Praful_Patil_HW_3

February 21, 2024

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**

2 Task 1 - Autodiff - 5 Points

[5]:
```
import torch
import torch.nn as nn
import torch.optim as optim
```

2.1 Q1 -Normalize Function (1 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.

[6]:
```
# Given Data
x = [[ 3,  60,  100, -100],
     [ 2,  20,  600, -600],
     [-5,  50,  900, -900]]
```

[7]:
```
# Convert to PyTorch Tensor and set to float
X = torch.tensor(x)
X= X.float()
```

[8]:
```
# Print shape and data type for verification
print(X.shape)
print(X.dtype)
```

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

[9]:
```
# Compute and display the mean and standard deviation of each column for␣
 ↪reference
X.mean(axis = 0)
```

[9]:
```
tensor([   0.0000,   43.3333,  533.3333, -533.3333])
```

[10]:
```
X.std(axis = 0)
```

[10]:
```
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.

[11]:
```
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)

  # 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 the standard deviation
  y = (X - mean) / std

  return y  # Return the normalized matrix
```

[12]:
```
Z = normalize_matrix(X)
Z
```

[12]:
```
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]])
```

[13]:
```
Z.mean(axis = 0)
```

[13]:
```
tensor([ 0.0000e+00,  4.9671e-08,  3.9736e-08, -3.9736e-08])
```

[14]:
```
Z.std(axis = 0)
```

[14]:
```
tensor([1., 1., 1., 1.])
```

2.2 Q2 -Calculate Gradients 1.5 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

[15]:
```
def fxy(x, y):
  # Calculate the numerator: Add x to the exponential of y
  num = x + torch.exp(y)

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

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

[16]:
```
# Create a single-element tensor 'x' containing the value 3.0
# make sure to set 'requires_grad=True' as you want to compute gradients with␣
 ↪respect to this tensor during backpropagation
x = torch.tensor(3.0, requires_grad = True)

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

[17]:
```
# 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
```

[17]:
```
tensor(584.0868, grad_fn=<DivBackward0>)
```

[18]:
```
# Perform backpropagation to compute the gradients of 'f' with respect to 'x'␣
 ↪and 'y'
# Hint use backward() function on f

f.backward()
```

[19]:
```
# 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)
```

2.3 Q6. Numerical Precision - 2.5 Points

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

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

.

[21]:
```
#Question
def log_exp(x, y):
    ## add your solution here and remove pass
    k = -torch.log(torch.exp(x)/(torch.exp(x)+torch.exp(y)))
    return k
```

Test your codes with normal inputs:

[22]:
```
# 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
```

[22]:
```
tensor([1.3133])
```

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

[23]:
```
def grad(forward_func, x, y):
  # Enable gradient tracking for x and y, set reauires_grad appropraitely
  x.requires_grad_(True)
  y.requires_grad_(True)

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

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

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

  # Reset the gradients for x and y to zero for the next iteration
  x.grad.zero_()
  y.grad.zero_()
```

Test your codes, it should print the results nicely.

[24]:
```
grad(log_exp, x, y)
```

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

But now let's try some "hard" inputs

[28]:
```
x, y = torch.tensor([50.0]), torch.tensor([100.0])
```

[29]:
```
# 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])
```

[30]:
```
# you may see nan/inf values as output, this is not an error
torch.exp(torch.tensor([100.0]))
```

[30]:
```
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/

[31]:
```
def stable_log_exp(x, y):
    # Find the maximum value between x and y to use for normalization
    result = torch.log(1+torch.exp(y-x))
    return result
```

[32]:
```
log_exp(x, y)
```

[32]:
```
tensor([inf], grad_fn=<NegBackward0>)
```

[33]:
```
stable_log_exp(x, y)
```

[33]:
```
tensor([50.], grad_fn=<LogBackward0>)
```

[34]:
```
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 $\mathrm{y} = b + w_1 \cdot \mathrm{x} + w_2 \cdot \mathrm{x}^2$.
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.

4.1 Data

[35]:
```
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])
```

[36]:
```
# Reshape the y tensor to have shape (n, 1), where n is the number of samples.
# This is done to match the expected input shape for PyTorch's loss functions.
y = y.view(-1, 1)

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

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

[37]:
```
# Concatenate the original x tensor and its squared values (x2) along dimension␣
 ↪1 (columns).
```

```
# This creates a new tensor with two features: the original x and x2 (its␣
 ↪square) . This can be useful for polynomial regression.
x_combined = torch.cat((x, x2), dim=1)
```

[38]:
```
print(x_combined.shape, x.shape)
```

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

##**Loss Function**

[39]:
```
# Initialize Mean Squared Error (MSE) loss function with mean reduction
# 'reduction="mean"' averages the squared differences between predicted and␣
 ↪target values
loss_function = torch.nn.MSELoss(reduction="mean")
```

4.2 Train Function

[40]:
```
def train(epochs, x, y, loss_function, log_interval, model, optimizer):
    """
    Train a PyTorch model using gradient descent.

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

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


    for epoch in range(epochs):

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

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

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

```
        optimizer.zero_grad()

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

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

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

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 `nn.Linear` 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 (He) 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 might observe faster convergence, but it could lead to instability or overshooting the minimum loss.

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

Write your observations HERE: With LEARNING_RATE = 0.0005, the convergence will be more stable but might be slower, requiring more epochs to reach a similar loss level.

[41]:
```
# 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(in_features=2, out_features=1, bias=True)

# Initialize the weights of the model using a normal distribution with mean = 0␣
 ↪and std = 0.01
# Hint: To initialize the model's weights, you can use the nn.init.normal_()␣
 ↪function.
# You will need to provide the 'model.weight' tensor and specify values for the␣
 ↪'mean' and 'std' arguments.
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 = 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 57924.375
epoch: 10001 --> loss 5.002683162689209
epoch: 20001 --> loss 3.094863176345825
epoch: 30001 --> loss 2.137568473815918
epoch: 40001 --> loss 1.6571563482284546
epoch: 50001 --> loss 1.4160845279693604
epoch: 60001 --> loss 1.294957160949707
epoch: 70001 --> loss 1.2341139316558838
epoch: 80001 --> loss 1.2035857439041138
epoch: 90001 --> loss 1.188189148902893
```

[42]:
```
print(f' Weights {model.weight.data}, \nBias: {model.bias.data}')
```

```
 Weights tensor([[4.1796e+01, 1.4830e-02]]),
Bias: tensor([0.9773])
```

4.4 Part 2

- **For Part 2, use x as input to the model, this means that you have only one input.**
- Use `nn.Linear` to specify the model, **think carefully what values the three arguments `(n_ins, n_outs, bias)` will take**..

[43]:
```
# 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(in_features=1, out_features=1, bias=True)

# Initialize the weights of the model using a normal distribution with mean = 0␣
 ↪and std = 0.01
# Hint: To initialize the model's weights, you can use the torch.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)


# 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 = optim.SGD(model.parameters(), lr=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 57994.58203125
epoch: 11 --> loss 6.977301597595215
epoch: 21 --> loss 6.6029839515686035
epoch: 31 --> loss 6.252808570861816
epoch: 41 --> loss 5.9252095222473145
epoch: 51 --> loss 5.6187567710876465
epoch: 61 --> loss 5.3320417404174805
epoch: 71 --> loss 5.06383752822876
epoch: 81 --> loss 4.812923908233643
epoch: 91 --> loss 4.5781569480896
```

```
epoch: 101 --> loss 4.35856819152832
epoch: 111 --> loss 4.153106212615967
epoch: 121 --> loss 3.9608993530273438
epoch: 131 --> loss 3.7811341285705566
epoch: 141 --> loss 3.61289644241333
epoch: 151 --> loss 3.455559492111206
epoch: 161 --> loss 3.308345317840576
epoch: 171 --> loss 3.1706182956695557
epoch: 181 --> loss 3.0417723655700684
epoch: 191 --> loss 2.921234607696533
epoch: 201 --> loss 2.808487892150879
epoch: 211 --> loss 2.7030045986175537
epoch: 221 --> loss 2.604321002960205
epoch: 231 --> loss 2.511998414993286
epoch: 241 --> loss 2.4256319999694824
epoch: 251 --> loss 2.3448214530944824
epoch: 261 --> loss 2.269249677658081
epoch: 271 --> loss 2.1985297203063965
epoch: 281 --> loss 2.1323745250701904
epoch: 291 --> loss 2.0704917907714844
epoch: 301 --> loss 2.01259183883667
epoch: 311 --> loss 1.9584195613861084
epoch: 321 --> loss 1.9077517986297607
epoch: 331 --> loss 1.860347032546997
epoch: 341 --> loss 1.8159974813461304
epoch: 351 --> loss 1.7745170593261719
epoch: 361 --> loss 1.7356983423233032
epoch: 371 --> loss 1.6994043588638306
epoch: 381 --> loss 1.6654207706451416
epoch: 391 --> loss 1.6336438655853271
epoch: 401 --> loss 1.603930115699768
epoch: 411 --> loss 1.5761160850524902
epoch: 421 --> loss 1.5500918626785278
epoch: 431 --> loss 1.525754690170288
epoch: 441 --> loss 1.5029786825180054
epoch: 451 --> loss 1.4816746711730957
epoch: 461 --> loss 1.461751937866211
epoch: 471 --> loss 1.4431097507476807
epoch: 481 --> loss 1.4256666898727417
epoch: 491 --> loss 1.4093551635742188
epoch: 501 --> loss 1.394081473350525
epoch: 511 --> loss 1.3798072338104248
epoch: 521 --> loss 1.3664414882659912
epoch: 531 --> loss 1.3539434671401978
epoch: 541 --> loss 1.3422625064849854
epoch: 551 --> loss 1.3313157558441162
epoch: 561 --> loss 1.321080207824707
epoch: 571 --> loss 1.3115088939666748
```

```
epoch: 581 --> loss 1.3025527000427246
epoch: 591 --> loss 1.2941821813583374
epoch: 601 --> loss 1.2863414287567139
epoch: 611 --> loss 1.2790089845657349
epoch: 621 --> loss 1.2721474170684814
epoch: 631 --> loss 1.265730857849121
epoch: 641 --> loss 1.259728193283081
epoch: 651 --> loss 1.2541078329086304
epoch: 661 --> loss 1.2488601207733154
epoch: 671 --> loss 1.2439463138580322
epoch: 681 --> loss 1.2393379211425781
epoch: 691 --> loss 1.2350441217422485
epoch: 701 --> loss 1.231020212173462
epoch: 711 --> loss 1.2272542715072632
epoch: 721 --> loss 1.223728895187378
epoch: 731 --> loss 1.2204393148422241
epoch: 741 --> loss 1.2173596620559692
epoch: 751 --> loss 1.2144701480865479
epoch: 761 --> loss 1.2117712497711182
epoch: 771 --> loss 1.2092499732971191
epoch: 781 --> loss 1.2068901062011719
epoch: 791 --> loss 1.2046858072280884
epoch: 801 --> loss 1.2026183605194092
epoch: 811 --> loss 1.2006747722625732
epoch: 821 --> loss 1.1988669633865356
epoch: 831 --> loss 1.1971734762191772
epoch: 841 --> loss 1.1955959796905518
epoch: 851 --> loss 1.1941182613372803
epoch: 861 --> loss 1.1927354335784912
epoch: 871 --> loss 1.191430926322937
epoch: 881 --> loss 1.1902213096618652
epoch: 891 --> loss 1.189084768295288
epoch: 901 --> loss 1.1880210638046265
epoch: 911 --> loss 1.1870372295379639
epoch: 921 --> loss 1.1861084699630737
epoch: 931 --> loss 1.1852365732192993
epoch: 941 --> loss 1.1844291687011719
epoch: 951 --> loss 1.1836612224578857
epoch: 961 --> loss 1.1829524040222168
epoch: 971 --> loss 1.182286262512207
epoch: 981 --> loss 1.181666612625122
epoch: 991 --> loss 1.1810797452926636
```

[44]:
```
print(f' Weights {model.weight.data}, \nBias: {model.bias.data}')
```

```
 Weights tensor([[41.9377]]),
Bias: tensor([0.7468])
```

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**

[45]:
```
# 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(in_features=1, out_features=1, bias=False)

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


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

# Create an SGD (Stochastic Gradient Descent) optimizer using the model's␣
 ↪parameters and a predefined learning rate
optimizer = optim.SGD(model.parameters(), lr=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 57952.36328125
epoch: 2 --> loss 6894.48828125
epoch: 3 --> loss 821.1724853515625
epoch: 4 --> loss 98.7537612915039
epoch: 5 --> loss 12.822349548339844
epoch: 6 --> loss 2.6008598804473877
epoch: 7 --> loss 1.3849833011627197
```

```
epoch: 8 --> loss 1.240365743637085
epoch: 9 --> loss 1.2231651544570923
epoch: 10 --> loss 1.2211220264434814
```

[46]:
```
print(f' Weights {model.weight.data}')
```

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

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

- You will implement only training loop (no splitting of data in to training/validation).
- We will use minibatch Gradient Descent - Hence we will have two for loops in his case.
- You should use Pytorch's nn.module and functions.

5.1 Data

[47]:
```
# 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
```

[48]:
```
# 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
```

[49]:
```
# 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.

[50]:
```
# 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)
```

[51]:
```
print(X.shape, y.shape)
```

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

[52]:
```
X[0:5]
```

[52]:
```
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]])
```

[53]:
```
print(y[0:10])
```

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

5.2 Dataset and Data Loaders

[54]:
```
# Convert the numpy arrays X and y to PyTorch Tensors.
# For X, we create a floating-point tensor since most PyTorch models expect␣
 ↪float inputs for features.
# This is a  multiclass classification problem.

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

x_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)
```

[56]:
```
# 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
```

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

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

[58]:
```
(tensor([-0.3944, -0.7803, -0.2501,  0.0912, -0.5691]), tensor(2))
```

[59]:
```
# Create Data loader from Dataset
# Use a batch size of 16
# Use shuffle = True
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
```

5.3 Model

[67]:
```
# 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))
```

There is no need to use **softmax** here as it will be automatically applied internally in **nn.crossEntropyloss**

5.4 Loss Function

[68]:
```
# 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()
```

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

[69]:
```
# 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.

[71]:
```
# Function to train a neural network model.
# Arguments include the number of epochs, loss function, learning rate, model␣
 ↪architecture, and optimizer.

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

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

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

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

      # Move input and target tensors to the device (GPU or CPU)
      x = x.to(device)
      targets = y.to(device)

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

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

      # Step 3: Backward pass - Compute the gradients
      # Zero out gradients from the previous iteration
```

```
        optimizer.zero_grad()

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

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

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

        # Evaluate model's performance without backpropagation for efficiency
        # `with torch.no_grad()` temporarily disables autograd, improving speed␣
 ↪and avoiding side effects during evaluation.
        with torch.no_grad():
            y_pred = torch.argmax(output, dim=1)  # Find the class index with the␣
 ↪maximum predicted probability
            correct = torch.sum((y_pred == targets).int(), dim=0).item() #␣
 ↪Compute the number of correct predictions in the batch
            running_train_correct += correct  # Update the cumulative count of␣
 ↪correct predictions for the current epoch


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

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

[72]:
```
# Fix the random seed to ensure reproducibility across runs
torch.manual_seed(100)

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

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

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

# Student Task: Configure the optimizer for model training.
```

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

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

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

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

```
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%
```

[73]:
```
# 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

- You will implement only training loop (no splitting of data in to training/validation).
- We will use minibatch Gradient Descent - Hence we will have two for loops in his case.
- You should use Pytorch's nn.module and functions.

6.1 Data

[74]:
```
# 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
```

[75]:
```
# 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
```

[76]:
```
# 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)
```

[77]:
```
# 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)
```

[78]:
```
# 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)
```

[79]:
```
X[0:5]
```

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

[80]:
```
# ================================
# 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

[82]:
```
# Student Task: Create Tensors from the numpy arrays.
# Earlier, we focused on multiclass classification; now, we are dealing with␣
 ↪multilabel classification.

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

```
x_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)
```

[84]:
```
# 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
```

[85]:
```
# 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)
```

[86]:
```
# 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]
```

[86]:
```
(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)
```

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))
```

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()
```

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-1.3.1-py3-none-any.whl (840 kB)
                            840.4/840.4
kB 8.5 MB/s eta 0:00:00
Requirement already satisfied: numpy>1.20.0 in
/usr/local/lib/python3.10/dist-packages (from torchmetrics) (1.25.2)
Requirement already satisfied: packaging>17.1 in /usr/local/lib/python3.10/dist-
packages (from torchmetrics) (23.2)
Requirement already satisfied: torch>=1.10.0 in /usr/local/lib/python3.10/dist-
packages (from torchmetrics) (2.1.0+cu121)
Collecting lightning-utilities>=0.8.0 (from torchmetrics)
  Downloading lightning_utilities-0.10.1-py3-none-any.whl (24 kB)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
packages (from lightning-utilities>=0.8.0->torchmetrics) (67.7.2)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from lightning-
utilities>=0.8.0->torchmetrics) (4.9.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->torchmetrics) (3.13.1)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->torchmetrics) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->torchmetrics) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->torchmetrics) (3.1.3)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->torchmetrics) (2023.6.0)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->torchmetrics) (2.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from
jinja2->torch>=1.10.0->torchmetrics) (2.1.5)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
packages (from sympy->torch>=1.10.0->torchmetrics) (1.3.0)
Installing collected packages: lightning-utilities, torchmetrics
Successfully installed lightning-utilities-0.10.1 torchmetrics-1.3.1
```

[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 dissimilar-

ity 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)
```

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 , 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>=0.5

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

        # Compute mean train loss for the epoch
        train_loss = running_train_loss / len(train_loader)

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

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

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

[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 = torch.optim.SGD(model.parameters(), lr = learning_rate)

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

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

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

```
Using device: 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
# 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])
```