ShritejShrikant_Chavan_HW_3

September 18, 2023

1 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**

Many cells are double because I have printed my output as well as professor's desired output in this notebook.

2 Task 1 - Tensors and Autodiff - 5 Points

[1]:
```
import torch
import torch.nn as nn
```

2.1 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.

[2]:
```
my_tensor = torch.zeros(5, 3) # CODE HERE
```

[3]:
```
my_tensor.shape
```

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

[4]:
```
my_tensor
```

[4]:
```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

[5]:
```
# 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_tensor".
# code here
# CODE HERE
```

[6]:
```
my_tensor[0, 2] = 10
my_tensor[2, 0] = 100
```

[7]:
```
my_tensor
```

[7]:
```
tensor([[  0.,   0.,  10.],
        [  0.,   0.,   0.],
        [100.,   0.,   0.],
        [  0.,   0.,   0.],
        [  0.,   0.,   0.]])
```

2.2 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]])
```

[8]:
```
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])
```

[9]:
```
# Reshape the tensor into the desired shape and transpose it
x = x.view(6, 4).permute(1, 0)
x
```

[9]:
```
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]])
```

[10]:
```
# CODE HERE
x
```

[10]:
```
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]])
```

2.3 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

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

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

[12]:
```
x.shape
```

[12]:
```
torch.Size([3, 5])
```

[13]:
```
last_row =  x[-1, :]# CODE HERE
last_row
```

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

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

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

[14]:
```
fourth_column = x[:, -2]# CODE HERE
fourth_column
```

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

[ ]:
```
# 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 = # CODE HERE
fourth_column
```

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

[15]:
```
first_3_rows_2_columns = x[:3, :2]# CODE HERE
first_3_rows_2_columns
```

[15]:
```
tensor([[ 1,  2],
        [ 6,  7],
        [11, 12]])
```

[ ]:
```
# 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 = # CODE HERE
first_3_rows_2_columns
```

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

[16]:
```
odd_valued_rows_columns = x[::2, ::2]# CODE HERE
odd_valued_rows_columns
```

[16]:
```
tensor([[ 1,  3,  5],
        [11, 13, 15]])
```

[ ]:
```
# Student Task: Retrieve the rows and columns with odd-indexed positions from␣
↪the tensor 'x'.
# Hint: Use stride slicing to extract the required subset of rows and columns␣
↪with odd indices.
odd_valued_rows_columns = # CODE HERE
odd_valued_rows_columns
```

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

2.4 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.

[17]:
```
# Given Data
x = [[ 3,  60,  100, -100],
     [ 2,  20,  600, -600],
     [-5,  50,  900, -900]]
```

[18]:
```
# Convert to PyTorch Tensor and set to float
X = torch.tensor(x)
X= X.float()
```

[19]:
```
# Print shape and data type for verification
print(X.shape)
```

```
print(X.dtype)
```

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

[20]:
```
# Compute and display the mean and standard deviation of each column for␣
↪reference
X.mean(axis = 0)
X.std(axis = 0)
```

[20]:
```
tensor([  4.3589,  20.8167, 404.1452, 404.1452])
```

[21]:
```
X.std(axis = 0)
```

[21]:
```
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.

[22]:
```
def normalize_matrix(x):
  # Calculate the mean along each column (think carefully , you will take mean␣
 ↪along axis = 0 or 1)
  mean = X.mean(axis = 0) # CODE HERE

  # Calculate the standard deviation along each column
  std = X.std(axis = 0) # CODE HERE

  # Normalize each element in the columns by subtracting the mean and dividing␣
 ↪by the standard deviation
  y = (X - mean)/std # CODE HERE

  return y  # Return the normalized matrix

Z = normalize_matrix(X)
Z
```

[22]:
```
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]])
```

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

[ ]:
```
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]])
```

[23]:
```
Z.mean(axis = 0)
```

[23]:
```
tensor([ 0.0000e+00,  4.9671e-08,  3.9736e-08, -3.9736e-08])
```

[24]:
```
Z.std(axis = 0)
```

[24]:
```
tensor([1., 1., 1., 1.])
```

2.5 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

[25]:
```
# prompt: Add x to the exponential of y

import torch
import torch.nn as nn
x = 3
y = 4
# Create a Variable of x
x = torch.tensor(x, dtype=torch.float32) # CODE HERE

# Create a Variable of y
y = torch.tensor(y, dtype=torch.float32) # CODE HERE
```

[26]:
```
def fxy(x, y):
  # Calculate the numerator: Add x to the exponential of y
  num = x + torch.exp(y)# CODE HERE

  # 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 # CODE HERE

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

[27]:
```
# 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) # CODE HERE

# 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) # CODE HERE
```

[28]:
```
# 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
```

[28]:
```
tensor([584.0868], grad_fn=<DivBackward0>)
```

[29]:
```
# Perform backpropagation to compute the gradients of 'f' with respect to 'x'␣
 ↪and 'y'
# Hint use backward() function on f
f.backward()

# CODE HERE
print('x.grad =', x.grad )# CODE HERE)
print('y.grad =', y.grad) # CODE HERE)
```

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

[ ]:
```
# 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 )# CODE HERE)
print('y.grad =', y.grad) # CODE HERE)
```

```
tensor([-19733.3965]) tensor([18322.8477])
```

2.6 Q6. Numerical Precision - 2 Points

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

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

.

[30]:
```
#Question
def log_exp(x, y):
    ## add your solution here and remove pass

    num = torch.exp(x)

    den = torch.exp(x) + torch.exp(y)
```

```
    return -torch.log(num/den)
    # CODE HERE
```

Test your codes with normal inputs:

[31]:
```
# 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
```

[31]:
```
tensor([1.3133])
```

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

[32]:
```
def grad(forward_func, x, y):
  # Enable gradient tracking for x and y, set reauires_grad appropriately

  x.requires_grad_(True)
  y.requires_grad_(True)
  # CODE HERE
  # 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()
  # CODE HERE

  # Print the gradients for x and y
  print('x.grad =', x.grad )# CODE HERE)
  print('y.grad =', y.grad) # CODE HERE)
  # Reset the gradients for x and y to zero for the next iteration

  x.grad.zero_()
  y.grad.zero_()
  # CODE HERE
  # CODE HERE


grad(log_exp, x, y)
```

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

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

[33]:
```
x, y = torch.tensor([50.0]), torch.tensor([100.0])
```

[34]:
```
# 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])
```

[35]:
```
# you may see nan/inf values as output, this is not an error
torch.exp(torch.tensor([100.0]))
```

[35]:
```
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/

[36]:
```
def stable_log_exp(x, y):

    a = max(x, y)

    rr = a + torch.log(torch.exp(x - a) + torch.exp(y - a))

    dd = x

    return rr - dd
```

[37]:
```
log_exp(x, y)
```

[37]:
```
tensor([inf], grad_fn=<NegBackward0>)
```

[38]:
```
stable_log_exp(x, y)
```

[38]:
```
tensor([50.], grad_fn=<SubBackward0>)
```

[39]:
```
grad(stable_log_exp, x, y)
```

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

3 Task 2 - Linear Regression using Batch Gradient Descent with PyTorch- 5 Points

4 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.

4.1 Data

[40]:
```
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])
```

[41]:
```
# 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(-1, 1) # Code HERE

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

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

[42]:
```
# prompt: original x tensor and its squared values (x2) along dimension 1␣
↪(columns).

x_combined = torch.cat((x, x2), 1)
```

[43]:
```
# 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), 1) # Code HERE
```

[44]:
```
print(x_combined.shape, x.shape)
```

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

##**Loss Function**

[45]:
```
# Initialize Mean Squared Error (MSE) loss function with mean reduction
# 'reduction="mean"' averages the squared differences between predicted and␣
↪target values
loss_function = nn.MSELoss(reduction='mean') # Code HERE
```

4.2 Train Function

[46]:
```
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) # Code HERE

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

        # Step 3: Zero Gradients - Clear previous gradient information to␣
↪prevent accumulation

        optimizer.zero_grad()

        # Code HERE

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

        # Code HERE

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

        # Log training information at specified intervals
        if epoch % log_interval == 0:
            print(f'epoch: {epoch + 1} --> loss {loss.item()}')
```

4.3 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:

With Learning Rate = 0.05, we get loss as NAN

In the second attempt - Now use a LEARNING_RATE = 0.0005, What do you observe?

Write your observations HERE:

With Learning Rate = 0.0005, we get appropriate loss values with each iteration

[47]:
```
from numpy.core.fromnumeric import mean
# 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.
model = nn.Linear(2, 1, True) # Code HERE

# 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.
# Code HERE

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.
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 57711.8359375
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
```

[48]:
```python
from numpy.core.fromnumeric import mean
# 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.
model = nn.Linear(2, 1, True) # Code HERE

# 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.
# Code HERE

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.
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(), lr = 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)
```

```
<ipython-input-48-cb606886d40b>:17: UserWarning: nn.init.normal is now
deprecated in favor of nn.init.normal_.
  nn.init.normal(model.weight, mean = 0, std = 0.01)
```

```
epoch: 1 --> loss 58000.4140625
epoch: 10001 --> loss 5.004487037658691
epoch: 20001 --> loss 3.0958104133605957
epoch: 30001 --> loss 2.1380457878112793
epoch: 40001 --> loss 1.6573864221572876
epoch: 50001 --> loss 1.4162007570266724
epoch: 60001 --> loss 1.2950093746185303
epoch: 70001 --> loss 1.2341467142105103
epoch: 80001 --> loss 1.2036006450653076
epoch: 90001 --> loss 1.1882147789001465
```

[ ]:
```python
from numpy.core.fromnumeric import mean
# 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.
model = nn.Linear(2, 1, True) # Code HERE

# 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.
# Code HERE

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.
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(), lr = 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 57919.14453125
epoch: 10001 --> loss 5.003323554992676
epoch: 20001 --> loss 3.095221519470215
epoch: 30001 --> loss 2.1377549171447754
epoch: 40001 --> loss 1.6572593450546265
epoch: 50001 --> loss 1.4161388874053955
epoch: 60001 --> loss 1.294983148574829
epoch: 70001 --> loss 1.2341328859329224
epoch: 80001 --> loss 1.203601360321045
epoch: 90001 --> loss 1.1882034540176392
```

[49]:
```
print(f' Weights {model.weight.data}, \nBias: {model.bias.data}')
```

```
 Weights tensor([[4.1796e+01, 1.4832e-02]]),
Bias: tensor([0.9774])
```

4.4 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**..

[50]:
```
# 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.
model = nn.Linear(1, 1, True) # Code HERE

# 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.
nn.init.normal_(model.weight, mean = 0, std = 0.01)
# Code HERE
```

```
# 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.
nn.init.zeros_(model.bias )
# Code HERE

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


# 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 57923.63671875
epoch: 11 --> loss 6.9695587158203125
epoch: 21 --> loss 6.595754146575928
epoch: 31 --> loss 6.246048450469971
epoch: 41 --> loss 5.918890953063965
epoch: 51 --> loss 5.612832069396973
epoch: 61 --> loss 5.326478958129883
epoch: 71 --> loss 5.0586347579956055
epoch: 81 --> loss 4.808050632476807
epoch: 91 --> loss 4.573611736297607
epoch: 101 --> loss 4.354315280914307
epoch: 111 --> loss 4.14913272857666
epoch: 121 --> loss 3.95723032951355
epoch: 131 --> loss 3.777653217315674
epoch: 141 --> loss 3.609668254852295
epoch: 151 --> loss 3.4525020122528076
epoch: 161 --> loss 3.3054981231689453
epoch: 171 --> loss 3.1679470539093018
epoch: 181 --> loss 3.0392870903015137
epoch: 191 --> loss 2.9189138412475586
epoch: 201 --> loss 2.8063266277313232
epoch: 211 --> loss 2.7009575366973877
epoch: 221 --> loss 2.6024136543273926
epoch: 231 --> loss 2.510205030441284
epoch: 241 --> loss 2.4239604473114014
epoch: 251 --> loss 2.3432838916778564
epoch: 261 --> loss 2.267784595489502
epoch: 271 --> loss 2.197157621383667
epoch: 281 --> loss 2.1310932636260986
```

```
epoch: 291 --> loss 2.0692989826202393
epoch: 301 --> loss 2.0114803314208984
epoch: 311 --> loss 1.957381010055542
epoch: 321 --> loss 1.9067809581756592
epoch: 331 --> loss 1.859434723854065
epoch: 341 --> loss 1.8151445388793945
epoch: 351 --> loss 1.7737195491790771
epoch: 361 --> loss 1.7349512577056885
epoch: 371 --> loss 1.6986852884292603
epoch: 381 --> loss 1.664771318435669
epoch: 391 --> loss 1.6330331563949585
epoch: 401 --> loss 1.603348970413208
epoch: 411 --> loss 1.575561761856079
epoch: 421 --> loss 1.5495884418487549
epoch: 431 --> loss 1.525285005569458
epoch: 441 --> loss 1.5025403499603271
epoch: 451 --> loss 1.4812707901000977
epoch: 461 --> loss 1.4613651037216187
epoch: 471 --> loss 1.4427378177642822
epoch: 481 --> loss 1.4253302812576294
epoch: 491 --> loss 1.4090254306793213
epoch: 501 --> loss 1.3937901258468628
epoch: 511 --> loss 1.3795316219329834
epoch: 521 --> loss 1.3661872148513794
epoch: 531 --> loss 1.3537026643753052
epoch: 541 --> loss 1.3420270681381226
epoch: 551 --> loss 1.3311102390289307
epoch: 561 --> loss 1.3208876848220825
epoch: 571 --> loss 1.3113231658935547
epoch: 581 --> loss 1.30238938331604
epoch: 591 --> loss 1.29401433467865
epoch: 601 --> loss 1.2861878871917725
epoch: 611 --> loss 1.278865098953247
epoch: 621 --> loss 1.2720192670822144
epoch: 631 --> loss 1.2656123638153076
epoch: 641 --> loss 1.2596216201782227
epoch: 651 --> loss 1.2540104389190674
epoch: 661 --> loss 1.2487608194351196
epoch: 671 --> loss 1.243851661682129
epoch: 681 --> loss 1.2392587661743164
epoch: 691 --> loss 1.234959363937378
epoch: 701 --> loss 1.2309415340423584
epoch: 711 --> loss 1.227179765701294
epoch: 721 --> loss 1.2236549854278564
epoch: 731 --> loss 1.2203748226165771
epoch: 741 --> loss 1.2172921895980835
epoch: 751 --> loss 1.214413046836853
epoch: 761 --> loss 1.211721658706665
```

```
epoch: 771 --> loss 1.2092022895812988
epoch: 781 --> loss 1.2068387269973755
epoch: 791 --> loss 1.2046363353729248
epoch: 801 --> loss 1.2025724649429321
epoch: 811 --> loss 1.200635313987732
epoch: 821 --> loss 1.1988335847854614
epoch: 831 --> loss 1.1971439123153687
epoch: 841 --> loss 1.1955697536468506
epoch: 851 --> loss 1.1940785646438599
epoch: 861 --> loss 1.1927040815353394
epoch: 871 --> loss 1.1914130449295044
epoch: 881 --> loss 1.190193772315979
epoch: 891 --> loss 1.1890735626220703
epoch: 901 --> loss 1.188012957572937
epoch: 911 --> loss 1.1870132684707642
epoch: 921 --> loss 1.1860859394073486
epoch: 931 --> loss 1.1852223873138428
epoch: 941 --> loss 1.1844003200531006
epoch: 951 --> loss 1.183647871017456
epoch: 961 --> loss 1.1829382181167603
epoch: 971 --> loss 1.1822694540023804
epoch: 981 --> loss 1.1816496849060059
epoch: 991 --> loss 1.1810719966888428
```

[ ]:
```python
# 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.
model = nn.Linear(1, 1, True) # Code HERE

# 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.
nn.init.normal_(model.weight, mean = 0, std = 0.01)
# Code HERE


# 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.
nn.init.zeros_(model.bias )
# Code HERE

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


# 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 57946.64453125
epoch: 11 --> loss 6.972099304199219
epoch: 21 --> loss 6.5980987548828125
epoch: 31 --> loss 6.24825382232666
epoch: 41 --> loss 5.920919418334961
epoch: 51 --> loss 5.614750862121582
epoch: 61 --> loss 5.328296184539795
epoch: 71 --> loss 5.0603132247924805
epoch: 81 --> loss 4.809639930725098
epoch: 91 --> loss 4.575124740600586
epoch: 101 --> loss 4.355687141418457
epoch: 111 --> loss 4.150455951690674
epoch: 121 --> loss 3.95841646194458
epoch: 131 --> loss 3.7787766456604004
epoch: 141 --> loss 3.6107280254364014
epoch: 151 --> loss 3.453505754470825
epoch: 161 --> loss 3.3064160346984863
epoch: 171 --> loss 3.1688246726989746
epoch: 181 --> loss 3.040102481842041
epoch: 191 --> loss 2.9196677207946777
epoch: 201 --> loss 2.807030200958252
epoch: 211 --> loss 2.7016286849975586
epoch: 221 --> loss 2.6030232906341553
epoch: 231 --> loss 2.5107998847961426
epoch: 241 --> loss 2.4244935512542725
epoch: 251 --> loss 2.343773603439331
epoch: 261 --> loss 2.268254041671753
epoch: 271 --> loss 2.19760799407959
epoch: 281 --> loss 2.1315178871154785
epoch: 291 --> loss 2.0696842670440674
epoch: 301 --> loss 2.011842727661133
```

```
epoch: 311 --> loss 1.9577220678329468
epoch: 321 --> loss 1.9070955514907837
epoch: 331 --> loss 1.8597275018692017
epoch: 341 --> loss 1.8154239654541016
epoch: 351 --> loss 1.7739827632904053
epoch: 361 --> loss 1.7352020740509033
epoch: 371 --> loss 1.6989233493804932
epoch: 381 --> loss 1.6649821996688843
epoch: 391 --> loss 1.6332435607910156
epoch: 401 --> loss 1.6035352945327759
epoch: 411 --> loss 1.575739860534668
epoch: 421 --> loss 1.5497491359710693
epoch: 431 --> loss 1.5254303216934204
epoch: 441 --> loss 1.5026837587356567
epoch: 451 --> loss 1.481398582458496
epoch: 461 --> loss 1.4614914655685425
epoch: 471 --> loss 1.442858338356018
epoch: 481 --> loss 1.425437331199646
epoch: 491 --> loss 1.409139633178711
epoch: 501 --> loss 1.3938827514648438
epoch: 511 --> loss 1.3796178102493286
epoch: 521 --> loss 1.3662664890289307
epoch: 531 --> loss 1.353784441947937
epoch: 541 --> loss 1.3420988321304321
epoch: 551 --> loss 1.3311737775802612
epoch: 561 --> loss 1.3209459781646729
epoch: 571 --> loss 1.3113890886306763
epoch: 581 --> loss 1.3024394512176514
epoch: 591 --> loss 1.2940703630447388
epoch: 601 --> loss 1.286230444908142
epoch: 611 --> loss 1.2789127826690674
epoch: 621 --> loss 1.2720648050308228
epoch: 631 --> loss 1.265655279159546
epoch: 641 --> loss 1.2596511840820312
epoch: 651 --> loss 1.2540349960327148
epoch: 661 --> loss 1.2487866878509521
epoch: 671 --> loss 1.2438724040985107
epoch: 681 --> loss 1.2392778396606445
epoch: 691 --> loss 1.234989047050476
epoch: 701 --> loss 1.2309608459472656
epoch: 711 --> loss 1.2271982431411743
epoch: 721 --> loss 1.2236838340759277
epoch: 731 --> loss 1.2203937768936157
epoch: 741 --> loss 1.2173058986663818
epoch: 751 --> loss 1.2144362926483154
epoch: 761 --> loss 1.2117373943328857
epoch: 771 --> loss 1.2092090845108032
epoch: 781 --> loss 1.2068579196929932
```

```
epoch: 791 --> loss 1.204651117324829
epoch: 801 --> loss 1.2025864124298096
epoch: 811 --> loss 1.2006529569625854
epoch: 821 --> loss 1.1988455057144165
epoch: 831 --> loss 1.1971585750579834
epoch: 841 --> loss 1.1955753564834595
epoch: 851 --> loss 1.1940969228744507
epoch: 861 --> loss 1.1927134990692139
epoch: 871 --> loss 1.1914150714874268
epoch: 881 --> loss 1.1902071237564087
epoch: 891 --> loss 1.1890735626220703
epoch: 901 --> loss 1.188014268875122
epoch: 911 --> loss 1.1870183944702148
epoch: 921 --> loss 1.1860883235931396
epoch: 931 --> loss 1.1852186918258667
epoch: 941 --> loss 1.1844128370285034
epoch: 951 --> loss 1.183648705482483
epoch: 961 --> loss 1.1829415559768677
epoch: 971 --> loss 1.1822807788848877
epoch: 981 --> loss 1.1816537380218506
epoch: 991 --> loss 1.1810705661773682
```

[51]:
```
print(f' Weights {model.weight.data}, \nBias: {model.bias.data}')
```

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

4.5 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**

[52]:
```
# 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␣
 ↪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.
model = nn.Linear(1, 1, False) # Code HERE

# 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.

nn.init.normal_(model.weight, mean=0, std = 0.01)
# Code HERE


# 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(), lr = LEARNING_RATE)
# Code HERE


# 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 57937.4375
epoch: 2 --> loss 6892.7119140625
epoch: 3 --> loss 820.9612426757812
epoch: 4 --> loss 98.72866821289062
epoch: 5 --> loss 12.819379806518555
epoch: 6 --> loss 2.600492477416992
epoch: 7 --> loss 1.3849482536315918
epoch: 8 --> loss 1.2403569221496582
epoch: 9 --> loss 1.2231651544570923
epoch: 10 --> loss 1.2211220264434814
```

[ ]:
```
# 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␣
 ↪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.
model = nn.Linear(1, 1, False) # Code HERE
```

```
# 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.

nn.init.normal_(model.weight, mean=0, std = 0.01)
# Code HERE


# 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(), lr = LEARNING_RATE)
# Code HERE


# 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 57974.01171875
epoch: 2 --> loss 6897.0625
epoch: 3 --> loss 821.4786376953125
epoch: 4 --> loss 98.79048156738281
epoch: 5 --> loss 12.826677322387695
epoch: 6 --> loss 2.601363182067871
epoch: 7 --> loss 1.3850539922714233
epoch: 8 --> loss 1.2403713464736938
epoch: 9 --> loss 1.223166584968567
epoch: 10 --> loss 1.2211220264434814
```

[53]:
```
print(f' Weights {model.weight.data}')
```

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

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

5.1 Data

[54]:
```
# 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
```

[55]:
```
# 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
```

[56]:
```
# 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.

[57]:
```
# 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)
```

[58]:
```
print(X.shape, y.shape)
```

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

[59]:
```
X[0:5]
```

[59]:
```
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]])
```

[60]:
```
print(y[0:10])
```

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

5.2 Dataset and Data Loaders

[61]:
```
# 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.FloatTensor(X)
y_tensor = torch.LongTensor(y)



# x_tensor = torch.from_numpy(X).type(torch.float) # Code HERE
# y_tensor =  torch.from_numpy(y).type(torch.long) # Code HERE
```

[62]:
```
x_tensor[:10]
```

[62]:
```
tensor([[-0.3944, -0.7803, -0.2501,  0.0912, -0.5691],
        [ 0.6428, -0.9584,  0.8360, -0.0844,  0.5054],
        [ 0.9910,  0.8581,  0.7879, -0.9114,  1.6262],
        [-0.9692,  0.8617, -1.3184, -1.2284, -0.0759],
        [ 0.9602,  0.9921,  1.0026, -0.2534,  1.1883],
        [ 0.4900,  1.4806,  1.4544, -0.0630,  0.4466],
        [ 2.7926, -0.7939,  1.5146,  1.3165,  2.2375],
        [-1.1350,  1.1599, -1.7242, -1.4563, -0.0137],
        [ 0.8151,  0.5456, -0.6059, -0.1101,  1.2915],
        [ 0.5674, -1.1583,  0.7301, -0.0045,  0.3711]])
```

[63]:
```
y_tensor[:10]
```

[63]:
```
tensor([2, 0, 1, 2, 1, 1, 0, 2, 0, 0])
```

[64]:
```
# 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
```

[65]:
```
# 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)
```

[66]:
```
train_dataset[0]
```

[66]:
```
(tensor([-0.3944, -0.7803, -0.2501,  0.0912, -0.5691]), tensor(2))
```

[75]:
```
# 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]
```

[75]:
```
(tensor([-0.3944, -0.7803, -0.2501,  0.0912, -0.5691]), tensor(2))
```

[67]:
```
# Create Data loader from Dataset
# Use a batch size of 16
# Use shuffle = True
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)

# Code HERE
```

5.3 Model

[68]:
```
x_tensor.shape
```

[68]:
```
torch.Size([1000, 5])
```

[69]:
```
y_tensor.shape
```

[69]:
```
torch.Size([1000])
```

[70]:
```
# 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.Sequential(nn.Linear(5, 3)) # Code HERE
```

5.4 Loss Function

[71]:
```
# 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 = nn.CrossEntropyLoss() # Code HERE
```

5.5 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

[72]:
```
# 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)
```

5.6 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.

[73]:
```
# Function to train a neural network model.
# Arguments include the number of epochs, loss function, learning rate, model␣
 ↪architecture, and optimizer.
```

```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


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)
      x = x.to(device) # Code HERE
      targets = y.to(device) # Code HERE

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

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

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

      # Code HERE

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

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

      # 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 = torch.max(output, dim= 1) # Code HERE  # Find the class␣
 ↪index with the maximum predicted probability
            correct = torch.sum(y_pred == targets) # Code HERE # 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}%')
```

[74]:
```
# 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)
# Code HERE

# 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)
```

```
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%
```

[ ]:
```
# 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 = # Code HERE

# 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: 0.8252 | Train Accuracy: 64.4000%
Epoch : 2 / 5
Train Loss: 0.8124 | Train Accuracy: 66.0000%
Epoch : 3 / 5
Train Loss: 0.7933 | Train Accuracy: 66.4000%
Epoch : 4 / 5
Train Loss: 0.8063 | Train Accuracy: 65.2000%
Epoch : 5 / 5
Train Loss: 0.8021 | Train Accuracy: 68.2000%
```

[75]:
```
# 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([[ 0.4345, -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]])
0.bias tensor([-0.2508, -0.0254,  0.2762])
```

[76]:
```
# 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([[ 0.4345, -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]])
0.bias tensor([-0.2508, -0.0254,  0.2762])
```

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

6.1 Data

[77]:
```
# 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
```

[78]:
```
# 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
```

[79]:
```
# 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)
```

[80]:
```
# 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)
```

[81]:
```
# 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)
```

[82]:
```
X[0:5]
```

[82]:
```
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]])
```

[83]:
```
# ================================
# 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]]
```

6.2 Dataset and Data Loaders

[84]:
```
# 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.FloatTensor(X) # CODE HERE
y_tensor = torch.FloatTensor(y) # CODE HERE
```

[85]:
```
# 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
```

[86]:
```
# 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) # CODE HERE
train_dataset[0]
```

[86]:
```
(tensor([ 1.6551,  0.2102,  0.5157, -2.0018,  0.4000]), tensor([0., 0., 1.]))
```

[30]:
```
# 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]
```

[30]:
```
(tensor([ 1.6551,  0.2102,  0.5157, -2.0018,  0.4000]), tensor([0, 0, 1]))
```

[87]:
```
# Create Data lOader from Dataset
# Use a batch size of 16
# Use shuffle = True
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True) # CODE␣
 ↪HERE
```

6.3 Model

[88]:
```
# 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.Sequential(nn.Linear(5, 3)) # CODE HERE
```

6.4 Loss Function

[89]:
```
# 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 = nn.BCEWithLogitsLoss() # CODE HERE
```

6.5 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

[90]:
```
# 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)
```

6.6 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.

[91]:
```
# Install the torchmetrics package, a PyTorch library for various machine␣
 ↪learning metrics,
# to facilitate model evaluation during and after training.
!pip install torchmetrics
```

```
Collecting torchmetrics
  Downloading torchmetrics-1.1.2-py3-none-any.whl (764 kB)
                           764.8/764.8
kB 7.0 MB/s eta 0:00:00
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)
Collecting lightning-utilities>=0.8.0 (from torchmetrics)
  Downloading lightning_utilities-0.9.0-py3-none-any.whl (23 kB)
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)
Installing collected packages: lightning-utilities, torchmetrics
Successfully installed lightning-utilities-0.9.0 torchmetrics-1.1.2
```

[92]:
```
# 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:

[93]:
```
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)
```

[93]:
```
tensor(0.2500)
```

[94]:
```
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)
```

[94]:
```
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.

[95]:
```
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).float() # CODE HERE
            targets = y.to(device).float() # CODE HERE

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

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

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

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

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

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

            with torch.no_grad():
                # Correct prediction using thresholding
                y_pred = (output>0.5).float()  # Code HERE

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

        # 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()
```

[96]:
```
# 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 = optim.SGD(model.parameters(), lr = learning_rate) # Code HERE

# 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: cpu
Epoch: 1 / 20
Train Loss: 0.5126 | Train Hamming Distance: 0.2947
Epoch: 2 / 20
```

```
Train Loss: 0.4856 | Train Hamming Distance: 0.2523
Epoch: 3 / 20
Train Loss: 0.4825 | Train Hamming Distance: 0.2547
Epoch: 4 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2540
Epoch: 5 / 20
Train Loss: 0.4866 | Train Hamming Distance: 0.2530
Epoch: 6 / 20
Train Loss: 0.4829 | Train Hamming Distance: 0.2563
Epoch: 7 / 20
Train Loss: 0.4856 | Train Hamming Distance: 0.2520
Epoch: 8 / 20
Train Loss: 0.4843 | Train Hamming Distance: 0.2553
Epoch: 9 / 20
Train Loss: 0.4858 | Train Hamming Distance: 0.2533
Epoch: 10 / 20
Train Loss: 0.4860 | Train Hamming Distance: 0.2540
Epoch: 11 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2620
Epoch: 12 / 20
Train Loss: 0.4842 | Train Hamming Distance: 0.2510
Epoch: 13 / 20
Train Loss: 0.4845 | Train Hamming Distance: 0.2563
Epoch: 14 / 20
Train Loss: 0.4848 | Train Hamming Distance: 0.2523
Epoch: 15 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2527
Epoch: 16 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2593
Epoch: 17 / 20
Train Loss: 0.4847 | Train Hamming Distance: 0.2533
Epoch: 18 / 20
Train Loss: 0.4854 | Train Hamming Distance: 0.2533
Epoch: 19 / 20
Train Loss: 0.4828 | Train Hamming Distance: 0.2560
Epoch: 20 / 20
Train Loss: 0.4822 | Train Hamming Distance: 0.2507
```

[ ]:
```python
# 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 = optim.SGD(model.parameters(), lr = learning_rate) # Code HERE

# 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: 0.2583
Epoch: 2 / 20
Train Loss: 0.4868 | Train Hamming Distance: 0.2453
Epoch: 3 / 20
Train Loss: 0.4826 | Train Hamming Distance: 0.2413
Epoch: 4 / 20
Train Loss: 0.4848 | Train Hamming Distance: 0.2427
Epoch: 5 / 20
Train Loss: 0.4835 | Train Hamming Distance: 0.2447
Epoch: 6 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2477
Epoch: 7 / 20
Train Loss: 0.4838 | Train Hamming Distance: 0.2427
Epoch: 8 / 20
Train Loss: 0.4863 | Train Hamming Distance: 0.2410
Epoch: 9 / 20
Train Loss: 0.4843 | Train Hamming Distance: 0.2470
Epoch: 10 / 20
Train Loss: 0.4848 | Train Hamming Distance: 0.2420
Epoch: 11 / 20
Train Loss: 0.4824 | Train Hamming Distance: 0.2437
Epoch: 12 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2477
```

```
Epoch: 13 / 20
Train Loss: 0.4866 | Train Hamming Distance: 0.2450
Epoch: 14 / 20
Train Loss: 0.4829 | Train Hamming Distance: 0.2443
Epoch: 15 / 20
Train Loss: 0.4856 | Train Hamming Distance: 0.2453
Epoch: 16 / 20
Train Loss: 0.4843 | Train Hamming Distance: 0.2453
Epoch: 17 / 20
Train Loss: 0.4858 | Train Hamming Distance: 0.2457
Epoch: 18 / 20
Train Loss: 0.4860 | Train Hamming Distance: 0.2450
Epoch: 19 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2487
Epoch: 20 / 20
Train Loss: 0.4842 | Train Hamming Distance: 0.2443
```

[97]:
```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)
```

```
0.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]])
0.bias tensor([-0.2102,  0.4204,  0.0613])
```

[ ]: