HW3 - 20 Points

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

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

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

  > FirstName_LastName_HW_3**

Task 1 - Autodiff - 5 Points

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

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.

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

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

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

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

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

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

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

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

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

In [ ]:
```
def normalize_matrix(X):
    # Calculate the mean along each column
    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 divi
    Y = (X - mean) / std

    return Y  # Return the normalized matrix
```

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

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

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

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

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

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

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

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

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

In [ ]:
```
# Create a single-element tensor 'x' containing the value 3.0
# make sure to set 'requires_grad=True' as you want to compute gradients wit
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 backpropagatio
y = torch.tensor(4.0, requires_grad=True)
```

In [ ]:
```
# Call the function 'fxy' with the tensors 'x' and 'y' as arguments
# The result 'f' will also be a tensor and will contain derivative informati
f = fxy(x, y)
f
```

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

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

f.backward()
```

In [ ]:
```
# 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 ope
# Print the gradients for x and y
print('x.grad =', x.grad)
print('y.grad =', y.grad)
```

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

Q6. Numerical Precision - 2.5 Points

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

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

.

In [ ]:
```
#Question
def log_exp(x, y):
    ## add your solution here and remove pass
    return -torch.log(1/(1 + (torch.exp(y)/torch.exp(x))))
```

Test your codes with normal inputs:

In [ ]:
```
# 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 outpu
z = log_exp(x, y)

# Display the computed value of z
z
```

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

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

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

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

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

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

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

Test your codes, it should print the results nicely.

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

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

But now let's try some "hard" inputs

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

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

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

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

Out[ ]:
```
tensor([inf])
```

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

Hint: (1) $\log\left(\frac{x}{y}\right) = log(x) - log(y)$

Hint: (2) See logsum Trick - https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/

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

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

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

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

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

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

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

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

Regression using Pytroch

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

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

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

- You will use **Batch gradient descent to estimate the model co-efficients.Batch gradient descent uses complete training data at each iteration.**
- You will implement only training loop (no splitting of data in to training/validation).

- The training loop will have only one `for loop`. We need to iterate over whole data in each epoch. We do not need to create batches.
- You may have to try different values of number of epochs/ learning rate to get good results.
- You should use Pytorch's nn.module and functions.

Data

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

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

# Reshape the x tensor to have shape (n, 1), similar to y, for consistency a
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
```

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

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

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

Loss Function

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

Train Function

In [ ]:
```
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 para

    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 feat
        y_hat = model(x)

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

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

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

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

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

Part 1

- **For Part 1, use x_combined (we need to use both x and x^2) as input to the model, this means that you have two inputs.**
- Use `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:

In the second attempt

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

Write your observations HERE:

In [ ]:
```
# 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 in
# Take into consideration the number of input features, the number of output
num_input_features = x_combined.shape[-1]
model = torch.nn.Linear(in_features=num_input_features, out_features=1, bias

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


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

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


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

```
epoch: 1 --> loss 57676.1171875
epoch: 10001 --> loss 5.006659507751465
epoch: 20001 --> loss 3.0969135761260986
epoch: 30001 --> loss 2.13856840133667
epoch: 40001 --> loss 1.6576560735702515
epoch: 50001 --> loss 1.4163352251052856
epoch: 60001 --> loss 1.2950727939605713
epoch: 70001 --> loss 1.2341704368591309
epoch: 80001 --> loss 1.2036234140396118
epoch: 90001 --> loss 1.188222050666809
```

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

```
 Weights tensor([[4.1796e+01, 1.4832e-02]]),
Bias: tensor([0.9775])
```

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.**.

In [ ]:

```
# 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 in
# Take into consideration the number of input features, the number of output
n_input = x.shape[-1]
model = torch.nn.Linear(in_features=n_input, out_features=1, bias=True)

# Initialize the weights of the model using a normal distribution with mean
# Hint: To initialize the model's weights, you can use the torch.nn.init.nor
# You will need to provide the 'model.weight' tensor and specify values for
torch.nn.init.normal_(model.weight,mean=0,std=0.01)

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

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


# Start the training process for the model with specified parameters and set
# 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 57948.6484375
epoch: 11 --> loss 6.972309112548828
epoch: 21 --> loss 6.598326206207275
epoch: 31 --> loss 6.248446941375732
epoch: 41 --> loss 5.921111106872559
epoch: 51 --> loss 5.614941596984863
epoch: 61 --> loss 5.328449726104736
epoch: 71 --> loss 5.060487747192383
epoch: 81 --> loss 4.809765815734863
epoch: 91 --> loss 4.57523250579834
epoch: 101 --> loss 4.355823516845703
epoch: 111 --> loss 4.150549411773682
epoch: 121 --> loss 3.9585349559783936
epoch: 131 --> loss 3.778874158859253
epoch: 141 --> loss 3.6108131408691406
epoch: 151 --> loss 3.4535880088806152
epoch: 161 --> loss 3.3064956665039062
epoch: 171 --> loss 3.1688990592956543
epoch: 181 --> loss 3.0401740074157715
epoch: 191 --> loss 2.9197487831115723
epoch: 201 --> loss 2.807086229324341
epoch: 211 --> loss 2.7016806602478027
epoch: 221 --> loss 2.6030921936035156
epoch: 231 --> loss 2.5108466148376465
epoch: 241 --> loss 2.4245550632476807
epoch: 251 --> loss 2.3438119888305664
epoch: 261 --> loss 2.2683048248291016
epoch: 271 --> loss 2.1976332664489746
epoch: 281 --> loss 2.1315486431121826
epoch: 291 --> loss 2.0697147846221924
epoch: 301 --> loss 2.0118567943573
epoch: 311 --> loss 1.9577556848526
epoch: 321 --> loss 1.9071323871612549
epoch: 331 --> loss 1.8597581386566162
epoch: 341 --> loss 1.8154510259628296
epoch: 351 --> loss 1.7740051746368408
epoch: 361 --> loss 1.735216736793518
epoch: 371 --> loss 1.698948621749878
epoch: 381 --> loss 1.6649967432022095
epoch: 391 --> loss 1.6332600116729736
epoch: 401 --> loss 1.6035537719726562
epoch: 411 --> loss 1.575767993927002
epoch: 421 --> loss 1.5497679710388184
epoch: 431 --> loss 1.5254453420639038
epoch: 441 --> loss 1.5026957988739014
epoch: 451 --> loss 1.4814112186431885
epoch: 461 --> loss 1.4614951610565186
epoch: 471 --> loss 1.4428763389587402
epoch: 481 --> loss 1.4254467487335205
epoch: 491 --> loss 1.4091451168060303
epoch: 501 --> loss 1.393888235092163
epoch: 511 --> loss 1.379632592201233
epoch: 521 --> loss 1.3662805557250977
epoch: 531 --> loss 1.353790521621704
epoch: 541 --> loss 1.3421128988265991
epoch: 551 --> loss 1.3311784267425537
```

```
epoch: 561 --> loss 1.3209583759307861
epoch: 571 --> loss 1.3113969564437866
epoch: 581 --> loss 1.3024475574493408
epoch: 591 --> loss 1.2940785884857178
epoch: 601 --> loss 1.2862406969070435
epoch: 611 --> loss 1.2789117097854614
epoch: 621 --> loss 1.2720615863800049
epoch: 631 --> loss 1.265657663345337
epoch: 641 --> loss 1.2596551179885864
epoch: 651 --> loss 1.2540409564971924
epoch: 661 --> loss 1.248786211013794
epoch: 671 --> loss 1.2438793182373047
epoch: 681 --> loss 1.2392820119857788
epoch: 691 --> loss 1.2349909543991089
epoch: 701 --> loss 1.230961561203003
epoch: 711 --> loss 1.2271995544433594
epoch: 721 --> loss 1.223686695098877
epoch: 731 --> loss 1.22039794921875
epoch: 741 --> loss 1.2173105478286743
epoch: 751 --> loss 1.2144358158111572
epoch: 761 --> loss 1.2117373943328857
epoch: 771 --> loss 1.209210753440857
epoch: 781 --> loss 1.2068642377853394
epoch: 791 --> loss 1.2046555280685425
epoch: 801 --> loss 1.202588677406311
epoch: 811 --> loss 1.200654149055481
epoch: 821 --> loss 1.1988499164581299
epoch: 831 --> loss 1.1971595287322998
epoch: 841 --> loss 1.1955804824829102
epoch: 851 --> loss 1.1940996646881104
epoch: 861 --> loss 1.1927158832550049
epoch: 871 --> loss 1.1914175748825073
epoch: 881 --> loss 1.1902055740356445
epoch: 891 --> loss 1.1890758275985718
epoch: 901 --> loss 1.1880148649215698
epoch: 911 --> loss 1.1870208978652954
epoch: 921 --> loss 1.1860928535461426
epoch: 931 --> loss 1.1852209568023682
epoch: 941 --> loss 1.1844114065170288
epoch: 951 --> loss 1.1836506128311157
epoch: 961 --> loss 1.1829437017440796
epoch: 971 --> loss 1.1822826862335205
epoch: 981 --> loss 1.1816556453704834
epoch: 991 --> loss 1.181072473526001
```

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

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

Part 3

- **Part 3 is similar to part 2, the only difference is that model has no bias term now.**

- **You will see that we are now running the model for only ten epochs and will get similar results**

In [ ]:
```
# 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 in
# Take into consideration the number of input features, the number of output
n_input = x.shape[-1]
model = torch.nn.Linear(in_features=n_input, out_features=1, bias=False)

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


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

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


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

```
epoch: 1 --> loss 57931.82421875
epoch: 2 --> loss 6892.04541015625
epoch: 3 --> loss 820.8811645507812
epoch: 4 --> loss 98.71935272216797
epoch: 5 --> loss 12.818188667297363
epoch: 6 --> loss 2.600346088409424
epoch: 7 --> loss 1.3849307298660278
epoch: 8 --> loss 1.2403569221496582
epoch: 9 --> loss 1.2231651544570923
epoch: 10 --> loss 1.2211220264434814
```

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

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

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.

Data

In [ ]:
```
# Import the make_classification function from the sklearn.datasets module
# This function is used to generate a synthetic dataset for classification t
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 an
from sklearn.preprocessing import StandardScaler
```

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

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

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

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

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

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

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

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

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

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

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

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

Out[ ]:
```
array([[-0.39443436, -0.78033571, -0.25005511,  0.09118536, -0.5690698 ],
       [ 0.64284479, -0.95837057,  0.83598996, -0.08438568,  0.50539358],
       [ 0.99102498,  0.8580679 ,  0.78786062, -0.9114329 ,  1.62615938],
       [-0.96923966,  0.86168226, -1.31837608, -1.22844863, -0.07591589],
       [ 0.96021518,  0.99206623,  1.0026402 , -0.25339161,  1.18831784]])
```

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

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

Dataset and Data Loaders

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

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

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

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

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

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

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

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

Out[ ]:
```
(tensor([-0.3944, -0.7803, -0.2501,  0.0912, -0.5691], dtype=torch.float6
4),
 tensor(2))
```

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

Model

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

Loss Function

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

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

Initialization

Create a function to initilaize weights.

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

In [ ]:
```
# 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
    torch.nn.init.normal_(layer.weight, mean=0, std=0.05)
    # Initialize the bias terms to zero.
    torch.nn.init.zeros_(layer.bias)
```

Training Loop

Model Training involves five steps:

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

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

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

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

We will now create functions for step 1 to 4.

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

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 pred
    running_train_loss = 0
    running_train_correct = 0

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

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

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


    # 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:
```

In [ ]:

```
# 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 pe
learning_rate = 1

# Student Task: Configure the optimizer for model training.
# Here, we're using Stochastic Gradient Descent (SGD). Think through what pa
# Reminder: Utilize the learning rate defined above when setting up your opt
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 t
# The `apply` function recursively applies a function to each submodule in a
# In the given context, it's used to apply the `init_weights` function to in
# The benefit is that it provides a convenient way to systematically apply c
# 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: 0.9122 | Train Accuracy: 66.6000%
Epoch : 2 / 5
Train Loss: 0.8610 | Train Accuracy: 70.1000%
Epoch : 3 / 5
Train Loss: 0.8531 | Train Accuracy: 70.1000%
Epoch : 4 / 5
Train Loss: 0.8487 | Train Accuracy: 70.4000%
Epoch : 5 / 5
Train Loss: 0.8460 | Train Accuracy: 70.5000%
```

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

```
0.weight tensor([[ 8.1974e-01, -2.3865e+00, -1.0792e+00, -1.2373e+00,  1.586
8e+00],
        [ 5.0272e-01,  2.7547e+00,  9.0071e-01,  1.2812e+00,  1.9409e-01],
        [-1.4689e+00, -3.7443e-01, -2.7421e-03,  1.5359e-02, -1.8309e+00]])
0.bias tensor([-0.0206, -0.4313,  0.4519])
```

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.

Data

In [ ]:
```python
# Import the function to generate a synthetic multilabel classification data
from sklearn.datasets import make_multilabel_classification

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

In [ ]:
```python
# 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 manipul
from torch.utils.data import DataLoader, TensorDataset, Dataset
```

In [ ]:
```
# 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_classe
```

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

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

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

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

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

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

In [ ]:
```
# ================================
# 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 intern
# For multilabel case we have to provide y in this format
# ================================

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

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

Dataset and Data Loaders

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

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

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

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

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

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

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

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

Out[ ]:
```
(tensor([ 1.6551,  0.2102,  0.5157, -2.0018,  0.4000], dtype=torch.float6
4),
 tensor([0, 0, 1]))
```

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

Model

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

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

Loss Function

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


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

Initialization

Create a function to initilaize weights.

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

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

Training Loop

Model Training involves five steps:

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

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

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

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

We will now create functions for step 1 to 4.

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

```
Requirement already satisfied: torchmetrics in /opt/homebrew/anaconda3/envs/
nlp/lib/python3.11/site-packages (1.3.1)
Requirement already satisfied: numpy>1.20.0 in /opt/homebrew/anaconda3/envs/
nlp/lib/python3.11/site-packages (from torchmetrics) (1.26.3)
Requirement already satisfied: packaging>17.1 in /opt/homebrew/anaconda3/env
s/nlp/lib/python3.11/site-packages (from torchmetrics) (23.2)
Requirement already satisfied: torch>=1.10.0 in /opt/homebrew/anaconda3/env
s/nlp/lib/python3.11/site-packages (from torchmetrics) (2.1.2)
Requirement already satisfied: lightning-utilities>=0.8.0 in /opt/homebrew/a
naconda3/envs/nlp/lib/python3.11/site-packages (from torchmetrics) (0.10.1)
Requirement already satisfied: setuptools in /opt/homebrew/anaconda3/envs/nl
p/lib/python3.11/site-packages (from lightning-utilities>=0.8.0->torchmetric
s) (69.0.3)
Requirement already satisfied: typing-extensions in /opt/homebrew/anaconda3/
envs/nlp/lib/python3.11/site-packages (from lightning-utilities>=0.8.0->torc
hmetrics) (4.9.0)
Requirement already satisfied: filelock in /opt/homebrew/anaconda3/envs/nlp/
lib/python3.11/site-packages (from torch>=1.10.0->torchmetrics) (3.13.1)
Requirement already satisfied: sympy in /opt/homebrew/anaconda3/envs/nlp/li
b/python3.11/site-packages (from torch>=1.10.0->torchmetrics) (1.12)
Requirement already satisfied: networkx in /opt/homebrew/anaconda3/envs/nlp/
lib/python3.11/site-packages (from torch>=1.10.0->torchmetrics) (3.2.1)
Requirement already satisfied: jinja2 in /opt/homebrew/anaconda3/envs/nlp/li
b/python3.11/site-packages (from torch>=1.10.0->torchmetrics) (3.1.3)
Requirement already satisfied: fsspec in /opt/homebrew/anaconda3/envs/nlp/li
b/python3.11/site-packages (from torch>=1.10.0->torchmetrics) (2023.12.2)
Requirement already satisfied: MarkupSafe>=2.0 in /opt/homebrew/anaconda3/en
vs/nlp/lib/python3.11/site-packages (from jinja2->torch>=1.10.0->torchmetric
s) (2.1.4)
Requirement already satisfied: mpmath>=0.19 in /opt/homebrew/anaconda3/envs/
nlp/lib/python3.11/site-packages (from sympy->torch>=1.10.0->torchmetrics)
(1.3.0)
```

In [ ]:
```
# Import HammingDistance from torchmetrics
# HammingDistance is useful for evaluating multi-label classification proble
from torchmetrics import HammingDistance
```

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

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

Let us understand this with an example:

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

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

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

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

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

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

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

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

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

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

In [ ]:
```
def train(epochs, loss_function, learning_rate, model, optimizer, train_loac

    train_hamming_distance = HammingDistance(task="multilabel", num_labels=3

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In [ ]:

```
# 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 pa
# Reminder: Utilize the learning rate defined above when setting up your opt
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 pa
train(epochs, loss_function, learning_rate, model, optimizer, train_loader,
```

```
Using device: cpu
Epoch: 1 / 20
Train Loss: 0.5126 | Train Hamming Distance: 0.3437
Epoch: 2 / 20
Train Loss: 0.4856 | Train Hamming Distance: 0.2933
Epoch: 3 / 20
Train Loss: 0.4825 | Train Hamming Distance: 0.2830
Epoch: 4 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2857
Epoch: 5 / 20
Train Loss: 0.4866 | Train Hamming Distance: 0.2867
Epoch: 6 / 20
Train Loss: 0.4829 | Train Hamming Distance: 0.2913
Epoch: 7 / 20
Train Loss: 0.4856 | Train Hamming Distance: 0.2837
Epoch: 8 / 20
Train Loss: 0.4843 | Train Hamming Distance: 0.2810
Epoch: 9 / 20
Train Loss: 0.4858 | Train Hamming Distance: 0.2863
Epoch: 10 / 20
Train Loss: 0.4860 | Train Hamming Distance: 0.2823
Epoch: 11 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2853
Epoch: 12 / 20
Train Loss: 0.4842 | Train Hamming Distance: 0.2817
Epoch: 13 / 20
Train Loss: 0.4845 | Train Hamming Distance: 0.2897
Epoch: 14 / 20
Train Loss: 0.4848 | Train Hamming Distance: 0.2867
Epoch: 15 / 20
Train Loss: 0.4833 | Train Hamming Distance: 0.2840
Epoch: 16 / 20
Train Loss: 0.4846 | Train Hamming Distance: 0.2897
Epoch: 17 / 20
Train Loss: 0.4847 | Train Hamming Distance: 0.2880
Epoch: 18 / 20
Train Loss: 0.4854 | Train Hamming Distance: 0.2860
Epoch: 19 / 20
Train Loss: 0.4828 | Train Hamming Distance: 0.2873
Epoch: 20 / 20
Train Loss: 0.4822 | Train Hamming Distance: 0.2793
```

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

```
    # 'param.data' will contain the parameter values
    print(name, param.data)
```

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