plot_images` function. Is the attack successful?

In [8]:
```
#######
# !!! YOUR CODE HERE !!!

delta = fgsm(nn_model, example_data_flattened, example_labels, 0.05)
yp = nn_model(example_data_flattened + delta)
plot_images(example_data, example_labels, yp, 3, 6)

#######
```



We can see that the attack was not successful, we had much more incorrect predictions.

(b) PGD attack: In this part you will create a few adversarial examples using PGD attack. Use an attack budget $\epsilon = 0.05$. (10 points)

Note: For the Projected Gradient Descent (PGD) attack, you create an adversarial example by iteratively performing gradient descent with a fixed step size α. The update rule is: $\delta := P(\delta + \alpha\nabla_\delta \ell(h_\theta(x+\delta), y))$, where δ is the perturbation, θ are the frozen DNN parameters, x and y is the training example and its ground truth label respectively, h_θ is the hypothesis function, ℓ denotes the loss function, and P denotes the projection onto a norm ball (l_∞, l_1, l_2, etc.) of interest. For l_∞ ball, this just means clamping the value of δ between $-\epsilon$ and ϵ.

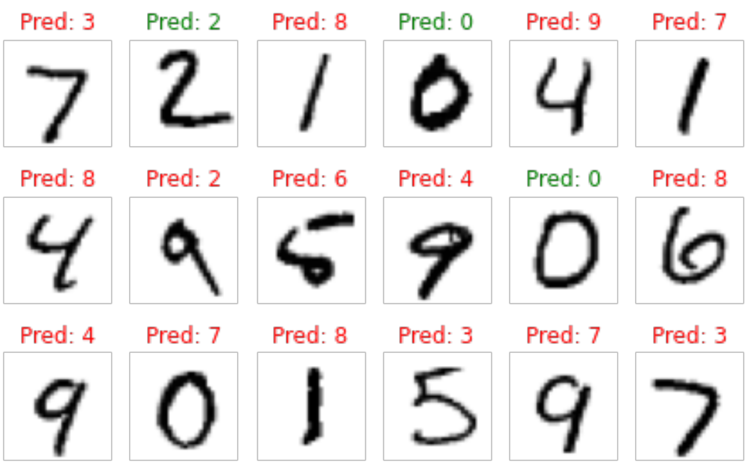
#1. Instead of using FGSM, now use Projected Gradient Descent (PGD) with projection on l_∞ ball for the attack. Define a function `pgd` that takes as input the neural network model (`model`), training examples (`X`), target labels (`y`), step size (`alpha`), attack budget (`epsilon`), and number of iterations (`num_iter`). Return the perturbation (δ) after `num_iter` gradient descent steps.

In [9]:
```
#######
# !!! YOUR CODE HERE !!!

def pgd(model, X, y, epsilon, alpha, num_iter):
    delta = torch.zeros_like(X, requires_grad=True)
    for t in range(num_iter):
        loss = nn.CrossEntropyLoss()(model(X + delta), y)
        loss.backward()
        delta.data = (delta + X.shape[0]*alpha*delta.grad.data).clamp(-epsilon,epsilon)
        delta.grad.zero_()
    return delta.detach()

#######
```

#2. Now use the PGD attack for the examples from `example_data_flattened`. Use `alpha=1000`, `num_iter=1000`, and create a similar plot as before. Is the attack successful?

The value of `alpha` is large because the neural network model is pretrained and is therefore at the local minima. The value of gradients here is extremely small, and we therefore need a huge value of step size to have any hope of moving out of the local minima.

In [10]:
```
#######
# !!! YOUR CODE HERE !!!

delta = pgd(nn_model, example_data_flattened, example_labels, 0.05, 1000, 1000)
yp = nn_model(example_data_flattened + delta)
plot_images(example_data, example_labels, yp, 3, 6)

#######
```



This attack does not seem successful either.

(c) Use FGSM and PGD to create adversarial examples using the complete test dataset. Create the datasets with different values of `epsilon: [0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.12, 0.14, 0.16, 0.18, 0.2]`. For each of the dataset created with different `epsilon` values and attack type, get the model accuracies. Plot a (single) graph of accuracy vs. epsilon for both attack types. Note that `epsilon=0` means no attack, so you can just get accuracy on the original dataset. (10 points)

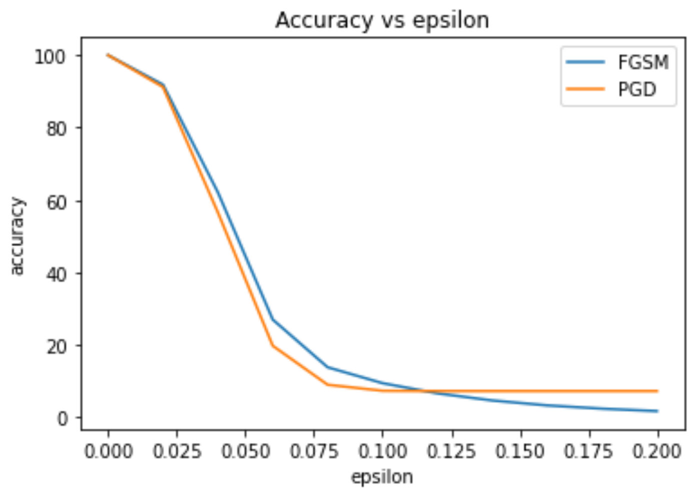
It is important that you use GPU accelaration for this part.

In [11]:
```
#######
# !!! YOUR CODE HERE !!!

epsilon = [0.02*i for i in range(11)]
accuracy_list_fgsm = []
accuracy_list_pgd = []


def get_accuracy(attack_method, X_train, y_train, epsilon):

    if attack_method == "fgsm":
        delta = fgsm(nn_model, X_train, y_train, epsilon=epsilon)
    else:
        delta = pgd(nn_model, X_train, y_train, alpha = 1000, epsilon = epsilon, num_iter=10
```

```
    perturbed_data_flat = X_train + delta
    fgsm_y_pred = nn_model(perturbed_data_flat)
    fgsm_y_pred_labels= torch.argmax(fgsm_y_pred, dim=1)

    accuracy = float(torch.eq(y_train, fgsm_y_pred_labels).sum() / len(y_train) * 100.0)
    return accuracy

n = len(X_train)

for eps in epsilon:
    fgsm_acc = get_accuracy("fgsm", X_train[:n], y_train[:n], epsilon=eps)
    pgd_acc = get_accuracy("pgd", X_train[:n], y_train[:n], epsilon=eps)
    accuracy_list_fgsm.append(fgsm_acc)
    accuracy_list_pgd.append(pgd_acc)
    print(f"{eps} -> {fgsm_acc} - {pgd_acc}")
```

```
0.0 -> 99.97666931152344 - 99.97666931152344
0.02 -> 91.83999633789062 - 91.18000030517578
0.04 -> 62.13333511352539 - 56.58833312988281
0.06 -> 27.0049991607666 - 19.773334503173828
0.08 -> 13.844999313354492 - 9.019999504089355
0.1 -> 9.458333015441895 - 7.360000133514404
0.12 -> 6.644999980926514 - 7.285000324249268
0.14 -> 4.681666374206543 - 7.285000324249268
0.16 -> 3.323333263397217 - 7.285000324249268
0.18 -> 2.4183332920074463 - 7.285000324249268
0.2 -> 1.746666669845581 - 7.285000324249268
```

In [12]:
```
plt.title("Accuracy vs epsilon")
plt.ylabel("accuracy")
plt.xlabel("epsilon")

plt.plot(epsilon,accuracy_list_fgsm, label = "FGSM")
plt.plot(epsilon,accuracy_list_pgd, label = "PGD")
plt.legend()
plt.show()
```



(d) Use the Interval-Bound-Propagation (IBP) technique to certify robustness of the model through lower bound with a given value of epsilon. (10 points)

- In this section, you will find the lower and upper bounds for each neuron of each of the linear layers of the neural network model.
- Note that the initial bound is the bound of the first layer, which is the input example. For the l_∞ perturbation,

the initial lower bound is simply $max(0, x - \epsilon)$, and the initial upper bound is $min(1, x + \epsilon)$, for an input example x (Note that each value of x must lie in between 0 and 1, thats why the min and max).

- In the function, propagate the initial bound across all layers of the neural network and return a list of tuples of *pre-activation* lower and upper bound for each layer. The *pre-activation* bounds are the bound before applying ReLU activation.
- Let's review a bit of the IBP bounds: let $z = Wx + b$ denote an intermediate linear layer of the model, and suppose $\hat{l} \leq x \leq \hat{u}, l \leq z \leq u$, we have:
  $l = W_+\hat{l} + W_-\hat{u} + b$
  $u = W_+\hat{u} + W_-\hat{l} + b$
  Note l, u here are the *pre-activation* bounds
- If a non-linear ReLU activation function $\sigma(\cdot)$ is applied to the layer $z = Wx + b$, then the bounds of $\sigma(z)$ will be: $l = \sigma(\hat{l}), u = \sigma(\hat{u})$ as σ is a monotonically non-decreasing function. I.e. $l \leq \sigma(z) \leq u$. The l, u here are the *post-activation* bounds. Note, here we use $\hat{l}$ and $\hat{u}$ to denote the bounds of the previous layer: $\hat{l} \leq z \leq \hat{u}$.

#1. Define a function `bound_propagation` which takes as input an ordered list of layers of the model (`model_layers`), a feature vector (`x`), and attack budget (`epsilon`). Return a list of tuples of `pre-activation` lower and upper bound tensors for each layer. Verify that your implementation is correct by verifying the results of your function on the unit tests given below.

In [13]:
```
#######
# !!! YOUR CODE HERE !!!

import torch.nn.functional as fn

def bound_propagation(model_layers, x, epsilon):
    initial_bound = ((x - epsilon).clamp(min=0), (x + epsilon).clamp(max=1))
    l, u = initial_bound
    bounds = []
    bounds.append((l, u))
    for i,layer in enumerate(model_layers):
        layer = layer.to(device)
        if isinstance(layer, nn.Linear):
            if i < len(model_layers):
                l = torch.nn.functional.relu(l)
                u = torch.nn.functional.relu(u)

            l_ = (layer.weight.clamp(min=0) @ l.t() + layer.weight.clamp(max=0) @ u.t()
                  + layer.bias[:,None]).t()
            u_ = (layer.weight.clamp(min=0) @ u.t() + layer.weight.clamp(max=0) @ l.t()
                  + layer.bias[:,None]).t()
        elif isinstance(layer, fn.relu):
        # else:
            print("Clamping done")
            l_ = l.clamp(min=0)
            u_ = u.clamp(min=0)

        bounds.append((l_, u_))
        l,u = l_, u_
    return bounds


      #######
```

In [15]:
```
# !!DO NOT EDIT!!
sample_epsilon = 0.2
# unit test - 1
x_1 = torch.tensor([[0.1, 0.9]], device=device)
test_bounds_1 = bound_propagation(test_model_layers, x_1, sample_epsilon)
assert torch.all(torch.eq(torch.round(test_bounds_1[0][0], decimals=2), torch.tensor([[0.000
assert torch.all(torch.eq(torch.round(test_bounds_1[0][1], decimals=2), torch.tensor([[0.300
```

```
assert torch.all(torch.eq(torch.round(test_bounds_1[1][0], decimals=2), torch.tensor([[0.000
assert torch.all(torch.eq(torch.round(test_bounds_1[1][1], decimals=2), torch.tensor([[0.450
assert torch.all(torch.eq(torch.round(test_bounds_1[2][0], decimals=2), torch.tensor([[2.650
assert torch.all(torch.eq(torch.round(test_bounds_1[2][1], decimals=2), torch.tensor([[6.700
assert torch.all(torch.eq(torch.round(test_bounds_1[3][0], decimals=2), torch.tensor([[4.200
assert torch.all(torch.eq(torch.round(test_bounds_1[3][1], decimals=2), torch.tensor([[9.470

# unit test - 2
x_2 = torch.tensor([[0.4, 0.5]], device=device)
test_bounds_2 = bound_propagation(test_model_layers, x_2, sample_epsilon)
assert torch.all(torch.eq(torch.round(test_bounds_2[0][0], decimals=2), torch.tensor([[0.200
assert torch.all(torch.eq(torch.round(test_bounds_2[0][1], decimals=2), torch.tensor([[0.600
assert torch.all(torch.eq(torch.round(test_bounds_2[1][0], decimals=2), torch.tensor([[0.400
assert torch.all(torch.eq(torch.round(test_bounds_2[1][1], decimals=2), torch.tensor([[1.000
assert torch.all(torch.eq(torch.round(test_bounds_2[2][0], decimals=2), torch.tensor([[-0.20
assert torch.all(torch.eq(torch.round(test_bounds_2[2][1], decimals=2), torch.tensor([[5.200
assert torch.all(torch.eq(torch.round(test_bounds_2[3][0], decimals=2), torch.tensor([[0.700
assert torch.all(torch.eq(torch.round(test_bounds_2[3][1], decimals=2), torch.tensor([[7.900
```

#2. Let the lower and upper bounds of the final layer of the model be l^{final} and u^{final} respectively. Then we say that an input example x has a robustness certificate ϵ if the criteria: $l^{final}[c] - u^{final}[i] > 0, \forall i \neq c$, where c denotes the ground truth class of the input x.

- We need to determine the maximum value of epsilon for certified robustness against an adversarial attack for a given example. We can do the same using binary search over a few values of epsilon.
- Define a function `binary_search` that takes as input a sorted array of epsilon values (`epsilons`), an ordered list of neural network model layers (`model_layers`), examples (`X`), corresponding targets (`y`), the number of target classes (`num_classes`). It should return `certified_epsilons` which is a python list of the final values of epsilon certification for each example in input. You can use `None` when unable to find an epsilon value from epsilons.
- Verify that your implementation is correct by verifying the results of your function on the unit tests given below.

In [16]:
```
#######
# !!! YOUR CODE HERE !!!
def _check(lf, uf, y):
    u_js = torch.cat((uf[0][:y], uf[0][y + 1:]))
    if torch.all(lf[0][y] > u_js):
        return True
    else:
        return False

def binary_search(epsilons, model_layers, X, y, num_classes):
    eps_f = []
    for i in range(len(y)):
        x = X[[i]]
        x = torch.reshape(x, (x.shape[0], x.shape[1]))
        j = y[i]
        if_found = False
        low = 0
        high = len(epsilons) - 1

        while(low <= high and not if_found):
            mid = (low + high) // 2
            bounds_mid = bound_propagation(model_layers, x, epsilon=epsilons[mid])
            lf_mid = bounds_mid[-1][0]
            uf_mid = bounds_mid[-1][1]
            bounds_hi = bound_propagation(model_layers, x, epsilon=epsilons[high])
            lf_hi = bounds_hi[-1][0]
            uf_hi = bounds_hi[-1][1]
            if _check(lf_hi, uf_hi, j):
```

```
                if_found = True
            else:
                if _check(lf_mid, uf_mid, j):
                    low = mid + 1
                else:
                    high = mid - 1
        if if_found==True:
            eps_f.append(epsilons[high])
        else:
            eps_f.append(epsilons[mid])

    return eps_f

#######
```

In [17]:

```
# !!DO NOT EDIT!!
epsilons = [x/10000 for x in range(1, 10000)]
# unit test - 1
sample_X = torch.tensor([[0.1, 0.9], [0.4, 0.5]], device=device)
sample_y = torch.tensor([0,0], device=device)
test_epsilons = binary_search(epsilons, test_model_layers, sample_X, sample_y, num_classes_te
assert test_epsilons==[0.0028, 0.0067]
```

#3. Report the certified values of epsilon on the first few examples (simply run the below cell).

In [18]:

```
# !!DO NOT EDIT!!
# finding epsilon for first few examples of MNIST dataset using IBP
epsilons = [x/10000 for x in range(1, 10000)]
X = example_data_flattened[0:2]
y = example_labels[0:2]
binary_search(epsilons, model_layers, X, y, num_classes)
```

Out[18]:

```
[0.0008, 0.0013]
```