HW2 - 25 Points

- **You have to submit two files for this part of the HW**

  (1) ipynb (colab notebook) and
  (2) pdf file (pdf version of the colab file).**

  (3) A separate ppt or pdf file for Task1 (pdf of handwritten calculations)

- **Files should be named as follows**:

  FirstName_LastName_HW_2_Task1 - for Task1** FirstName_LastName_HW_2 - for other Tasks**

Task 1 : Manual Backpropogation (5 Points)

For the network below update the weights using back propogation.

- X1, x2 are inputs. The values of the inputs are provided.
- h1, h2 are hidden neurons. You will need to calculate the values of h1 and h2 in forward pass. -**o1 and o2 are outputs. 0.01 and 0.99 are the true values of the output. You will calculate the predicted values of o1 and o1 in forward pass.**
- W1-W8 are weights. The initial values are provided to you. You will need to calculate the updated values in backward pass.
- bh1, bh2, bo1, bo2 are bias terms. The initial values are provided. You will need to calculate the updated values in backward pass.
- You will apply sigmoid activation on hidden layer.
- You will apply Linear activation function on output neurons.
- You will use the squared error as the loss function. where $ E_1 =1/2 *(\hat{o_1}-o_1)^2 $ $ E_2 =1/2* (\hat{o_2}-o_2)^2 $ $ E = E_1 + E_2$

Here E is the total loss. $\widehat{o_2}$ and $\widehat{o_2}$ are predicted values of o_1 and o_2.

- **Assume a Learning Rate of 10.**

Requirements

- Show caluclations for one forward and one backward pass.
- Show all the steps of your calculations. You will get partial credit for the steps even if the final answers are not accurate.
- You will do this question manually.

- **For this question you can submit - ppt or pdf file (pdf of handwritten calculations).**

```
import math

def sigmoid(x):
  return 1 / (1 + math.exp(-x))

h1 = sigmoid(0*0.1+0.5*0.3+0.1*1)
h2 = sigmoid(0*0.4+0.3*0.3+0.2*1)
print(h1)
print(h2)
```

```
0.5621765008857981
0.5719961329315186
```

```
o1 = 0.2*h1+0.6*h2+0.3*1
o2 = 0.5*h1+0.8*h2+0.4*1
print(o1,o2)
```

```
0.7556329799360708 1.138685156788114
```

```
e1 = 0.5*(o1-0.01)**2
e2 = 0.5*(o2-0.99)**2
E = e1+e2
print(e1,e2,E)
```

```
0.2779842703841725 0.011053637924553023 0.28903790830872556
```

```
del_3_1 = ((o1-0.01)+(o2-0.99)) * 1
del_3_1
```

```
0.8943181367241848
```

```
w5 = 0.2 - (del_3_1*h1)*10
w6 = 0.5 - (del_3_1*h1)*10
w7 = 0.6 - (del_3_1*h2)*10
w8 = 0.8 - (del_3_1*h2)*10
b01 = 0.3 - (del_3_1)*10
b02 = 0.4 - (del_3_1)*10
print(w5,w6,w7,w8,b01,b02)
```

```
-4.82764640782309 -4.52764640782309 -4.515465158167549 -
4.315465158167549 -8.643181367241848 -8.543181367241848
```

```
del_2_1 = h1*(1-h1)*(0.2*(o1-0.01)+0.5*(o2-0.99))
print(del_2_1)
```

```
0.05500338025646259
```

```
del_2_2 = h2*(1-h2)*(0.6*(o1-0.01)+0.8*(o2-0.99))
print(del_2_2)
```

```
0.0990657717129053
```

```
w1 = 0.1 - (del_2_1*0)*10
w2 = 0.4 - (del_2_1*0)*10
w3 = 0.5 - (del_2_2*0.3)*10
w4 = 0.3 - (del_2_2*0.3)*10
bh1 = 0.3 - (del_2_1)*10
bh2 = 0.2 - (del_3_1)*10
print(w1,w2,w3,w4,bh1,bh2)
```

```
0.1 0.4 0.2028026848612841 0.0028026848612840993 -0.2500338025646259 -
8.74318136724185
```

Task 2 - Tensors and Autodiff - 5 Points

```
import torch
import torch.nn as nn
```

Q1 -Calculate Gradients 2 Point

Compute Gradient using PyTorch Autograd - 2 Points

$$f(x,y)=\frac{x+\exp(y)}{\log(x)+(x-y)^3}$$

Compute dx and dy at x=3 and y=4

```
def fxy(x, y):
  # Calculate the numerator: Add x to the exponential of y
  num = x + torch.exp(y)

  # Calculate the denominator: Sum of the logarithm of x and cube of
the difference between x and y
  den = torch.log(x) + (x - y)**3

  # Perform element-wise division of the numerator by the denominator
  return num / den

# Create a single-element tensor 'x' containing the value 3.0
# make sure to set 'requires_grad=True' as you want to compute
gradients with respect to this tensor during backpropagation
x = torch.tensor(3.0, requires_grad=True)

# Create a single-element tensor 'y' containing the value 4.0
# Similar to 'x', we want to compute gradients for 'y' during
backpropagation, hence make sure to set 'requires_grad=True'
y = torch.tensor(4.0, requires_grad=True)

# Call the function 'fxy' with the tensors 'x' and 'y' as arguments
# The result 'f' will also be a tensor and will contain derivative
```

```
information because 'x' and 'y' have 'requires_grad=True'
f = fxy(x, y)
f
```

```
tensor(584.0868, grad_fn=<DivBackward0>)
```

```
# Perform backpropagation to compute the gradients of 'f' with respect
to 'x' and 'y'
# Hint use backward() function on f

# Perform backpropagation to compute the gradients of 'f' with respect
to 'x' and 'y'
f.backward()

# Display the computed gradients of 'f' with respect to 'x' and 'y'
# These gradients are stored as attributes of x and y after the
backward operation
# Print the gradients for x and y
print('x.grad =', x.grad)
print('y.grad =', y.grad)
```

```
x.grad = tensor(-19733.3965)
y.grad = tensor(18322.8477)
```

Q2. Numerical Precision - 3 Points

Given scalars `x` and `y`, implement the following `log_exp` function such that it returns

$$-\log\left(\frac{e^x}{e^x+e^y}\right)$$

.

```
#Question
def log_exp(x, y):
  return -torch.log(1/(1 + (torch.exp(y)/torch.exp(x))))
```

Test your codes with normal inputs:

```
# Create tensors x and y with initial values 2.0 and 3.0, respectively
x, y = torch.tensor([2.0]), torch.tensor([3.0])

# Evaluate the function log_exp() for the given x and y, and store the
output in z
z = log_exp(x, y)

# Display the computed value of z
z
```

```
tensor([1.3133])
```

Now implement a function to compute $\partial z/\partial x$ and $\partial z/\partial y$ with `autograd`

```
def grad(forward_func, x, y):
  # Enable gradient tracking for x and y, set reauires_grad
appropraitely
  x.requires_grad = True
  y.requires_grad = True

  # Evaluate the forward function to get the output 'z'
  z = forward_func(x, y)

  # Perform the backward pass to compute gradients
  # Hint use backward() function on z
  z.backward()

  # Print the gradients for x and y
  print('x.grad =', x.grad)
  print('y.grad =', y.grad)

  # Reset the gradients for x and y to zero for the next iteration
  x.grad = None
  y.grad = None
```

Test your codes, it should print the results nicely.

```
grad(log_exp, x, y)

x.grad = tensor([-0.7311])
y.grad = tensor([0.7311])
```

But now let's try some "hard" inputs

```
x, y = torch.tensor([50.0]), torch.tensor([100.0])

# you may see nan/inf values as output, this is not an error
grad(log_exp, x, y)

x.grad = tensor([nan])
y.grad = tensor([nan])

# you may see nan/inf values as output, this is not an error
torch.exp(torch.tensor([100.0]))

tensor([inf])
```

Does your code return correct results? If not, try to understand the reason. (Hint, evaluate `exp(100)`). Now develop a new function `stable_log_exp` that is identical to `log_exp` in

math, but returns a more numerical stable result. Hint: (1) $\log\left(\frac{x}{y}\right) = log(x) - log(y)$ Hint: (2)

See logsum Trick - https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/

```
def stable_log_exp(x, y):
  max_val = torch.max(x, y)
  result = -x + max_val + torch.log(torch.exp(x - max_val) +
torch.exp(y - max_val))
  return result

log_exp(x, y)
```

```
tensor([inf], grad_fn=<NegBackward0>)
```

```
stable_log_exp(x, y)
```

```
tensor([50.], grad_fn=<AddBackward0>)
```

```
grad(stable_log_exp, x, y)
```

```
x.grad = tensor([-1.])
y.grad = tensor([1.])
```

Task 3 - Linear Regression using Batch Gradient Descent with PyTorch- 5 Points

Regression using Pytroch

Imagine that you're trying to figure out relationship between two variables x and y . You have some idea but you aren't quite sure yet whether the dependence is linear or quadratic.

Your goal is to use least mean squares regression to identify the coefficients for the following three models. The three models are:

1. Quadratic model where $y = b + w_1 \cdot x + w_2 \cdot x^2$.
2. Linear model where $y = b + w_1 \cdot x$.
3. Linear model with no bias where $y = w_1 \cdot x$.

- You will use **Batch gradient descent to estimate the model co-efficients.Batch gradient descent uses complete training data at each iteration.**
- You will implement only training loop (no splitting of data in to training/validation).
- The training loop will have only one `for loop`. We need to iterate over whole data in each epoch. We do not need to create batches.
- You may have to try different values of number of epochs/ learning rate to get good results.

- You should use Pytorch's nn.module and functions.

Data

```
x = torch.tensor([1.5420291, 1.8935232, 2.1603365, 2.5381863,
2.893443, \
                  3.838855, 3.925425, 4.2233696, 4.235571, 4.273397,
\
                  4.9332876, 6.4704757, 6.517571, 6.87826,
7.0009003, \
                  7.035741, 7.278681, 7.7561755, 9.121138,
9.728281])
y = torch.tensor([63.802246, 80.036026, 91.4903, 108.28776,
122.781975, \
                  161.36314, 166.50816, 176.16772, 180.29395,
179.09758, \
                  206.21027, 272.71857, 272.24033, 289.54745,
293.8488, \
                  295.2281, 306.62274, 327.93243, 383.16296,
408.65967])

# Reshape the y tensor to have shape (n, 1), where n is the number of
samples.
# This is done to match the expected input shape for PyTorch's loss
functions.
y = y.view(-1,1)

# Reshape the x tensor to have shape (n, 1), similar to y, for
consistency and to work with matrix operations.
x = x.view(-1,1)

# Compute the square of each element in x.
# This may be used for polynomial features in regression models.
x2 = x * x

# Concatenate the original x tensor and its squared values (x2) along
dimension 1 (columns).
# This creates a new tensor with two features: the original x and x2
(its square) . This can be useful for polynomial regression.
x_combined = torch.cat((x,x2),dim = 1)

print(x_combined.shape, x.shape)
```

```
torch.Size([20, 2]) torch.Size([20, 1])
```

##Loss Function

```
# Initialize Mean Squared Error (MSE) loss function with mean
reduction
# 'reduction="mean"' averages the squared differences between
```

```
predicted and target values
loss_function = torch.nn.MSELoss(reduction='mean')
```

Train Function

```
def train(epochs, x, y, loss_function, log_interval, model,
optimizer):
    """
    Train a PyTorch model using gradient descent.

    Parameters:
    epochs (int): The number of training epochs.
    x (torch.Tensor): The input features.
    y (torch.Tensor): The ground truth labels.
    loss_function (torch.nn.Module): The loss function to be
minimized.
    log_interval (int): The interval at which training information is
logged.
    model (torch.nn.Module): The PyTorch model to be trained.
    optimizer (torch.optim.Optimizer): The optimizer for updating
model parameters.

    Side Effects:
    - Modifies the input model's internal parameters during training.
    - Outputs training log information at specified intervals.
    """


    for epoch in range(epochs):

        # Step 1: Forward pass - Compute predictions based on the
input features
        y_hat = model(x)

        # Step 2: Compute Loss
        loss = loss_function(y_hat,y)

        # Step 3: Zero Gradients - Clear previous gradient information
to prevent accumulation
        optimizer.zero_grad()

        # Step 4: Calculate Gradients - Backpropagate the error to
compute gradients for each parameter
        loss.backward()

        # Step 5: Update Model Parameters - Adjust weights based on
computed gradients
        optimizer.step()

        # Log training information at specified intervals
```

```
        if epoch % log_interval == 0:
            print(f'epoch: {epoch + 1} --> loss {loss.item()}')
```

Part 1

- **For Part 1, use x_combined (we need to use both x and $x^2 $) as input to the model, this means that you have two inputs.**
- Use linear_reg function to specify the model, **think carefully what values the three arguments `(n_ins, n_outs, bias)` will take**..
- In PyTorch, the `nn.Linear` layer initializes its weights using Kaiming initialization by default, which is well-suited for ReLU activation functions. The bias terms are initialized to zero.
- In this assignment you will use `nn.init` functions like `nn.init.normal_` and `nn.init.zeros_`, to explicitly override these default initializations to use your specified methods.

Run the cell below twice

In the first attempt

- Use LEARNING_RATE = 0.05 What do you observe?

Write your observations HERE:

In the second attempt

- Now use a LEARNING_RATE = 0.0005, What do you observe?

Write your observations HERE:

This is due to exploding gradients. This can happen when the weights of your model become too large or too small. When the weights are too large, the gradients can become very large, which can cause the loss to explode. When the weights are too small, the gradients can become very small, which can cause the loss to vanish.

```
# model 1
LEARNING_RATE = 0.05
EPOCHS = 100000
LOG_INTERVAL= 10000

# Use PyTorch's nn.Linear to create the model for your task.
# Based on your understanding of the problem at hand, decide how you
will initialize the nn.Linear layer.
# Take into consideration the number of input features, the number of
output features, and whether or not to include a bias term.
num_input_features = x_combined.shape[-1]
model = torch.nn.Linear(in_features=num_input_features,
out_features=1, bias=True)

# Initialize the weights of the model using a normal distribution with
```

```
mean = 0 and std = 0.01
# Hint: To initialize the model's weights, you can use the
nn.init.normal_() function.
# You will need to provide the 'model.weight' tensor and specify
values for the 'mean' and 'std' arguments.
torch.nn.init.normal_(model.weight,mean=0,std=0.01)


# Initialize the model's bias terms to zero
# Hint: To set the model's bias terms to zero, consider using the
nn.init.zeros_() function.
# You'll need to supply 'model.bias' as an argument.
torch.nn.init.zeros_(model.bias)

# Create an SGD (Stochastic Gradient Descent) optimizer using the
model's parameters and a predefined learning rate
optimizer = torch.optim.SGD(model.parameters(),LEARNING_RATE)

# Start the training process for the model with specified parameters
and settings
train(EPOCHS, x_combined, y, loss_function, LOG_INTERVAL, model,
optimizer)
```

```
epoch: 1 --> loss 58098.30078125
epoch: 10001 --> loss nan
epoch: 20001 --> loss nan
epoch: 30001 --> loss nan
epoch: 40001 --> loss nan
epoch: 50001 --> loss nan
epoch: 60001 --> loss nan
epoch: 70001 --> loss nan
epoch: 80001 --> loss nan
epoch: 90001 --> loss nan
```

```
# model 1
LEARNING_RATE = 0.0005
EPOCHS = 100000
LOG_INTERVAL= 10000

# Use PyTorch's nn.Linear to create the model for your task.
# Based on your understanding of the problem at hand, decide how you
will initialize the nn.Linear layer.
# Take into consideration the number of input features, the number of
output features, and whether or not to include a bias term.
num_input_features = x_combined.shape[-1]
model = torch.nn.Linear(in_features=num_input_features,
out_features=1, bias=True)

# Initialize the weights of the model using a normal distribution with
mean = 0 and std = 0.01
```

```
# Hint: To initialize the model's weights, you can use the
nn.init.normal_() function.
# You will need to provide the 'model.weight' tensor and specify
values for the 'mean' and 'std' arguments.
torch.nn.init.normal_(model.weight,mean=0,std=0.01)


# Initialize the model's bias terms to zero
# Hint: To set the model's bias terms to zero, consider using the
nn.init.zeros_() function.
# You'll need to supply 'model.bias' as an argument.
torch.nn.init.zeros_(model.bias)

# Create an SGD (Stochastic Gradient Descent) optimizer using the
model's parameters and a predefined learning rate
optimizer = torch.optim.SGD(model.parameters(),LEARNING_RATE)


# Start the training process for the model with specified parameters
and settings
train(EPOCHS, x_combined, y, loss_function, LOG_INTERVAL, model,
optimizer)
```

```
epoch: 1 --> loss 57737.01171875
epoch: 10001 --> loss 5.00191593170166
epoch: 20001 --> loss 3.0944876670837402
epoch: 30001 --> loss 2.137355327606201
epoch: 40001 --> loss 1.6570346355438232
epoch: 50001 --> loss 1.4160265922546387
epoch: 60001 --> loss 1.2949260473251343
epoch: 70001 --> loss 1.2341095209121704
epoch: 80001 --> loss 1.2035775184631348
epoch: 90001 --> loss 1.1882011890411377
```

```
print(f' Weights {model.weight.data}, \nBias: {model.bias.data}')
```

```
 Weights tensor([[4.1796e+01, 1.4828e-02]]),
Bias: tensor([0.9773])
```

Part 2

- **For Part 1, use x as input to the model, this means that you have only one input.**
- Use linear_reg function to specify the model, **think carefully what values the three arguments `(n_ins, n_outs, bias)` will take**..

```
# model 2
LEARNING_RATE = 0.01
EPOCHS = 1000
LOG_INTERVAL= 10

# Use PyTorch's nn.Linear to create the model for your task.
```

```
# Based on your understanding of the problem at hand, decide how you
will initialize the nn.Linear layer.
# Take into consideration the number of input features, the number of
output features, and whether or not to include a bias term.
n_input = x.shape[-1]
model = torch.nn.Linear(in_features=n_input, out_features=1,
bias=True)

# Initialize the weights of the model using a normal distribution with
mean = 0 and std = 0.01
# Hint: To initialize the model's weights, you can use the
nn.init.normal_() function.
# You will need to provide the 'model.weight' tensor and specify
values for the 'mean' and 'std' arguments.
torch.nn.init.normal_(model.weight,mean=0,std=0.01)


# Initialize the model's bias terms to zero
# Hint: To set the model's bias terms to zero, consider using the
nn.init.zeros_() function.
# You'll need to supply 'model.bias' as an argument.
torch.nn.init.zeros_(model.bias)

# Create an SGD (Stochastic Gradient Descent) optimizer using the
model's parameters and a predefined learning rate
optimizer = torch.optim.SGD(model.parameters(),LEARNING_RATE)


# Start the training process for the model with specified parameters
and settings
# Note that we are passing x as an input for this part
train(EPOCHS, x, y, loss_function, LOG_INTERVAL, model, optimizer)
```

```
epoch: 1 --> loss 57931.89453125
epoch: 11 --> loss 6.970487117767334
epoch: 21 --> loss 6.596597194671631
epoch: 31 --> loss 6.246827602386475
epoch: 41 --> loss 5.919613838195801
epoch: 51 --> loss 5.61353063583374
epoch: 61 --> loss 5.327154636383057
epoch: 71 --> loss 5.059247016906738
epoch: 81 --> loss 4.808634281158447
epoch: 91 --> loss 4.574170112609863
epoch: 101 --> loss 4.354818820953369
epoch: 111 --> loss 4.149626731872559
epoch: 121 --> loss 3.9576339721679688
epoch: 131 --> loss 3.7780513763427734
epoch: 141 --> loss 3.610041379928589
epoch: 151 --> loss 3.452868700027466
epoch: 161 --> loss 3.305838108062744
```

```
epoch: 171 --> loss 3.168278217315674
epoch: 181 --> loss 3.039578676223755
epoch: 191 --> loss 2.919196844100952
epoch: 201 --> loss 2.8065707683563232
epoch: 211 --> loss 2.7012176513671875
epoch: 221 --> loss 2.6026482582092285
epoch: 231 --> loss 2.5104150772094727
epoch: 241 --> loss 2.4241530895233154
epoch: 251 --> loss 2.343452215194702
epoch: 261 --> loss 2.2679524421691895
epoch: 271 --> loss 2.197324275970459
epoch: 281 --> loss 2.131232500076294
epoch: 291 --> loss 2.0694222450256348
epoch: 301 --> loss 2.0115981101989746
epoch: 311 --> loss 1.9575061798095703
epoch: 321 --> loss 1.9068794250488281
epoch: 331 --> loss 1.8595447540283203
epoch: 341 --> loss 1.8152507543563843
epoch: 351 --> loss 1.7738040685653687
epoch: 361 --> loss 1.7350494861602783
epoch: 371 --> loss 1.6987781524658203
epoch: 381 --> loss 1.66484797000885
epoch: 391 --> loss 1.6331058740615845
epoch: 401 --> loss 1.6034133434295654
epoch: 411 --> loss 1.5756399631500244
epoch: 421 --> loss 1.5496450662612915
epoch: 431 --> loss 1.525335669517517
epoch: 441 --> loss 1.502586007118225
epoch: 451 --> loss 1.4813083410263062
epoch: 461 --> loss 1.461414098739624
epoch: 471 --> loss 1.442789912223816
epoch: 481 --> loss 1.425371766090393
epoch: 491 --> loss 1.4090648889541626
epoch: 501 --> loss 1.3938194513320923
epoch: 511 --> loss 1.3795570135116577
epoch: 521 --> loss 1.3662182092666626
epoch: 531 --> loss 1.3537347316741943
epoch: 541 --> loss 1.3420519828796387
epoch: 551 --> loss 1.33112633228302
epoch: 561 --> loss 1.3209125995635986
epoch: 571 --> loss 1.3113502264022827
epoch: 581 --> loss 1.3024013042449951
epoch: 591 --> loss 1.2940305471420288
epoch: 601 --> loss 1.2862099409103394
epoch: 611 --> loss 1.2788809537887573
epoch: 621 --> loss 1.2720329761505127
epoch: 631 --> loss 1.2656220197677612
epoch: 641 --> loss 1.2596309185028076
epoch: 651 --> loss 1.25401771068573
```

```
epoch: 661 --> loss 1.248769998550415
epoch: 671 --> loss 1.2438604831695557
epoch: 681 --> loss 1.2392663955688477
epoch: 691 --> loss 1.2349704504013062
epoch: 701 --> loss 1.2309538125991821
epoch: 711 --> loss 1.2271960973739624
epoch: 721 --> loss 1.2236711978912354
epoch: 731 --> loss 1.2203751802444458
epoch: 741 --> loss 1.2173010110855103
epoch: 751 --> loss 1.2144218683242798
epoch: 761 --> loss 1.211728811264038
epoch: 771 --> loss 1.2092041969299316
epoch: 781 --> loss 1.2068431377410889
epoch: 791 --> loss 1.2046411037445068
epoch: 801 --> loss 1.202578067779541
epoch: 811 --> loss 1.2006399631500244
epoch: 821 --> loss 1.198833703994751
epoch: 831 --> loss 1.1971498727798462
epoch: 841 --> loss 1.1955734491348267
epoch: 851 --> loss 1.1940834522247314
epoch: 861 --> loss 1.192704677581787
epoch: 871 --> loss 1.1914142370224
epoch: 881 --> loss 1.1901978254318237
epoch: 891 --> loss 1.1890780925750732
epoch: 901 --> loss 1.1880154609680176
epoch: 911 --> loss 1.1870167255401611
epoch: 921 --> loss 1.186085820198059
epoch: 931 --> loss 1.185221791267395
epoch: 941 --> loss 1.1844091415405273
epoch: 951 --> loss 1.1836483478546143
epoch: 961 --> loss 1.18294358253479
epoch: 971 --> loss 1.182273268699646
epoch: 981 --> loss 1.181653380393982
epoch: 991 --> loss 1.1810722351074219
```

```
print(f' Weights {model.weight.data}, \nBias: {model.bias.data}')
```

```
 Weights tensor([[41.9377]]),
Bias: tensor([0.7467])
```

Part 3

- **Part 3 is similar to part 2, the only difference is that model has no bias term now.**
- **You will see that we are now running the model for only ten epochs and will get similar results**

```
# model 3
LEARNING_RATE = 0.01
EPOCHS = 10
LOG_INTERVAL= 1
```

```python
# Use PyTorch's nn.Linear to create the model for your task.
# Based on your understanding of the problem at hand, decide how you
will initialize the nn.Linear layer.
# Take into consideration the number of input features, the number of
output features, and whether or not to include a bias term.
n_input = x.shape[-1]
model = torch.nn.Linear(in_features=n_input, out_features=1,
bias=False)

# Initialize the weights of the model using a normal distribution with
mean = 0 and std = 0.01
# Hint: To initialize the model's weights, you can use the
nn.init.normal_() function.
# You will need to provide the 'model.weight' tensor and specify
values for the 'mean' and 'std' arguments.
torch.nn.init.normal_(model.weight,mean=0,std=0.01)


# We do not need to initilaize the bias term as there is no bias term
in this model

# Create an SGD (Stochastic Gradient Descent) optimizer using the
model's parameters and a predefined learning rate
optimizer = torch.optim.SGD(model.parameters(),LEARNING_RATE)


# Start the training process for the model with specified parameters
and settings
# Note that we are passing x as an input for this part
train(EPOCHS, x, y, loss_function, LOG_INTERVAL, model, optimizer)
```

```
epoch: 1 --> loss 57932.1953125
epoch: 2 --> loss 6892.0888671875
epoch: 3 --> loss 820.8875122070312
epoch: 4 --> loss 98.71975708007812
epoch: 5 --> loss 12.81834602355957
epoch: 6 --> loss 2.600346088409424
epoch: 7 --> loss 1.3849307298660278
epoch: 8 --> loss 1.2403569221496582
epoch: 9 --> loss 1.2231651544570923
epoch: 10 --> loss 1.2211220264434814
```

```python
print(f' Weights {model.weight.data}')
```

```
 Weights tensor([[42.0557]])
```

Task 4 - MultiClass Classification using Mini Batch Gradient Descent with PyTorch- 5 Points

Data

```
# Import the make_classification function from the sklearn.datasets module
# This function is used to generate a synthetic dataset for classification tasks.
from sklearn.datasets import make_classification

# Import the StandardScaler class from the sklearn.preprocessing module
# StandardScaler is used to standardize the features by removing the mean and scaling to unit variance.
from sklearn.preprocessing import StandardScaler

# Import the main PyTorch library, which provides the essential building blocks for constructing neural networks.
import torch

# Import the 'optim' module from PyTorch for various optimization algorithms like SGD, Adam, etc.
import torch.optim as optim

# Import the 'nn' module from PyTorch, which contains pre-defined layers, loss functions, etc., for neural networks.
import torch.nn as nn

# Import the 'functional' module from PyTorch; incorrect import here, it should be 'import torch.nn.functional as F'
# This module contains functional forms of layers, loss functions, and other operations.
import torch.functional as F  # Should be 'import torch.nn.functional as F'

# Import DataLoader and Dataset classes from PyTorch's utility library.
# DataLoader helps with batching, shuffling, and loading data in parallel.
# Dataset provides an abstract interface for easier data manipulation.
from torch.utils.data import DataLoader, Dataset

# Generate a synthetic dataset for classification using make_classification function.
# Parameters:
# - n_samples=1000: The total number of samples in the generated
```

```
dataset.
# - n_features=5: The total number of features for each sample.
# - n_classes=3: The number of classes for the classification task.
# - n_informative=4: The number of informative features, i.e.,
features that are actually useful for classification.
# - n_redundant=1: The number of redundant features, i.e., features
that can be linearly derived from informative features.
# - random_state=0: The seed for the random number generator to ensure
reproducibility.

X, y = make_classification(n_samples=1000, n_features=5, n_classes=3,
n_informative=4, n_redundant=1, random_state=0)
```

In this example, you're using `make_classification` to **generate a dataset with 1,000 samples, 5 features per sample, and 3 classes for the classification problem**. Of the 5 features, 4 are informative (useful for classification), and 1 is redundant (can be derived from the informative features). The `random_state` parameter ensures that the data generation is reproducible.

```
# Initialize the StandardScaler object from the sklearn.preprocessing
module.
# This will be used to standardize the features of the dataset.
preprocessor = StandardScaler()

# Fit the StandardScaler on the dataset (X) and then transform it.
# The fit_transform() method computes the mean and standard deviation
of each feature,
# and then standardizes the features by subtracting the mean and
dividing by the standard deviation.
X = preprocessor.fit_transform(X)

print(X.shape, y.shape)
```

```
(1000, 5) (1000,)
```

```
import numpy as np
np.unique(y)
```

```
array([0, 1, 2])
```

```
X[0:5]
```

```
array([[-0.39443436, -0.78033571, -0.25005511,  0.09118536, -0.5690698
],
       [ 0.64284479, -0.95837057,  0.83598996, -0.08438568,
0.50539358],
       [ 0.99102498,  0.8580679 ,  0.78786062, -0.9114329 ,
1.62615938],
       [-0.96923966,  0.86168226, -1.31837608, -1.22844863, -
0.07591589],
```

```
       [ 0.96021518,  0.99206623,  1.0026402 , -0.25339161,
1.18831784]])

print(y[0:10])

[2 0 1 2 1 1 0 2 0 0]
```

Dataset and Data Loaders

```
# Convert the numpy arrays X and y to PyTorch Tensors.
# For X, we create a floating-point tensor since most PyTorch models
expect float inputs for features.
# This is a  multiclass classification problem.

# ================================
# IMPORTANT: # Consider what cost function you will use and whether it
expects the label tensor (y)  to be float or long type.
# ================================

x_tensor = torch.tensor(X)
y_tensor = torch.tensor(y)

# Define a custom PyTorch Dataset class for handling our data
class MyDataset(Dataset):
    # Constructor: Initialize the dataset with features and labels
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    # Method to return the length of the dataset
    def __len__(self):
        return self.labels.shape[0]

    # Method to get a data point by index
    def __getitem__(self, index):
        x = self.features[index]
        y = self.labels[index]
        return x, y

# Create an instance of the custom MyDataset class, passing in the
feature and label tensors.
# This will allow the data to be used with PyTorch's DataLoader for
efficient batch processing.
train_dataset = MyDataset(x_tensor, y_tensor)

# Access the first element (feature-label pair) from the train_dataset
using indexing.
# The __getitem__ method of MyDataset class will be called to return
this element.
train_dataset[0]
```

```
(tensor([-0.3944, -0.7803, -0.2501,  0.0912, -0.5691],
dtype=torch.float64),
 tensor(2))

# Create Data loader from Dataset
# Use a batch size of 16
# Use shuffle = True
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=16, shuffle=True)
```

Model

```
# Student Task: Define your neural network model for multi-class
classification.
# Think through what layers you should add. Note: Your task is to
create a model that uses Softmax for
# classification but doesn't include any hidden layers.
# You can use nn.Linear or nn.Sequential for this task
model = nn.Linear(in_features=5, out_features=3)
model = nn.Sequential(model, nn.Softmax(dim=1))
```

Loss Function

```
# Student Task: Specify the loss function for your model.
# Consider the architecture of your model, especially the last layer,
when choosing the loss function.
# Reminder: The last layer in the previous step should guide your
choice for an appropriate loss function for multi-class
classification.

loss_function = torch.nn.CrossEntropyLoss()
```

Initialization

Create a function to initilaize weights.

- Initialize weights using normal distribution with mean = 0 and std = 0.05
- Initilaize the bias term with zeros

```
# Function to initialize the weights and biases of a neural network
layer.
# This function specifically targets layers of type nn.Linear.
def init_weights(layer):
  # Check if the layer is of the type nn.Linear.
  if type(layer) == nn.Linear:
    # Initialize the weights with a normal distribution, centered at 0
with a standard deviation of 0.05.
    torch.nn.init.normal_(layer.weight, mean=0, std=0.05)
```

```
    # Initialize the bias terms to zero.
    torch.nn.init.zeros_(layer.bias)
```

Training Loop

Model Training involves five steps:

- Step 0: Randomly initialize parameters / weights
- Step 1: Compute model's predictions - forward pass
- Step 2: Compute loss
- Step 3: Compute the gradients
- Step 4: Update the parameters
- Step 5: Repeat steps 1 - 4

Model training is repeating this process over and over, for many **epochs**.

We will specify number of ***epochs*** and during each epoch we will iterate over the complete dataset and will keep on updating the parameters.

Learning rate and ***epochs*** are known as hyperparameters. We have to adjust the values of these two based on validation dataset.

We will now create functions for step 1 to 4.

```
# Function to train a neural network model.
# Arguments include the number of epochs, loss function, learning
rate, model architecture, and optimizer.

def train(epochs, loss_function, learning_rate, model, optimizer):

  # Loop through each epoch
  for epoch in range(epochs):

    # Initialize variables to hold aggregated training loss and
correct prediction count for each epoch
    running_train_loss = 0
    running_train_correct = 0

    # Loop through each batch in the training dataset using
train_loader
    for x, y in train_loader:

      # Move input and target tensors to the device (GPU or CPU)
      device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
      x = x.to(device,dtype=torch.float32)
      targets = y.to(device,dtype=torch.long)
      # print(x.dtype)

      # Step 1: Forward Pass: Compute model's predictions
```

```
      output = model(x)

      # Step 2: Compute loss
      loss = loss_function(output, targets)

      # Step 3: Backward pass - Compute the gradients
      # Zero out gradients from the previous iteration
      optimizer.zero_grad()

      # Backward pass: Compute gradients based on the loss
      loss.backward()


      # Step 4: Update the parameters
      optimizer.step()

      # Accumulate the loss for the batch
      running_train_loss += loss.item()

      # Evaluate model's performance without backpropagation for
efficiency
      # `with torch.no_grad()` temporarily disables autograd,
improving speed and avoiding side effects during evaluation.
      with torch.no_grad():
          y_pred = output.argmax(dim=1) # Find the class index with
the maximum predicted probability
          correct = (y_pred == targets).sum().item() # Compute the
number of correct predictions in the batch
          running_train_correct += correct  # Update the cumulative
count of correct predictions for the current epoch


    # Compute average training loss and accuracy for the epoch
    train_loss = running_train_loss / len(train_loader)
    train_acc = running_train_correct / len(train_loader.dataset)

    # Display training loss and accuracy metrics for the current epoch
    print(f'Epoch : {epoch + 1} / {epochs}')
    print(f'Train Loss: {train_loss:.4f} | Train Accuracy: {train_acc
* 100:.4f}%')

# Fix the random seed to ensure reproducibility across runs
torch.manual_seed(100)

# Define the total number of epochs for which the model will be
trained
epochs = 5

# Detect if a GPU is available and use it; otherwise, use CPU
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
```

```
print(device)  # Output the device being used

# Define the learning rate for optimization; consider its impact on
model performance
learning_rate = 1

# Student Task: Configure the optimizer for model training.
# Here, we're using Stochastic Gradient Descent (SGD). Think through
what parameters are needed.
# Reminder: Utilize the learning rate defined above when setting up
your optimizer.
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Relocate the model to the appropriate compute device (GPU or CPU)
model.to(device)

# Apply custom weight initialization; this can affect the model's
learning trajectory
# The `apply` function recursively applies a function to each
submodule in a PyTorch model.
# In the given context, it's used to apply the `init_weights` function
to initialize the weights of all layers in the model.
# The benefit is that it provides a convenient way to systematically
apply custom weight initialization across complex models,
# potentially improving model convergence and performance.
model.apply(init_weights)

# Kick off the training process using the specified settings
train(epochs, loss_function, learning_rate, model, optimizer)
```

```
cuda:0
Epoch : 1 / 5
Train Loss: 0.9132 | Train Accuracy: 67.4000%
Epoch : 2 / 5
Train Loss: 0.8653 | Train Accuracy: 68.9000%
Epoch : 3 / 5
Train Loss: 0.8549 | Train Accuracy: 70.1000%
Epoch : 4 / 5
Train Loss: 0.8496 | Train Accuracy: 70.7000%
Epoch : 5 / 5
Train Loss: 0.8482 | Train Accuracy: 70.3000%
```

```
# Output the learned parameters (weights and biases) of the model
after training
for name, param in model.named_parameters():
  # Print the name and the values of each parameter
  print(name, param.data)
```

```
0.weight tensor([[ 0.8682, -2.4038, -1.0038, -1.3080,  1.4973],
        [ 0.6010,  2.7676,  0.9057,  1.2223,  0.1562],
```

```
        [-1.4159, -0.4914,  0.1928,  0.0637, -1.8261]],
device='cuda:0')
0.bias tensor([-0.0192, -0.4509,  0.4701], device='cuda:0')
```

Task 5 - MultiLabel Classification using Mini Batch Gradient Descent with PyTorch- 5 Points

Data

```
# Import the function to generate a synthetic multilabel
classification dataset
from sklearn.datasets import make_multilabel_classification

# Import the StandardScaler for feature normalization
from sklearn.preprocessing import StandardScaler

# Import PyTorch library for tensor computation and neural network
modules
import torch

# Import PyTorch's optimization algorithms package
import torch.optim as optim

# Import PyTorch's neural network module for defining layers and
models
import torch.nn as nn

# Import PyTorch's functional API for stateless operations
import torch.functional as F

# Import DataLoader, TensorDataset, and Dataset for data loading and
manipulation
from torch.utils.data import DataLoader, TensorDataset, Dataset

# Generate a synthetic multilabel classification dataset
# n_samples: Number of samples in the dataset
# n_features: Number of feature variables
# n_classes: Number of distinct labels (or classes)
# n_labels: Average number of labels per instance
# random_state: Seed for reproducibility
X, y = make_multilabel_classification(n_samples=1000, n_features=5,
n_classes=3, n_labels=2, random_state=0)

# Initialize the StandardScaler for feature normalization
preprocessor = StandardScaler()
```

```
# Fit the preprocessor to the data and transform the features for zero
mean and unit variance
X = preprocessor.fit_transform(X)

# Print the shape of the feature matrix X and the label matrix y
# Students: Pay attention to these shapes as they will guide you in
defining your neural network model
print(X.shape, y.shape)
```

```
(1000, 5) (1000, 3)
```

```
X[0:5]
```

```
array([[ 1.65506353,  0.2101857 ,  0.51570947, -2.00177184,
0.40001786],
       [-0.02349989, -0.51376047,  2.34771468,  0.78787635, -
1.04334554],
       [ 1.09554239,  0.93413188, -0.09495894, -0.00916599, -
0.01237169],
       [-0.58302103,  1.17544727,  0.21037527, -0.80620833,  0.8124074
],
       [ 1.09554239,  0.69281649, -1.92696415,  1.18639752, -
1.24954031]])
```

```
# ================================
# IMPORTANT: # NOTE: The y in this case is one hot encoded.
# This is different from Multiclass Classification.
# The loss function we use for multiclass classification handles this
internally
# For multilabel case we have to provide y in this format
# ================================

print(y[0:10])
```

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

Dataset and Data Loaders

```
# Student Task: Create Tensors from the numpy arrays.
# Earlier, we focused on multiclass classification; now, we are
dealing with multilabel classification.
```

```
# ================================
# IMPORTANT: # Consider what cost function you will use for multilabel
classification and whether it expects the label tensor (y) to be float
or long type.
# ================================

x_tensor = torch.tensor(X)
y_tensor = torch.tensor(y)

# Define a custom PyTorch Dataset class for handling our data
class MyDataset(Dataset):
    # Constructor: Initialize the dataset with features and labels
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    # Method to return the length of the dataset
    def __len__(self):
        return self.labels.shape[0]

    # Method to get a data point by index
    def __getitem__(self, index):
        x = self.features[index]
        y = self.labels[index]
        return x, y

# Initialize an instance of the custom MyDataset class
# This will be our training dataset, holding our features and labels
as PyTorch tensors
train_dataset = MyDataset(x_tensor, y_tensor)

# Access the first element (feature-label pair) from the train_dataset
using indexing.
# The __getitem__ method of MyDataset class will be called to return
this element.
# This is useful for debugging and understanding the data structure
train_dataset[0]
```

```
(tensor([ 1.6551,  0.2102,  0.5157, -2.0018,  0.4000],
dtype=torch.float64),
 tensor([0, 0, 1]))
```

```
# Create Data lOader from Dataset
# Use a batch size of 16
# Use shuffle = True
train_loader =  torch.utils.data.DataLoader(train_dataset,
batch_size=16, shuffle=True)
```

Model

```
# Student Task: Specify your model architecture here.
# This is a multilabel problem. Think through what layers you should
add to handle this.
# Remember, the architecture of your last layer will also depend on
your choice of loss function.
# Additional Note: No hidden layers should be added for this exercise.
# You can use nn.Linear or nn.Sequential for this task

model =  nn.Linear(in_features=5, out_features=3)
```

Loss Function

```
# Student Task: Specify the loss function for your model.
# Consider the architecture of your model, especially the last layer,
when choosing the loss function.
# This is a multilabel problem, so make sure your choice reflects
that.

loss_function =  torch.nn.BCEWithLogitsLoss()
```

Initialization

Create a function to initilaize weights.

- Initialize weights using normal distribution with mean = 0 and std = 0.05
- Initilaize the bias term with zeros

```
# Function to initialize the weights and biases of the model's layers
# This is provided to you and is not a student task
def init_weights(layer):
  # Check if the layer is a Linear layer
  if type(layer) == nn.Linear:
    # Initialize the weights with a normal distribution, mean=0,
std=0.05
    torch.nn.init.normal_(layer.weight, mean = 0, std = 0.05)
    # Initialize the bias terms to zero
    torch.nn.init.zeros_(layer.bias)
```

Training Loop

Model Training involves five steps:

- Step 0: Randomly initialize parameters / weights
- Step 1: Compute model's predictions - forward pass
- Step 2: Compute loss
- Step 3: Compute the gradients
- Step 4: Update the parameters

- Step 5: Repeat steps 1 - 4

Model training is repeating this process over and over, for many **epochs**.

We will specify number of ***epochs*** and during each epoch we will iterate over the complete dataset and will keep on updating the parameters.

Learning rate and ***epochs*** are known as hyperparameters. We have to adjust the values of these two based on validation dataset.

We will now create functions for step 1 to 4.

```
# Install the torchmetrics package, a PyTorch library for various
machine learning metrics,
# to facilitate model evaluation during and after training.
!pip install torchmetrics
```

```
Requirement already satisfied: torchmetrics in
/usr/local/lib/python3.10/dist-packages (1.1.2)
Requirement already satisfied: numpy>1.20.0 in
/usr/local/lib/python3.10/dist-packages (from torchmetrics) (1.23.5)
Requirement already satisfied: torch>=1.8.1 in
/usr/local/lib/python3.10/dist-packages (from torchmetrics)
(2.0.1+cu118)
Requirement already satisfied: lightning-utilities>=0.8.0 in
/usr/local/lib/python3.10/dist-packages (from torchmetrics) (0.9.0)
Requirement already satisfied: packaging>=17.1 in
/usr/local/lib/python3.10/dist-packages (from lightning-
utilities>=0.8.0->torchmetrics) (23.1)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from lightning-
utilities>=0.8.0->torchmetrics) (4.5.0)
Requirement already satisfied: filelock in
/usr/local/lib/python3.10/dist-packages (from torch>=1.8.1-
>torchmetrics) (3.12.2)
Requirement already satisfied: sympy in
/usr/local/lib/python3.10/dist-packages (from torch>=1.8.1-
>torchmetrics) (1.12)
Requirement already satisfied: networkx in
/usr/local/lib/python3.10/dist-packages (from torch>=1.8.1-
>torchmetrics) (3.1)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.10/dist-packages (from torch>=1.8.1-
>torchmetrics) (3.1.2)
Requirement already satisfied: triton==2.0.0 in
/usr/local/lib/python3.10/dist-packages (from torch>=1.8.1-
>torchmetrics) (2.0.0)
Requirement already satisfied: cmake in
/usr/local/lib/python3.10/dist-packages (from triton==2.0.0-
>torch>=1.8.1->torchmetrics) (3.27.4.1)
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-
```

```
packages (from triton==2.0.0->torch>=1.8.1->torchmetrics) (16.0.6)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.8.1-
>torchmetrics) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in
/usr/local/lib/python3.10/dist-packages (from sympy->torch>=1.8.1-
>torchmetrics) (1.3.0)
```

```
# Import HammingDistance from torchmetrics
# HammingDistance is useful for evaluating multi-label classification
problems.
from torchmetrics import HammingDistance
```

Hamming Distance is often used in multi-label classification problems to quantify the dissimilarity between the predicted and true labels. It does this by measuring the number of label positions where predicted and true labels differ for each sample. It is a useful metric because it offers a granular level of understanding of the discrepancies between the predicted and actual labels, taking into account each label in a multi-label setting.

Unlike accuracy, which is all-or-nothing, Hamming Distance can give partial credit by considering the labels that were correctly classified , thereby providing a more granular insight into the model's performance.

Let us understand this with an example:

```
target = torch.tensor([[0, 1], [1, 1]])
preds = torch.tensor([[0, 1], [0, 1]])
hamming_distance = HammingDistance(task="multilabel", num_labels=2)
hamming_distance(preds, target)
```

```
tensor(0.2500)
```

In the given example, the Hamming Distance is calculated for multi-label classification with two labels (0 and 1).

1. The target tensor has shape (2, 2): `[[0, 1], [1, 1]]`
2. The prediction tensor also has shape (2, 2): `[[0, 1], [0, 1]]`

Let's examine the individual sample pairs to understand the distance:

- For the first sample pair (target = `[0, 1]`, prediction = `[0, 1]`), the Hamming Distance is 0 because the prediction is accurate.
- For the second sample pair (target = `[1, 1]`, prediction = `[0, 1]`), the Hamming Distance is 1 for the first label (predicted 0, true label 1).

To calculate the overall Hamming Distance, we can take the number of label mismatches and divide by the total number of labels:

- Total Mismatches = 1 (from the second sample pair)
- Total Number of Labels = 2 samples * 2 labels per sample = 4

Therefore, the overall Hamming Distance is (1 / 4 = 0.25), which matches the output `tensor(0.2500)`.

Hamming Distance is a good metric for multi-label classification as it can capture the difference between sets of labels per sample, thereby providing a more granular measure of the model's performance.

```
def train(epochs, loss_function, learning_rate, model, optimizer, train_loader, device):

    train_hamming_distance = HammingDistance(task="multilabel", num_labels=3).to(device)

    for epoch in range(epochs):
        # Initialize train_loss at the start of the epoch
        running_train_loss = 0.0

        # Iterate on batches from the dataset using train_loader
        for x, y in train_loader:
            # Move inputs and outputs to GPUs
            x =  x.to(device, dtype=torch.float32)
            y =  y.to(device, dtype=torch.float32)

            # Step 1: Forward Pass: Compute model's predictions
            output =  model(x)

            # Step 2: Compute loss
            loss =  loss_function(output, y)

            # Step 3: Backward pass - Compute the gradients
            # Zero out gradients from the previous iteration
            optimizer.zero_grad()

            # Backward pass: Compute gradients based on the loss
            loss.backward()

            # Step 4: Update the parameters
            optimizer.step()

            # Update running loss
            running_train_loss += loss.item()

            with torch.no_grad():
                # Correct prediction using thresholding
                threshold = 0.95
                y_pred = (output > threshold).float()

                # Update Hamming Distance metric
                train_hamming_distance.update(y_pred, y)

        # Compute mean train loss for the epoch
```

```
        train_loss = running_train_loss / len(train_loader)

        # Compute Hamming Distance for the epoch
        epoch_hamming_distance = train_hamming_distance.compute()

        # Print the train loss and Hamming Distance for the epoch
        print(f'Epoch: {epoch + 1} / {epochs}')
        print(f'Train Loss: {train_loss:.4f} | Train Hamming Distance:
{epoch_hamming_distance:.4f}')

        # Reset metric states for the next epoch
        train_hamming_distance.reset()

# Set a manual seed for reproducibility across runs
torch.manual_seed(100)

# Define hyperparameters: learning rate and the number of epochs
learning_rate = 1
epochs = 20

# Determine the computing device (GPU if available, otherwise CPU)
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
print(f"Using device: {device}")

# Student Task: Configure the optimizer for model training.
# Here, we're using Stochastic Gradient Descent (SGD). Think through
what parameters are needed.
# Reminder: Utilize the learning rate defined above when setting up
your optimizer.
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# Transfer the model to the selected device (CPU or GPU)
model.to(device)

# Apply custom weight initialization function to the model layers
# Note: Weight initialization can significantly affect training
dynamics
model.apply(init_weights)

# Call the training function to start the training process
# Note: All elements like epochs, loss function, learning rate, etc.,
are passed as arguments
train(epochs, loss_function, learning_rate, model, optimizer,
train_loader, device)

Using device: cuda:0
Epoch: 1 / 20
Train Loss: 0.5119 | Train Hamming Distance: 0.3323
Epoch: 2 / 20
Train Loss: 0.4868 | Train Hamming Distance: 0.2933
```

```
Epoch: 3 / 20
Train Loss: 0.4826 | Train Hamming Distance: 0.2823
Epoch: 4 / 20
Train Loss: 0.4848 | Train Hamming Distance: 0.2893
Epoch: 5 / 20
Train Loss: 0.4835 | Train Hamming Distance: 0.2807
Epoch: 6 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2917
Epoch: 7 / 20
Train Loss: 0.4838 | Train Hamming Distance: 0.2847
Epoch: 8 / 20
Train Loss: 0.4863 | Train Hamming Distance: 0.2823
Epoch: 9 / 20
Train Loss: 0.4843 | Train Hamming Distance: 0.2850
Epoch: 10 / 20
Train Loss: 0.4848 | Train Hamming Distance: 0.2873
Epoch: 11 / 20
Train Loss: 0.4824 | Train Hamming Distance: 0.2813
Epoch: 12 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2857
Epoch: 13 / 20
Train Loss: 0.4866 | Train Hamming Distance: 0.2867
Epoch: 14 / 20
Train Loss: 0.4829 | Train Hamming Distance: 0.2910
Epoch: 15 / 20
Train Loss: 0.4856 | Train Hamming Distance: 0.2837
Epoch: 16 / 20
Train Loss: 0.4843 | Train Hamming Distance: 0.2810
Epoch: 17 / 20
Train Loss: 0.4858 | Train Hamming Distance: 0.2863
Epoch: 18 / 20
Train Loss: 0.4860 | Train Hamming Distance: 0.2823
Epoch: 19 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2853
Epoch: 20 / 20
Train Loss: 0.4842 | Train Hamming Distance: 0.2817
```

```python
# Loop through the model's parameters to display them
# This is helpful for debugging and understanding how well the model
has learned
for name, param in model.named_parameters():
    # 'name' will contain the name of the parameter (e.g.,
'layer1.weight')
    # 'param.data' will contain the parameter values
    print(name, param.data)
```

```
weight tensor([[ 0.9557, -0.1176, -0.1493,  0.3218, -0.8933],
        [-0.8538,  0.5978,  0.7190,  0.2157, -1.5057],
        [ 0.2417,  0.7477,  0.1337, -1.6168,  0.6500]],
```

```
device='cuda:0')
bias tensor([-0.1273,  0.3485,  0.1630], device='cuda:0')
```