HW3 - 20 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).**

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

  > FirstName_LastName_HW_3**

Task 1 - Tensors and Autodiff - 5 Points

In [1]:
```
import torch
import torch.nn as nn
from torch.utils import data
import torch.functional as F
from torch.utils.data import DataLoader, TensorDataset, Dataset
```

Q1 : Create Tensor (1/2 Point)

Create a torch Tensor of shape (5, 3) which is filled with zeros. Modify the tensor to set element (0, 2) to 10 and element (2, 0) to 100.

In [2]:
```
ten = torch.zeros(5, 3)
print(ten)
# Manually set the value at the first row and third column to 10,
# and the value at the third row and first column to 100 in the tensor named "my_te
# code here
ten[0,2] = 10
ten[2, 0] = 100
ten
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

Out[2]:
```
tensor([[  0.,   0.,  10.],
        [  0.,   0.,   0.],
        [100.,   0.,   0.],
        [  0.,   0.,   0.],
        [  0.,   0.,   0.]])
```

In [3]:
```
ten.shape
```

```
Out[3]: torch.Size([5, 3])
```

Q2: Reshape tensor (1/2 Point)

You have following tensor as input:

`x=torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23])`

Using only reshaping functions (like view, reshape, transpose, permute), you need to get at the following tensor as output:

```
tensor([[ 0,  4,  8, 12, 16, 20],
        [ 1,  5,  9, 13, 17, 21],
        [ 2,  6, 10, 14, 18, 22],
        [ 3,  7, 11, 15, 19, 23]])
```

```
In [4]: x=torch.tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 1
```

```
In [5]: x = torch.reshape(x, (6, 4))
        x = torch.transpose(x, 0, 1)
        x
```

```
Out[5]: tensor([[ 0,  4,  8, 12, 16, 20],
                [ 1,  5,  9, 13, 17, 21],
                [ 2,  6, 10, 14, 18, 22],
                [ 3,  7, 11, 15, 19, 23]])
```

Q3: Slice tensor (1/2 Point)

- Slice the tensor x to get the following

> - last row of x
> - fourth column of x
> - first three rows and first two columns - the shape of subtensor should be (3,2)
> - odd valued rows and columns

```
In [6]: x = torch.tensor([[1, 2, 3, 4, 5], [6, 7, 8, 8, 10], [11, 12, 13, 14, 15]])
        x
```

```
Out[6]: tensor([[ 1,  2,  3,  4,  5],
                [ 6,  7,  8,  8, 10],
                [11, 12, 13, 14, 15]])
```

```
In [7]: x.shape
```

```
Out[7]: torch.Size([3, 5])
```

In [8]:
```
# Student Task: Retrieve the last row of the tensor 'x'
# Hint: Negative indexing can help you select rows or columns counting from the end
# Think about how you can select all columns for the desired row.
last_row = x[-1]
last_row
```

Out[8]:
```
tensor([11, 12, 13, 14, 15])
```

In [9]:
```
# Student Task: Retrieve the fourth column of the tensor 'x'
# Hint: Pay attention to the indexing for both rows and columns.
# Remember that indexing in Python starts from zero.
fourth_column = x[:,-2]
fourth_column
```

Out[9]:
```
tensor([ 4,  8, 14])
```

In [10]:
```
# Student Task: Retrieve the first 3 rows and first 2 columns from the tensor 'x'.
# Hint: Use slicing to extract the required subset of rows and columns.
first_3_rows_2_columns = x[0:3, 0:2]
first_3_rows_2_columns
```

Out[10]:
```
tensor([[ 1,  2],
        [ 6,  7],
        [11, 12]])
```

In [11]:
```
# Student Task: Retrieve the rows and columns with odd-indexed positions from the t
# Hint: Use stride slicing to extract the required subset of rows and columns with
odd_valued_rows_columns = x[::2, ::2]
odd_valued_rows_columns
```

Out[11]:
```
tensor([[ 1,  3,  5],
        [11, 13, 15]])
```

Q4 -Normalize Function (1/2 Points)

Write the function that normalizes the columns of a matrix. You have to compute the mean and standard deviation of each column. Then for each element of the column, you subtract the mean and divide by the standard deviation.

In [12]:
```
# Given Data
x = [[ 3,  60,  100, -100],
     [ 2,  20,  600, -600],
     [-5,  50,  900, -900]]
```

In [13]:
```
# Convert to PyTorch Tensor and set to float
X = torch.tensor(x)
X= X.float()
```

In [14]:
```
# Print shape and data type for verification
print(X.shape)
print(X.dtype)
```

```
torch.Size([3, 4])
torch.float32
```

In [15]:
```
# Compute and display the mean and standard deviation of each column for reference
X.mean(axis = 0)
X.std(axis = 0)
```

Out[15]:
```
tensor([  4.3589,  20.8167, 404.1452, 404.1452])
```

In [16]:
```
X.std(axis = 0)
```

Out[16]:
```
tensor([  4.3589,  20.8167, 404.1452, 404.1452])
```

- Your task starts here
- Your normalize_matrix function should take a PyTorch tensor x as input.
- It should return a tensor where the columns are normalized.
- After implementing your function, use the code provided to verify if the mean for each column in Z is close to zero and the standard deviation is 1.

In [17]:
```
def normalize_matrix(x):
  # Calculate the mean along each column (think carefully , you will take mean alon
  mean = x.mean(axis=0)

  # Calculate the standard deviation along each column
  std = x.std(axis=0)

  # Normalize each element in the columns by subtracting the mean and dividing by t
  y = (x-mean)/std

  return y  # Return the normalized matrix
```

In [18]:
```
Z = normalize_matrix(X)
Z
```

Out[18]:
```
tensor([[ 0.6882,  0.8006, -1.0722,  1.0722],
        [ 0.4588, -1.1209,  0.1650, -0.1650],
        [-1.1471,  0.3203,  0.9073, -0.9073]])
```

In [19]:
```
Z.mean(axis = 0)
```

Out[19]:
```
tensor([ 0.0000e+00,  4.9671e-08,  3.9736e-08, -3.9736e-08])
```

In [20]:
```
Z.std(axis = 0)
```

Out[20]:
```
tensor([1., 1., 1., 1.])
```

Q5 -Calculate Gradients 1 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

In [21]:
```
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
  den = torch.log(x)+(x-y)**3

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

In [22]:
```
# 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 respe
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, henc
y = torch.tensor([4.0], requires_grad = True)
```

In [23]:
```
# 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 beca
f = fxy(x, y)
f
```

Out[23]:
```
tensor([584.0868], grad_fn=<DivBackward0>)
```

In [24]:
```
# Perform backpropagation to compute the gradients of 'f' with respect to 'x' and '
# Hint use backward() function on f

f.backward()
```

In [25]:
```
# 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])
```

Q6. Numerical Precision - 2 Points

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

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

.

In [26]:
```
#Question
def log_exp(x, y):
    ## add your solution here and remove pass
    log_output = torch.tensor([-1]) * (torch.log(torch.div(torch.exp(x), torch.add(
    return log_output
```

Test your codes with normal inputs:

In [27]:
```
# 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
```

Out[27]:
```
tensor([1.3133])
```

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

In [28]:
```
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)

  # CODE HERE

  # 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 = x.grad.fill_(0)
    y.grad = y.grad.fill_(0)
```

Test your codes, it should print the results nicely.

In [29]:
```
grad(log_exp, x, y)
```

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

But now let's try some "hard" inputs

In [30]:
```
x, y = torch.tensor([50.0]), torch.tensor([100.0])
```

In [31]:
```
# 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])
```

In [32]:
```
# you may see nan/inf values as output, this is not an error
torch.exp(torch.tensor([100.0]))
```

Out[32]:
```
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/

In [33]:
```
def stable_log_exp(x, y):
    z = torch.max(x, y)

    stable_log = -x + z + torch.log(torch.add(torch.exp(x - z), torch.exp(y - z)))
    #torch.subtract((torch.log(x)), torch.log(y))

    return stable_log
```

In [34]:
```
log_exp(x, y)
```

Out[34]:
```
tensor([inf], grad_fn=<MulBackward0>)
```

In [35]:
```
stable_log_exp(x, y)
```

Out[35]:
```
tensor([50.], grad_fn=<AddBackward0>)
```

In [36]:
```
grad(stable_log_exp, x, y)
```

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

Task 2 - 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 $\mathrm{y} = b + w_1 \cdot \mathrm{x} + w_2 \cdot \mathrm{x}^2$.
2. Linear model where $\mathrm{y} = b + w_1 \cdot \mathrm{x}$.
3. Linear model with no bias where $\mathrm{y} = w_1 \cdot \mathrm{x}$.

- 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

In [37]:
```
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])
```

In [38]:
```
# 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.reshape(y.size()[0], 1)

# Reshape the x tensor to have shape (n, 1), similar to y, for consistency and to w
x = x.reshape(x.size()[0], 1)

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

In [39]:
```
# Concatenate the original x tensor and its squared values (x2) along dimension 1 (
# This creates a new tensor with two features: the original x and x2 (its square) .
x_combined = torch.cat((x, x2), 1)
```

In [40]:
```
print(x_combined.shape, x.shape)
```

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

In [41]:
```
x_combined.shape[-1]
```

Out[41]:
```
2
```

Loss Function

In [42]:
```
# Initialize Mean Squared Error (MSE) loss function with mean reduction
# 'reduction="mean"' averages the squared differences between predicted and target
loss_function = nn.MSELoss(reduction = 'mean')
```

Train Function

In [43]:
```python
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 a
        optimizer.zero_grad()

        # Step 4: Calculate Gradients - Backpropagate the error to compute gradient
        loss.backward()


        # Step 5: Update Model Parameters - Adjust weights based on computed gradie
        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:

In [44]:
```
# 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 initializ
# Take into consideration the number of input features, the number of output featur
# print(x_combined.shape[-1])
# print(y.size()[-1])
model = nn.Linear(in_features = x_combined.shape[-1], out_features = y.size()[-1],

# Initialize the weights of the model using a normal distribution with mean = 0 and
# Hint: To initialize the model's weights, you can use the nn.init.normal_() functi
# You will need to provide the 'model.weight' tensor and specify values for the 'me
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_()
# You'll need to supply 'model.bias' as an argument.
nn.init.zeros_(model.bias)

# Create an SGD (Stochastic Gradient Descent) optimizer using the model's parameter
optimizer = torch.optim.SGD(model.parameters(), LEARNING_RATE)

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

```
epoch: 1 --> loss 57629.94921875
epoch: 10001 --> loss 5.004045486450195
epoch: 20001 --> loss 3.09553861618042
epoch: 30001 --> loss 2.1379129886627197
epoch: 40001 --> loss 1.6573289632797241
epoch: 50001 --> loss 1.4161760807037354
epoch: 60001 --> loss 1.294985055923462
epoch: 70001 --> loss 1.2341334819793701
epoch: 80001 --> loss 1.2036077976226807
epoch: 90001 --> loss 1.1882110834121704
```

In [45]:
```
print(f' Weights {model.weight.data}, \nBias: {model.bias.data}')
```

```
 Weights tensor([[4.1796e+01, 1.4833e-02]]),
Bias: tensor([0.9774])
```

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**..

In [46]:
```
# 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 initializ
# Take into consideration the number of input features, the number of output featur
model = nn.Linear(in_features = x.shape[-1], out_features = y.size()[-1], bias = Tr

# Initialize the weights of the model using a normal distribution with mean = 0 and
# Hint: To initialize the model's weights, you can use the nn.init.normal_() functi
# You will need to provide the 'model.weight' tensor and specify values for the 'me
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_()
# You'll need to supply 'model.bias' as an argument.
nn.init.zeros_(model.bias)

# Create an SGD (Stochastic Gradient Descent) optimizer using the model's parameter
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 57931.1484375
epoch: 11 --> loss 6.970406532287598
epoch: 21 --> loss 6.596502780914307
epoch: 31 --> loss 6.246757984161377
epoch: 41 --> loss 5.919557094573975
epoch: 51 --> loss 5.613437175750732
epoch: 61 --> loss 5.327097415924072
epoch: 71 --> loss 5.059176445007324
epoch: 81 --> loss 4.808563709259033
epoch: 91 --> loss 4.574114799499512
epoch: 101 --> loss 4.354766368865967
epoch: 111 --> loss 4.149575233459473
epoch: 121 --> loss 3.957587718963623
epoch: 131 --> loss 3.7780089378356934
epoch: 141 --> loss 3.6100101470947266
epoch: 151 --> loss 3.4528374671936035
epoch: 161 --> loss 3.305802583694458
epoch: 171 --> loss 3.168247938156128
epoch: 181 --> loss 3.039552688598633
epoch: 191 --> loss 2.9191665649414062
epoch: 201 --> loss 2.8065433502197266
epoch: 211 --> loss 2.7011678218841553
epoch: 221 --> loss 2.602609157562256
epoch: 231 --> loss 2.5104026794433594
epoch: 241 --> loss 2.424147129058838
epoch: 251 --> loss 2.343447208404541
epoch: 261 --> loss 2.267942190170288
epoch: 271 --> loss 2.197319746017456
epoch: 281 --> loss 2.131227493286133
epoch: 291 --> loss 2.0694172382354736
epoch: 301 --> loss 2.0115818977355957
epoch: 311 --> loss 1.9574825763702393
epoch: 321 --> loss 1.906874656677246
epoch: 331 --> loss 1.8595330715179443
epoch: 341 --> loss 1.8152427673339844
epoch: 351 --> loss 1.773810625076294
epoch: 361 --> loss 1.735033392906189
epoch: 371 --> loss 1.698767900466919
epoch: 381 --> loss 1.6648380756378174
epoch: 391 --> loss 1.6331119537353516
epoch: 401 --> loss 1.6034069061279297
epoch: 411 --> loss 1.5756254196166992
epoch: 421 --> loss 1.549647569656372
epoch: 431 --> loss 1.525327444076538
epoch: 441 --> loss 1.5025887489318848
epoch: 451 --> loss 1.4813003540039062
epoch: 461 --> loss 1.4614076614379883
epoch: 471 --> loss 1.4427732229232788
epoch: 481 --> loss 1.425364375114441
epoch: 491 --> loss 1.4090664386749268
epoch: 501 --> loss 1.3938236236572266
epoch: 511 --> loss 1.3795610666275024
epoch: 521 --> loss 1.3662116527557373
epoch: 531 --> loss 1.353738784790039
epoch: 541 --> loss 1.3420501947402954
epoch: 551 --> loss 1.3311245441436768
```

```
epoch: 561 --> loss 1.3209110498428345
epoch: 571 --> loss 1.3113486766815186
epoch: 581 --> loss 1.3023998737335205
epoch: 591 --> loss 1.2940291166305542
epoch: 601 --> loss 1.2862087488174438
epoch: 611 --> loss 1.2788854837417603
epoch: 621 --> loss 1.2720317840576172
epoch: 631 --> loss 1.2656220197677612
epoch: 641 --> loss 1.2596298456192017
epoch: 651 --> loss 1.2540167570114136
epoch: 661 --> loss 1.248769998550415
epoch: 671 --> loss 1.2438617944717407
epoch: 681 --> loss 1.2392655611038208
epoch: 691 --> loss 1.234969973564148
epoch: 701 --> loss 1.2309528589248657
epoch: 711 --> loss 1.2271960973739624
epoch: 721 --> loss 1.2236711978912354
epoch: 731 --> loss 1.2203744649887085
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 1.2006373405456543
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
```

In [47]:
```python
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**

In [48]:
```
# model 3
LEARNING_RATE = 0.01
EPOCHS = 10
LOG_INTERVAL= 1

# 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 initializ
# Take into consideration the number of input features, the number of output featur
model = nn.Linear(in_features = x.shape[-1], out_features = y.size()[-1], bias = Fa

# Initialize the weights of the model using a normal distribution with mean = 0 and
# Hint: To initialize the model's weights, you can use the nn.init.normal_() functi
# You will need to provide the 'model.weight' tensor and specify values for the 'me
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 parameter
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 57945.9453125
epoch: 2 --> loss 6893.7275390625
epoch: 3 --> loss 821.08251953125
epoch: 4 --> loss 98.74296569824219
epoch: 5 --> loss 12.821006774902344
epoch: 6 --> loss 2.600693702697754
epoch: 7 --> loss 1.3849643468856812
epoch: 8 --> loss 1.240365743637085
epoch: 9 --> loss 1.2231651544570923
epoch: 10 --> loss 1.2211220264434814
```

In [49]:
```
print(f' Weights {model.weight.data}')
```

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

Task 3 - MultiClass Classification using Mini Batch Gradient Descent with PyTorch- 5 Points

Data

In [50]:
```
# 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 scali
from sklearn.preprocessing import StandardScaler
```

In [51]:
```
# Import the main PyTorch library, which provides the essential building blocks for
import torch

# Import the 'optim' module from PyTorch for various optimization algorithms like S
import torch.optim as optim

# Import the 'nn' module from PyTorch, which contains pre-defined layers, loss func
import torch.nn as nn

# Import the 'functional' module from PyTorch; incorrect import here, it should be
# This module contains functional forms of layers, loss functions, and other operat
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
```

In [52]:
```
# Generate a synthetic dataset for classification using make_classification functio
# 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 ac
# - n_redundant=1: The number of redundant features, i.e., features that can be lin
# - random_state=0: The seed for the random number generator to ensure reproducibil

X, y = make_classification(n_samples=1000, n_features=5, n_classes=3, n_informative
```

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.

In [53]:
```
# 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 featu
# and then standardizes the features by subtracting the mean and dividing by the st
X = preprocessor.fit_transform(X)
```

In [54]:
```
print(X.shape, y.shape)
```

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

In [55]:
```
X[0:5]
```

Out[55]:
```
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]])
```

In [56]:
```
print(y[0:10])
```

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

Dataset and Data Loaders

In [57]:
```
# Convert the numpy arrays X and y to PyTorch Tensors.
# For X, we create a floating-point tensor since most PyTorch models expect float i
# This is a  multiclass classification problem.

# ================================
# IMPORTANT: # Consider what cost function you will use and whether it expects the
# ================================

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

In [58]:
```
# 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
```

In [59]:
```
# Create an instance of the custom MyDataset class, passing in the feature and labe
# This will allow the data to be used with PyTorch's DataLoader for efficient batch
train_dataset = MyDataset(x_tensor, y_tensor)
```

In [60]:
```
# Access the first element (feature-label pair) from the train_dataset using indexi
# The __getitem__ method of MyDataset class will be called to return this element.
train_dataset[0]
```

Out[60]:
```
(tensor([-0.3944, -0.7803, -0.2501,  0.0912, -0.5691], dtype=torch.float64),
 tensor(2, dtype=torch.int32))
```

In [61]:
```
# Create Data loader from Dataset
# Use a batch size of 16
# Use shuffle = True
train_loader = data.DataLoader(train_dataset, batch_size = 16, shuffle = True)
```

In [62]:
```
for X_batch, y_batch in train_loader:
    print(X_batch, '\n', y_batch)
    break
```

```
tensor([[-0.2185,  0.9655, -1.3025,  1.1737, -0.4569],
        [-0.8762,  0.9241, -0.0717,  0.4633, -1.2034],
        [ 0.9676, -0.9052,  1.0567,  0.6395,  0.4591],
        [ 0.6487, -0.8149, -0.4108, -0.0971,  0.8600],
        [ 0.6102, -1.0195,  0.8864,  0.1992,  0.2890],
        [-0.7050, -2.6669, -0.8674, -1.6803, -0.0552],
        [-1.8729,  1.4647, -1.1924, -0.3540, -1.6296],
        [ 0.5527, -0.1956,  0.2116, -0.1725,  0.6984],
        [-0.0641, -0.7978, -0.6218, -1.0031,  0.5330],
        [-0.4552, -0.7291,  1.3423,  0.3070, -1.1682],
        [ 0.8297,  1.4415,  1.3723,  0.4348,  0.6096],
        [ 2.5057,  0.4694,  0.9073,  3.2727,  1.1241],
        [-0.2374, -3.0383, -0.9911, -0.9811,  0.1251],
        [-1.5176,  0.3580,  0.0854, -0.7736, -1.4282],
        [-0.5085,  0.4061,  0.1271, -1.5419,  0.2346],
        [ 0.1211,  0.1431, -0.1639,  0.1702,  0.1164]], dtype=torch.float64)
 tensor([1, 2, 0, 1, 0, 0, 2, 2, 0, 2, 1, 1, 0, 2, 0, 2], dtype=torch.int32)
```

Model

In [63]:
```
# 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 th
# classification but doesn't include any hidden layers.
# You can use nn.Linear or nn.Sequential for this task
model = nn.Linear(5, 3)  #, nn.Softmax(dim = 1)
# result = model(train_loader)
# print(f'output:{result}\n')
```

Loss Function

In [64]:
```
# Student Task: Specify the loss function for your model.
# Consider the architecture of your model, especially the last layer, when choosing
# Reminder: The last layer in the previous step should guide your choice for an app

loss_function = 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

In [65]:
```
# 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 stand
    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.

In [66]:
```
# Function to train a neural network model.
# Arguments include the number of epochs, loss function, learning rate, model archi

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
    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 a
        with torch.no_grad():
            y_pred = output.argmax(dim=1) # Find the class index with the maximum pre
            correct = (y_pred == targets).sum().item() # Compute the number of correc
            running_train_correct += correct  # Update the cumulative count of correc


    # 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}%')
```

In [67]:
```
# 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 performan
learning_rate = 1

# Student Task: Configure the optimizer for model training.
# Here, we're using Stochastic Gradient Descent (SGD). Think through what parameter
# Reminder: Utilize the learning rate defined above when setting up your optimizer.
```

```
optimizer = torch.optim.SGD(model.parameters(), 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 trajecto
# The `apply` function recursively applies a function to each submodule in a PyTorc
# In the given context, it's used to apply the `init_weights` function to initializ
# The benefit is that it provides a convenient way to systematically apply custom w
# 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)
```

```
cpu
Epoch : 1 / 5
Train Loss: 0.8130 | Train Accuracy: 65.4000%
Epoch : 2 / 5
Train Loss: 0.8005 | Train Accuracy: 66.7000%
Epoch : 3 / 5
Train Loss: 0.7915 | Train Accuracy: 67.0000%
Epoch : 4 / 5
Train Loss: 0.7978 | Train Accuracy: 67.9000%
Epoch : 5 / 5
Train Loss: 0.8008 | Train Accuracy: 67.0000%
```

In [68]:
```
# 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)
```

```
weight tensor([[ 0.4345, -0.9407, -0.5891, -0.4591,  0.7364],
        [ 0.0557,  1.0332,  0.1920,  0.4732,  0.0554],
        [-0.6367, -0.0987,  0.2159,  0.0453, -0.8418]])
bias tensor([-0.2508, -0.0254,  0.2762])
```

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

Data

In [69]:
```
# 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
```

In [70]:
```
# 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
```

In [71]:
```
# 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_
```

In [72]:
```
# Initialize the StandardScaler for feature normalization
preprocessor = StandardScaler()

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

In [73]:
```
# 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 n
print(X.shape, y.shape)
```

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

In [74]:
```
X[0:5]
```

Out[74]:
```
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]])
```

In [75]:
```
# ================================
# 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

In [76]:
```
# Student Task: Create Tensors from the numpy arrays.
# Earlier, we focused on multiclass classification; now, we are dealing with multil

# ================================
# IMPORTANT: # Consider what cost function you will use for multilabel classificati
# ================================

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

In [77]:
```
# 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
```

In [78]:
```
# Initialize an instance of the custom MyDataset class
# This will be our training dataset, holding our features and labels as PyTorch ten
train_dataset = MyDataset(x_tensor, y_tensor)
```

In [79]:
```
# Access the first element (feature-label pair) from the train_dataset using indexi
# 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]
```

Out[79]:
```
(tensor([ 1.6551,  0.2102,  0.5157, -2.0018,  0.4000], dtype=torch.float64),
 tensor([0, 0, 1], dtype=torch.int32))
```

In [80]:
```
# Create Data lOader from Dataset
# Use a batch size of 16
# Use shuffle = True
train_loader =  data.DataLoader(train_dataset, batch_size = 16, shuffle = True)
```

Model

In [81]:
```
# Student Task: Specify your model architecture here.
# This is a multilabel problem. Think through what layers you should add to handle
# Remember, the architecture of your last layer will also depend on your choice of
# 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(5, 3)
```

Loss Function

In [82]:
```
# Student Task: Specify the loss function for your model.
# Consider the architecture of your model, especially the last layer, when choosing
# This is a multilabel problem, so make sure your choice reflects that.

loss_function = 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

In [83]:
```
# 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.

In [84]:
```
# Install the torchmetrics package, a PyTorch library for various machine learning
# to facilitate model evaluation during and after training.
#!pip install torchmetrics
```

In [85]:
```
# 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:

In [86]:
```
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)
```

Out[86]:
```
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.

In [87]:
```
def train(epochs, loss_function, learning_rate, model, optimizer, train_loader, dev

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

    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
            device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
            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
                y_pred = (output > 0.9).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_hammi

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

In [88]:

```
# 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 parameter
# 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 sel ected 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
train(epochs, loss_function, learning_rate, model, optimizer, train_loader, device)
```

```
Using device: cpu
Epoch: 1 / 20
Train Loss: 0.5126 | Train Hamming Distance: 0.3363
Epoch: 2 / 20
Train Loss: 0.4856 | Train Hamming Distance: 0.2877
Epoch: 3 / 20
Train Loss: 0.4825 | Train Hamming Distance: 0.2817
Epoch: 4 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2833
Epoch: 5 / 20
Train Loss: 0.4866 | Train Hamming Distance: 0.2823
Epoch: 6 / 20
Train Loss: 0.4829 | Train Hamming Distance: 0.2867
Epoch: 7 / 20
Train Loss: 0.4856 | Train Hamming Distance: 0.2757
Epoch: 8 / 20
Train Loss: 0.4843 | Train Hamming Distance: 0.2790
Epoch: 9 / 20
Train Loss: 0.4858 | Train Hamming Distance: 0.2840
Epoch: 10 / 20
Train Loss: 0.4860 | Train Hamming Distance: 0.2787
Epoch: 11 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2817
Epoch: 12 / 20
Train Loss: 0.4842 | Train Hamming Distance: 0.2773
Epoch: 13 / 20
Train Loss: 0.4845 | Train Hamming Distance: 0.2840
Epoch: 14 / 20
Train Loss: 0.4848 | Train Hamming Distance: 0.2800
Epoch: 15 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2800
Epoch: 16 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2840
Epoch: 17 / 20
Train Loss: 0.4847 | Train Hamming Distance: 0.2830
Epoch: 18 / 20
Train Loss: 0.4854 | Train Hamming Distance: 0.2800
Epoch: 19 / 20
Train Loss: 0.4828 | Train Hamming Distance: 0.2783
Epoch: 20 / 20
Train Loss: 0.4822 | Train Hamming Distance: 0.2747
```

In [89]:
```
# 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([[ 0.9856, -0.1141, -0.2715,  0.0869, -0.9284],
        [-0.9591,  0.7973,  0.5119,  0.1237, -1.4677],
        [ 0.1281,  0.7948, -0.0565, -1.6264,  0.5559]])
bias tensor([-0.2102,  0.4204,  0.0613])
```