In [70]:
```
!jupyter nbconvert --to html /content/Assignment1_Final.ipynb
```

```
[NbConvertApp] Converting notebook /content/Assignment1_Final.ipynb to html
[NbConvertApp] Writing 5970900 bytes to /content/Assignment1_Final.html
```

FIT3181: Deep Learning (2024)

Lecturer (Malaysia): **Dr Arghya Pal** | arghya.pal@monash.edu
Lecturer (Malaysia): **Dr Lim Chern Hong** | lim.chernhong@monash.edu

CE/Lecturer (Clayton): **Dr Trung Le** | trunglm@monash.edu
Lecturer (Clayton): **Prof Dinh Phung** | dinh.phung@monash.edu

School of Information Technology, Monash University, Malaysia
Faculty of Information Technology, Monash University, Australia

---

Student Information

Surname: **Sarawanan**
Firstname: **Lekxment**
Student ID: **33530025**
Email: **lsar0007@student.monash.edu**
Your tutorial time: **Thursday 12-2pm**

[Very Important]
Make a copy of thus Google colab notebook including the traces and progresses of model training before submitting.**

Deep Neural Networks

Due: 11:55pm Wednesday, 11 September 2024 (Wednesday)

Important note: **This is an individual assignment. It contributes 25% to your final mark. Read the assignment instructions carefully.**

What to submit

This assignment is to be completed individually and submitted to Moodle unit site. **By the due date, you are required to submit one single zip file, named**

xxx_assignment01_solution.zip where `xxx` is your student ID, to the corresponding Assignment (Dropbox) in Moodle. You can use Google Colab to do Assigmnent 1 but you need to save it to an `*.ipynb` file to submit to the unit Moodle.

More importantly, if you use Google Colab to do this assignment, you need to first make a copy of this notebook on your Google drive.

For example, if your student ID is 12356, then gather all of your assignment solution to folder, create a zip file named 123456_assignment01_solution.zip and submit this file.

Within this zip folder, you **must** submit the following files:

1. **Assignment01_solution.ipynb**: this is your Python notebook solution source file.
2. **Assignment01_output.html**: this is the output of your Python notebook solution *exported* in html format.
3. Any **extra files or folder** needed to complete your assignment (e.g., images used in your answers).

Since the notebook is quite big to load and work together, one recommended option is to split solution into three parts and work on them seperately. In that case, replace **Assignment01_solution.ipynb** by three notebooks: **Assignment01_Part1_solution.ipynb**, **Assignment01_Part2_solution.ipynb** and **Assignment01_Part3_solution.ipynb**

Part 1: Theory and Knowledge Questions

[Total marks for this part: 30 points]

The first part of this assignment is to demonstrate your knowledge in deep learning that you have acquired from the lectures and tutorials materials. Most of the contents in this assignment are drawn from **the lectures and tutorials from weeks 1 to 4**. Going through these materials before attempting this part is highly recommended.

Question 1.1 Activation function plays an important role in modern Deep NNs. For each of the activation functions below, state its output range, find its derivative (show your steps), and plot the activation fuction and its derivative

(a) Exponential linear unit (ELU): $\mathrm{ELU}(x) = \begin{cases} 0.1\left(\exp(x) - 1\right) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$

[1.5 points]

(b) Gaussian Error Linear Unit (GELU): $\mathrm{GELU}(x) = x\Phi(x)$ where $\Phi(x)$ is the `probability cummulative function` of the standard Gaussian distribution or $\Phi(x) = \mathbb{P}\left(X \leq x\right)$ where $X \sim N\left(0, 1\right)$. In addition, the GELU activation fuction (the link for the main paper) has been widely used in the state-of-the-art Vision for Transformers (e.g., here is the link for the main ViT paper).

[1.5 points]

Part 1

1.1 a)

$$ELU(x) = \begin{cases} 0.1(\exp(x)-1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

ELU Range:

- For $x \leq 0$, the exponential function approaches 0 as $x \to -\infty$. So, the output will approach -0.1. At $x = 0$, output is 0.
- For $x > 0$, since it is more than 0, $ELU(x) = x$. The larger x gets, the larger the output. So, the range will be $(0, +\infty)$

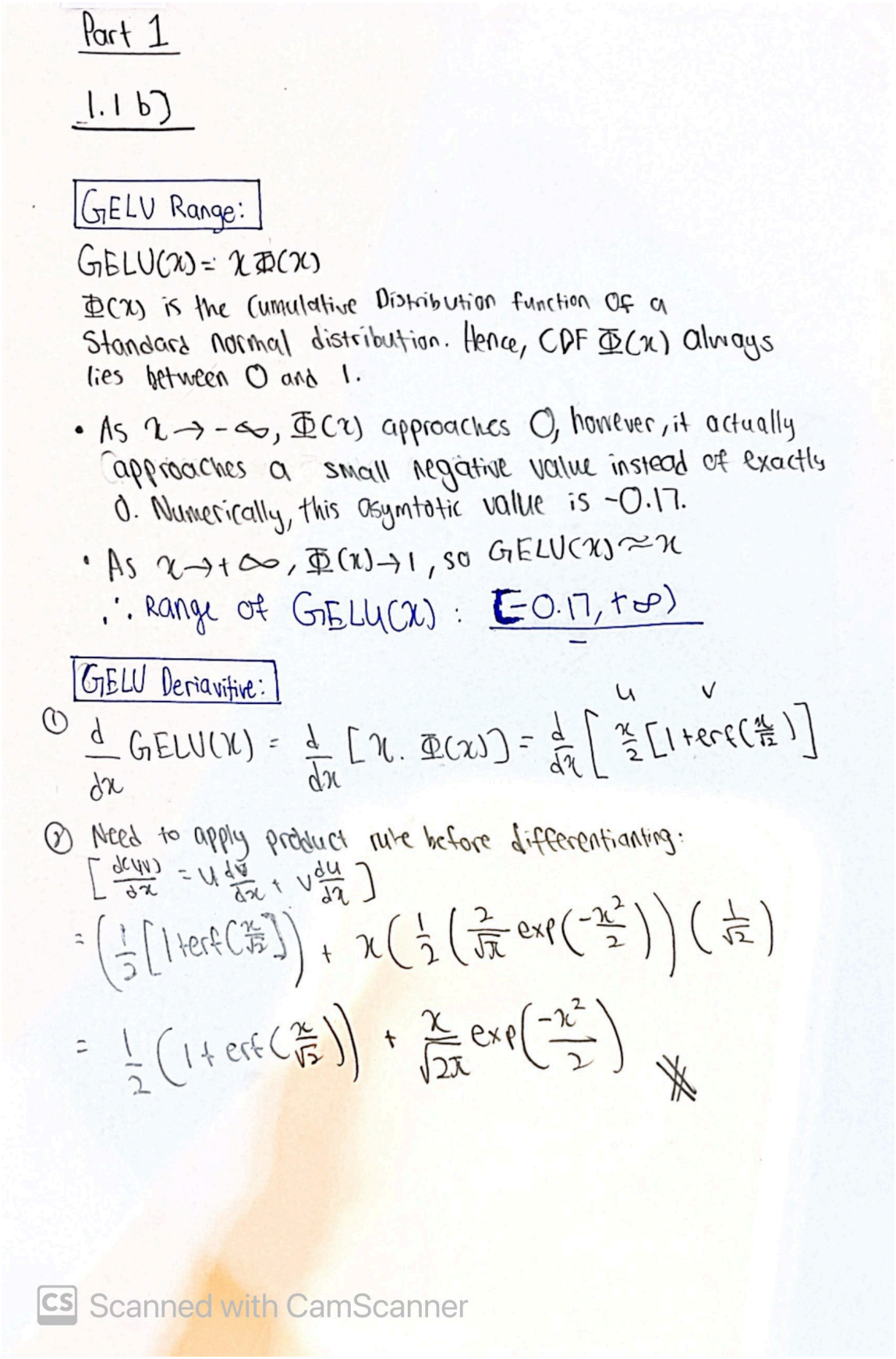
∴ Range of ELU is $(-0.1, +\infty)$

ELU Derivative:

- For $x > 0$, $f'(x) = 1$
  ↳ Positive x values, derivative is 1.
- For $x \leq 0$,

  $f'(x) = 0.1(\exp(x))$

$\frac{d}{dx}[e^x] = e^x \cdot (1$



Write your answer here. You can add more cells if needed.

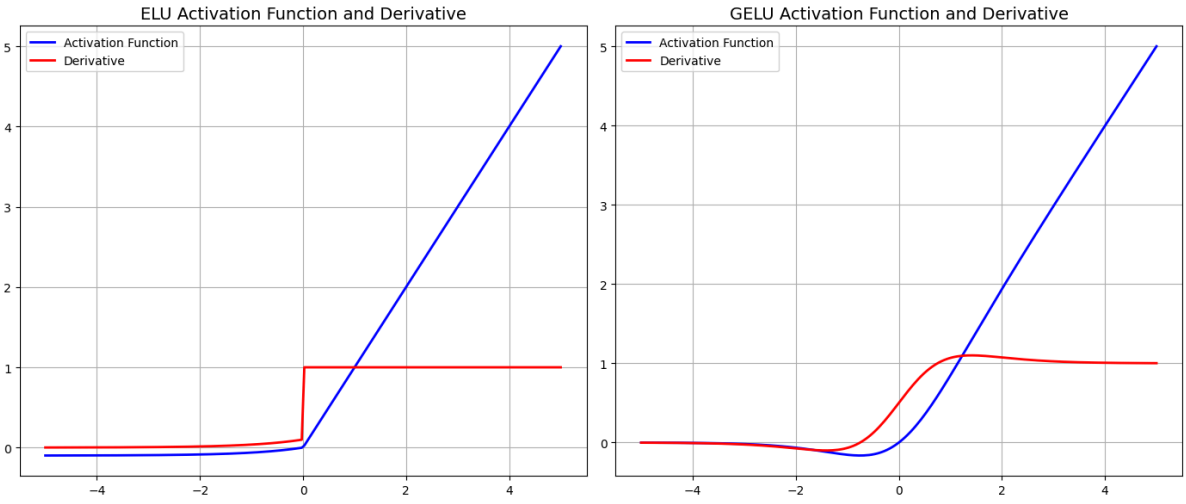
In [ ]:
```
# This code is acquired from week 5 tutorial to plot the activation function's deri
from torch import nn
import matplotlib.pyplot as plt

# Function to plot activation function and its derivative
def plot_activation_function_with_derivative(ax, activation_function, title=''):
```

```
    linespace = torch.linspace(-5, 5, 200)
    activation = activation_function(linespace)

    # Compute the derivative
    linespace.requires_grad_(True)
    activation = activation_function(linespace)
    activation.backward(torch.ones_like(linespace))
    grads = linespace.grad

    # Plot activation function
    ax.plot(linespace.detach(), activation.detach(), 'b', label='Activation Functic

    # Plot derivative
    ax.plot(linespace.detach(), grads.detach(), 'r', label='Derivative', linewidth=

    # Set the title and labels
    ax.set_title(title, fontsize=14)
    ax.grid(True)
    ax.legend()
```

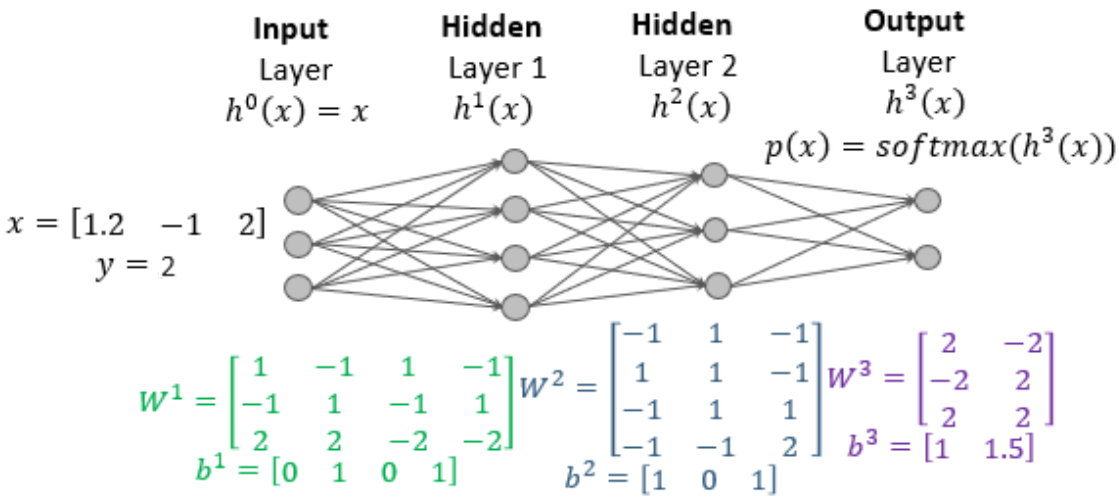
In [ ]:
```
# First we make the elu class which is from the week 5 tutorial

class ELU(nn.Module):
    def __init__(self, alpha=0.1):
        super(ELU, self).__init__()
        # initialise the alpha
        self.alpha = alpha

    def forward(self, x):
        return torch.where(x < 0, self.alpha * (torch.exp(x) - 1), x) # Follow the
```

In [ ]:
```
class GELU(nn.Module):
    def __init__(self):
        super(GELU, self).__init__()

    def forward(self, x):
        # This formula is obtained from the paper provided
        return x * torch.nn.functional.sigmoid(1.702 * x)
```

In [ ]:
```
# Plotting the derivatives, this code is from tutorial week 5 as well
elu_activation_function = ELU()
gelu_activation_function = GELU()

# Plotting
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

plot_activation_function_with_derivative(ax1, elu_activation_function, title="ELU A
plot_activation_function_with_derivative(ax2, gelu_activation_function, title="GELU

plt.tight_layout()
plt.show()
```

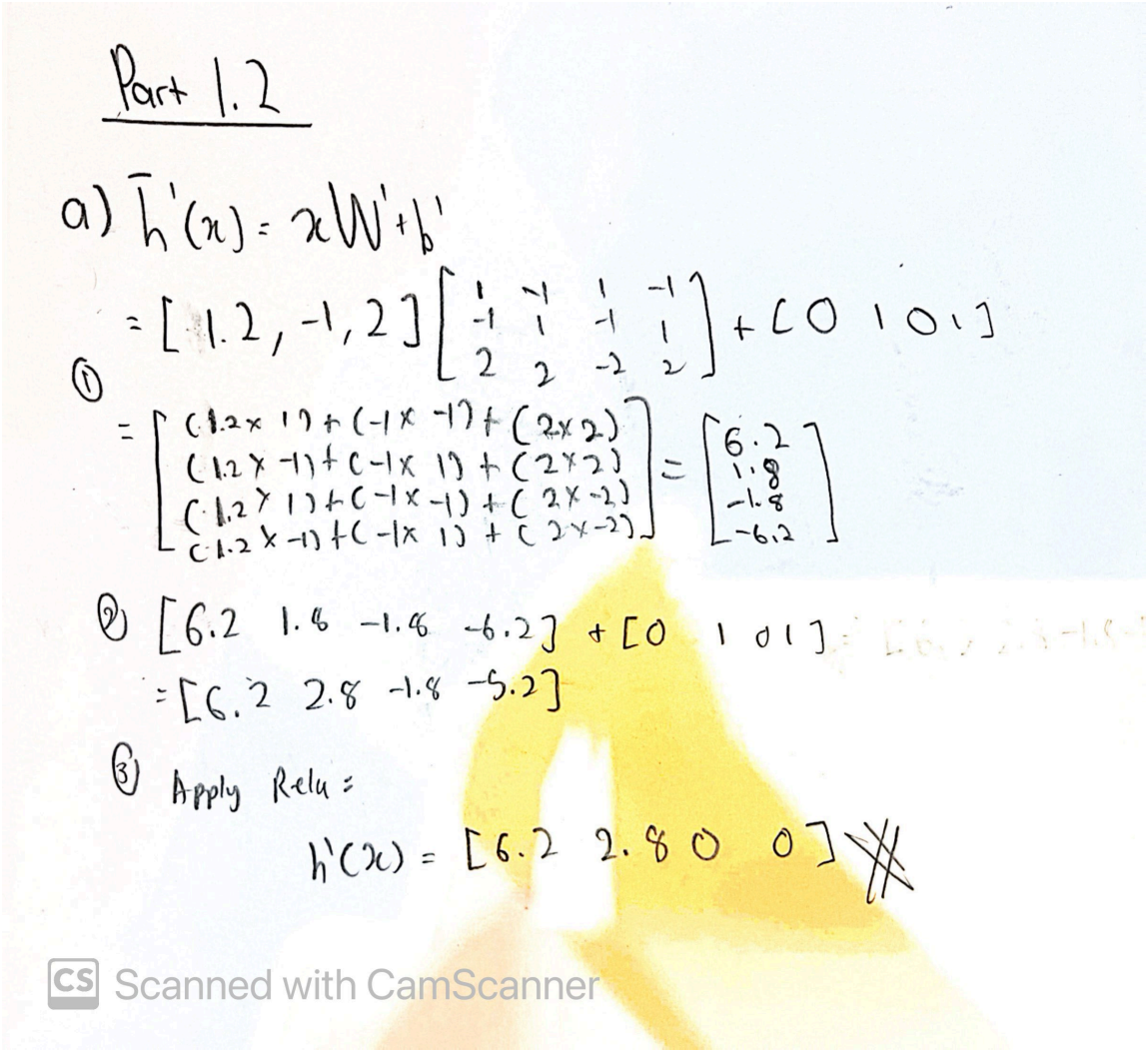


Question 1.2: **Assume that we feed a data point x with a ground-truth label $y = 2$ to the feed-forward neural network with the `ReLU activation` function as shown in the following figure**



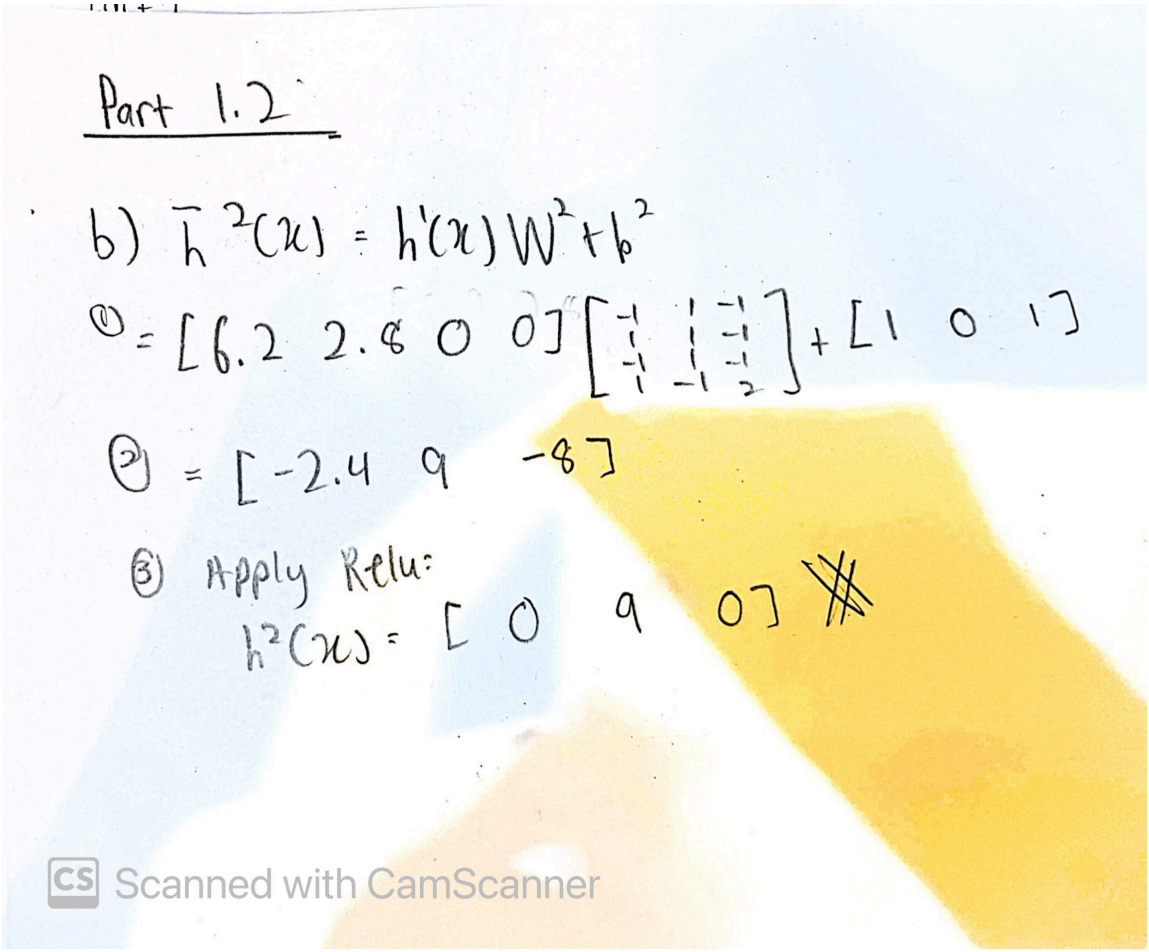
(a) What is the numerical value of the latent presentation $h^1(x)$?

[1 point]



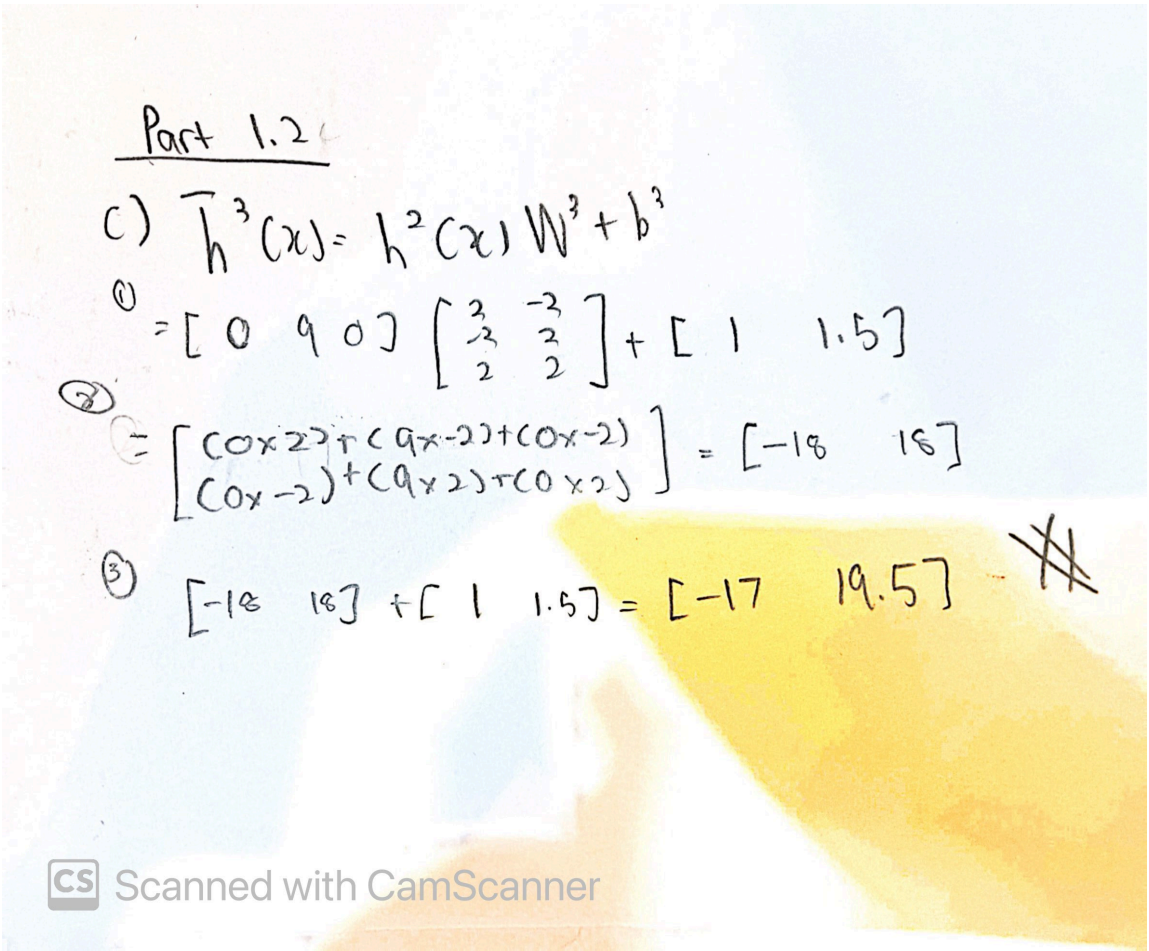
(b) What is the numerical value of the latent presentation $h^2(x)$?

[1 point]



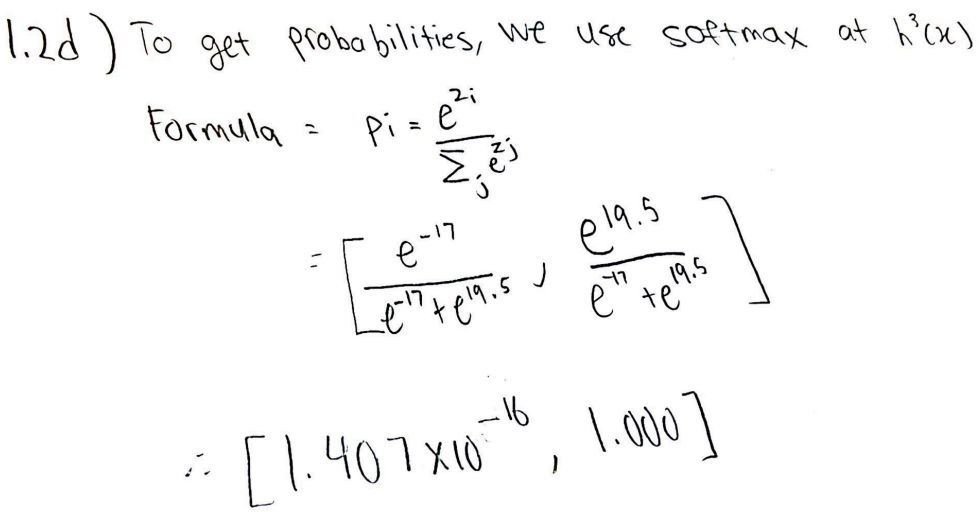
(c) What is the numerical value of the logit $h^3(x)$?

[1 point]



(d) What is the corresonding prediction probabilities $p(x)$?

[1 point]



(e) What is the predicted label $\hat{y}$? Is it a correct and an incorect prediction? Remind that $y = 2$.

[1 point]

Answer: Since the ground truth label is y=2 and the second class has the highest probability , this means that the model accurately predicted the class. The prediction is correct.

(f) What is the cross-entropy loss caused by the feed-forward neural network at (x, y)? Remind that $y = 2$.

[1 point]

Answer: The cross-entropy loss formula used is CE(y,p(x)) = -log(p_y). Since the true class is y=2 , we will take the second class probability which is 1.000. Hence , CE(y,p(x)) = -log(p_y) = -log(1.000) = 0.

(g) Why is the cross-entropy loss caused by the feed-forward neural network at (x, y) (i.e., $\mathrm{CE}(1_y, p(x))$) always non-negative? When does this $\mathrm{CE}(1_y, p(x))$ loss get the value 0 ? Note that you need to answer this question for a general pair (x, y) and a general feed-forward neural network with, for example $M = 4$ classes?

[1 point]

Answer: The cross-entropy loss measures the difference between the predicted probabilities and the true labels. It is important to note that probability , p_y ranges from 0 to 1 only.

The logarithmic function is applied to the probability and this results in a negative value. However, when two negatives multiply, it becomes positive. Hence that is why, CE loss will always be positive

For example, let's say we have a feed forward neural network of 4 classes with a true label y=3 and for instance , the predicted probabilites by the model are [0.1,0.15,0.7,0.05]. The correct class is y=3 so p_y = 0.7. The CE loss is , -log(0.7) = -(-0.357) = 0.357.

If the model predicts [0,0,1,0] where p_y = 1 , the loss becomes -log(1) = 0. This means that the model is perfectly confident in predicted y=3 as the ground truth label , leadign to zero loss. Hence, CE loss will be zero when the predicted probability is 1.

You must show both formulas and numerical results for earning full mark. Although it is optional, it is great if you show your PyTorch code for your computation.

Question 1.3:

For **Question 1.3**, you have two options:

- **(1)** *perform the forward, backward propagation, and SGD update for* `one mini-batch` (**10 points**), or
- **(2)** *manually implement a feed-forward neural network* that can work on real tabular datasets (**20 points**).

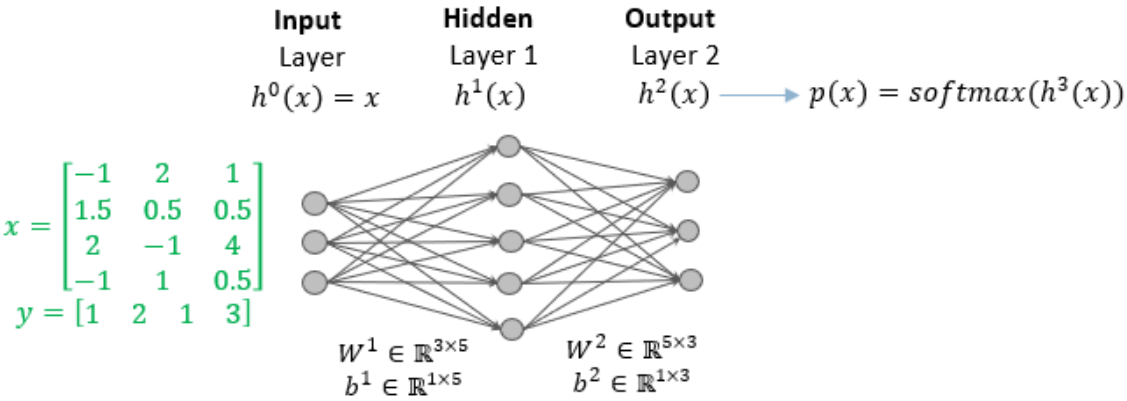
You can choose either **(1)** or **(2)** to proceed.

Option 1

[Total marks for this option: 10 points]

Assume that we are constructing a multilayered feed-forward neural network for a classification problem with three classes where the model parameters will be generated randomly using your student ID. The architecture of this network is $3(Input) \rightarrow 5(ELU) \rightarrow 3(Output)$ as shown in the following figure. Note that the ELU has the same formula as the one in Q1.1.

We feed a batch X with the labels Y as shown in the figure. Answer the following questions.



You need to show both formulas, numerical results, and your PyTorch code for your computation for earning full marks.

In [ ]:
```
import torch
student_id = 1234          #insert your student id here for example 1234
```

```
torch.manual_seed(student_id)
```

Out[ ]:
```
<torch._C.Generator at 0x7dc439f98810>
```

In [ ]:
```
#Code to generate random matrices and biases for W1, b1, W2, b2
```

Forward propagation

(a) What is the value of $\bar{h}^1(x)$ (the pre-activation values of h^1)?

[0.5 point]

In [ ]:
```
#Show your code
```

(b) What is the value of $h^1(x)$?

[0.5 point]

In [ ]:
```
#Show your code
```

(c) What is the predicted value $\hat{y}$?

[0.5 point]

In [ ]:
```
#Show your code
```

(d) Suppose that we use the cross-entropy (CE) loss. What is the value of the CE loss l incurred by the mini-batch?

[0.5 point]

In [ ]:
```
#Show your code
```

Backward propagation

(e) What are the derivatives $\frac{\partial l}{\partial h^2}$, $\frac{\partial l}{\partial W^2}$, and $\frac{\partial l}{\partial b^2}$?

[3 points]

In [ ]:
```
#Show your code
```

(f) What are the derivatives $\frac{\partial l}{\partial h^1}$, $\frac{\partial l}{\partial \bar{h}^1}$, $\frac{\partial l}{\partial W^1}$, and $\frac{\partial l}{\partial b^1}$?

[3 points]

In [ ]:
```
#Show your code
```

SGD update

(g) Assume that we use SGD with learning rate $\eta = 0.01$ to update the model parameters. What are the values of W^2, b^2 and W^1, b^1 after updating?

[2 points]

In [ ]:
```
#Show your code
```

Option 2

[Total marks for this option: 20 points]

In [ ]:
```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

In Option 2, you need to implement a feed-forward NN manually using PyTorch and auto-differentiation of PyTorch. We then manually train the model on the MNIST dataset.

We first download the `MNIST` dataset and preprocess it.

In [ ]:
```
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert the image to a tensor with shape [C, H, W]
    transforms.Normalize((0.5,), (0.5,)),  # Normalize to [-1, 1]

    transforms.Lambda(lambda x: x.view(28*28)) # Flatten the tensor to shape [-1,HW
])

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=

train_data, train_labels = train_dataset.data, train_dataset.targets
test_data, test_labels = test_dataset.data, test_dataset.targets
print(train_data.shape, train_labels.shape)
print(test_data.shape, test_labels.shape)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.
gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.
gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:10<00:00, 905170.64it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.
gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.
gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 57396.27it/s]
```

```
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.g
z
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.g
z to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:01<00:00, 1244889.26it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.g
z
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.g
z to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 5063936.41it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

torch.Size([60000, 28, 28]) torch.Size([60000])
torch.Size([10000, 28, 28]) torch.Size([10000])
```

Each data point has dimension `[28,28]`. We need to flatten it to a vector to input to our FFN.

In [ ]:
```
train_dataset.data = train_data.data.view(-1, 28*28)
test_dataset.data = test_data.data.view(-1, 28*28)

train_data, train_labels = train_dataset.data, train_dataset.targets
test_data, test_labels = test_dataset.data, test_dataset.targets
print(train_data.shape, train_labels.shape)
print(test_data.shape, test_labels.shape)
```

```
torch.Size([60000, 784]) torch.Size([60000])
torch.Size([10000, 784]) torch.Size([10000])
```

In [ ]:
```
train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)
```

Develop the feed-forward neural networks

(a) You need to develop the class `MyLinear` with the following skeleton

[3 points]

In [ ]:
```
class MyLinear(torch.nn.Module):
  def __init__(self, input_size, output_size):
    """
    input_size: the size of the input
    output_size: the size of the output
    """
    super().__init__()
    #Your code here
    self.input_size = input_size
    self.output_size = output_size
    # torch.nn.Parameter assigns random weights to the tensor and the gradients wil
```

```
    self.weights = torch.nn.Parameter(torch.randn(input_size, output_size))
    self.biases = torch.nn.Parameter(torch.randn(output_size))

  #forward propagation
  def forward(self, x): #x is a mini-batch
    #Your code here
    return torch.mm(x, self.weights) + self.biases

  # String representation method of MyLinear class
  def __repr__(self):
        return f"MyLinear(in_features={self.input_size}, out_features={self.output_
```

(b) You need to develop the class `MyFFN` with the following skeleton

[7 points]

In [ ]:
```
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
```

In [ ]:
```
class MyFFN(torch.nn.Module):
  def __init__(self, input_size, num_classes, hidden_sizes, act = torch.nn.ReLU()):
    """
    input_size: the size of the input
    num_classes: the number of classes
    act is the activation function
    hidden_sizes is the list of hidden sizes
    for example input_size = 3, hidden_sizes = [5, 7], num_classes = 4, and act = t
    means that we are building up a FFN with the confirguration
    (3 (Input) -> 5 (ReLU) -> 7 (ReLU) -> 4 (Output))
    """
    super(MyFFN,self).__init__()
    self.input_size = input_size
    self.number_of_classes = num_classes
    self.activation_function = act
    self.hidden_sizes = hidden_sizes


  def create_FFN(self):
    """
    This function creates the feed-forward neural network
    We stack many MyLinear layers
    """
    layers = []
    previous_input_size = self.input_size

    for hidden_size in self.hidden_sizes:
        layers.append(MyLinear(previous_input_size, hidden_size))
        layers.append(self.activation_function)
        previous_input_size = hidden_size

    layers.append(MyLinear(previous_input_size, self.number_of_classes))
    # Making the FFN a sequential model
    self.layers = nn.Sequential(*layers)


  def forward(self,x):
    """
    This implements the forward propagation of the batch x
    This needs to return the prediction probabilities of x
    """
    #Your code here
    return self.layers(x)
```

```
  def compute_loss(self, x, y):
    """
    This function computes the cross-entropy loss
    """
    #Your code here
    pred = self.forward(x)
    loss_function = nn.CrossEntropyLoss()  # Instantiate loss function
    loss = loss_function(pred, y)  # Compute loss
    return loss

  def update_SGD(self, x, y, learning_rate = 0.01):
    """
    This function updates the model parameters using SGD using the batch (x,y)
    """
    #Your code here
    sgd_optimizer = torch.optim.SGD(self.parameters(), lr=learning_rate)
    # clear gradients and set the back to zero
    sgd_optimizer.zero_grad()
    # Calculates the current CE loss of the batch
    current_loss = self.compute_loss(x, y)
    current_loss.backward()
    sgd_optimizer.step()    # Updates the parameters based on the gradients compute

  def update_SGDwithMomentum(self, x, y, learning_rate = 0.01, momentum = 0.9):
    """
    This function updates the model parameters using SGD with momentum using the ba
    """
    #Your code here
    momentum_sgd_optimizer = torch.optim.SGD(self.parameters(), lr=learning_rate, m
    momentum_sgd_optimizer.zero_grad()
    current_loss = self.compute_loss(x, y)
    current_loss.backward()
    momentum_sgd_optimizer.step()

  def update_AdaGrad(self, x, y, learning_rate = 0.01):
    """
    This function updates the model parameters using AdaGrad using the batch (x,y)
    """
    #Your code here
    ada_optimizer = torch.optim.Adagrad(self.parameters(), lr=learning_rate)
    ada_optimizer.zero_grad()
    current_loss = self.compute_loss(x, y)
    current_loss.backward()
    ada_optimizer.step()
```

In [ ]:
```
myFFN = MyFFN(input_size = 28*28, num_classes = 26, hidden_sizes = [100, 100], act
myFFN.create_FFN()
print(myFFN)
```

```
MyFFN(
  (activation_function): ReLU()
  (layers): Sequential(
    (0): MyLinear(in_features=784, out_features=100)
    (1): ReLU()
    (2): MyLinear(in_features=100, out_features=100)
    (3): ReLU()
    (4): MyLinear(in_features=100, out_features=26)
  )
)
```

(c) Write the code to evaluate the accuracy of the current `myFFN` model on a data loader (e.g., train_loader or test_loader).

[2.5 points]

In [ ]:
```
# This code is obtained from WEEK3B Tutorial
def compute_acc(model, data_loader):
    """
    This function computes the accuracy of the model on a data loader
    """
    correct = 0
    total = 0
    model.eval()  # Set the model to evaluation mode

     # Disable the gradient calculation
    for batchX, batchY in data_loader:
        outputs = model(batchX.type(torch.float32)) # Get the predictions
        total += batchY.size(0)  # Obtain the total number of samples
        predicted_class = torch.argmax(outputs, 1)  # Get the predicted class
        correct += (predicted_class == batchY.type(torch.long)).sum().item()  # Cou

    return correct / total  # Return  accuracy
```

(c) Write the code to evaluate the loss of the current `myFFN` model on a data loader (e.g., train_loader or test_loader).

[2.5 points]

In [ ]:
```
loss_fn = nn.CrossEntropyLoss()
```

In [ ]:
```
# This code is obtained from WEEK3B Tutorial
def compute_loss(model, data_loader):
  """
  This function computes the loss of the model on a data loader
  """
  #Your code here
  total_loss = 0
  total_samples = 0
  model.eval()

  with torch.no_grad(): # No need to track the gradients
    for batchX , batchY in data_loader:
      loss = model.compute_loss(batchX, batchY) # get loss for current batch
      total_loss += loss.item() * batchX.size(0) # accumulate the loss for the curr
      total_samples += batchX.size(0) # update the count of total samples that has

  # Set back model to training mode
  model.train()
  return total_loss / total_samples
```

Train on the `MNIST` data with 50 epochs using `updateSGD`.

In [ ]:
```
num_epochs = 50
for epoch in range(num_epochs):
    for i, (x, y) in enumerate(train_loader):
      myFFN.update_SGD(x, y, learning_rate = 0.01)
    train_acc = compute_acc(myFFN, train_loader)
    train_loss = compute_loss(myFFN, train_loader)
    test_acc = compute_acc(myFFN, test_loader)
    test_loss = compute_loss(myFFN, test_loader)
    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Acc:
```

```
Epoch 1/50, Train Loss: 2.5927, Train Acc: 21.88%, Test Loss: 2.6737, Test Acc: 2
1.87%
Epoch 2/50, Train Loss: 2.5076, Train Acc: 26.16%, Test Loss: 2.5655, Test Acc: 2
6.18%
Epoch 3/50, Train Loss: 2.6735, Train Acc: 20.42%, Test Loss: 2.7393, Test Acc: 2
0.44%
Epoch 4/50, Train Loss: 2.0682, Train Acc: 23.56%, Test Loss: 2.1610, Test Acc: 2
3.50%
Epoch 5/50, Train Loss: 1.8917, Train Acc: 32.90%, Test Loss: 1.9715, Test Acc: 3
3.16%
Epoch 6/50, Train Loss: 1.8338, Train Acc: 34.44%, Test Loss: 1.9061, Test Acc: 3
4.79%
Epoch 7/50, Train Loss: 1.7945, Train Acc: 35.67%, Test Loss: 1.8767, Test Acc: 3
5.89%
Epoch 8/50, Train Loss: 1.7367, Train Acc: 36.13%, Test Loss: 1.8097, Test Acc: 3
6.75%
Epoch 9/50, Train Loss: 1.9138, Train Acc: 28.87%, Test Loss: 1.9580, Test Acc: 2
9.48%
Epoch 10/50, Train Loss: 1.7301, Train Acc: 37.76%, Test Loss: 1.8032, Test Acc: 3
8.15%
Epoch 11/50, Train Loss: 1.6922, Train Acc: 36.31%, Test Loss: 1.7744, Test Acc: 3
7.16%
Epoch 12/50, Train Loss: 1.6786, Train Acc: 39.50%, Test Loss: 1.7663, Test Acc: 3
9.98%
Epoch 13/50, Train Loss: 1.6215, Train Acc: 35.87%, Test Loss: 1.7215, Test Acc: 3
5.41%
Epoch 14/50, Train Loss: 1.5869, Train Acc: 42.81%, Test Loss: 1.6764, Test Acc: 4
2.82%
Epoch 15/50, Train Loss: 1.5562, Train Acc: 43.41%, Test Loss: 1.6454, Test Acc: 4
4.44%
Epoch 16/50, Train Loss: 1.5502, Train Acc: 44.26%, Test Loss: 1.6373, Test Acc: 4
4.88%
Epoch 17/50, Train Loss: 1.5646, Train Acc: 43.61%, Test Loss: 1.6385, Test Acc: 4
3.86%
Epoch 18/50, Train Loss: 1.5457, Train Acc: 45.56%, Test Loss: 1.6322, Test Acc: 4
5.86%
Epoch 19/50, Train Loss: 1.4869, Train Acc: 47.05%, Test Loss: 1.5890, Test Acc: 4
7.18%
Epoch 20/50, Train Loss: 1.4772, Train Acc: 48.25%, Test Loss: 1.5720, Test Acc: 4
8.75%
Epoch 21/50, Train Loss: 1.4533, Train Acc: 48.65%, Test Loss: 1.5562, Test Acc: 4
8.95%
Epoch 22/50, Train Loss: 1.4349, Train Acc: 49.50%, Test Loss: 1.5227, Test Acc: 4
9.38%
Epoch 23/50, Train Loss: 1.4232, Train Acc: 49.69%, Test Loss: 1.5080, Test Acc: 5
0.29%
Epoch 24/50, Train Loss: 1.4191, Train Acc: 51.03%, Test Loss: 1.5396, Test Acc: 5
1.34%
Epoch 25/50, Train Loss: 1.3925, Train Acc: 51.10%, Test Loss: 1.5055, Test Acc: 5
0.99%
Epoch 26/50, Train Loss: 1.4133, Train Acc: 51.80%, Test Loss: 1.5323, Test Acc: 5
2.37%
Epoch 27/50, Train Loss: 1.3940, Train Acc: 46.57%, Test Loss: 1.5189, Test Acc: 4
5.91%
Epoch 28/50, Train Loss: 1.3174, Train Acc: 54.35%, Test Loss: 1.4425, Test Acc: 5
4.10%
Epoch 29/50, Train Loss: 1.3015, Train Acc: 55.11%, Test Loss: 1.4466, Test Acc: 5
4.73%
Epoch 30/50, Train Loss: 1.2773, Train Acc: 55.74%, Test Loss: 1.4147, Test Acc: 5
6.00%
Epoch 31/50, Train Loss: 1.2488, Train Acc: 56.21%, Test Loss: 1.3839, Test Acc: 5
5.90%
Epoch 32/50, Train Loss: 1.2425, Train Acc: 56.53%, Test Loss: 1.3823, Test Acc: 5
6.75%
```

```
Epoch 33/50, Train Loss: 1.2278, Train Acc: 57.26%, Test Loss: 1.3659, Test Acc: 5
7.68%
Epoch 34/50, Train Loss: 1.2282, Train Acc: 56.94%, Test Loss: 1.3651, Test Acc: 5
6.66%
Epoch 35/50, Train Loss: 1.2066, Train Acc: 57.81%, Test Loss: 1.3427, Test Acc: 5
8.27%
Epoch 36/50, Train Loss: 1.2419, Train Acc: 57.05%, Test Loss: 1.3727, Test Acc: 5
6.61%
Epoch 37/50, Train Loss: 1.1824, Train Acc: 58.40%, Test Loss: 1.3248, Test Acc: 5
8.49%
Epoch 38/50, Train Loss: 1.1912, Train Acc: 59.05%, Test Loss: 1.3187, Test Acc: 5
9.31%
Epoch 39/50, Train Loss: 1.1843, Train Acc: 58.76%, Test Loss: 1.3078, Test Acc: 5
8.93%
Epoch 40/50, Train Loss: 1.1326, Train Acc: 59.84%, Test Loss: 1.2714, Test Acc: 5
9.72%
Epoch 41/50, Train Loss: 1.1194, Train Acc: 61.20%, Test Loss: 1.2497, Test Acc: 6
1.09%
Epoch 42/50, Train Loss: 1.1201, Train Acc: 60.63%, Test Loss: 1.2465, Test Acc: 6
0.79%
Epoch 43/50, Train Loss: 1.1745, Train Acc: 61.64%, Test Loss: 1.2712, Test Acc: 6
0.97%
Epoch 44/50, Train Loss: 1.1341, Train Acc: 62.21%, Test Loss: 1.2616, Test Acc: 6
1.92%
Epoch 45/50, Train Loss: 1.0756, Train Acc: 62.33%, Test Loss: 1.2144, Test Acc: 6
2.22%
Epoch 46/50, Train Loss: 1.0709, Train Acc: 62.84%, Test Loss: 1.2076, Test Acc: 6
2.63%
Epoch 47/50, Train Loss: 1.0695, Train Acc: 62.36%, Test Loss: 1.2106, Test Acc: 6
1.78%
Epoch 48/50, Train Loss: 1.0548, Train Acc: 63.25%, Test Loss: 1.1852, Test Acc: 6
2.71%
Epoch 49/50, Train Loss: 1.0378, Train Acc: 63.54%, Test Loss: 1.1831, Test Acc: 6
3.43%
Epoch 50/50, Train Loss: 1.0890, Train Acc: 63.34%, Test Loss: 1.2169, Test Acc: 6
2.53%
```

(d) Implement the function `updateSGDMomentum` in the class and train the model with this optimizer in `50` epochs. You can update the corresponding function in the `MyFNN` class.

[2.5 points]

In [ ]:
```python
#Your code here
#Your code here
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
myFFN.to(device)

num_epochs = 50
for epoch in range(num_epochs):
    for i, (x, y) in enumerate(train_loader):
      myFFN.update_SGDwithMomentum(x, y, learning_rate = 0.05)
    train_acc = compute_acc(myFFN, train_loader)
    train_loss = compute_loss(myFFN, train_loader)
    test_acc = compute_acc(myFFN, test_loader)
    test_loss = compute_loss(myFFN, test_loader)
    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Acc:
```

```
Epoch 1/50, Train Loss: 0.7893, Train Acc: 74.37%, Test Loss: 0.7761, Test Acc: 7
4.78%
Epoch 2/50, Train Loss: 0.8159, Train Acc: 73.72%, Test Loss: 0.7916, Test Acc: 7
4.14%
Epoch 3/50, Train Loss: 0.7117, Train Acc: 77.95%, Test Loss: 0.6976, Test Acc: 7
8.33%
Epoch 4/50, Train Loss: 0.7011, Train Acc: 78.24%, Test Loss: 0.6848, Test Acc: 7
8.56%
Epoch 5/50, Train Loss: 0.6754, Train Acc: 78.45%, Test Loss: 0.6655, Test Acc: 7
8.58%
Epoch 6/50, Train Loss: 0.6980, Train Acc: 77.50%, Test Loss: 0.6821, Test Acc: 7
7.73%
Epoch 7/50, Train Loss: 0.6937, Train Acc: 76.93%, Test Loss: 0.7048, Test Acc: 7
6.54%
Epoch 8/50, Train Loss: 0.6469, Train Acc: 79.78%, Test Loss: 0.6419, Test Acc: 7
9.81%
Epoch 9/50, Train Loss: 0.7039, Train Acc: 77.22%, Test Loss: 0.7017, Test Acc: 7
7.75%
Epoch 10/50, Train Loss: 0.6274, Train Acc: 80.00%, Test Loss: 0.6201, Test Acc: 8
0.70%
Epoch 11/50, Train Loss: 0.6219, Train Acc: 79.53%, Test Loss: 0.6048, Test Acc: 7
9.94%
Epoch 12/50, Train Loss: 0.6726, Train Acc: 78.41%, Test Loss: 0.6514, Test Acc: 7
8.87%
Epoch 13/50, Train Loss: 0.5944, Train Acc: 80.69%, Test Loss: 0.5831, Test Acc: 8
1.18%
Epoch 14/50, Train Loss: 0.5689, Train Acc: 82.10%, Test Loss: 0.5670, Test Acc: 8
2.25%
Epoch 15/50, Train Loss: 0.5588, Train Acc: 82.50%, Test Loss: 0.5548, Test Acc: 8
2.59%
Epoch 16/50, Train Loss: 0.5399, Train Acc: 83.09%, Test Loss: 0.5434, Test Acc: 8
3.15%
Epoch 17/50, Train Loss: 0.5555, Train Acc: 82.93%, Test Loss: 0.5530, Test Acc: 8
3.17%
Epoch 18/50, Train Loss: 0.6113, Train Acc: 80.28%, Test Loss: 0.6140, Test Acc: 8
0.36%
Epoch 19/50, Train Loss: 0.5435, Train Acc: 83.14%, Test Loss: 0.5521, Test Acc: 8
2.92%
Epoch 20/50, Train Loss: 0.5319, Train Acc: 83.47%, Test Loss: 0.5363, Test Acc: 8
3.42%
Epoch 21/50, Train Loss: 0.5333, Train Acc: 83.26%, Test Loss: 0.5308, Test Acc: 8
3.41%
Epoch 22/50, Train Loss: 0.5061, Train Acc: 84.29%, Test Loss: 0.5138, Test Acc: 8
4.29%
Epoch 23/50, Train Loss: 0.5089, Train Acc: 84.17%, Test Loss: 0.5137, Test Acc: 8
4.25%
Epoch 24/50, Train Loss: 0.5174, Train Acc: 83.87%, Test Loss: 0.5156, Test Acc: 8
4.10%
Epoch 25/50, Train Loss: 0.4938, Train Acc: 84.83%, Test Loss: 0.4891, Test Acc: 8
5.04%
Epoch 26/50, Train Loss: 0.5462, Train Acc: 82.49%, Test Loss: 0.5597, Test Acc: 8
2.43%
Epoch 27/50, Train Loss: 0.5607, Train Acc: 83.22%, Test Loss: 0.5591, Test Acc: 8
3.10%
Epoch 28/50, Train Loss: 0.4703, Train Acc: 85.69%, Test Loss: 0.4712, Test Acc: 8
5.69%
Epoch 29/50, Train Loss: 0.4715, Train Acc: 85.87%, Test Loss: 0.4753, Test Acc: 8
5.79%
Epoch 30/50, Train Loss: 0.4737, Train Acc: 85.73%, Test Loss: 0.4826, Test Acc: 8
5.52%
Epoch 31/50, Train Loss: 0.4646, Train Acc: 85.82%, Test Loss: 0.4654, Test Acc: 8
5.68%
Epoch 32/50, Train Loss: 0.4796, Train Acc: 85.21%, Test Loss: 0.4842, Test Acc: 8
5.28%
```

```
Epoch 33/50, Train Loss: 0.4636, Train Acc: 85.85%, Test Loss: 0.4701, Test Acc: 8
5.51%
Epoch 34/50, Train Loss: 0.4529, Train Acc: 86.18%, Test Loss: 0.4621, Test Acc: 8
6.19%
Epoch 35/50, Train Loss: 0.4655, Train Acc: 85.76%, Test Loss: 0.4805, Test Acc: 8
5.41%
Epoch 36/50, Train Loss: 0.4385, Train Acc: 86.50%, Test Loss: 0.4491, Test Acc: 8
6.31%
Epoch 37/50, Train Loss: 0.4502, Train Acc: 86.19%, Test Loss: 0.4541, Test Acc: 8
5.96%
Epoch 38/50, Train Loss: 0.4557, Train Acc: 85.97%, Test Loss: 0.4607, Test Acc: 8
5.50%
Epoch 39/50, Train Loss: 0.4310, Train Acc: 86.78%, Test Loss: 0.4414, Test Acc: 8
6.68%
Epoch 40/50, Train Loss: 0.4436, Train Acc: 86.39%, Test Loss: 0.4419, Test Acc: 8
6.35%
Epoch 41/50, Train Loss: 0.4991, Train Acc: 84.61%, Test Loss: 0.5021, Test Acc: 8
4.46%
Epoch 42/50, Train Loss: 0.4333, Train Acc: 86.97%, Test Loss: 0.4407, Test Acc: 8
6.93%
Epoch 43/50, Train Loss: 0.4567, Train Acc: 86.24%, Test Loss: 0.4591, Test Acc: 8
6.04%
Epoch 44/50, Train Loss: 0.4301, Train Acc: 87.00%, Test Loss: 0.4413, Test Acc: 8
6.68%
Epoch 45/50, Train Loss: 0.4521, Train Acc: 86.29%, Test Loss: 0.4588, Test Acc: 8
6.23%
Epoch 46/50, Train Loss: 0.4202, Train Acc: 87.38%, Test Loss: 0.4248, Test Acc: 8
7.54%
Epoch 47/50, Train Loss: 0.4286, Train Acc: 86.86%, Test Loss: 0.4375, Test Acc: 8
6.75%
Epoch 48/50, Train Loss: 0.4305, Train Acc: 87.14%, Test Loss: 0.4397, Test Acc: 8
6.40%
Epoch 49/50, Train Loss: 0.4180, Train Acc: 87.41%, Test Loss: 0.4303, Test Acc: 8
6.99%
Epoch 50/50, Train Loss: 0.4281, Train Acc: 87.03%, Test Loss: 0.4370, Test Acc: 8
6.57%
```

(e) Implement the function `updateAdagrad` in the class and train the model with this optimizer in `50` epochs. You can update the corresponding function in the `MyFNN` class.

[2.5 points]

In [ ]:
```python
#Your code here
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
myFFN.to(device)

num_epochs = 50
for epoch in range(num_epochs):
    for i, (x, y) in enumerate(train_loader):
      myFFN.update_AdaGrad(x, y, learning_rate = 0.001)
    train_acc = compute_acc(myFFN, train_loader)
    train_loss = compute_loss(myFFN, train_loader)
    test_acc = compute_acc(myFFN, test_loader)
    test_loss = compute_loss(myFFN, test_loader)
    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Train Acc:
```

```
Epoch 1/50, Train Loss: 0.6316, Train Acc: 86.80%, Test Loss: 0.6626, Test Acc: 8
6.33%
Epoch 2/50, Train Loss: 0.8358, Train Acc: 85.22%, Test Loss: 0.8898, Test Acc: 8
5.32%
Epoch 3/50, Train Loss: 1.0602, Train Acc: 83.49%, Test Loss: 1.1350, Test Acc: 8
3.59%
Epoch 4/50, Train Loss: 1.3024, Train Acc: 82.78%, Test Loss: 1.4407, Test Acc: 8
2.56%
Epoch 5/50, Train Loss: 1.3579, Train Acc: 78.87%, Test Loss: 1.4633, Test Acc: 7
9.05%
Epoch 6/50, Train Loss: 1.4937, Train Acc: 76.70%, Test Loss: 1.5943, Test Acc: 7
6.83%
Epoch 7/50, Train Loss: 1.4133, Train Acc: 79.81%, Test Loss: 1.5297, Test Acc: 8
0.74%
Epoch 8/50, Train Loss: 1.7865, Train Acc: 71.77%, Test Loss: 1.8934, Test Acc: 7
2.40%
Epoch 9/50, Train Loss: 1.9511, Train Acc: 79.11%, Test Loss: 2.1911, Test Acc: 7
9.55%
Epoch 10/50, Train Loss: 2.1134, Train Acc: 80.94%, Test Loss: 2.2468, Test Acc: 8
1.68%
Epoch 11/50, Train Loss: 2.2809, Train Acc: 78.08%, Test Loss: 2.4435, Test Acc: 7
8.39%
Epoch 12/50, Train Loss: 2.3168, Train Acc: 76.91%, Test Loss: 2.5057, Test Acc: 7
7.00%
Epoch 13/50, Train Loss: 2.2779, Train Acc: 77.89%, Test Loss: 2.4558, Test Acc: 7
7.64%
Epoch 14/50, Train Loss: 2.2571, Train Acc: 74.94%, Test Loss: 2.4338, Test Acc: 7
5.19%
Epoch 15/50, Train Loss: 1.9382, Train Acc: 78.93%, Test Loss: 2.1054, Test Acc: 7
9.42%
Epoch 16/50, Train Loss: 1.8516, Train Acc: 75.97%, Test Loss: 2.0315, Test Acc: 7
6.29%
Epoch 17/50, Train Loss: 1.9928, Train Acc: 75.75%, Test Loss: 2.2633, Test Acc: 7
6.35%
Epoch 18/50, Train Loss: 1.7103, Train Acc: 76.26%, Test Loss: 1.9394, Test Acc: 7
6.83%
Epoch 19/50, Train Loss: 1.9854, Train Acc: 78.37%, Test Loss: 2.2942, Test Acc: 7
8.81%
Epoch 20/50, Train Loss: 2.0646, Train Acc: 75.62%, Test Loss: 2.2769, Test Acc: 7
5.79%
Epoch 21/50, Train Loss: 1.9066, Train Acc: 76.26%, Test Loss: 2.2222, Test Acc: 7
6.82%
Epoch 22/50, Train Loss: 2.4741, Train Acc: 79.87%, Test Loss: 2.8400, Test Acc: 8
0.21%
Epoch 23/50, Train Loss: 2.1099, Train Acc: 77.87%, Test Loss: 2.5008, Test Acc: 7
7.84%
Epoch 24/50, Train Loss: 2.4022, Train Acc: 78.27%, Test Loss: 2.8250, Test Acc: 7
8.81%
Epoch 25/50, Train Loss: 1.9244, Train Acc: 74.74%, Test Loss: 2.3581, Test Acc: 7
4.76%
Epoch 26/50, Train Loss: 2.8078, Train Acc: 74.12%, Test Loss: 3.4675, Test Acc: 7
3.94%
Epoch 27/50, Train Loss: 2.8388, Train Acc: 74.27%, Test Loss: 3.4992, Test Acc: 7
4.74%
Epoch 28/50, Train Loss: 1.7971, Train Acc: 72.99%, Test Loss: 2.3199, Test Acc: 7
2.86%
Epoch 29/50, Train Loss: 2.8494, Train Acc: 73.28%, Test Loss: 3.5209, Test Acc: 7
3.89%
Epoch 30/50, Train Loss: 2.5550, Train Acc: 73.05%, Test Loss: 3.0572, Test Acc: 7
3.67%
Epoch 31/50, Train Loss: 2.6272, Train Acc: 79.61%, Test Loss: 3.6039, Test Acc: 7
9.17%
Epoch 32/50, Train Loss: 3.1041, Train Acc: 72.85%, Test Loss: 4.0168, Test Acc: 7
3.58%
```

```
Epoch 33/50, Train Loss: 2.5061, Train Acc: 78.92%, Test Loss: 3.3905, Test Acc: 7
8.74%
Epoch 34/50, Train Loss: 3.3912, Train Acc: 73.12%, Test Loss: 3.9905, Test Acc: 7
3.59%
Epoch 35/50, Train Loss: 2.9683, Train Acc: 76.41%, Test Loss: 4.0687, Test Acc: 7
6.36%
Epoch 36/50, Train Loss: 2.4581, Train Acc: 76.76%, Test Loss: 3.1612, Test Acc: 7
7.18%
Epoch 37/50, Train Loss: 2.9308, Train Acc: 77.08%, Test Loss: 3.8661, Test Acc: 7
7.21%
Epoch 38/50, Train Loss: 2.2982, Train Acc: 80.56%, Test Loss: 3.2300, Test Acc: 8
0.25%
Epoch 39/50, Train Loss: 2.4370, Train Acc: 74.18%, Test Loss: 3.3833, Test Acc: 7
4.35%
Epoch 40/50, Train Loss: 3.3733, Train Acc: 74.25%, Test Loss: 4.4418, Test Acc: 7
4.60%
Epoch 41/50, Train Loss: 2.8858, Train Acc: 74.78%, Test Loss: 3.6347, Test Acc: 7
5.20%
Epoch 42/50, Train Loss: 4.2526, Train Acc: 80.98%, Test Loss: 5.1840, Test Acc: 8
0.95%
Epoch 43/50, Train Loss: 5.7809, Train Acc: 81.23%, Test Loss: 6.4078, Test Acc: 8
1.27%
Epoch 44/50, Train Loss: 5.7643, Train Acc: 82.44%, Test Loss: 6.5181, Test Acc: 8
2.91%
Epoch 45/50, Train Loss: 5.1115, Train Acc: 77.27%, Test Loss: 6.2681, Test Acc: 7
7.85%
Epoch 46/50, Train Loss: 7.9303, Train Acc: 74.39%, Test Loss: 8.8067, Test Acc: 7
4.79%
Epoch 47/50, Train Loss: 7.1828, Train Acc: 80.88%, Test Loss: 8.6053, Test Acc: 8
1.52%
Epoch 48/50, Train Loss: 6.4062, Train Acc: 76.83%, Test Loss: 7.6891, Test Acc: 7
7.15%
Epoch 49/50, Train Loss: 6.1829, Train Acc: 83.67%, Test Loss: 7.6885, Test Acc: 8
3.59%
Epoch 50/50, Train Loss: 8.0389, Train Acc: 74.47%, Test Loss: 8.7539, Test Acc: 7
4.25%
```

Part 2: Deep Neural Networks (DNN)

[Total marks for this part: 25 points]

The second part of this assignment is to demonstrate your basis knowledge in deep learning that you have acquired from the lectures and tutorials materials. Most of the contents in this assignment are drawn from **the tutorials covered from weeks 1 to 2**. Going through these materials before attempting this assignment is highly recommended.

In the second part of this assignment, you are going to work with the FashionMNIST dataset for image recognition task. It has the exact same format as MNIST (70,000 grayscale images of 28 × 28 pixels each with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem is significantly more challenging than MNIST.

In [ ]:
```
import torch
import torch.optim as optim
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
```

```
torch.manual_seed(1234)
```

Out[ ]:
```
<torch._C.Generator at 0x7e6d5e371610>
```

Load the Fashion MNIST using `torchvision`

In [ ]:
```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,),

train_dataset_orgin = datasets.FashionMNIST(root='./data', train=True, download=Tru
test_dataset = datasets.FashionMNIST(root='./data', train=False, download=True, tra

print(train_dataset_orgin.data.shape, train_dataset_orgin.targets.shape)
print(test_dataset.data.shape, test_dataset.targets.shape)

train_dataset_orgin.data = train_dataset_orgin.data.view(-1, 28*28)
test_dataset.data = test_dataset.data.view(-1, 28*28)

print(train_dataset_orgin.data.shape, train_dataset_orgin.targets.shape)
print(test_dataset.data.shape, test_dataset.targets.shape)

N = len(train_dataset_orgin)
print(f"Number of training samples: {N}")
N_train = int(0.9*N)
N_val = N - N_train
print(f"Number of training samples: {N_train}")
print(f"Number of validation samples: {N_val}")

train_dataset, val_dataset = torch.utils.data.random_split(train_dataset_orgin, [N_


train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(dataset=test_dataset, batch_size=1000, shuffle=False)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-image
s-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-image
s-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 26421880/26421880 [00:03<00:00, 8587713.06it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNI
ST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-label
s-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-label
s-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 172414.83it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNI
ST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images
-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images
-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:01<00:00, 3235516.45it/s]
```

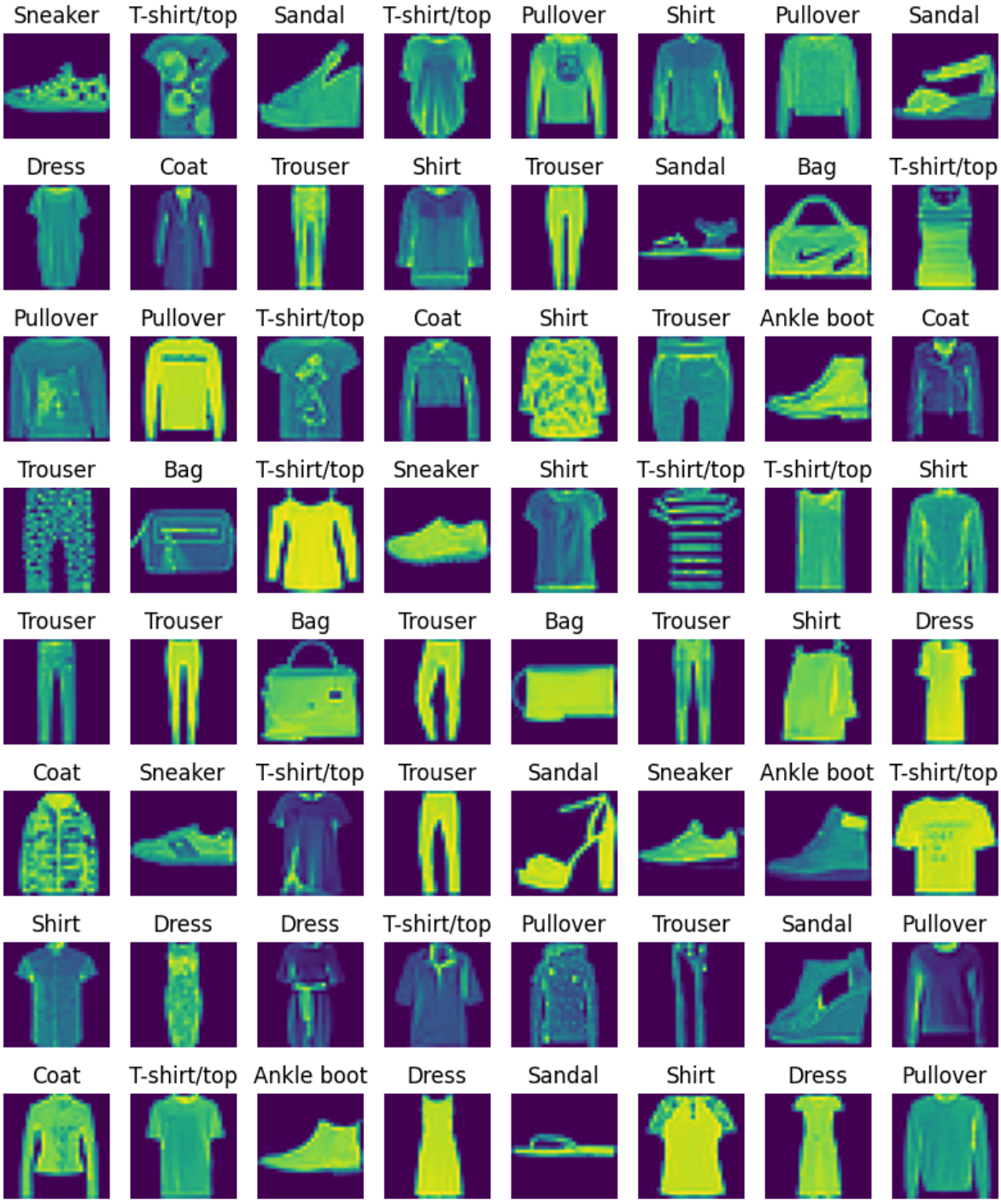
```
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIS
T/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels
-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels
-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 8200636.91it/s]
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIS
T/raw

torch.Size([60000, 28, 28]) torch.Size([60000])
torch.Size([10000, 28, 28]) torch.Size([10000])
torch.Size([60000, 784]) torch.Size([60000])
torch.Size([10000, 784]) torch.Size([10000])
Number of training samples: 60000
Number of training samples: 54000
Number of validation samples: 6000
```

Question 2.1: Write the code to visualize a mini-batch in `train_loader` including its images and labels.

[5 points]

In [ ]:
```python
from torch.nn import Linear, Sequential
```

In [ ]:
```python
#Your code here
# This code is from week3B and I modified it a little
def visualize_images(images, labels, classes):
    plt.figure(figsize=(10, 12))
    for i in range(len(images)):
        plt.subplot(8,8, i+1) # 8 by 8
        plt.axis('off')
        # Convert the image tensor to numpy arrays for visualization
        plt.imshow(images[i].view(28,28).numpy().squeeze()) # gray colour for image
        plt.title(classes[labels[i].item()]) # Obtain each image label
    plt.show()

# This code is from week 2 tutorial
# Get a mini-batch of images and labels
dataiter = iter(train_loader)
images, labels = next(dataiter)

# Get the classes names
classes_names = train_dataset_orgin.classes
visualize_images(images, labels, classes_names)
```



Question 2.2: Write the code for the feed-forward neural net using PyTorch

[5 points]

We now develop a feed-forward neural network with the architecture $784 \rightarrow 40(ReLU) \rightarrow 30(ReLU) \rightarrow 10(softmax)$. You can choose your own way to implement your network and an optimizer of interest. You should train model in 50 epochs and evaluate the trained model on the test set.

In [ ]:
```
#Your code here
number_of_features = train_dataset_orgin.data.shape[1] # 784 (28*28)

# Get the number of classes
number_of_classes = len(np.unique(train_dataset_orgin.targets.numpy())) # 10 classe
```

```
# Print to verify
class FFN(nn.Module):
  def __init__(self, number_of_features, number_of_classes):
    super(FFN, self).__init__()
    self.layer_1 = nn.Linear(number_of_features, 40)
    self.relu1 = nn.ReLU()
    self.layer_2 = nn.Linear(40, 30)
    self.relu2 = nn.ReLU()
    self.layer_3 = nn.Linear(30, number_of_classes)
    self.softmax = nn.Softmax(dim=1)

  def forward(self, x):
    x = self.layer_1(x)
    x = self.relu1(x)
    x = self.layer_2(x)
    x = self.relu2(x)
    x = self.layer_3(x)
    x = self.softmax(x)
    return x


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
ffn_model = FFN(number_of_features, number_of_classes).to(device)

# Print the model architecture
print(ffn_model)
```

```
FFN(
  (layer_1): Linear(in_features=784, out_features=40, bias=True)
  (relu1): ReLU()
  (layer_2): Linear(in_features=40, out_features=30, bias=True)
  (relu2): ReLU()
  (layer_3): Linear(in_features=30, out_features=10, bias=True)
  (softmax): Softmax(dim=1)
)
```

In [ ]:

```
# This code is obtained from tutorial in WEEK5A for training and validating epochs

# Define the loss function and optimizer
cross_entropy_loss = nn.CrossEntropyLoss()  #  multiclass classification
optimizer = optim.Adamax(ffn_model.parameters(), lr=0.001, weight_decay=0.001)  # d

def training_epoch(model, dataloader, criterion, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in dataloader:
        images = images.view(-1, 28*28).to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    loss = running_loss / len(dataloader)
```

```
    accuracy = correct / total
    return loss, accuracy

def validating_epoch(model, dataloader, criterion, device):
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images = images.view(-1, 28*28).to(device)
            labels = labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    loss = val_loss / len(dataloader)
    accuracy = correct / total
    return loss, accuracy

# Training loop
epochs = 50
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")

    train_loss, train_acc = training_epoch(ffn_model, train_loader, cross_entropy_l
    val_loss, val_acc = validating_epoch(ffn_model, val_loader, cross_entropy_loss,

    print(f"train loss= {train_loss:.4f} - train acc= {train_acc*100:.2f}% - valid
```

```
Epoch 1/50
train loss= 1.7774 - train acc= 71.85% - valid loss= 1.7043 - valid acc= 77.25%
Epoch 2/50
train loss= 1.6829 - train acc= 79.07% - valid loss= 1.6821 - valid acc= 79.18%
Epoch 3/50
train loss= 1.6680 - train acc= 80.45% - valid loss= 1.6729 - valid acc= 79.82%
Epoch 4/50
train loss= 1.6559 - train acc= 81.76% - valid loss= 1.6561 - valid acc= 81.85%
Epoch 5/50
train loss= 1.6378 - train acc= 83.59% - valid loss= 1.6448 - valid acc= 82.57%
Epoch 6/50
train loss= 1.6288 - train acc= 84.29% - valid loss= 1.6389 - valid acc= 83.30%
Epoch 7/50
train loss= 1.6229 - train acc= 84.84% - valid loss= 1.6371 - valid acc= 83.33%
Epoch 8/50
train loss= 1.6186 - train acc= 85.27% - valid loss= 1.6353 - valid acc= 83.57%
Epoch 9/50
train loss= 1.6159 - train acc= 85.52% - valid loss= 1.6284 - valid acc= 84.12%
Epoch 10/50
train loss= 1.6133 - train acc= 85.74% - valid loss= 1.6319 - valid acc= 83.67%
Epoch 11/50
train loss= 1.6113 - train acc= 85.95% - valid loss= 1.6271 - valid acc= 84.28%
Epoch 12/50
train loss= 1.6087 - train acc= 86.20% - valid loss= 1.6245 - valid acc= 84.38%
Epoch 13/50
train loss= 1.6073 - train acc= 86.32% - valid loss= 1.6250 - valid acc= 84.47%
Epoch 14/50
train loss= 1.6063 - train acc= 86.44% - valid loss= 1.6290 - valid acc= 84.18%
Epoch 15/50
train loss= 1.6046 - train acc= 86.58% - valid loss= 1.6255 - valid acc= 84.23%
Epoch 16/50
train loss= 1.6033 - train acc= 86.74% - valid loss= 1.6181 - valid acc= 85.10%
Epoch 17/50
train loss= 1.6021 - train acc= 86.89% - valid loss= 1.6186 - valid acc= 85.22%
Epoch 18/50
train loss= 1.6009 - train acc= 86.95% - valid loss= 1.6269 - valid acc= 84.10%
Epoch 19/50
train loss= 1.6002 - train acc= 87.08% - valid loss= 1.6164 - valid acc= 85.07%
Epoch 20/50
train loss= 1.5996 - train acc= 87.00% - valid loss= 1.6159 - valid acc= 85.42%
Epoch 21/50
train loss= 1.5978 - train acc= 87.27% - valid loss= 1.6189 - valid acc= 84.82%
Epoch 22/50
train loss= 1.5975 - train acc= 87.23% - valid loss= 1.6185 - valid acc= 85.12%
Epoch 23/50
train loss= 1.5958 - train acc= 87.44% - valid loss= 1.6139 - valid acc= 85.77%
Epoch 24/50
train loss= 1.5943 - train acc= 87.65% - valid loss= 1.6132 - valid acc= 85.70%
Epoch 25/50
train loss= 1.5948 - train acc= 87.54% - valid loss= 1.6159 - valid acc= 85.20%
Epoch 26/50
train loss= 1.5930 - train acc= 87.76% - valid loss= 1.6142 - valid acc= 85.50%
Epoch 27/50
train loss= 1.5929 - train acc= 87.66% - valid loss= 1.6139 - valid acc= 85.42%
Epoch 28/50
train loss= 1.5918 - train acc= 87.88% - valid loss= 1.6161 - valid acc= 85.32%
Epoch 29/50
train loss= 1.5919 - train acc= 87.89% - valid loss= 1.6117 - valid acc= 85.60%
Epoch 30/50
train loss= 1.5906 - train acc= 88.03% - valid loss= 1.6197 - valid acc= 84.82%
Epoch 31/50
train loss= 1.5901 - train acc= 88.01% - valid loss= 1.6096 - valid acc= 85.98%
Epoch 32/50
train loss= 1.5898 - train acc= 88.19% - valid loss= 1.6132 - valid acc= 85.50%
```

```
Epoch 33/50
train loss= 1.5885 - train acc= 88.20% - valid loss= 1.6089 - valid acc= 86.28%
Epoch 34/50
train loss= 1.5878 - train acc= 88.31% - valid loss= 1.6077 - valid acc= 86.05%
Epoch 35/50
train loss= 1.5874 - train acc= 88.36% - valid loss= 1.6133 - valid acc= 85.53%
Epoch 36/50
train loss= 1.5871 - train acc= 88.37% - valid loss= 1.6117 - valid acc= 85.68%
Epoch 37/50
train loss= 1.5867 - train acc= 88.45% - valid loss= 1.6060 - valid acc= 86.13%
Epoch 38/50
train loss= 1.5859 - train acc= 88.57% - valid loss= 1.6076 - valid acc= 86.00%
Epoch 39/50
train loss= 1.5850 - train acc= 88.60% - valid loss= 1.6123 - valid acc= 85.42%
Epoch 40/50
train loss= 1.5843 - train acc= 88.67% - valid loss= 1.6078 - valid acc= 86.02%
Epoch 41/50
train loss= 1.5846 - train acc= 88.64% - valid loss= 1.6039 - valid acc= 86.23%
Epoch 42/50
train loss= 1.5843 - train acc= 88.64% - valid loss= 1.6056 - valid acc= 86.38%
Epoch 43/50
train loss= 1.5836 - train acc= 88.82% - valid loss= 1.6043 - valid acc= 86.62%
Epoch 44/50
train loss= 1.5837 - train acc= 88.65% - valid loss= 1.6074 - valid acc= 86.12%
Epoch 45/50
train loss= 1.5826 - train acc= 88.88% - valid loss= 1.6118 - valid acc= 85.60%
Epoch 46/50
train loss= 1.5823 - train acc= 88.97% - valid loss= 1.6068 - valid acc= 86.30%
Epoch 47/50
train loss= 1.5820 - train acc= 88.92% - valid loss= 1.6006 - valid acc= 86.83%
Epoch 48/50
train loss= 1.5808 - train acc= 89.09% - valid loss= 1.6028 - valid acc= 86.57%
Epoch 49/50
train loss= 1.5811 - train acc= 88.99% - valid loss= 1.6055 - valid acc= 86.28%
Epoch 50/50
train loss= 1.5805 - train acc= 89.06% - valid loss= 1.6035 - valid acc= 86.58%
```

Question 2.3: Tuning hyper-parameters with grid search

[5 points]

Assume that you need to tune the number of neurons on the first and second hidden layers $n_1 \in \{20, 40\}$, $n_2 \in \{20, 40\}$ and the used activation function $act \in \{sigmoid, tanh, relu\}$. The network has the architecture pattern $784 \rightarrow n_1(act) \rightarrow n_2(act) \rightarrow 10(softmax)$ where n_1, n_2, and act are in their grides. Write the code to tune the hyper-parameters n_1, n_2, and act. Note that you can freely choose the optimizer and learning rate of interest for this task.

In [ ]:
```
#Your code here

# New model with variable hyperparameters
class NewNN(nn.Module):
    def __init__(self, n1, n2, activation_function):
        super(NewNN, self).__init__()
        self.activation_function = activation_function
        self.fc_layer_1 = nn.Linear(784, n1)
        self.fc_layer_2 = nn.Linear(n1, n2)
        self.fc_layer_3 = nn.Linear(n2, 10)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
```

```
        x = self.activation_function(self.fc_layer_1(x))
        x = self.activation_function(self.fc_layer_2(x))
        x = self.fc_layer_3(x)
        x = self.softmax(x)
        return x

# Function to train and validate the model
def tune_hyperparameters(n1_values, n2_values, activation_functions, train_loader,
    best_model = None
    best_val_acc = 0.0
    best_parameters = {'n1': None, 'n2': None, 'activation': None}

    for n1 in n1_values:
        for n2 in n2_values:
            for activation_name, act_function in activation_functions.items():
              '''
              O(n^3) complexity :)
              '''
              print()
              print(f"Parameters: n1={n1}, n2={n2}, activation={activation_name}")

              # Initialize model, criterion, and optimizer
              model = NewNN(n1, n2, act_function).to(device)
              criterion = nn.CrossEntropyLoss()
              optimizer = optim.Adamax(model.parameters(), lr=0.001 , weight_decay=

              # Train and validate from previous functions
              for epoch in range(num_epochs):
                  train_loss, train_acc = training_epoch(model, train_loader, crite
                  val_loss, val_acc = validating_epoch(model, val_loader, criterior


                  print(f"Epoch {epoch+1}/{num_epochs} - train loss= {train_loss:.4

              # Check if this model is the best so far
              if val_acc > best_val_acc:
                  best_val_acc = val_acc
                  best_model = model
                  best_parameters = {'n1': n1, 'n2': n2, 'activation': activation_r

    return best_model, best_parameters , best_val_acc

# Hyperparameter for tuning
n1_values = [20, 40]
n2_values = [20, 40]

activation_functions = {
    'sigmoid': nn.Sigmoid(),
    'tanh': nn.Tanh(),
    'relu': nn.ReLU()
}

# Find the best hyperparameters
best_model, best_parameters , best_valid_accuracy= tune_hyperparameters(n1_values,

# Print best parameters
print()
print(f"Best parameters: {best_parameters} with validation accuracy of {best_valid_
```

```
Training with n1=20, n2=20, activation=sigmoid
Epoch 1/10 - train loss= 2.2328 - train acc= 22.64% - val loss= 2.1590 - val acc=
29.05%
Epoch 2/10 - train loss= 2.0901 - train acc= 38.88% - val loss= 2.0456 - val acc=
45.77%
Epoch 3/10 - train loss= 2.0064 - train acc= 49.49% - val loss= 1.9827 - val acc=
55.35%
Epoch 4/10 - train loss= 1.9486 - train acc= 57.53% - val loss= 1.9246 - val acc=
62.10%
Epoch 5/10 - train loss= 1.8941 - train acc= 64.16% - val loss= 1.8813 - val acc=
63.77%
Epoch 6/10 - train loss= 1.8626 - train acc= 65.04% - val loss= 1.8623 - val acc=
64.20%
Epoch 7/10 - train loss= 1.8472 - train acc= 65.35% - val loss= 1.8491 - val acc=
64.57%
Epoch 8/10 - train loss= 1.8358 - train acc= 65.68% - val loss= 1.8352 - val acc=
67.42%
Epoch 9/10 - train loss= 1.8164 - train acc= 71.93% - val loss= 1.8109 - val acc=
72.32%
Epoch 10/10 - train loss= 1.7943 - train acc= 72.92% - val loss= 1.7931 - val acc=
72.52%

Training with n1=20, n2=20, activation=tanh
Epoch 1/10 - train loss= 1.8917 - train acc= 70.64% - val loss= 1.7295 - val acc=
77.48%
Epoch 2/10 - train loss= 1.6908 - train acc= 79.69% - val loss= 1.6874 - val acc=
79.17%
Epoch 3/10 - train loss= 1.6690 - train acc= 80.69% - val loss= 1.6724 - val acc=
80.23%
Epoch 4/10 - train loss= 1.6614 - train acc= 81.20% - val loss= 1.6683 - val acc=
80.52%
Epoch 5/10 - train loss= 1.6576 - train acc= 81.54% - val loss= 1.6663 - val acc=
80.47%
Epoch 6/10 - train loss= 1.6541 - train acc= 81.83% - val loss= 1.6644 - val acc=
80.77%
Epoch 7/10 - train loss= 1.6525 - train acc= 81.95% - val loss= 1.6630 - val acc=
80.90%
Epoch 8/10 - train loss= 1.6508 - train acc= 82.08% - val loss= 1.6594 - val acc=
81.20%
Epoch 9/10 - train loss= 1.6490 - train acc= 82.26% - val loss= 1.6623 - val acc=
80.60%
Epoch 10/10 - train loss= 1.6478 - train acc= 82.44% - val loss= 1.6608 - val acc=
80.90%

Training with n1=20, n2=20, activation=relu
Epoch 1/10 - train loss= 1.8736 - train acc= 62.10% - val loss= 1.7880 - val acc=
68.87%
Epoch 2/10 - train loss= 1.7563 - train acc= 71.91% - val loss= 1.7601 - val acc=
71.27%
Epoch 3/10 - train loss= 1.7309 - train acc= 74.44% - val loss= 1.7427 - val acc=
73.03%
Epoch 4/10 - train loss= 1.6947 - train acc= 78.34% - val loss= 1.6683 - val acc=
81.07%
Epoch 5/10 - train loss= 1.6458 - train acc= 83.10% - val loss= 1.6532 - val acc=
81.92%
Epoch 6/10 - train loss= 1.6361 - train acc= 83.76% - val loss= 1.6472 - val acc=
82.55%
Epoch 7/10 - train loss= 1.6303 - train acc= 84.39% - val loss= 1.6419 - val acc=
82.98%
Epoch 8/10 - train loss= 1.6265 - train acc= 84.69% - val loss= 1.6380 - val acc=
83.58%
Epoch 9/10 - train loss= 1.6234 - train acc= 85.01% - val loss= 1.6383 - val acc=
83.17%
Epoch 10/10 - train loss= 1.6202 - train acc= 85.24% - val loss= 1.6372 - val acc=
```

```
83.42%

Training with n1=20, n2=40, activation=sigmoid
Epoch 1/10 - train loss= 2.2160 - train acc= 27.13% - val loss= 2.1112 - val acc=
38.32%
Epoch 2/10 - train loss= 2.0612 - train acc= 43.39% - val loss= 2.0220 - val acc=
54.12%
Epoch 3/10 - train loss= 1.9602 - train acc= 55.92% - val loss= 1.9323 - val acc=
55.65%
Epoch 4/10 - train loss= 1.8994 - train acc= 59.66% - val loss= 1.8737 - val acc=
67.38%
Epoch 5/10 - train loss= 1.8313 - train acc= 70.46% - val loss= 1.8088 - val acc=
72.13%
Epoch 6/10 - train loss= 1.7861 - train acc= 74.35% - val loss= 1.7794 - val acc=
75.65%
Epoch 7/10 - train loss= 1.7630 - train acc= 77.15% - val loss= 1.7593 - val acc=
77.02%
Epoch 8/10 - train loss= 1.7403 - train acc= 78.26% - val loss= 1.7413 - val acc=
77.13%
Epoch 9/10 - train loss= 1.7237 - train acc= 78.79% - val loss= 1.7303 - val acc=
77.68%
Epoch 10/10 - train loss= 1.7155 - train acc= 79.03% - val loss= 1.7249 - val acc=
77.62%

Training with n1=20, n2=40, activation=tanh
Epoch 1/10 - train loss= 1.8332 - train acc= 72.54% - val loss= 1.7014 - val acc=
77.98%
Epoch 2/10 - train loss= 1.6756 - train acc= 79.93% - val loss= 1.6749 - val acc=
79.75%
Epoch 3/10 - train loss= 1.6618 - train acc= 80.90% - val loss= 1.6711 - val acc=
79.97%
Epoch 4/10 - train loss= 1.6520 - train acc= 82.03% - val loss= 1.6535 - val acc=
82.23%
Epoch 5/10 - train loss= 1.6305 - train acc= 84.54% - val loss= 1.6379 - val acc=
83.48%
Epoch 6/10 - train loss= 1.6214 - train acc= 85.26% - val loss= 1.6333 - val acc=
83.93%
Epoch 7/10 - train loss= 1.6160 - train acc= 85.75% - val loss= 1.6277 - val acc=
84.47%
Epoch 8/10 - train loss= 1.6124 - train acc= 86.13% - val loss= 1.6263 - val acc=
84.48%
Epoch 9/10 - train loss= 1.6095 - train acc= 86.44% - val loss= 1.6245 - val acc=
84.68%
Epoch 10/10 - train loss= 1.6071 - train acc= 86.70% - val loss= 1.6231 - val acc=
84.80%

Training with n1=20, n2=40, activation=relu
Epoch 1/10 - train loss= 1.7893 - train acc= 70.43% - val loss= 1.7069 - val acc=
77.13%
Epoch 2/10 - train loss= 1.6846 - train acc= 79.12% - val loss= 1.6872 - val acc=
78.33%
Epoch 3/10 - train loss= 1.6716 - train acc= 80.05% - val loss= 1.6752 - val acc=
79.47%
Epoch 4/10 - train loss= 1.6611 - train acc= 81.16% - val loss= 1.6639 - val acc=
81.32%
Epoch 5/10 - train loss= 1.6446 - train acc= 83.02% - val loss= 1.6489 - val acc=
82.60%
Epoch 6/10 - train loss= 1.6334 - train acc= 83.89% - val loss= 1.6418 - val acc=
82.78%
Epoch 7/10 - train loss= 1.6265 - train acc= 84.60% - val loss= 1.6383 - val acc=
83.20%
Epoch 8/10 - train loss= 1.6221 - train acc= 84.94% - val loss= 1.6395 - val acc=
83.00%
Epoch 9/10 - train loss= 1.6193 - train acc= 85.17% - val loss= 1.6311 - val acc=
```

```
84.05%
Epoch 10/10 - train loss= 1.6163 - train acc= 85.52% - val loss= 1.6273 - val acc=
84.27%

Training with n1=40, n2=20, activation=sigmoid
Epoch 1/10 - train loss= 2.1982 - train acc= 36.38% - val loss= 2.0723 - val acc=
46.43%
Epoch 2/10 - train loss= 1.9898 - train acc= 57.44% - val loss= 1.9337 - val acc=
63.85%
Epoch 3/10 - train loss= 1.8897 - train acc= 64.29% - val loss= 1.8652 - val acc=
64.22%
Epoch 4/10 - train loss= 1.8470 - train acc= 64.75% - val loss= 1.8400 - val acc=
64.70%
Epoch 5/10 - train loss= 1.8198 - train acc= 69.46% - val loss= 1.8068 - val acc=
71.32%
Epoch 6/10 - train loss= 1.7906 - train acc= 72.55% - val loss= 1.7868 - val acc=
71.90%
Epoch 7/10 - train loss= 1.7759 - train acc= 72.98% - val loss= 1.7794 - val acc=
71.98%
Epoch 8/10 - train loss= 1.7680 - train acc= 73.26% - val loss= 1.7731 - val acc=
72.47%
Epoch 9/10 - train loss= 1.7640 - train acc= 73.38% - val loss= 1.7695 - val acc=
72.52%
Epoch 10/10 - train loss= 1.7612 - train acc= 73.43% - val loss= 1.7673 - val acc=
72.73%

Training with n1=40, n2=20, activation=tanh
Epoch 1/10 - train loss= 1.8702 - train acc= 71.62% - val loss= 1.7240 - val acc=
77.95%
Epoch 2/10 - train loss= 1.6891 - train acc= 79.93% - val loss= 1.6809 - val acc=
79.83%
Epoch 3/10 - train loss= 1.6663 - train acc= 80.89% - val loss= 1.6744 - val acc=
79.95%
Epoch 4/10 - train loss= 1.6579 - train acc= 81.54% - val loss= 1.6664 - val acc=
80.50%
Epoch 5/10 - train loss= 1.6535 - train acc= 81.82% - val loss= 1.6641 - val acc=
80.57%
Epoch 6/10 - train loss= 1.6508 - train acc= 82.03% - val loss= 1.6611 - val acc=
80.95%
Epoch 7/10 - train loss= 1.6484 - train acc= 82.33% - val loss= 1.6597 - val acc=
81.05%
Epoch 8/10 - train loss= 1.6469 - train acc= 82.43% - val loss= 1.6587 - val acc=
81.30%
Epoch 9/10 - train loss= 1.6450 - train acc= 82.68% - val loss= 1.6549 - val acc=
81.73%
Epoch 10/10 - train loss= 1.6444 - train acc= 82.75% - val loss= 1.6630 - val acc=
80.65%

Training with n1=40, n2=20, activation=relu
Epoch 1/10 - train loss= 1.8004 - train acc= 69.23% - val loss= 1.7049 - val acc=
77.43%
Epoch 2/10 - train loss= 1.6695 - train acc= 81.27% - val loss= 1.6675 - val acc=
81.10%
Epoch 3/10 - train loss= 1.6475 - train acc= 83.03% - val loss= 1.6555 - val acc=
82.18%
Epoch 4/10 - train loss= 1.6371 - train acc= 83.91% - val loss= 1.6478 - val acc=
82.52%
Epoch 5/10 - train loss= 1.6312 - train acc= 84.31% - val loss= 1.6415 - val acc=
82.85%
Epoch 6/10 - train loss= 1.6263 - train acc= 84.72% - val loss= 1.6386 - val acc=
83.47%
Epoch 7/10 - train loss= 1.6222 - train acc= 85.05% - val loss= 1.6366 - val acc=
83.38%
Epoch 8/10 - train loss= 1.6199 - train acc= 85.34% - val loss= 1.6339 - val acc=
```

```
83.43%
Epoch 9/10 - train loss= 1.6171 - train acc= 85.47% - val loss= 1.6354 - val acc=
83.27%
Epoch 10/10 - train loss= 1.6152 - train acc= 85.58% - val loss= 1.6280 - val acc=
84.23%

Training with n1=40, n2=40, activation=sigmoid
Epoch 1/10 - train loss= 2.1570 - train acc= 38.15% - val loss= 2.0165 - val acc=
54.90%
Epoch 2/10 - train loss= 1.9184 - train acc= 61.90% - val loss= 1.8726 - val acc=
62.72%
Epoch 3/10 - train loss= 1.8156 - train acc= 70.34% - val loss= 1.7886 - val acc=
71.53%
Epoch 4/10 - train loss= 1.7637 - train acc= 74.64% - val loss= 1.7572 - val acc=
76.27%
Epoch 5/10 - train loss= 1.7345 - train acc= 77.89% - val loss= 1.7332 - val acc=
77.30%
Epoch 6/10 - train loss= 1.7160 - train acc= 78.86% - val loss= 1.7206 - val acc=
78.13%
Epoch 7/10 - train loss= 1.7068 - train acc= 79.41% - val loss= 1.7127 - val acc=
79.05%
Epoch 8/10 - train loss= 1.7019 - train acc= 79.72% - val loss= 1.7106 - val acc=
78.67%
Epoch 9/10 - train loss= 1.6991 - train acc= 79.88% - val loss= 1.7064 - val acc=
79.20%
Epoch 10/10 - train loss= 1.6969 - train acc= 80.05% - val loss= 1.7047 - val acc=
79.47%

Training with n1=40, n2=40, activation=tanh
Epoch 1/10 - train loss= 1.7989 - train acc= 73.16% - val loss= 1.6931 - val acc=
78.63%
Epoch 2/10 - train loss= 1.6613 - train acc= 81.74% - val loss= 1.6577 - val acc=
82.05%
Epoch 3/10 - train loss= 1.6331 - train acc= 84.13% - val loss= 1.6385 - val acc=
83.68%
Epoch 4/10 - train loss= 1.6225 - train acc= 85.00% - val loss= 1.6341 - val acc=
83.88%
Epoch 5/10 - train loss= 1.6157 - train acc= 85.73% - val loss= 1.6280 - val acc=
84.18%
Epoch 6/10 - train loss= 1.6119 - train acc= 86.11% - val loss= 1.6299 - val acc=
84.43%
Epoch 7/10 - train loss= 1.6087 - train acc= 86.42% - val loss= 1.6226 - val acc=
84.88%
Epoch 8/10 - train loss= 1.6068 - train acc= 86.60% - val loss= 1.6232 - val acc=
84.78%
Epoch 9/10 - train loss= 1.6032 - train acc= 86.95% - val loss= 1.6279 - val acc=
84.32%
Epoch 10/10 - train loss= 1.6014 - train acc= 87.15% - val loss= 1.6166 - val acc=
85.78%

Training with n1=40, n2=40, activation=relu
Epoch 1/10 - train loss= 1.7601 - train acc= 73.40% - val loss= 1.6847 - val acc=
79.27%
Epoch 2/10 - train loss= 1.6563 - train acc= 82.29% - val loss= 1.6585 - val acc=
81.13%
Epoch 3/10 - train loss= 1.6391 - train acc= 83.56% - val loss= 1.6437 - val acc=
83.03%
Epoch 4/10 - train loss= 1.6305 - train acc= 84.19% - val loss= 1.6403 - val acc=
83.00%
Epoch 5/10 - train loss= 1.6255 - train acc= 84.65% - val loss= 1.6355 - val acc=
83.72%
Epoch 6/10 - train loss= 1.6219 - train acc= 84.97% - val loss= 1.6349 - val acc=
83.32%
Epoch 7/10 - train loss= 1.6183 - train acc= 85.26% - val loss= 1.6365 - val acc=
```

```
83.42%
Epoch 8/10 - train loss= 1.6161 - train acc= 85.51% - val loss= 1.6320 - val acc=
83.93%
Epoch 9/10 - train loss= 1.6136 - train acc= 85.77% - val loss= 1.6322 - val acc=
83.77%
Epoch 10/10 - train loss= 1.6115 - train acc= 85.87% - val loss= 1.6377 - val acc=
83.05%

Best parameters: {'n1': 40, 'n2': 40, 'activation': 'tanh'} with validation accura
cy of 85.78%
```

Question 2.4: Implement the loss with the form: $loss(p, y) = CE(1_y, p) + \lambda H(p)$ where $H(p) = -\sum_{i=1}^{M} p_i \log p_i$ is the entropy of p, p is the prediction probabilities of a data point x with the ground-truth label y, 1_y is an one-hot label, and $\lambda > 0$ is a trade-off parameter. Set $\lambda = 0.1$ to train a model.

[5 points]

In [ ]:
```
# Improved model with new hidden layer parameters
number_of_features = train_dataset_orgin.data.shape[1]

# Get the number of classes
number_of_classes = len(np.unique(train_dataset_orgin.targets.numpy()))

#
class FNN_improved(nn.Module):
  def __init__(self, number_of_features, number_of_classes):
    super(FNN_improved, self).__init__()
    self.layer_1 = nn.Linear(number_of_features, 40)
    self.tanh1 = nn.Tanh()
    self.layer_2 = nn.Linear(40, 40)
    self.tanh2 = nn.Tanh()
    self.layer_3 = nn.Linear(40, number_of_classes)


  def forward(self, x):
    x = self.layer_1(x)
    x = self.tanh1(x)
    x = self.layer_2(x)
    x = self.tanh2(x)
    x = self.layer_3(x)
    return x

fnn_improved = FNN_improved(number_of_features , number_of_classes)
print(fnn_improved)
```

```
FNN_improved(
  (layer_1): Linear(in_features=784, out_features=40, bias=True)
  (tanh1): Tanh()
  (layer_2): Linear(in_features=40, out_features=40, bias=True)
  (tanh2): Tanh()
  (layer_3): Linear(in_features=40, out_features=10, bias=True)
)
```

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

class LambdaLoss(nn.Module):
    def __init__(self, lambda_regularization = 0.1):
        super(LambdaLoss, self).__init__()
```

```
        self.lambda_regularization = lambda_regularization
        self.cross_entropy = nn.CrossEntropyLoss()

    def forward(self, outputs, actual_labels):
        """
        This function computes the custom loss formula given

        outputs: The logits from the network (before applying softmax).
        actual_labels: The ground truth labels for the batch (not one-hot encoded).
        """
        # Compute the cross-entropy loss
        ce_loss = self.cross_entropy(outputs, actual_labels)

        # added epsilon to avoid log(0) issues
        epsilon = 1e-10
        probabilities = F.softmax(outputs, dim=1)

        # Calculate H(p) where it is the -sum(p * log(p))
        entropy_p = -torch.sum(probabilities * torch.log(probabilities + epsilon) ,
        # Did the mean of entropy because CE does that as well , so we remain consi

        # Final loss formula
        final_loss = ce_loss + self.lambda_regularization * entropy_p

        return final_loss
```

In [ ]:

```
lambda_regularization = 0.1
new_loss_function = LambdaLoss(lambda_regularization = lambda_regularization)
optimizer = optim.Adamax(fnn_improved.parameters(), lr=0.001)

fnn_improved.to(device)

# Training loop
epochs = 50
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")

    train_loss, train_acc = training_epoch(fnn_improved, train_loader, new_loss_fur
    val_loss, val_acc = validating_epoch(fnn_improved, val_loader, new_loss_functic

    print(f"train loss= {train_loss:.4f} - train acc= {train_acc*100:.2f}% - valid
```

```
Epoch 1/50
train loss= 0.8297 - train acc= 76.24% - valid loss= 0.5979 - valid acc= 80.62%
Epoch 2/50
train loss= 0.4936 - train acc= 84.15% - valid loss= 0.4941 - valid acc= 83.97%
Epoch 3/50
train loss= 0.4400 - train acc= 85.71% - valid loss= 0.4631 - valid acc= 84.62%
Epoch 4/50
train loss= 0.4067 - train acc= 86.81% - valid loss= 0.4505 - valid acc= 85.15%
Epoch 5/50
train loss= 0.3855 - train acc= 87.46% - valid loss= 0.4299 - valid acc= 85.88%
Epoch 6/50
train loss= 0.3706 - train acc= 87.87% - valid loss= 0.4217 - valid acc= 86.27%
Epoch 7/50
train loss= 0.3567 - train acc= 88.21% - valid loss= 0.4147 - valid acc= 86.55%
Epoch 8/50
train loss= 0.3458 - train acc= 88.68% - valid loss= 0.4008 - valid acc= 87.22%
Epoch 9/50
train loss= 0.3356 - train acc= 89.04% - valid loss= 0.3945 - valid acc= 87.67%
Epoch 10/50
train loss= 0.3263 - train acc= 89.31% - valid loss= 0.3923 - valid acc= 87.50%
Epoch 11/50
train loss= 0.3194 - train acc= 89.43% - valid loss= 0.3912 - valid acc= 87.55%
Epoch 12/50
train loss= 0.3098 - train acc= 89.78% - valid loss= 0.3859 - valid acc= 87.85%
Epoch 13/50
train loss= 0.3048 - train acc= 89.91% - valid loss= 0.3937 - valid acc= 87.58%
Epoch 14/50
train loss= 0.2976 - train acc= 90.19% - valid loss= 0.3879 - valid acc= 87.82%
Epoch 15/50
train loss= 0.2920 - train acc= 90.29% - valid loss= 0.3915 - valid acc= 87.78%
Epoch 16/50
train loss= 0.2877 - train acc= 90.48% - valid loss= 0.3796 - valid acc= 88.05%
Epoch 17/50
train loss= 0.2825 - train acc= 90.66% - valid loss= 0.3859 - valid acc= 87.78%
Epoch 18/50
train loss= 0.2770 - train acc= 90.91% - valid loss= 0.3773 - valid acc= 88.05%
Epoch 19/50
train loss= 0.2720 - train acc= 91.06% - valid loss= 0.3803 - valid acc= 87.97%
Epoch 20/50
train loss= 0.2673 - train acc= 91.18% - valid loss= 0.3837 - valid acc= 88.07%
Epoch 21/50
train loss= 0.2634 - train acc= 91.24% - valid loss= 0.3838 - valid acc= 88.40%
Epoch 22/50
train loss= 0.2603 - train acc= 91.33% - valid loss= 0.3791 - valid acc= 88.20%
Epoch 23/50
train loss= 0.2551 - train acc= 91.62% - valid loss= 0.3853 - valid acc= 88.03%
Epoch 24/50
train loss= 0.2540 - train acc= 91.56% - valid loss= 0.3777 - valid acc= 88.45%
Epoch 25/50
train loss= 0.2485 - train acc= 91.78% - valid loss= 0.3782 - valid acc= 88.88%
Epoch 26/50
train loss= 0.2452 - train acc= 91.99% - valid loss= 0.3926 - valid acc= 87.95%
Epoch 27/50
train loss= 0.2429 - train acc= 92.00% - valid loss= 0.4036 - valid acc= 88.00%
Epoch 28/50
train loss= 0.2400 - train acc= 92.06% - valid loss= 0.3808 - valid acc= 88.33%
Epoch 29/50
train loss= 0.2366 - train acc= 92.19% - valid loss= 0.3879 - valid acc= 88.22%
Epoch 30/50
train loss= 0.2327 - train acc= 92.29% - valid loss= 0.3915 - valid acc= 88.37%
Epoch 31/50
train loss= 0.2314 - train acc= 92.28% - valid loss= 0.3926 - valid acc= 88.18%
Epoch 32/50
train loss= 0.2288 - train acc= 92.40% - valid loss= 0.3889 - valid acc= 88.43%
```

```
Epoch 33/50
train loss= 0.2242 - train acc= 92.56% - valid loss= 0.3947 - valid acc= 88.08%
Epoch 34/50
train loss= 0.2243 - train acc= 92.62% - valid loss= 0.3902 - valid acc= 88.35%
Epoch 35/50
train loss= 0.2199 - train acc= 92.65% - valid loss= 0.3883 - valid acc= 88.17%
Epoch 36/50
train loss= 0.2170 - train acc= 92.88% - valid loss= 0.3939 - valid acc= 88.07%
Epoch 37/50
train loss= 0.2146 - train acc= 92.94% - valid loss= 0.3950 - valid acc= 88.10%
Epoch 38/50
train loss= 0.2117 - train acc= 93.00% - valid loss= 0.4044 - valid acc= 87.72%
Epoch 39/50
train loss= 0.2094 - train acc= 93.07% - valid loss= 0.4038 - valid acc= 87.63%
Epoch 40/50
train loss= 0.2080 - train acc= 93.20% - valid loss= 0.4056 - valid acc= 87.98%
Epoch 41/50
train loss= 0.2051 - train acc= 93.23% - valid loss= 0.4005 - valid acc= 88.23%
Epoch 42/50
train loss= 0.2020 - train acc= 93.36% - valid loss= 0.4031 - valid acc= 88.33%
Epoch 43/50
train loss= 0.1991 - train acc= 93.56% - valid loss= 0.4228 - valid acc= 87.65%
Epoch 44/50
train loss= 0.1994 - train acc= 93.39% - valid loss= 0.4117 - valid acc= 88.10%
Epoch 45/50
train loss= 0.1971 - train acc= 93.63% - valid loss= 0.4054 - valid acc= 88.42%
Epoch 46/50
train loss= 0.1957 - train acc= 93.54% - valid loss= 0.4186 - valid acc= 87.92%
Epoch 47/50
train loss= 0.1930 - train acc= 93.67% - valid loss= 0.4036 - valid acc= 88.25%
Epoch 48/50
train loss= 0.1900 - train acc= 93.68% - valid loss= 0.4086 - valid acc= 88.33%
Epoch 49/50
train loss= 0.1883 - train acc= 93.91% - valid loss= 0.4166 - valid acc= 87.78%
Epoch 50/50
train loss= 0.1883 - train acc= 93.81% - valid loss= 0.4155 - valid acc= 88.23%
```

Question 2.5: Experimenting with **sharpness-aware minimization technique**

[5 points]

Sharpness-aware minimization (SAM) (i.e., link for main paper from Google Deepmind) is a simple yet but efficient technique to improve the generalization ability of deep learning models on unseen data examples. In your research or your work, you might potentially use this idea. Your task is to read the paper and implement *Sharpness-aware minimization (SAM)*. Finally, you need to apply SAM to the best architecture found in **Question 2.3**.

In [ ]:
```python
#Your code here

# To do this , we first need to calculate the gradient ascent to find
# parameters that maximise the loss withing a small neighbourhood

# Then , after calculating the worst-case parameters we perform a standard
# gradient descent to minimize the loss with respect to those parameters

# Parts of the code was obtained from the github repository of the paper provided
# such as the get_sam_gradient and global_norm method in flax_training.py

class SAM:
    def __init__(self, parameters, base_optimizer, rho=0.05, lr=0.001):
        self.parameters = list(parameters)  # List of model parameters
```

```
        self.base_optimizer = base_optimizer(self.parameters, lr=lr)  # Initialize
        self.rho = rho  # rho is the size of the pertubation

    def first_step(self):
        # Perform sharpness-aware update (gradient ascent)
        for i in self.parameters:
            if i.grad is None:
                continue
            # Calculate epsilon
            perturbation  = i.grad * (self.rho / (self.gradient_normalization() + 1
            i.data.add_(perturbation)  # Move parameters in the direction that incr

    def second_step(self):
        for i in self.parameters:
            if i.grad is None:
                continue
            perturbation  = i.grad * (self.rho / (self.gradient_normalization() + 1
            i.data.sub_(2 * perturbation)  # Revert back the parameters to optimal

    def gradient_normalization(self):
        # Compute the L2 norm of the gradients
        return torch.sqrt(sum((i.grad ** 2).sum() for i in self.parameters))

    def step(self, closure):
        # First forward-backward pass
        loss = closure()  # Run the closure
        self.base_optimizer.zero_grad()
        loss.backward()
        self.first_step()  # Perform sharpness-aware update

        # Second forward-backward pass
        loss = closure()  # Run closure again for the second step
        self.base_optimizer.zero_grad()
        loss.backward()
        self.base_optimizer.step()  # Take a regular step
        self.second_step()  # Revert the parameters
```

In [ ]:

```
from sklearn.metrics import accuracy_score


# Function to train and evaluate the model with SAM using the best configuration
def train_and_evaluate_sam(model, train_loader, val_loader, test_loader, epochs=50)
    criterion = nn.CrossEntropyLoss()  # Loss function
    base_optimizer = optim.Adamax  # Base optimizer for SAM
    sam_optimizer = SAM(model.parameters(), base_optimizer, lr=0.001, rho=0.05)

    # Train the model with SAM
    # This code combines previous training and evaluation methods with adjustments
    for epoch in range(epochs):
        model.train()
        total_loss = 0.0
        correct_train = 0
        total_train = 0

        for images, labels in train_loader:
            images = images.view(-1, 28*28)  # Flatten the images

            def forward_and_loss():
                outputs = model(images)  # Forward pass
                loss = criterion(outputs, labels)
                sam_optimizer.base_optimizer.zero_grad()  # Zero out the gradients
                return loss

            sam_optimizer.step(forward_and_loss)  # Apply SAM optimization
```

```
            loss = forward_and_loss() # Get the loss again to update the loss

            total_loss += loss.item()
            _, predicted = torch.max(model(images).data, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        train_loss = total_loss / len(train_loader)
        train_acc = 100 * correct_train / total_train

        # Validate the model
        model.eval()
        val_loss = 0.0
        correct_val = 0
        total_val = 0

        with torch.no_grad():
            for images, labels in val_loader:
                images = images.view(-1, 28*28)  # Flatten the images
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                _, predicted = torch.max(outputs.data, 1)
                total_val += labels.size(0)
                correct_val += (predicted == labels).sum().item()

        val_loss /= len(val_loader)
        val_acc = 100 * correct_val / total_val

        print(f"Epoch [{epoch + 1}/{epochs}]")
        print(f"train loss= {train_loss:.4f} - train acc= {train_acc:.2f}% - valid


model = FNN_improved(number_of_features=number_of_features , number_of_classes=numb
accuracy = train_and_evaluate_sam(model, train_loader, val_loader, test_loader, epo
```

```
Epoch [1/50]
train loss= 0.5762 - train acc= 81.82% - valid loss= 0.5020 - valid acc= 81.25%

Epoch [2/50]
train loss= 0.3910 - train acc= 86.77% - valid loss= 0.4408 - valid acc= 84.08%

Epoch [3/50]
train loss= 0.3541 - train acc= 88.02% - valid loss= 0.4174 - valid acc= 84.88%

Epoch [4/50]
train loss= 0.3321 - train acc= 88.80% - valid loss= 0.4002 - valid acc= 85.70%

Epoch [5/50]
train loss= 0.3150 - train acc= 89.35% - valid loss= 0.3990 - valid acc= 85.55%

Epoch [6/50]
train loss= 0.3042 - train acc= 89.74% - valid loss= 0.3756 - valid acc= 86.52%

Epoch [7/50]
train loss= 0.2943 - train acc= 90.07% - valid loss= 0.3712 - valid acc= 86.95%

Epoch [8/50]
train loss= 0.2860 - train acc= 90.43% - valid loss= 0.3750 - valid acc= 86.65%

Epoch [9/50]
train loss= 0.2778 - train acc= 90.74% - valid loss= 0.3739 - valid acc= 86.70%

Epoch [10/50]
train loss= 0.2728 - train acc= 90.80% - valid loss= 0.3620 - valid acc= 86.88%

Epoch [11/50]
train loss= 0.2668 - train acc= 91.01% - valid loss= 0.3662 - valid acc= 86.63%

Epoch [12/50]
train loss= 0.2616 - train acc= 91.19% - valid loss= 0.3598 - valid acc= 87.18%

Epoch [13/50]
train loss= 0.2574 - train acc= 91.36% - valid loss= 0.3563 - valid acc= 87.30%

Epoch [14/50]
train loss= 0.2526 - train acc= 91.54% - valid loss= 0.3593 - valid acc= 87.12%

Epoch [15/50]
train loss= 0.2490 - train acc= 91.73% - valid loss= 0.3540 - valid acc= 87.07%

Epoch [16/50]
train loss= 0.2448 - train acc= 91.77% - valid loss= 0.3522 - valid acc= 87.30%

Epoch [17/50]
train loss= 0.2420 - train acc= 91.84% - valid loss= 0.3507 - valid acc= 87.42%

Epoch [18/50]
train loss= 0.2386 - train acc= 91.94% - valid loss= 0.3459 - valid acc= 87.62%

Epoch [19/50]
train loss= 0.2345 - train acc= 92.19% - valid loss= 0.3470 - valid acc= 87.60%

Epoch [20/50]
train loss= 0.2316 - train acc= 92.24% - valid loss= 0.3514 - valid acc= 87.35%

Epoch [21/50]
train loss= 0.2277 - train acc= 92.47% - valid loss= 0.3455 - valid acc= 87.78%

Epoch [22/50]
```

```
train loss= 0.2249 - train acc= 92.42% - valid loss= 0.3489 - valid acc= 87.78%

Epoch [23/50]
train loss= 0.2228 - train acc= 92.56% - valid loss= 0.3457 - valid acc= 87.38%

Epoch [24/50]
train loss= 0.2201 - train acc= 92.70% - valid loss= 0.3453 - valid acc= 87.65%

Epoch [25/50]
train loss= 0.2176 - train acc= 92.84% - valid loss= 0.3432 - valid acc= 87.70%

Epoch [26/50]
train loss= 0.2146 - train acc= 92.93% - valid loss= 0.3448 - valid acc= 88.00%

Epoch [27/50]
train loss= 0.2128 - train acc= 93.01% - valid loss= 0.3400 - valid acc= 87.73%

Epoch [28/50]
train loss= 0.2098 - train acc= 93.09% - valid loss= 0.3417 - valid acc= 87.73%

Epoch [29/50]
train loss= 0.2087 - train acc= 93.17% - valid loss= 0.3379 - valid acc= 88.10%

Epoch [30/50]
train loss= 0.2057 - train acc= 93.22% - valid loss= 0.3388 - valid acc= 87.95%

Epoch [31/50]
train loss= 0.2039 - train acc= 93.29% - valid loss= 0.3386 - valid acc= 87.83%

Epoch [32/50]
train loss= 0.2021 - train acc= 93.39% - valid loss= 0.3389 - valid acc= 87.92%

Epoch [33/50]
train loss= 0.1996 - train acc= 93.51% - valid loss= 0.3416 - valid acc= 88.00%

Epoch [34/50]
train loss= 0.1977 - train acc= 93.54% - valid loss= 0.3417 - valid acc= 87.70%

Epoch [35/50]
train loss= 0.1956 - train acc= 93.65% - valid loss= 0.3443 - valid acc= 87.73%

Epoch [36/50]
train loss= 0.1941 - train acc= 93.68% - valid loss= 0.3364 - valid acc= 87.95%

Epoch [37/50]
train loss= 0.1920 - train acc= 93.79% - valid loss= 0.3378 - valid acc= 88.03%

Epoch [38/50]
train loss= 0.1905 - train acc= 93.76% - valid loss= 0.3373 - valid acc= 88.12%

Epoch [39/50]
train loss= 0.1889 - train acc= 93.78% - valid loss= 0.3449 - valid acc= 88.08%

Epoch [40/50]
train loss= 0.1868 - train acc= 93.93% - valid loss= 0.3409 - valid acc= 87.88%

Epoch [41/50]
train loss= 0.1856 - train acc= 93.97% - valid loss= 0.3392 - valid acc= 87.77%

Epoch [42/50]
train loss= 0.1847 - train acc= 93.99% - valid loss= 0.3398 - valid acc= 88.15%

Epoch [43/50]
train loss= 0.1816 - train acc= 94.09% - valid loss= 0.3469 - valid acc= 87.37%
```

```
Epoch [44/50]
train loss= 0.1808 - train acc= 94.09% - valid loss= 0.3391 - valid acc= 88.08%

Epoch [45/50]
train loss= 0.1790 - train acc= 94.24% - valid loss= 0.3400 - valid acc= 88.12%

Epoch [46/50]
train loss= 0.1775 - train acc= 94.25% - valid loss= 0.3445 - valid acc= 88.00%

Epoch [47/50]
train loss= 0.1765 - train acc= 94.31% - valid loss= 0.3414 - valid acc= 88.02%

Epoch [48/50]
train loss= 0.1757 - train acc= 94.30% - valid loss= 0.3482 - valid acc= 87.68%

Epoch [49/50]
train loss= 0.1736 - train acc= 94.39% - valid loss= 0.3422 - valid acc= 87.92%

Epoch [50/50]
train loss= 0.1722 - train acc= 94.37% - valid loss= 0.3466 - valid acc= 88.05%
```

Part 3: Convolutional Neural Networks and Image Classification

[Total marks for this part: 45 points]

The third part of this assignment is to demonstrate your basis knowledge in deep learning that you have acquired from the lectures and tutorials materials. Most of the contents in this assignment are drawn from **the tutorials covered from weeks 3 to 6**. Going through these materials before attempting this assignment is highly recommended.

The dataset used for this part is a specific dataset for this unit consisting of approximately $10,000$ images of 20 classes of Animals, each of which has approximately 500 images. You can download the dataset at download here if you want to do your assignment on your machine.

In [ ]:
```
import os
import requests
import tarfile
import time
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split
import torchvision.models as models
import torch.nn as nn
import torch
from torch import optim
import PIL.Image
import pathlib
from torchsummary import summary
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

# check if CUDA is available
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
```

```
    print('CUDA is not available.  Training on CPU ...')
else:
    print('CUDA is available!  Training on GPU ...')

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(1234)
```

```
CUDA is available!  Training on GPU ...
<torch._C.Generator at 0x7e6d5e371610>
```

Out[ ]:

Download the dataset to the folder of this Google Colab.

In [ ]:
```
!gdown --fuzzy https://drive.google.com/file/d/1aEkxNWaD02Z8ZNvZzeMefUoY97C-3wTG/vi
```

```
Failed to retrieve file url:

        Too many users have viewed or downloaded this file recently. Please
        try accessing the file again later. If the file you are trying to
        access is particularly large or is shared with many people, it may
        take up to 24 hours to be able to view or download the file. If you
        still can't access a file after 24 hours, contact your domain
        administrator.

You may still be able to access the file from the browser:

        https://drive.google.com/uc?id=1aEkxNWaD02Z8ZNvZzeMefUoY97C-3wTG

but Gdown can't. Please check connections and permissions.
```

We unzip the dataset to the folder.

In [ ]:
```
!unzip -q Animals_Dataset.zip
```

In [ ]:
```
data_dir = "FIT5215_Dataset"

# We resize the images to [3,64,64]
transform = transforms.Compose([transforms.Resize((64,64)),  #resises the image so
                                transforms.RandomHorizontalFlip(), # FLips th
                                #transforms.RandomRotation(4),     #Rotates t
                                #transforms.RandomAffine(0, shear=10, scale=(
                                transforms.ColorJitter(brightness=0.2, contra
                                transforms.ToTensor(), # convert the image to
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0
                                ])


# Load the dataset using torchvision.datasets.ImageFolder and apply transformations
dataset = datasets.ImageFolder(data_dir, transform=transform)

# Split the dataset into training and validation sets
train_size = int(0.9 * len(dataset))
valid_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, valid_size])

# Example of DataLoader creation for training and validation
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

print("Number of instance in train_set: %s" % len(train_dataset))
print("Number of instance in val_set: %s" % len(val_dataset))
```

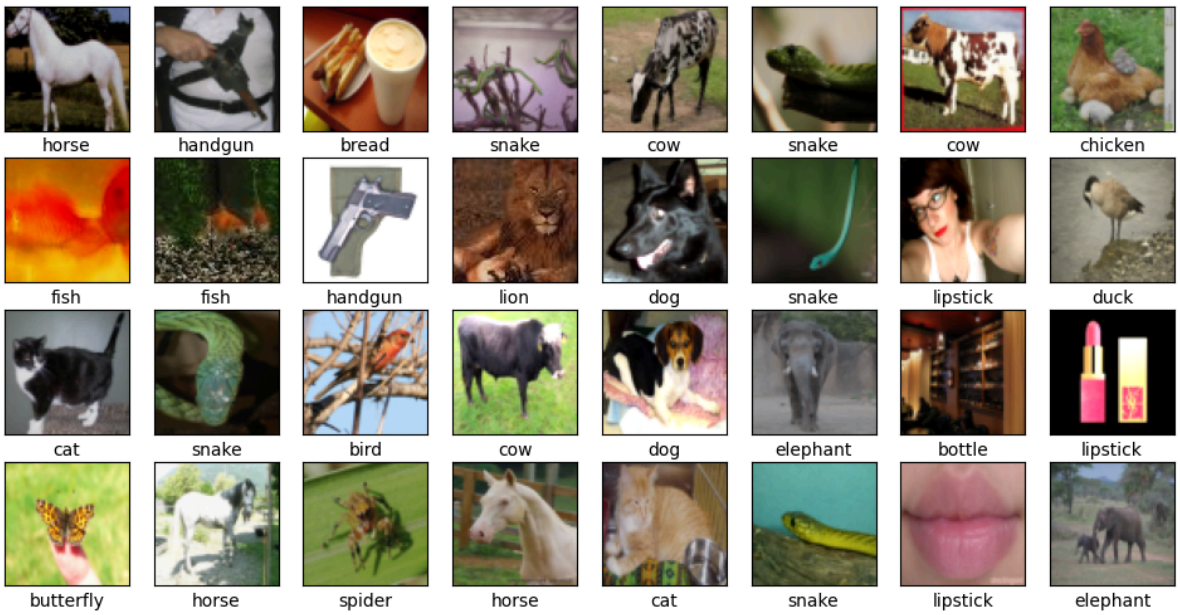
```
Number of instance in train_set: 8519
Number of instance in val_set: 947
```

In [ ]:
```
class_names = ['bird', 'bottle', 'bread', 'butterfly', 'cake', 'cat', 'chicken', 'c
               'elephant', 'fish', 'handgun', 'horse', 'lion', 'lipstick', 'seal
```

In [ ]:
```
# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = next(dataiter)
images = images.numpy() # convert images to numpy for display
```

In [ ]:
```
import math

def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image

def visualize_data(images, categories, images_per_row = 8):
    class_names = ['bird', 'bottle', 'bread', 'butterfly', 'cake', 'cat', 'chicken'
                   'elephant', 'fish', 'handgun', 'horse', 'lion', 'lipstick', 'seal
    n_images = len(images)
    n_rows = math.ceil(float(n_images)/images_per_row)
    fig = plt.figure(figsize=(1.5*images_per_row, 1.5*n_rows))
    fig.patch.set_facecolor('white')
    for i in range(n_images):
        plt.subplot(n_rows, images_per_row, i+1)
        plt.xticks([])
        plt.yticks([])
        imshow(images[i])
        class_index = categories[i]
        plt.xlabel(class_names[class_index])
    plt.show()
```

In [ ]:
```
visualize_data(images, labels)
```



For questions 3.1 to 3.7, you'll need to write your own model in a way that makes it easy for you to experiment with different architectures and parameters. The goal is to be able to pass the parameters to initialize a new instance of `YourModel` to build different network architectures with different parameters. Below are descriptions of some parameters for `YourModel` :

1. `Block confirguration` : Our network consists of many blocks. Each block has the pattern `[conv, batch norm, activation, conv, batch norm, activation, max pool, dropout]` . All convolutional layers have filter size $(3, 3)$, strides $(1, 1)$ and padding = 1, and all max pool layers have strides $(2, 2)$, kernel size 2, and padding = 0. The network will consists of a few blocks before applying a linear layer to output the logits for the softmax layer.

2. `list_feature_maps` : the number of feature maps in the blocks of the network. For example, if `list_feature_maps = [16, 32, 64]` , our network has three blocks with the input_channels or number of feature maps are `16, 32` , and `64` respectively.

3. `drop_rate` : the keep probability for dropout. Setting `drop_rate` to 0.0 means not using dropout.
4. `batch_norm` : the batch normalization function is used or not. Setting `batch_norm` to `false` means not using batch normalization.
5. `use_skip` : the skip connection is used in the blocks or not. Setting this to `true` means that we use `1x1` Conv2D with `strides=2` for the skip connection.
6. At the end, you need to apply `global average pooling (GAP)` (`AdaptiveAvgPool2d((1, 1))`) to flatten the 3D output tensor before defining the output linear layer for predicting the labels.

Here is the model confirguration of `YourCNN` if the `list_feature_maps = [16, 32, 64]` and `batch_norm = true` .

```
YourCNN(
  (block): ModuleList(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=same)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): YourBlock(
      (block): ModuleList(
        (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
        (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace=True)
        (6): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
        (7): Dropout(p=0.2, inplace=False)
      )
      (skip_conv): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2))
    )
    (4): YourBlock(
      (block): ModuleList(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace=True)
        (6): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
        (7): Dropout(p=0.2, inplace=False)
      )
      (skip_conv): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2))
    )
    (5): AdaptiveAvgPool2d(output_size=(1, 1))
    (6): Flatten(start_dim=1, end_dim=-1)
    (7): Linear(in_features=64, out_features=20, bias=True)
  )
)
```

Question 3.1: **You need to implement the aforementioned CNN**.

First, you need to implement the block of our CNN in the class `YourBlock`. You can ignore `use_skip` and `skip connection` for simplicity. However, you cannot earn full marks for this question.

[6 points]

In [ ]:
```
class CustomReLU(nn.Module):
    def forward(self, x):
        return torch.max(torch.tensor(0.0, device=x.device), x)
```

In [ ]:
```
class YourBlock(nn.Module):
    def __init__(self, in_feature_maps, out_feature_maps, drop_rate=0.2, batch_norm
        super(YourBlock, self).__init__()

        self.use_skip = use_skip
        self.batch_norm = batch_norm

        self.block = nn.ModuleList()

        # First convolutional layer
        self.block.append(nn.Conv2d(in_feature_maps, out_feature_maps, kernel_size=

        # First Batch Normalization
        if self.batch_norm:
            self.block.append(nn.BatchNorm2d(out_feature_maps, eps=0.00001, momentu

        # First ReLU activation
```

```
        self.block.append(CustomReLU())

        # Second convolutional layer
        self.block.append(nn.Conv2d(out_feature_maps, out_feature_maps, kernel_size

        # Second Batch Normalization
        if self.batch_norm:
            self.block.append(nn.BatchNorm2d(out_feature_maps, eps=0.00001, momentu

        # Second ReLU activation
        self.block.append(CustomReLU())

        # Max Pooling Layer (with adjusted padding)
        self.block.append(nn.MaxPool2d(kernel_size=2, stride=2, padding=0))  # Remo

        # Dropout layer
        self.block.append(nn.Dropout2d(p=drop_rate, inplace=False))

        # Skip connection: 1x1 convolution with stride 2 to match the dimensions
        if self.use_skip:
          self.skip_connection = nn.Conv2d(in_feature_maps, out_feature_maps, kerne
        else:
          self.skip_connection = nn.Identity()


    def forward(self, x):
        # Save the input for skip connection
        identity = x

        # Pass input through the main block layers
        for layer in self.block:
            x = layer(x)

        # Apply skip connection if use_skip is True
        if self.use_skip:
            # Apply the 1x1 convolution to the input for the skip path
            identity = self.skip_connection(identity)
            # Add the skip connection to the output of the main path
            x += identity

        return x
```

In [ ]:
```
block = YourBlock(in_feature_maps=32 ,out_feature_maps=64, drop_rate=0.2, batch_nor
print(block)
```

```
YourBlock(
  (block): ModuleList(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
    (2): CustomReLU()
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
    (5): CustomReLU()
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (7): Dropout2d(p=0.2, inplace=False)
  )
  (skip_connection): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2))
)
```

Second, you need to use the above `YourBlock` to implement the class `YourCNN`.

[6 points]

In [ ]:
```
class YourCNN(nn.Module):
  def __init__(self, list_feature_maps = [16, 32, 64], drop_rate = 0.2, batch_norm=
    super(YourCNN, self).__init__()
    layers = []
    #Write your code here

    # First conv layer
    layers.append(nn.Conv2d(3, list_feature_maps[0], kernel_size=3, stride=1, paddi
    layers.append(nn.BatchNorm2d(list_feature_maps[0], eps=1e-05, momentum=0.1, aff
    layers.append(CustomReLU())

    # Blocks
    total_number_of_blocks = len(list_feature_maps) - 1

    for i in range(total_number_of_blocks):
        input = list_feature_maps[i]
        output = list_feature_maps[i + 1]
        layers.append(YourBlock(in_feature_maps=input,
                                out_feature_maps=output,
                                drop_rate=drop_rate,
                                batch_norm=batch_norm,
                                use_skip=use_skip))

    # Adaptive average pooling, flattening, and fully connected layer
    layers.append(nn.AdaptiveAvgPool2d(output_size=(1, 1)))
    layers.append(nn.Flatten(1))
    layers.append(nn.Linear(list_feature_maps[-1], 20, bias=True))

    # Store the layers in a ModuleList
    self.block = nn.ModuleList(layers)


  def forward(self, x):
    #Write your code here

    for layer in self.block:
      x = layer(x)

    return x
```

We declare `my_cnn` from `YourCNN` as follows.

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
my_cnn = YourCNN(list_feature_maps = [16, 32, 64], use_skip = True)
my_cnn = my_cnn.to(device)
print(my_cnn)
```

```
YourCNN(
  (block): ModuleList(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
    (2): CustomReLU()
    (3): YourBlock(
      (block): ModuleList(
        (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
        (2): CustomReLU()
        (3): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
        (5): CustomReLU()
        (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
        (7): Dropout2d(p=0.2, inplace=False)
      )
      (skip_connection): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2))
    )
    (4): YourBlock(
      (block): ModuleList(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
        (2): CustomReLU()
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
        (5): CustomReLU()
        (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
        (7): Dropout2d(p=0.2, inplace=False)
      )
      (skip_connection): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2))
    )
    (5): AdaptiveAvgPool2d(output_size=(1, 1))
    (6): Flatten(start_dim=1, end_dim=-1)
    (7): Linear(in_features=64, out_features=20, bias=True)
  )
)
```

We declare the optimizer and the loss function.

In [ ]:
```
# Loss and optimizer
learning_rate = 0.001
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(my_cnn.parameters(), lr=learning_rate , weight_decay=
```

Here are the codes to compute the loss and accuracy.

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def compute_loss(model, loss_fn, loader):
    total_loss = 0
    total_samples = 0
    model.eval()
    with torch.no_grad():
        for (X, y) in loader:
            X, y = X.to(device), y.to(device)
            outputs = model(X)
```

```
            loss = loss_fn(outputs, y)
            total_loss += loss.item() * y.size(0)
            total_samples += y.size(0)
    return total_loss / total_samples
```

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def compute_acc(model, loader):
    correct = 0
    totals = 0
    model.eval()
    with torch.no_grad():
        for (batchX, batchY) in loader:
            batchX, batchY = batchX.to(device), batchY.to(device)
            outputs = model(batchX)
            totals += batchY.size(0)
            predicted = torch.argmax(outputs.data, 1)
            correct += (predicted == batchY).sum().item()
    return correct / totals
```

Here is the code to train our model.

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def fit(model= None, train_loader = None, valid_loader= None, optimizer = None,
        num_epochs = 50, verbose = True, seed= 1234):
  torch.manual_seed(seed)
  # Move the model to the device before initializing the optimizer
  model.to(device) # Move the model to the GPU

  if optimizer == None:
    optim = torch.optim.adamw(model.parameters(), lr = 0.001) # Now initialize opti
  else:
    optim = optimizer
  history = dict()
  history['val_loss'] = list()
  history['val_acc'] = list()
  history['train_loss'] = list()
  history['train_acc'] = list()

  for epoch in range(num_epochs):
    model.train()
    for (X, y) in train_loader:
      # Move input data to the same device as the model
      X,y = X.to(device), y.to(device)
      # Forward pass
      outputs = model(X.type(torch.float32)) # X is already on the correct device
      loss = loss_fn(outputs, y.type(torch.long))
      # Backward and optimize
      optim.zero_grad()
      loss.backward()
      optim.step()
    #Losses and accuracies for epoch
    val_loss = compute_loss(model, loss_fn, valid_loader)
    val_acc = compute_acc(model, valid_loader)
    train_loss = compute_loss(model, loss_fn, train_loader)
    train_acc = compute_acc(model, train_loader)
    history['val_loss'].append(val_loss)
    history['val_acc'].append(val_acc)
    history['train_loss'].append(train_loss)
    history['train_acc'].append(train_acc)
    if not verbose: #verbose = True means we do show the training information durir
      print(f"Epoch {epoch+1}/{num_epochs}")
```

```
      print(f"train loss= {train_loss:.4f} - train acc= {train_acc*100:.2f}% - vali

  return history
```

In [ ]:
```
history = fit(model= my_cnn, train_loader=train_loader, valid_loader = val_loader,
```

```
Epoch 1/15
train loss= 2.3589 - train acc= 27.13% - valid loss= 2.3705 - valid acc= 25.77%
Epoch 2/15
train loss= 2.1931 - train acc= 31.60% - valid loss= 2.2286 - valid acc= 27.67%
Epoch 3/15
train loss= 2.1508 - train acc= 32.34% - valid loss= 2.2084 - valid acc= 31.15%
Epoch 4/15
train loss= 1.9748 - train acc= 38.74% - valid loss= 2.0078 - valid acc= 35.16%
Epoch 5/15
train loss= 1.9026 - train acc= 40.93% - valid loss= 1.9429 - valid acc= 37.17%
Epoch 6/15
train loss= 1.8673 - train acc= 41.77% - valid loss= 1.8880 - valid acc= 40.76%
Epoch 7/15
train loss= 1.8966 - train acc= 39.99% - valid loss= 1.9430 - valid acc= 37.91%
Epoch 8/15
train loss= 1.7379 - train acc= 45.85% - valid loss= 1.7534 - valid acc= 42.66%
Epoch 9/15
train loss= 1.6814 - train acc= 48.35% - valid loss= 1.7292 - valid acc= 45.30%
Epoch 10/15
train loss= 1.7256 - train acc= 45.26% - valid loss= 1.7439 - valid acc= 43.40%
Epoch 11/15
train loss= 1.6164 - train acc= 50.10% - valid loss= 1.6753 - valid acc= 46.15%
Epoch 12/15
train loss= 1.5604 - train acc= 51.10% - valid loss= 1.5938 - valid acc= 50.16%
Epoch 13/15
train loss= 1.5700 - train acc= 51.11% - valid loss= 1.6049 - valid acc= 49.95%
Epoch 14/15
train loss= 1.5252 - train acc= 52.85% - valid loss= 1.5564 - valid acc= 51.53%
Epoch 15/15
train loss= 1.6156 - train acc= 49.47% - valid loss= 1.6327 - valid acc= 48.05%
```

Question 3.2: **Now, let us tune the number of blocks $use\_skip \in \{true, false\}$ and $learning\_rate \in \{0.001, 0.0005\}$. Write your code for this tuning and report the result of the best model on the testing set. Note that you need to show your code for tuning and evaluating on the test set to earn the full marks. During tuning, you can set the instance variable `verbose` of your model to `True` for not showing the training details of each epoch.**

Note that for this question, depending on your computational resource, you can choose `list_feature_maps= [32, 64]` or `list_feature_maps= [16, 32, 64]`.

[3 points]

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
use_skips = [True, False]
learning_rates = [0.001, 0.0005]
number_of_epochs = 5
feature_maps = [16, 32, 64]

best_valid_accuracy = 0
best_model = None
best_parameters = {'learning_rate': None, 'use_skip': None}

for lr in learning_rates:
    for use_skip in use_skips:
```

```
        print(f"Training parameters: learning_rate={lr} and use_skip={use_skip}")

        model = YourCNN(list_feature_maps=feature_maps, drop_rate=0.2, batch_norm=T

        optimizer = optim.AdamW(model.parameters(), lr=lr , weight_decay=1e-4)

        # previous training method
        history = fit(model=model, train_loader=train_loader, valid_loader=val_loac

        valid_accuracy = history["val_acc"][-1]  # Get the last epoch accuracy for

        print()
        print(f"Completed training, Validation accuracy: {valid_accuracy:.4f}")
        print()

        # Compare the accuracies to the previous one
        # If the current accuracy is better , use that model
        if valid_accuracy > best_valid_accuracy:
            best_valid_accuracy = valid_accuracy
            best_parameters = {'learning_rate': lr, 'use_skip': use_skip}
            best_model = model

# After running through all the combinations and epochs
# We get the best model and train it on the test set
test_acc = compute_acc(best_model, val_loader)
print()
print(f"Best model parameters: {best_parameters}")
print(f"Best validation accuracy: {best_valid_accuracy:.4f}")
print(f"Test accuracy of the best model: {test_acc:.4f}")
```

```
Training parameters: learning_rate=0.001 and use_skip=True
Epoch 1/5
train loss= 2.4268 - train acc= 25.32% - valid loss= 2.4353 - valid acc= 23.34%
Epoch 2/5
train loss= 2.2812 - train acc= 28.00% - valid loss= 2.3053 - valid acc= 25.66%
Epoch 3/5
train loss= 2.2039 - train acc= 30.50% - valid loss= 2.2374 - valid acc= 30.10%
Epoch 4/5
train loss= 1.9998 - train acc= 38.22% - valid loss= 2.0275 - valid acc= 36.96%
Epoch 5/5
train loss= 1.9157 - train acc= 40.83% - valid loss= 1.9395 - valid acc= 38.86%

Completed training, Validation accuracy: 0.3886

Training parameters: learning_rate=0.001 and use_skip=False
Epoch 1/5
train loss= 2.4204 - train acc= 26.29% - valid loss= 2.4336 - valid acc= 26.08%
Epoch 2/5
train loss= 2.2397 - train acc= 31.79% - valid loss= 2.2708 - valid acc= 28.19%
Epoch 3/5
train loss= 2.0917 - train acc= 35.41% - valid loss= 2.1285 - valid acc= 32.84%
Epoch 4/5
train loss= 1.9904 - train acc= 37.84% - valid loss= 2.0268 - valid acc= 34.53%
Epoch 5/5
train loss= 1.9639 - train acc= 38.96% - valid loss= 1.9812 - valid acc= 38.01%

Completed training, Validation accuracy: 0.3801

Training parameters: learning_rate=0.0005 and use_skip=True
Epoch 1/5
train loss= 2.4285 - train acc= 27.22% - valid loss= 2.4357 - valid acc= 27.03%
Epoch 2/5
train loss= 2.2553 - train acc= 31.22% - valid loss= 2.2812 - valid acc= 27.67%
Epoch 3/5
train loss= 2.1572 - train acc= 33.85% - valid loss= 2.1788 - valid acc= 32.42%
Epoch 4/5
train loss= 2.0519 - train acc= 37.66% - valid loss= 2.0805 - valid acc= 35.59%
Epoch 5/5
train loss= 2.0013 - train acc= 38.46% - valid loss= 2.0304 - valid acc= 36.54%

Completed training, Validation accuracy: 0.3654

Training parameters: learning_rate=0.0005 and use_skip=False
Epoch 1/5
train loss= 2.4746 - train acc= 25.66% - valid loss= 2.4956 - valid acc= 24.50%
Epoch 2/5
train loss= 2.3323 - train acc= 30.40% - valid loss= 2.3619 - valid acc= 27.24%
Epoch 3/5
train loss= 2.1983 - train acc= 34.08% - valid loss= 2.2372 - valid acc= 31.57%
Epoch 4/5
train loss= 2.0953 - train acc= 35.49% - valid loss= 2.1190 - valid acc= 32.52%
Epoch 5/5
train loss= 2.0216 - train acc= 38.76% - valid loss= 2.0386 - valid acc= 36.54%

Completed training, Validation accuracy: 0.3654


Best model parameters: {'learning_rate': 0.001, 'use_skip': True}
Best validation accuracy: 0.3886
Test accuracy of the best model: 0.3970
```

Please note that you are struggling in implementing the aforementioned CNN. You can use the MiniVGG network in our labs for doing the following questions. However, you

cannot earn any mark for 3.1 and 3.2.

Question 3.3: **Exploring Data Mixup Technique for Improving Generalization Ability.**

[4 points]

Data mixup is another super-simple technique used to boost the generalization ability of deep learning models. You need to incoroporate data mixup technique to the above deep learning model and experiment its performance. There are some papers and documents for data mixup as follows:

- Main paper for data mixup link for main paper and a good article article link.

You need to extend your model developed above, train a model using data mixup, and write your observations and comments about the result.

In [60]:
```
# The functionality of cutmix code is obtained from the github repository in the
# paper provided in train.py

def mixup_data(x, y, alpha=0.1):
    '''Returns mixed inputs, pairs of targets, and lambda'''
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1  # No mixup will be performed

    batch_size = x.size()[0]
    index = torch.randperm(batch_size)

    # mixed input
    mixed_x = lam * x + (1 - lam) * x[index, :]
    # Original and shuffled labels
    y_a, y_b = y, y[index]
    return mixed_x, y_a, y_b, lam

def mixup_criterion(criterion, pred, y_a, y_b, lam):
  # calculate the weighted loss of the mixup data based on original and shuffled ta
    return lam * criterion(pred, y_a) + (1 - lam) * criterion(pred, y_b)

# Training the model with mixup code is obtained from the history method above with
def fit_with_mixup(model, train_loader, valid_loader, optimizer, num_epochs=50, alp

    history = {'val_loss': [], 'val_acc': [], 'train_loss': [], 'train_acc': []}

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for x, y in train_loader:
            x, y = x.to(device), y.to(device)

            # Apply mixup
            mixed_X, y_a, y_b, lam = mixup_data(x, y, alpha)
            outputs = model(mixed_X)

            loss = mixup_criterion(loss_fn, outputs, y_a, y_b, lam)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```
            running_loss += loss.item()

        # Calculate losses and accuracies
        train_loss = running_loss / len(train_loader)
        val_loss = compute_loss(model, loss_fn, valid_loader)
        val_acc = compute_acc(model, valid_loader)
        train_acc = compute_acc(model, train_loader)

        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)

        if not verbose:
            print(f"Epoch {epoch + 1}/{num_epochs}")
            print(f"train loss={train_loss:.4f} - train acc={train_acc * 100:.2f}%

    return history

alpha = 0.1 # Hyperparameter for the mixup which can be tuned
best_model_with_mixup = YourCNN(list_feature_maps=[16, 32, 64], drop_rate=0.2, batc
optimizer = torch.optim.AdamW(best_model_with_mixup.parameters(), lr=0.001, weight_
history_with_mixup = fit_with_mixup(model=best_model_with_mixup, train_loader=trair
                                    optimizer=optimizer, num_epochs=15, alpha=alpha

# Evaluate the best model with mixup on the test set
test_acc_with_mixup = compute_acc(best_model_with_mixup, val_loader)
print(f"Test accuracy with mixup: {test_acc_with_mixup* 100:.2f}")
```

```
Epoch 1/15
train loss=2.6456 - train acc=28.31% - valid loss=2.3942 - valid acc=25.87%
Epoch 2/15
train loss=2.4553 - train acc=29.06% - valid loss=2.2964 - valid acc=26.93%
Epoch 3/15
train loss=2.3464 - train acc=36.06% - valid loss=2.1206 - valid acc=34.74%
Epoch 4/15
train loss=2.2905 - train acc=37.79% - valid loss=2.0978 - valid acc=33.47%
Epoch 5/15
train loss=2.2364 - train acc=35.61% - valid loss=2.1019 - valid acc=32.10%
Epoch 6/15
train loss=2.1776 - train acc=39.09% - valid loss=2.0144 - valid acc=37.49%
Epoch 7/15
train loss=2.0849 - train acc=43.01% - valid loss=1.8673 - valid acc=41.18%
Epoch 8/15
train loss=2.0807 - train acc=46.34% - valid loss=1.7928 - valid acc=43.82%
Epoch 9/15
train loss=2.0348 - train acc=45.31% - valid loss=1.8160 - valid acc=44.35%
Epoch 10/15
train loss=2.0113 - train acc=48.10% - valid loss=1.7353 - valid acc=45.93%
Epoch 11/15
train loss=1.9677 - train acc=47.51% - valid loss=1.7368 - valid acc=45.93%
Epoch 12/15
train loss=1.9505 - train acc=49.84% - valid loss=1.6660 - valid acc=47.41%
Epoch 13/15
train loss=1.9152 - train acc=50.02% - valid loss=1.6532 - valid acc=48.15%
Epoch 14/15
train loss=1.8954 - train acc=50.60% - valid loss=1.6321 - valid acc=49.21%
Epoch 15/15
train loss=1.8693 - train acc=51.97% - valid loss=1.5805 - valid acc=49.63%
Test accuracy with mixup: 51.32
```

Observation:

- Data Mixup model has a higher test accuracy compared to during training which indicate that the model generalizes well and performs slightly better on the test set
- The gap between training and validation accuracy is narrowing, which is a good sign. It indicates that the Data Mixup model is helping reduce overfitting and improving the model's ability to generalize.

Question 3.4: **Exploring CutMix Technique for Improving Generalization Ability.**

[4 points]

Data mixup is another super-simple technique used to boost the generalization ability of deep learning models. You need to incoroporate data mixup technique to the above deep learning model and experiment its performance. There are some papers and documents for data mixup as follows:

- Main paper for Cutmix link for main paper and a good article article link.

You need to extend your model developed above, train a model using data mixup, and write your observations and comments about the result.

In [ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# The code for cutmix functionality is obtained from the github repository
# from the paper provided in train.py

def cutmix_technique(x, y, alpha=1.0, device='cuda'):
    '''
    Apply CutMix augmentation technique on input data.
    x: input data (batch of images)
    y: target labels corresponding to the input batches
    alpha: parameter for the Beta distribution, set to 1.0 for balance
    '''
    lam = np.random.beta(alpha, alpha)
    random_index = torch.randperm(x.size()[0]).to(device)
    y_a = y
    y_b = y[random_index]

    # Decide the size of the patch
    bbx1, bby1, bbx2, bby2 = rand_bbox(x.size(), lam)
    x[:, :, bbx1:bbx2, bby1:bby2] = x[random_index, :, bbx1:bbx2, bby1:bby2]

    # Adjust lambda to exactly match pixel ratio
    lam = 1 - ((bbx2 - bbx1) * (bby2 - bby1) / (x.size()[-1] * x.size()[-2]))
    return x, y_a, y_b, lam

def rand_bbox(size, lam):
    W = size[2]
    H = size[3]
    cut_rat = np.sqrt(1. - lam)
    cut_w = int(W * cut_rat)  # Replaced np.int with int
    cut_h = int(H * cut_rat)  # Replaced np.int with int

    # uniform
    cx = np.random.randint(W)
    cy = np.random.randint(H)
```

```
    bbx1 = np.clip(cx - cut_w // 2, 0, W)
    bby1 = np.clip(cy - cut_h // 2, 0, H)
    bbx2 = np.clip(cx + cut_w // 2, 0, W)
    bby2 = np.clip(cy + cut_h // 2, 0, H)

    return bbx1, bby1, bbx2, bby2

# Function to compute CutMix loss
def cutmix_criterion(criterion, pred, y_a, y_b, lam):
    return lam * criterion(pred, y_a) + (1 - lam) * criterion(pred, y_b)
```

In [ ]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Training the model with cutmix code is obtained from the history method above wit
def fit_cutmix(model, train_loader, valid_loader, optimizer, num_epochs=10, alpha=1
    criterion = nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            # Apply CutMix
            inputs, targets_a, targets_b, lam = cutmix_technique(inputs, labels, al
            outputs = model(inputs)
            loss = cutmix_criterion(criterion, outputs, targets_a, targets_b, lam)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            # Creat boolean tensor
            correct += (lam * predicted.eq(targets_a).sum().item() + (1 - lam) * pr

        train_loss = running_loss / len(train_loader)
        train_acc = correct / total

        valid_acc = compute_acc(model, valid_loader)
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Trair


best_model_cutmix = YourCNN(list_feature_maps=[16, 32, 64], drop_rate=0.25, batch_r
optimizer_cutmix = optim.Adam(best_model_cutmix.parameters(), lr=0.001)

# Fit the model using CutMix
history_with_cutmix = fit_cutmix(best_model_cutmix, train_loader, val_loader, optim

# Evaluate on the test set
test_acc_cutmix = compute_acc(best_model_cutmix, val_loader)
print(f'Test accuracy of the CutMix model: {test_acc_cutmix * 100:.2f}%')
```

```
Epoch [1/15], Train Loss: 2.7781, Train Acc: 0.1504, Valid Acc: 0.2460
Epoch [2/15], Train Loss: 2.6371, Train Acc: 0.1958, Valid Acc: 0.2471
Epoch [3/15], Train Loss: 2.5968, Train Acc: 0.2028, Valid Acc: 0.2862
Epoch [4/15], Train Loss: 2.5333, Train Acc: 0.2301, Valid Acc: 0.3221
Epoch [5/15], Train Loss: 2.4752, Train Acc: 0.2404, Valid Acc: 0.3485
Epoch [6/15], Train Loss: 2.4333, Train Acc: 0.2611, Valid Acc: 0.3897
Epoch [7/15], Train Loss: 2.4027, Train Acc: 0.2649, Valid Acc: 0.3780
Epoch [8/15], Train Loss: 2.3687, Train Acc: 0.2769, Valid Acc: 0.3812
Epoch [9/15], Train Loss: 2.3713, Train Acc: 0.2776, Valid Acc: 0.4118
Epoch [10/15], Train Loss: 2.3357, Train Acc: 0.2908, Valid Acc: 0.4361
Epoch [11/15], Train Loss: 2.3021, Train Acc: 0.3004, Valid Acc: 0.3685
Epoch [12/15], Train Loss: 2.2548, Train Acc: 0.3146, Valid Acc: 0.4435
Epoch [13/15], Train Loss: 2.2375, Train Acc: 0.3189, Valid Acc: 0.4192
Epoch [14/15], Train Loss: 2.2323, Train Acc: 0.3219, Valid Acc: 0.4403
Epoch [15/15], Train Loss: 2.2115, Train Acc: 0.3260, Valid Acc: 0.4477
Test accuracy of the CutMix model: 46.67%
```

Observation:

- The cutmix model performed slighly worse than the data mixup model.
- The training and validation accuracies increases gradually. The training loss also has a steady decrease which means the model is learning well
- The gap between training and validation accuracy is moderate , this means that the model generalizes well and does not overfit

Question 3.5: Implement the **one-versus-all (OVA)** loss. The details are as follows:

- You need to apply `the sigmoid activation function` to logits $h = [h_1, h_2, \ldots, h_M]$ instead of `the softmax activation` function as usual to obtain $p = [p_1, p_2, \ldots, p_M]$, meaning that $p_i = sigmoid(h_i), i = 1, \ldots, M$. Note that M is the number of classes.
- Given a data example x with the ground-truth label y, the idea is to maximize the likelihood p_y and to minimize the likelihoods $p_i, i \neq y$. Therefore, the objective function is to find the model parameters to
  - $\max \left\{ \log p_y + \sum_{i \neq y} \log(1 - p_i) \right\}$ or equivalently $\min \left\{ -\log p_y - \sum_{i \neq y} \log(1 - p_i) \right\}$.
  - For example, if $M = 3$ and $y = 2$, you need to minimize $\min \{ -\log(1 - p_1) - \log p_2 - \log(1 - p_3) \}$.

Compare the model trained with the OVA loss and the same model trained with the standard cross-entropy loss.

[4 points]

In [ ]:
```python
import torch.nn.functional as F

#Your code here
class OvaLoss(nn.Module):
  '''
  The main purpose of this class is to maximise the likelihood
  of the true lable and minimize the likelihood of all other
  classes
```

```
  '''
  def __init__(self):
    super(OvaLoss,self).__init__()

  def forward(self,logits,target):
    # Applying the sigmoid function to the logits instead of softmax
    probs = torch.sigmoid(logits)

    # We need to do one-hot-encoding for the target labels
    # In order to identify the true labels from the incorrect labels
    one_hot_targets = F.one_hot(target, num_classes=logits.size(1)).float()

    # Ova loss calculation
    actual_loss = -torch.log(probs) * one_hot_targets
    incorrect_losses = -torch.log(1 - probs) * (1 - one_hot_targets)

    # sum up the losses
    final_loss = (actual_loss + incorrect_losses).sum(dim = 1).mean()

    return final_loss
```

In [ ]:

```
best_model_with_ova_loss = YourCNN(list_feature_maps=[16, 32, 64], drop_rate=0.2, b
criterion = OvaLoss()
optimizer = optim.AdamW(best_model_with_ova_loss.parameters(), lr=0.001)

def ova_loss_train(model, train_loader, valid_loader, optimizer, num_epochs, verbos
    history = {'val_loss': [], 'val_acc': [], 'train_loss': [], 'train_acc': []}

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for x, y in train_loader:
            x, y = x.to(device), y.to(device)

            outputs = model(x)
            loss = criterion(outputs, y)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate accuracy
            _, predicted = torch.max(torch.sigmoid(outputs), 1)
            total += y.size(0)
            correct += (predicted == y).sum().item()

        train_loss = running_loss / len(train_loader)
        train_acc = correct / total
        val_loss = compute_loss(model, criterion, valid_loader)
        val_acc = compute_acc(model, valid_loader)

        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)

        if verbose:
            print(f"Epoch {epoch + 1}/{num_epochs}")
            print(f"train loss={train_loss:.4f} - train acc={train_acc * 100:.2f}%
```

```
    return history

# Train the model with OVA loss
history_with_ova = ova_loss_train(best_model_with_ova_loss, train_loader=train_loac
                                  optimizer=optimizer, num_epochs=15, verbose=Tru

test_acc_ovaloss= compute_acc(best_model_with_ova_loss, val_loader)
print(f'Test accuracy of the Ova loss model: {test_acc_ovaloss * 100:.2f}%')
```

```
Epoch 1/15
train loss=4.1684 - train acc=16.46% - valid loss=3.4132 - valid acc=22.60%
Epoch 2/15
train loss=3.3874 - train acc=24.69% - valid loss=3.2746 - valid acc=25.34%
Epoch 3/15
train loss=3.2552 - train acc=27.48% - valid loss=3.0851 - valid acc=31.57%
Epoch 4/15
train loss=3.1416 - train acc=31.11% - valid loss=2.9842 - valid acc=36.22%
Epoch 5/15
train loss=3.0649 - train acc=33.03% - valid loss=2.8761 - valid acc=37.38%
Epoch 6/15
train loss=2.9863 - train acc=35.98% - valid loss=2.8275 - valid acc=40.13%
Epoch 7/15
train loss=2.9332 - train acc=37.21% - valid loss=2.8390 - valid acc=40.23%
Epoch 8/15
train loss=2.8937 - train acc=38.07% - valid loss=2.7785 - valid acc=39.39%
Epoch 9/15
train loss=2.8438 - train acc=40.20% - valid loss=2.7046 - valid acc=42.45%
Epoch 10/15
train loss=2.8053 - train acc=40.79% - valid loss=2.7519 - valid acc=40.55%
Epoch 11/15
train loss=2.7731 - train acc=41.14% - valid loss=2.6436 - valid acc=43.82%
Epoch 12/15
train loss=2.7353 - train acc=42.12% - valid loss=2.6507 - valid acc=45.72%
Epoch 13/15
train loss=2.7156 - train acc=42.96% - valid loss=2.5595 - valid acc=46.36%
Epoch 14/15
train loss=2.6886 - train acc=43.40% - valid loss=2.5437 - valid acc=46.78%
Epoch 15/15
train loss=2.6691 - train acc=43.95% - valid loss=2.5534 - valid acc=47.73%
Test accuracy of the Ova loss model: 47.94%
```

Question 3.6: **Attack your best obtained model with PGD attacks with $\epsilon = 0.0313, k = 20, \eta = 0.002$ on the testing set. Write the code for the attacks and report the robust accuracies. Also choose a random set of 20 clean images in the testing set and visualize the original and attacked images.**

[4 points]

In [66]:
```
# this code is obtained from tutorial week 6 colab

import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt

def pgd_attack(model, input_image, input_label=None,
               epsilon=0.0313,
               num_steps=20,
               step_size=0.002,
               clip_value_min=0.,
               clip_value_max=1.0):
```

```
    if isinstance(input_image, np.ndarray):
        input_image = torch.tensor(input_image, dtype=torch.float32).to(device)

    if isinstance(input_label, np.ndarray):
        input_label = torch.tensor(input_label, dtype=torch.long).to(device)

    # Ensure the model is in evaluation mode
    model.eval()

    # Create a copy of the input image and set it to require gradients
    adv_image = input_image.clone().detach().requires_grad_(True)

    # Random initialization around input_image
    random_noise = torch.FloatTensor(input_image.shape).uniform_(-epsilon, epsilon)
    adv_image = adv_image + random_noise
    adv_image = torch.clamp(adv_image, clip_value_min, clip_value_max).detach().rec

    # If no input label is provided, use the model's prediction
    if input_label is None:
        output = model(adv_image)
        input_label = torch.argmax(output, dim=1)


    for _ in range(int(num_steps)):  # Ensure this is an integer
        adv_image.requires_grad_(True)  # Ensure requires_grad is True in each iter
        output = model(adv_image)
        loss = nn.CrossEntropyLoss()(output, input_label)
        model.zero_grad()
        loss.backward()

        # Check if gradient is available before accessing 'data'
        if adv_image.grad is not None:
            gradient = adv_image.grad.data
            adv_image = adv_image + step_size * gradient.sign()
            adv_image = torch.clamp(adv_image, input_image - epsilon, input_image +
            adv_image = torch.clamp(adv_image, clip_value_min, clip_value_max)  # C
            adv_image = adv_image.detach().requires_grad_(True)  # Ensure requires_
        else:
            print("Warning: Gradient is None. Check for detach operations.")

    return adv_image.detach()
```

In [67]:

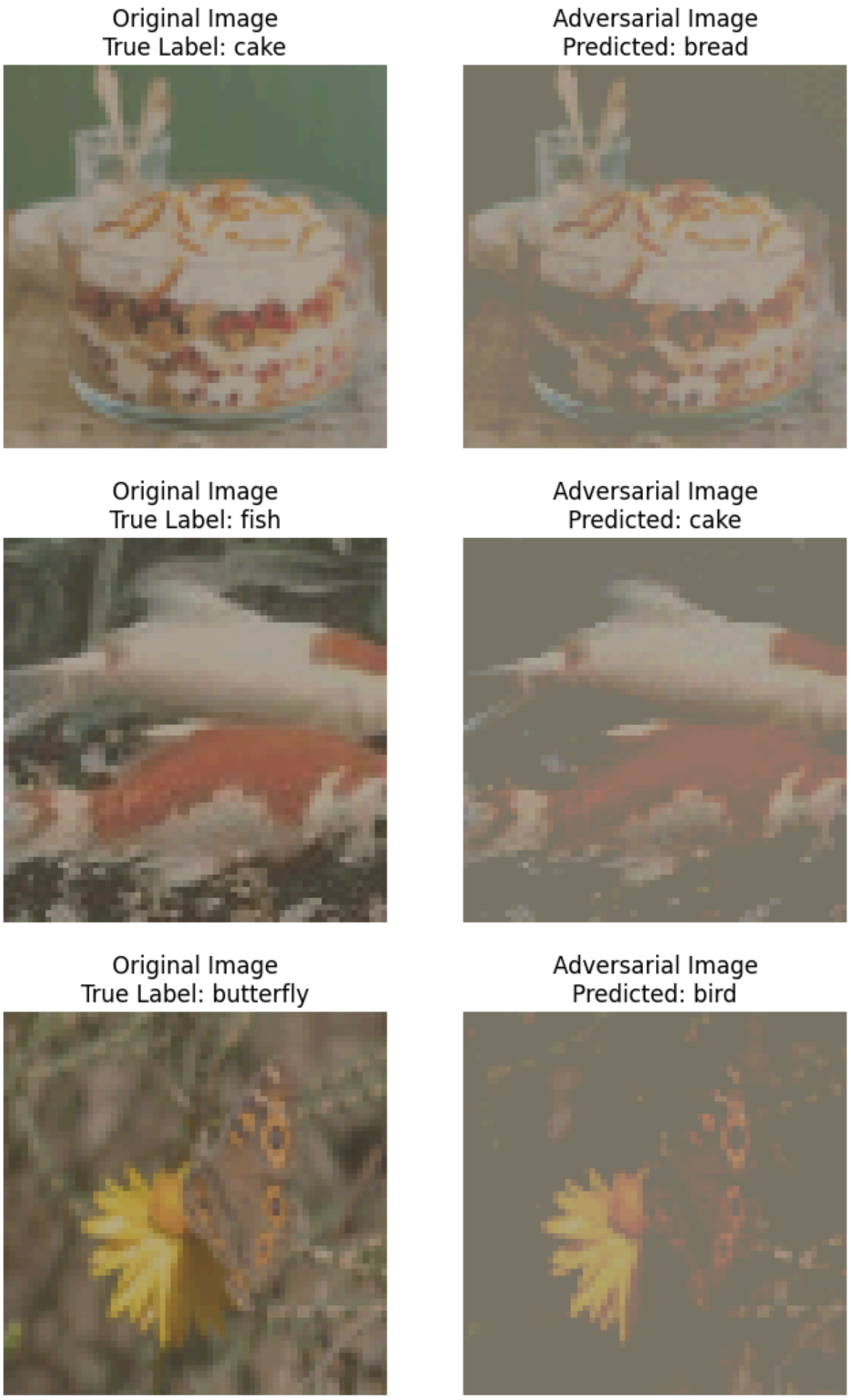
```
# This code is obtained from tutorial week 6 colab


def revert_preprocess(tensor):
    # Define the inverse normalization
    mean = torch.tensor([0.485, 0.456, 0.406])
    std = torch.tensor([0.229, 0.224, 0.225])
    inv_normalize = transforms.Normalize(
        mean=-mean / std,
        std=1.0 / std
    )

    # Apply inverse normalization
    tensor = inv_normalize(tensor)

    # Convert tensor to PIL image
    to_pil = transforms.ToPILImage()
    img = to_pil(tensor)

    return img
```

In [ ]:

```
# Tutorial week 6  code for robust accuracy
def evaluate_robust_accuracy(model, test_loader, epsilon=0.0313, step_size=0.002, r
    model.eval()
    y_adv = []
    y_true = []

    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        adv_images = pgd_attack(model, images, labels, epsilon, num_steps=num_steps
        output_adv = model(adv_images)
        pred_adv = output_adv.argmax(dim=1, keepdim=True)
        y_adv.extend(pred_adv.squeeze().cpu().numpy())
        y_true.extend(labels.cpu().numpy())  # Move labels to CPU before converting

    test_adv_acc = accuracy_score(y_true, y_adv)
    return test_adv_acc

def visualize_attack_with_labels(model, test_loader, epsilon=0.0313, num_steps=20,
    model.eval()

    # Select random images from the test loader
    dataiter = iter(test_loader)
    images, labels = next(dataiter)
    images, labels = images.to(device), labels.to(device)

    # Generate adversarial examples using PGD attack
    adv_images = pgd_attack(model, images, labels, epsilon=epsilon, num_steps=num_s

    # Loop over the images
    for i in range(num_images):
        original_image = images[i]
        adversarial_image = adv_images[i]
        true_label = labels[i].item()

        output_adv = model(adversarial_image.unsqueeze(0))

        # Decode the predictions to get the predicted class indices
        adversarial_prediction_index = output_adv.argmax(dim=1).item()

        # Map the predicted class indices to class names
        adversarial_label = class_names[adversarial_prediction_index]
        true_class_name = class_names[true_label]

        # Revert preprocessing
        original_image = revert_preprocess(original_image.cpu())
        adversarial_image= revert_preprocess(adversarial_image.cpu())

        # side by side plotting
        fig, ax = plt.subplots(1, 2, figsize=(8, 5))
        ax[0].imshow(original_image)
        ax[0].set_title(f'Original Image\nTrue Label: {true_class_name}')
        ax[0].axis('off')

        ax[1].imshow(adversarial_image)
        ax[1].set_title(f'Adversarial Image\nPredicted: {adversarial_label}')
        ax[1].axis('off')

        plt.show()

robust_accuracy = evaluate_robust_accuracy(best_model_with_mixup, val_loader, epsil
print(f"Robust accuracy with PGD attack: {robust_accuracy * 100:.2f}%")
```

```
# Visualizing 20 random clean and adversarial images from the test set
visualize_attack_with_labels(best_model_with_mixup , val_loader, epsilon=0.0313, nu
```

```
Robust accuracy with PGD attack: 8.24%
```



Original Image
True Label: elephant

Adversarial Image
Predicted: duck

Original Image
True Label: duck

Adversarial Image
Predicted: spider

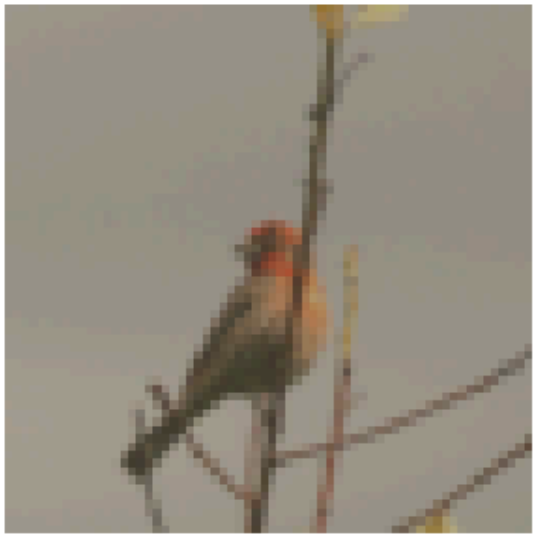
Original Image
True Label: snake

Adversarial Image
Predicted: duck

Original Image
True Label: cat

Adversarial Image
Predicted: duck

Original Image
True Label: bird



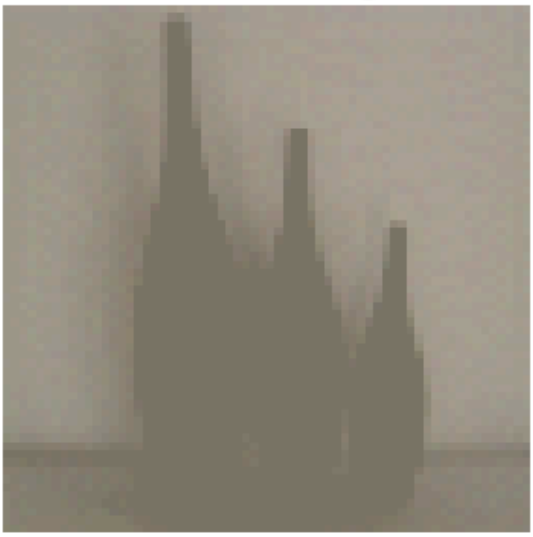
Adversarial Image
Predicted: spider

Original Image
True Label: bread

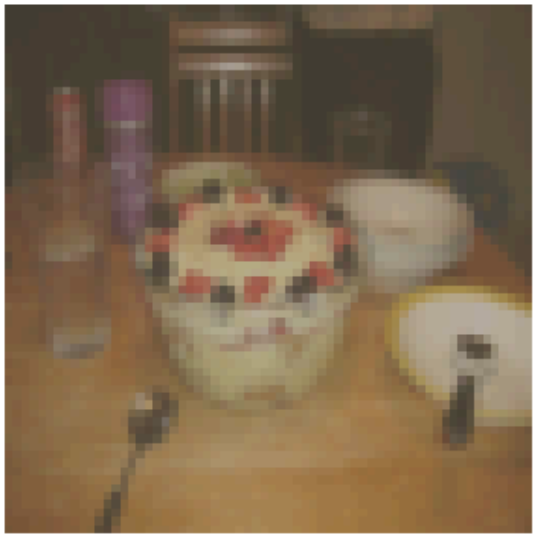
Adversarial Image
Predicted: bread

Original Image
True Label: vase

Adversarial Image
Predicted: handgun



Original Image
True Label: cake



Adversarial Image
Predicted: spider

Original Image
True Label: cat

Adversarial Image
Predicted: lipstick

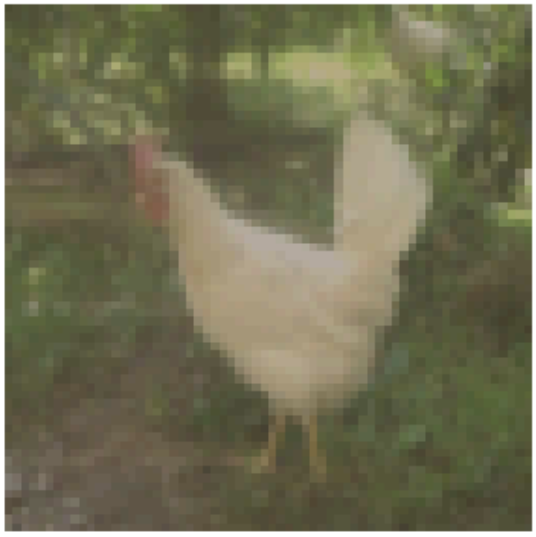
Original Image
True Label: cow

Adversarial Image
Predicted: spider

Original Image
True Label: bird

Adversarial Image
Predicted: cat

Original Image
True Label: chicken



Adversarial Image
Predicted: bird

Original Image
True Label: chicken

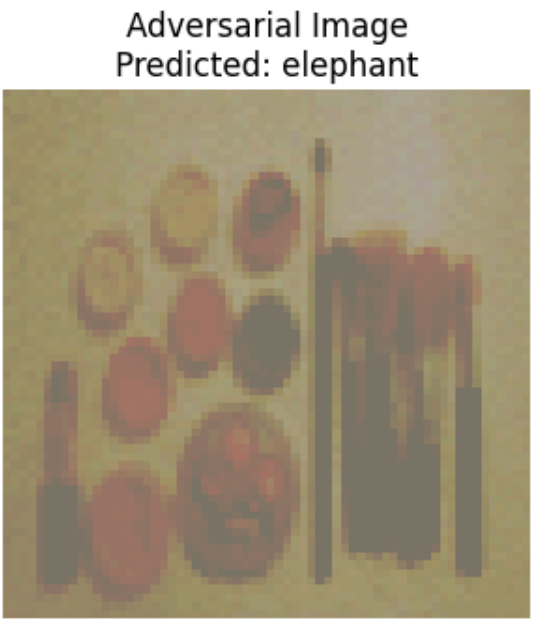
Adversarial Image
Predicted: handgun

Original Image
True Label: vase

Adversarial Image
Predicted: spider

Original Image
True Label: chicken

Adversarial Image
Predicted: seal



Question 3.7: **Train a robust model using adversarial training with PGD $\epsilon = 0.0313, k = 10, \eta = 0.002$. Write the code for the adversarial training and report the robust accuracies. After finishing the training, you need to store your best robust model in the folder `./models` and load the model to evaluate the robust accuracies for PGD and FGSM attacks with $\epsilon = 0.0313, k = 20, \eta = 0.002$ on the testing set.**

[4 points]

In [61]:
```
import torch
import torch.nn.functional as F
import numpy as np

# INITIALISE FGSM ATTACK FIRST
# This code is obtained from week 6 tutorial

def fgsm_attack(model, input_image, input_label=None, epsilon=0.3, num_steps = 20 ,
    """
    FGSM attack function to create adversarial examples by applying a perturbation

    Args:
        model: pretrained model (set to eval mode)
```

```
        input_image: original (clean) input image (tensor)
        input_label: original label (tensor, categorical representation) [Optional]
        epsilon: perturbation boundary (strength of the attack)
        clip_value_min: minimum value to clip the adversarial image to
        clip_value_max: maximum value to clip the adversarial image to

    Returns:
        adv_image: adversarial image (perturbed input image)
    """

    # Ensure input_image and input_label are on the correct device
    input_image = input_image.to(device)
    if input_label is not None:
        input_label = input_label.to(device)

    # Set the model to evaluation mode
    model.eval()

    # Ensure requires_grad is set to True for the attack
    adv_image = input_image.clone().detach().requires_grad_(True)

    # Forward pass: compute logits from the model
    output = model(adv_image)

    # If input label is provided, use it, else use the predicted label
    if input_label is not None:
        loss = F.cross_entropy(output, input_label)  # Targeted attack using the tr
    else:
        pred_label = output.argmax(dim=1)  # Untargeted attack using the predicted
        loss = F.cross_entropy(output, pred_label)

    # Zero all existing gradients in the model
    model.zero_grad()

    # Backward pass: compute gradients with respect to the adversarial image
    loss.backward()

    # Collect the sign of the gradients on the adversarial image
    gradient = adv_image.grad.data

    # Apply the perturbation (FGSM step)
    adv_image = adv_image + epsilon * gradient.sign()

    # Clip the adversarial image to maintain it within valid range
    adv_image = torch.clamp(adv_image, clip_value_min, clip_value_max)

    return adv_image.detach()
```

In [62]:

```
# ADVERSARIAL TRAINING FUNCTION , THIS  CODE IS OBTAINED FROM WEEK 6 TUTORIAL
def train_step_adv(model, x, x_adv, y, optimizer, criterion):
    model.train()
    optimizer.zero_grad()

    # Forward pass on both clean and adversarial examples
    logits = model(x)
    logits_adv = model(x_adv)

    # Loss on clean and adversarial examples
    loss = (criterion(logits, y) + criterion(logits_adv, y)) / 2
    loss.backward()
    optimizer.step()

    pred_adv = logits_adv.argmax(dim=1, keepdim=True)
    return loss.item(), pred_adv
```

In [ ]:
```
from sklearn.metrics import accuracy_score

# Metrics lists to track progress
train_loss = []
train_acc = []
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(best_model_with_mixup.parameters(), lr=0.001)

# Training loop with best model so far
epochs = 5
for epoch in range(epochs):
    best_model_with_mixup.train()
    total_loss = 0.0
    y_pred = []
    y_true = []

    for batch_idx, (x, y) in enumerate(train_loader):
        x, y = x.to(device), y.to(device)

        # Create adversarial examples using pgd
        x_adv = pgd_attack(best_model_with_mixup, x, y, epsilon=0.0313, num_steps=1

        # Perform adversarial training
        loss, pred_adversarial = train_step_adv(best_model_with_mixup, x, x_adv, y,

        # Log metrics
        total_loss += loss
        y_pred.extend(pred_adversarial.squeeze().cpu().numpy())
        y_true.extend(y.cpu().numpy())

    train_loss_epoch = total_loss / len(train_loader)
    train_acc_epoch = accuracy_score(y_true, y_pred)

    train_loss.append(train_loss_epoch)
    train_acc.append(train_acc_epoch)

    print(f"Epoch {epoch+1}/{epochs}, Training Loss: {train_loss_epoch:.4f}, Traini
```

```
Epoch 1/5, Training Loss: 2.3522, Training Accuracy: 16.76%
Epoch 2/5, Training Loss: 2.2472, Training Accuracy: 20.71%
Epoch 3/5, Training Loss: 2.2010, Training Accuracy: 22.50%
Epoch 4/5, Training Loss: 2.1688, Training Accuracy: 23.77%
Epoch 5/5, Training Loss: 2.1271, Training Accuracy: 24.76%
```

In [63]:
```
# Save the trained model
model_save_path = './models/best_adversarial_model.pth'
if not os.path.exists('./models'):
  os.makedirs('./models')  # Ensure the models directory exists
torch.save(best_model_with_mixup.state_dict(), model_save_path)  # Save best model
```

In [64]:
```
def evaluate_accuracy(model, test_loader, epsilon, num_steps, step_size):
    model.eval()  # Keep model in evaluation mode
    pgd_correct = 0
    fgsm_correct = 0
    total = 0

    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)

        # PGD attack evaluation
        pgd_adversarial = pgd_attack(model, images, labels, epsilon=epsilon, num_st
        # Test the model with input data that has been perturbed by PGD attack
        output_pgd = model(images + pgd_adversarial)
```

```
        _, prediction_pgd = torch.max(output_pgd, 1)
        pgd_correct += (prediction_pgd == labels).sum().item()

        # FGSM attack evaluation
        perturbed_images = fgsm_attack(model, images, labels, epsilon=epsilon, num_
        output_fgsm = model(perturbed_images)
        _, pred_fgsm = torch.max(output_fgsm, 1)
        fgsm_correct += (pred_fgsm == labels).sum().item()

        total += labels.size(0)

    pgd_acc = pgd_correct / total
    fgsm_acc = fgsm_correct / total

    print(f"Robust Accuracy under PGD attack: {pgd_acc*100:.2f}%")
    print(f"Robust Accuracy under FGSM attack: {fgsm_acc*100:.2f}%")

    return pgd_acc, fgsm_acc
```

In [68]:
```
model = YourCNN(list_feature_maps=[16, 32, 64], use_skip=True).to(device)
model.load_state_dict(torch.load('./models/best_adversarial_model.pth'))

pgd_acc, fgsm_acc = evaluate_accuracy(model, val_loader, epsilon = 0.0313, num_step
```

```
<ipython-input-68-7118ce71d122>:2: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle mo
dule implicitly. It is possible to construct malicious pickle data which will exec
ute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/
main/SECURITY.md#untrusted-models for more details). In a future release, the defa
ult value for `weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no longer be allo
wed to be loaded via this mode unless they are explicitly allowlisted by the user
via `torch.serialization.add_safe_globals`. We recommend you start setting `weight
s_only=True` for any use case where you don't have full control of the loaded fil
e. Please open an issue on GitHub for any issues related to this experimental feat
ure.
  model.load_state_dict(torch.load('./models/best_adversarial_model.pth'))
Robust Accuracy under PGD attack: 31.68%
Robust Accuracy under FGSM attack: 9.50%
```

In [ ]:

Question 3.8 (Kaggle competition)

[10 points] </div10

You can reuse the best model obtained in this assignment or develop new models to evaluate on the **testing set of the FIT3181/5215 Kaggle competion**. However, to gain any points for this question, your testing accuracy must **exceed** the accuracy threshold from a base model developed by us as shown in the leader board of the competition.

The marks for this question are as follows:

- If you are in *top 10%* of your cohort, you gain *10 points*.
- If you are in *top 20%* of your cohort, you gain *8 points*.
- If you are in *top 30%* of your cohort, you gain *6 points*.
- If you *beat* our base model, you gain *4 points*.

END OF ASSIGNMENT

GOOD LUCK WITH YOUR ASSIGNMENT 1!