PA_Part_I

April 24, 2025

1 *PA - Part I: Basic Vision Models [Experiments with MNIST]* (55pt)

Keywords: Multiclass Image Classification, Neural Networks, PyTorch

MNIST * The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. * The MNIST database contains 70,000 labeled images. Each datapoint is a 28×28 pixels grayscale image. * However to speed up computations, we will use a much smaller dataset with size 8×8 images. These images are loaded from `sklearn.datasets`.

Agenda: * The PA is split into three parts, the first part dealing with miniature models which we will build from scratch, the second part dealing with modern architectures and the bonus third part dealing with training vision models robust to adversarial attacks. * In this part, we will be performing multiclass classification on the simplified MNIST dataset from scratch. * We will be applying and analysing different loss functions, optimization techniques and learning methods.

- We will be using PyTorch to do most of the heavylifting for modeling and training.

Note: * Hardware acceleration (GPU) is recommended but not required for this part. * A note on working with GPU: * Take care that whenever declaring new tensors, set `device=device` in parameters. * You can also move a declared torch tensor/model to device using `.to(device)`. * To move a torch model/tensor to cpu, use `.to('cpu')` * Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU). * Run all the cells in order. * Only **add your code** to cells marked with "TODO:" or with "…" * You should not have to change variable names where provided, but you are free to if required for your implementation.

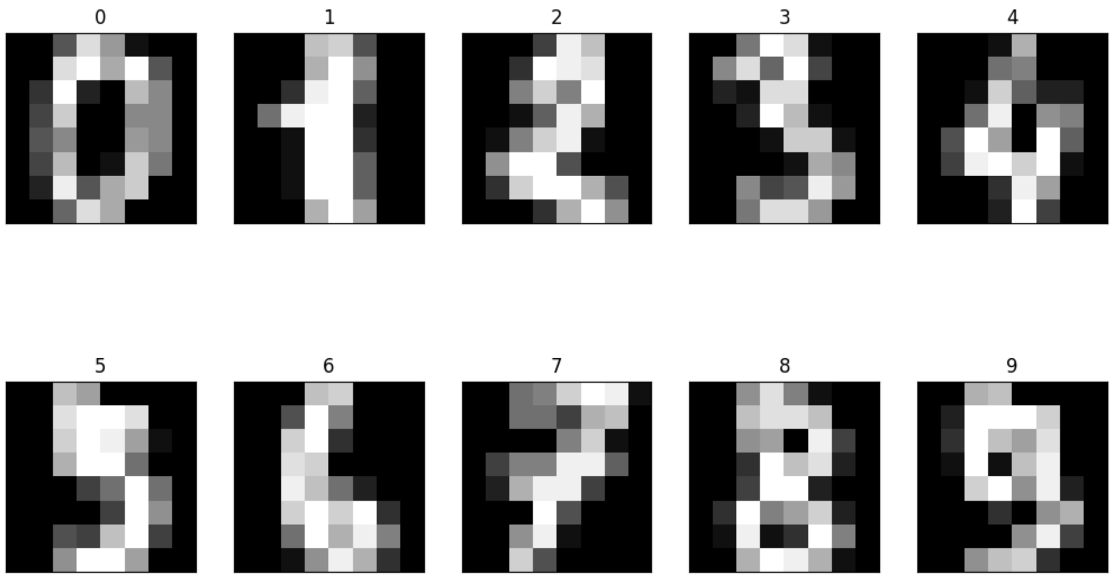
1.0.1 *Setup: Imports and Utils*

[86]:
```
# imports
import torch
import numpy as np
import math
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# loading the dataset directly from the scikit-learn library
dataset = load_digits()
X = dataset.data
y = dataset.target
print('Number of images:', X.shape[0])
print('Number of features per image:', X.shape[1])
```

```
Number of images: 1797
Number of features per image: 64
```

[87]:
```
# utility function to plot gallery of images
def plot_gallery(images, titles, height, width, n_row=2, n_col=4):
    plt.figure(figsize=(2* n_col, 3 * n_row))
    plt.subplots_adjust(bottom=0, left=0.01, right=0.99, top=0.90, hspace=0.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((height, width)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

# visualize some of the images of the MNIST dataset
plot_gallery(X, y, 8, 8, 2, 5)
```



[88]:
```
# Let us split the dataset into training and test sets in a stratified manner.
# Note that we are not creating evaluation datset as we will not be tuning␣
↪hyper-parameters
```

```
# The split ratio is 4:1
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42, stratify=y)
print('Shape of train dataset:', X_train.shape)
print('Shape of evaluation dataset:', X_test.shape)
```

```
Shape of train dataset: (1437, 64)
Shape of evaluation dataset: (360, 64)
```

[89]:
```
# define some constants - useful for later
num_classes = len(np.unique(y)) # number of target classes = 10 --␣
 ↪(0,1,2,3,4,5,6,7,8,9)
num_features = X.shape[1]       # number of features = 64
max_epochs = 100000             # max number of epochs for training
lr = 1e-2                       # learning rate
tolerance = 1e-6                # tolerance for early stopping during training
```

[90]:
```
# Hardware Acceleration: to set device if using GPU.
# You can change runtime in colab by naviagting to (Runtime->Change runtime␣
 ↪type), and selecting GPU in hardware accelarator.
# NOTE that you can run this homework without GPU.
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

1.0.2 *(a) Utils and pre-processing* (5pt)

- Scale the image data between 0 and 1
- One-hot encode the target data
- Convert the data to tensors and move them to the required device

[91]:
```
# 1. Scale the features between 0 and 1
# To scale, you can directly use the MinMaxScaler from sklearn.
#######
#TODO:
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range = (0,1))
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

# output variable names -  X_train, X_test
#######
print(X_train)
print(X_test)
```

```
[[0.     0.     0.6875 … 0.125  0.     0.    ]
 [0.     0.     0.125  … 0.0625 0.     0.    ]
 [0.     0.125  0.9375 … 0.     0.     0.    ]
 …
 [0.     0.     0.0625 … 0.1875 0.     0.    ]
 [0.     0.     0.25   … 0.     0.     0.    ]
 [0.     0.     0.1875 … 1.     1.     0.1875]]
[[0.         0.         0.5        … 0.25       0.         0.        ]
 [0.         0.42857143 1.         … 0.5        0.1875     0.        ]
 [0.         0.         0.125      … 0.0625     0.         0.        ]
 …
 [0.         0.         0.625      … 0.1875     0.         0.        ]
 [0.         0.         0.4375     … 0.75       0.         0.        ]
 [0.         0.         0.         … 0.125      0.         0.        ]]
```

[92]:
```
# 2. One-Hot encode the target labels
# To one-hot encode, you can use the OneHotEncoder from sklearn
#######
#TODO:
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(sparse_output=False) # To make the ohe not sparse (to avoid␣
 ↪the ambiguous length of the sparse array when converting them to torch)
y_train_ohe = ohe.fit_transform(y_train.reshape(-1,1)) # Reshape: ohe need a␣
 ↪2d-array input
y_test_ohe = ohe.fit_transform(y_test.reshape(-1,1))

# output variable names -  y_train_ohe, y_test_ohe
#######
print('Shape of y_train_ohe:',y_train_ohe.shape)
print('Shape of y_test_ohe:',y_test_ohe.shape)


# move X and y to the defined device and convert them to torch.float32
X_train_torch = torch.tensor(X_train, dtype=torch.float32, device=device)
X_test_torch = torch.tensor(X_test, dtype=torch.float32, device=device)
y_train_ohe_torch = torch.tensor(y_train_ohe, dtype=torch.float32,␣
 ↪device=device)
y_test_ohe_torch = torch.tensor(y_test_ohe, dtype=torch.float32, device=device)

# output variable names -  X_train_torch, X_test_torch, y_train_ohe_torch,␣
 ↪y_test_ohe_torch
print(X_train_torch)
print(X_test_torch)
print(y_train_ohe_torch)
print(y_test_ohe_torch)
```

```
Shape of y_train_ohe: (1437, 10)
Shape of y_test_ohe: (360, 10)
tensor([[0.0000, 0.0000, 0.6875,  …, 0.1250, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.1250,  …, 0.0625, 0.0000, 0.0000],
        [0.0000, 0.1250, 0.9375,  …, 0.0000, 0.0000, 0.0000],
        …,
        [0.0000, 0.0000, 0.0625,  …, 0.1875, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.2500,  …, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.1875,  …, 1.0000, 1.0000, 0.1875]],
       device='cuda:0')
tensor([[0.0000, 0.0000, 0.5000,  …, 0.2500, 0.0000, 0.0000],
        [0.0000, 0.4286, 1.0000,  …, 0.5000, 0.1875, 0.0000],
        [0.0000, 0.0000, 0.1250,  …, 0.0625, 0.0000, 0.0000],
        …,
        [0.0000, 0.0000, 0.6250,  …, 0.1875, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.4375,  …, 0.7500, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000,  …, 0.1250, 0.0000, 0.0000]],
       device='cuda:0')
tensor([[0., 0., 0.,  …, 0., 0., 0.],
        [1., 0., 0.,  …, 0., 0., 0.],
        [0., 0., 0.,  …, 0., 0., 0.],
        …,
        [0., 0., 0.,  …, 0., 0., 1.],
        [0., 0., 0.,  …, 1., 0., 0.],
        [0., 0., 0.,  …, 0., 0., 1.]], device='cuda:0')
tensor([[0., 0., 0.,  …, 0., 0., 0.],
        [0., 0., 1.,  …, 0., 0., 0.],
        [0., 0., 0.,  …, 0., 1., 0.],
        …,
        [1., 0., 0.,  …, 0., 0., 0.],
        [0., 0., 0.,  …, 0., 0., 0.],
        [0., 0., 0.,  …, 0., 0., 0.]], device='cuda:0')
```

1.0.3 *(b) Implement Multi-Class Logistic Regression from scratch* (15pt)

In this problem, we will apply multiclass logistic regression from scratch trained using gradient descent (GD) with Mean Squared Error (MSE) loss as the objective.

We will be using a linear model $y^{(i)} = W\mathbf{x}^{(i)}$, with the following notations:

- Weight matrix: $ W_{p×n}=

  $$\begin{bmatrix} \leftarrow & \mathbf{w}_1^\top & \rightarrow \\ \leftarrow & \mathbf{w}_2^\top & \rightarrow \\ & \vdots & \\ \leftarrow & \mathbf{w}_p^\top & \rightarrow \end{bmatrix}$$

  $, where p is the number of target classes

- data points: $\mathbf{x}^{(i)} \in \mathbb{R}^n, y^{(i)} \in \mathbb{R}$, and $X = \begin{bmatrix} \uparrow & \uparrow & \cdots & \uparrow \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ \downarrow & \downarrow & \cdots & \downarrow \end{bmatrix}, Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$, where m is the number of datapoints.

Note: Here we need to define the model prediction. The input matrix is $X_{n\times m}$ where m is the number of examples, and n is the number of features. The linear predictions can be given by: $Y = WX + b$ where W is a $p \times n$ weight matrix and b is a p size bias vector. p is the number of target classes.

#1. Define a function `linear_model`

- This function takes as input a weight matrix (`W`), bias vector (`b`), and input data matrix of size $m \times n$ (`XT`).
- This function should return the predictions $\hat{y}$, which is an $m \times p$ matrix. For each datapoint row which is a $1 \times p$ vector, the prediction can be seen as the scores that the model gives each target class for that datapoint

[93]:
```
#######
def linear_model(W, b, XT):
    #TODO:
    y_hat = XT @ W.T + b
    return y_hat

#######
```

Note: The loss function that we would be using is the Mean Square Error (L2) Loss: $MSE = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2$, where m is the number of examples, $\hat{y}^{(i)}$ is the predicted value of $x^{(i)}$ and $y^{(i)}$ is the ground truth of $x^{(i)}$.

#2. Define a function `mse_loss`

- This function takes as input prediction (`y_pred`) and ground-truth label (`y`), and returns the MSE loss.

[94]:
```
#######
def mse_loss(y_pred, y):
    #TODO:
    loss = torch.mean((y_pred - y) ** 2)
    return loss

#######
```

#3. Define a function: `initializeWeightsAndBiases` In this part, we will do some setup required for training (such as initializing weights and biases) and move everything to torch tensors.

- This function returns tuple `(W, b)`, where `W` is a randomly generated torch tensor of size `num_classes x num_features`, and `b` is a randomly generated torch vector of size `num_classes`.
- For both the tensors, set `requires_grad=True` in parameters.

[95]:
```
#######
def initializeWeightsAndBiases(num_classes, num_features):
    #TODO:
    W = torch.randn(num_classes, num_features, device=device,␣
 ↪requires_grad=True) # requires_grad: allow Gradient Updates
    b = torch.randn(num_classes, device=device, requires_grad=True)
    return W, b
#######
```

#4 Training code

- Given below is a function: `train_linear_regression_model` that takes as inputs as max number of epochs (`max_epochs`), weights (`W`), biases (`b`), training data (`X_train, y_train`), learning rate (`lr`), tolerance for stopping (`tolerance`).
- This function returns a tuple `(W,b,losses)` where `W,b` are the trained weights and biases respectively, and `losses` is a list of tuples of loss logged every 100th epoch.
- You can go through this article for reference.

[96]:
```
# Define a function train_linear_regression_model
def train_linear_regression_model(max_epochs, W, b, X_train, y_train, lr,␣
 ↪tolerance):
  #TODO:
  losses = []
  prev_loss = float('inf')

  for epoch in tqdm(range(max_epochs)):

      #######
      # 7. do prediction
      y_pred = linear_model(W, b, X_train)

      # 8. get the loss
      loss = mse_loss(y_pred, y_train)

      # 9. backpropagate loss
      loss.backward()

      # 10. update the weights and biasees
      with torch.no_grad():
        W -= lr * W.grad
        b -= lr * b.grad

      # 11. set the gradients to zero
      W.grad = None # Equivalent to W.grad.zero()
      b.grad = None

      #######

      # log the loss every 100th epoch and print every 5000th epoch:
        if epoch%100==0:
          losses.append((epoch, loss.item())) # loss.item(): tensor -> float num
          if epoch%5000==0:
            print('Epoch: {}, Loss: {}'.format(epoch, loss.item()))

        # break if decrease in loss is less than threshold
        if abs(prev_loss - loss.item()) < tolerance:
          print(f"Break! Early stopping at epoch {epoch} due to low improvement␣
↪in loss.")
          break
        prev_loss = loss.item()

    # return updated weights, biases, and logged losses
    return W, b, losses
```

#5. Initialize parameters and train your own model

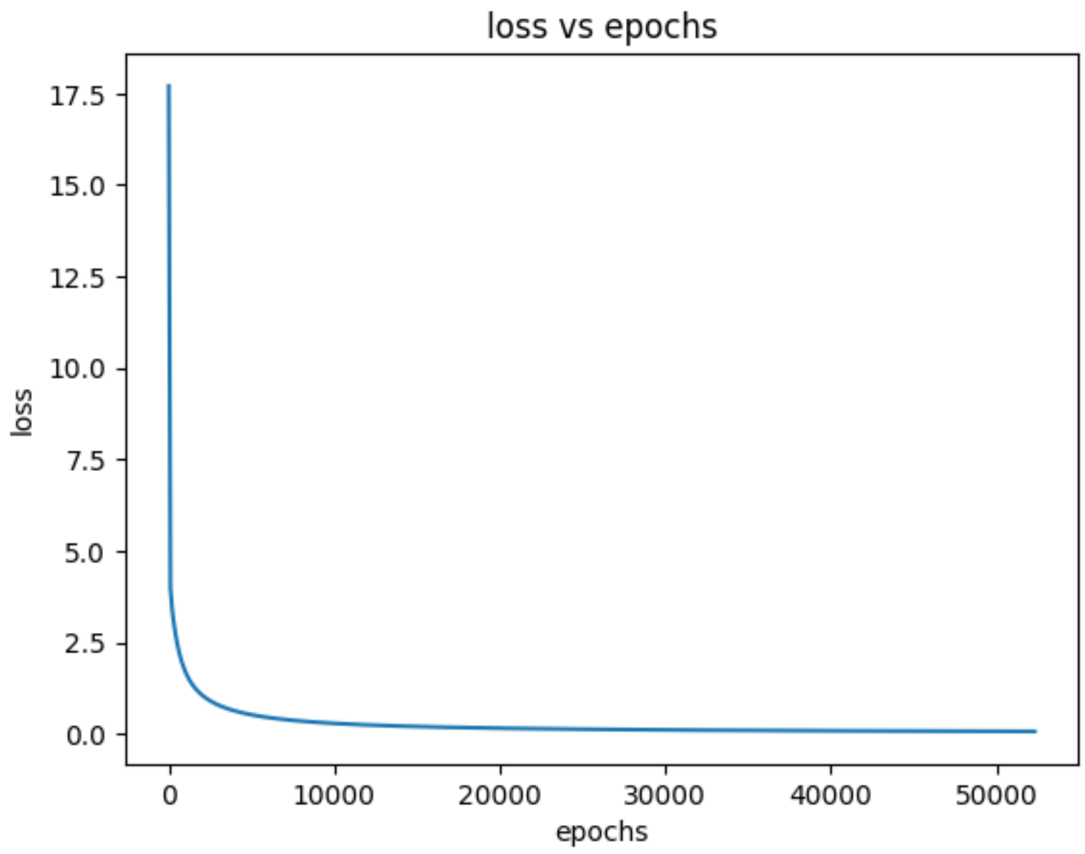
- Initialize weights and biases using the `initializeWeightsAndBiases` function that you defined earlier
- Train your model using function `train_linear_regression_model` defined above.
- Use full batch (set `batch_size=len(X_train)` for training (Gradient Descent).
- Also plot the graph of loss vs number of epochs (Recall that values for learning rate (`lr`) and tolerance (`tolerance`) are already defined above).

[97]:
```
#######
#TODO:
W, b = initializeWeightsAndBiases(num_classes, num_features)

W, b, losses = train_linear_regression_model(
    max_epochs=max_epochs,
    W=W,
    b=b,
    X_train=X_train_torch,
    y_train=y_train_ohe_torch,
    lr=lr,
    tolerance=tolerance
)

import matplotlib.pyplot as plt
plt.plot([x[0] for x in losses],[x[1] for x in losses])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('loss vs epochs')
#######
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]

Epoch: 0, Loss: 17.706771850585938
Epoch: 5000, Loss: 0.5214282870292664
```

```
Epoch: 10000, Loss: 0.2865198254585266
Epoch: 15000, Loss: 0.19979605078697205
Epoch: 20000, Loss: 0.15468721091747284
Epoch: 25000, Loss: 0.12691180408000946
Epoch: 30000, Loss: 0.1080709770321846
Epoch: 35000, Loss: 0.09449117630720139
Epoch: 40000, Loss: 0.08429207652807236
Epoch: 45000, Loss: 0.07640080899000168
Epoch: 50000, Loss: 0.07015533745288849
Break! Early stopping at epoch 52374 due to low improvement in loss.
```

[97]:
```
Text(0.5, 1.0, 'loss vs epochs')
```



[98]:
```
# print accuracies of model
predictions_train = linear_model(W, b, X_train_torch)
predictions_test = linear_model(W, b, X_test_torch)

# Get the predicted label with the maximum score in score matrix
y_train_pred = torch.argmax(predictions_train, dim=1).cpu().numpy() #␣
 ↪accuracy_score() can only process np.ndarray
```

```
y_test_pred = torch.argmax(predictions_test, dim=1).cpu().numpy()

print("Train accuracy:",accuracy_score(y_train_pred, np.asarray(y_train,␣
 ↪dtype=np.float32)))
print("Test accuracy:",accuracy_score(y_test_pred, np.asarray(y_test, dtype=np.
 ↪float32)))
```

```
Train accuracy: 0.8413361169102297
Test accuracy: 0.7861111111111111
```

1.0.4 *(c) Use PyTorch for training* (10pt)

- In the previous part, we defined the model, loss, and even the gradient update step. We also had to manually set the gradients to zero.
- In this part, we will re-implement the linear model and see how we can directly use Pytorch to do all this for us in a few simple steps.

[99]:
```
# common utility function to print accuracies
def print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test):
  predictions_train = model(X_train_torch)
  predictions_test = model(X_test_torch)
  y_train_pred = torch.argmax(predictions_train, dim=1).cpu().numpy()
  y_test_pred = torch.argmax(predictions_test, dim=1).cpu().numpy()
  print("Train accuracy:",accuracy_score(y_train_pred, np.asarray(y_train,␣
 ↪dtype=np.float32)))
  print("Test accuracy:",accuracy_score(y_test_pred, np.asarray(y_test,␣
 ↪dtype=np.float32)))
```

#1. Define the linear model using PyTorch

- Use the inbuilt PyTorch `torch.nn.Module` to define a model class.
- Use the `torch.nn.Linear` to define the linear layer of the model

[100]:
```
#######

# Define a model class using torch.nn
class Linear_Model(torch.nn.Module):
  def __init__(self):
    super(Linear_Model, self).__init__()
    # Initalize various layers of model as below
    # 1. initialze one linear layer: num_features -> num_targets
    self.linear = torch.nn.Linear(num_features, num_classes)

  def forward(self, X):
    # 2. define the feedforward algorithm of the model and return the final␣
 ↪output
    output = self.linear(X)
    return output
```

```
#######
```

#2. A general function for training a PyTorch model. Define a general training function: `train_torch_model`. * This function takes as input an initialized torch model (`model`), batch size (`batch_size`), initialized loss (`criterion`), max number of epochs (`max_epochs`), training data (`X_train, y_train`), learning rate (`lr`), tolerance for stopping (`tolerance`). * This function will return a tuple `(model, losses)`, where `model` is the trained model, and `losses` is a list of tuples of loss logged every 100^{th} epoch.

Note: You can reuse a lot of components from the `train_linear_regression_model` function from the previous section

You can go through this article for reference.

[101]:
```
# Define a function train_torch_model
def train_torch_model(model, batch_size, criterion, max_epochs, X_train,␣
 ↪y_train, lr, tolerance):
  losses = []
  prev_loss = float('inf')
  number_of_batches = math.ceil(len(X_train)/batch_size)

  #######
  # 3. move model to device
  model.to(device)

  # 4. define optimizer (use torch.optim.SGD (Stochastic Gradient Descent))
  # Set learning rate to lr and also set model parameters
  optimizer = torch.optim.SGD(model.parameters(), lr=lr)

  for epoch in tqdm(range(max_epochs)):
    for i in range(number_of_batches):
      start = i * batch_size
      end = start + batch_size
      X_train_batch = X_train[start:end]
      y_train_batch = y_train[start:end]

      # 5. reset gradients
      optimizer.zero_grad()

      # 6. prediction
      prediction = model(X_train_batch)

      # 7. calculate loss
      loss = criterion(prediction, y_train_batch)

      # 8. backpropagate loss
      loss.backward()
```

```
        # 9. perform a single gradient update step
        optimizer.step()

    #######

      # log loss every 100th epoch and print every 5000th epoch:
      if epoch%100==0:
        losses.append((epoch, loss.item()))
        if epoch%5000==0:
          print('Epoch: {}, Loss: {}'.format(epoch, loss.item()))

      # break if decrease in loss is less than threshold
      if abs(prev_loss - loss.item()) < tolerance:
        print(f"Break! Early stopping at epoch {epoch}")
        break
      prev_loss = loss.item()

    # return updated model and logged losses
    return model, losses
```

1.0.5 *(d) Working with different model types* (15pt)

- Now, we retrain the above model with `batch_size=64`
- Use Stochastic/Mini-batch Gradient Descent and keep everything else the same.
- Like before, we plot the graph between loss and number of epochs.

#1. MSE Loss and Gradient Descent (GD)

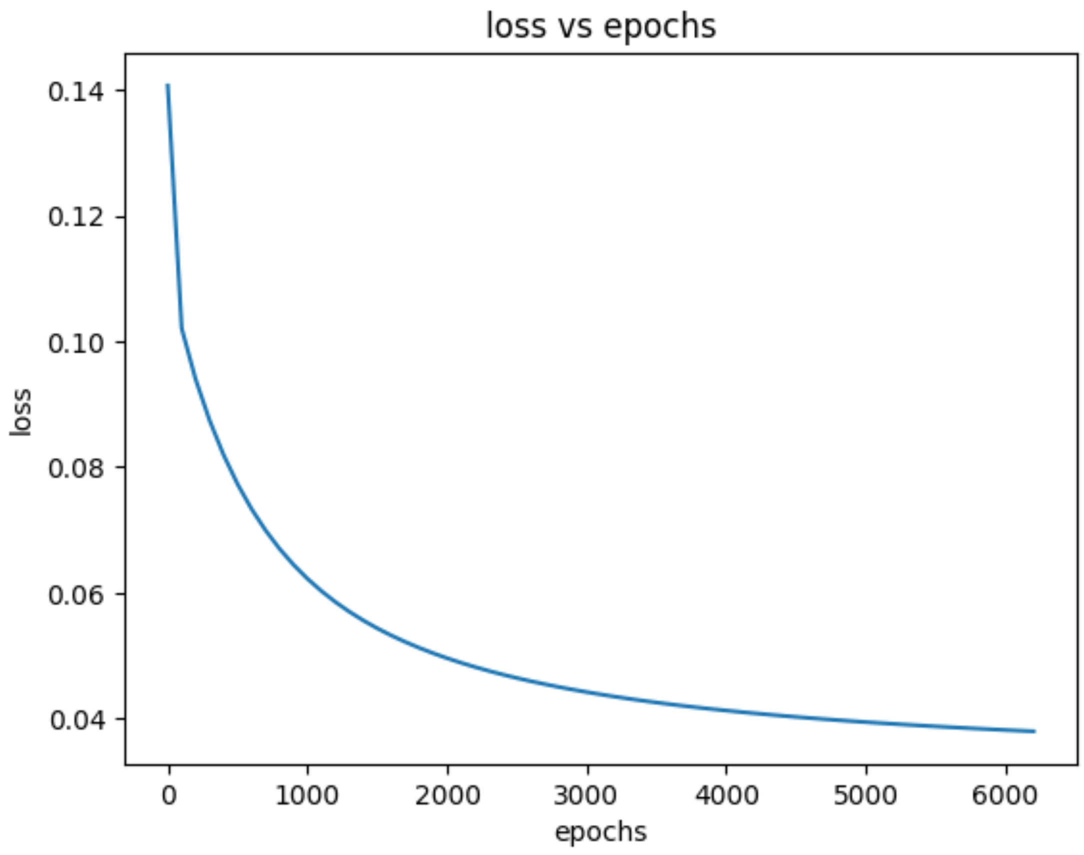
- Use `nn.MSELoss`.
- Use full batch for training (Gradient Descent).
- Also plot the graph of loss vs number of epochs.

[102]:
```
#######
model = Linear_Model()
criterion = torch.nn.MSELoss()
batch_size = X_train_torch.shape[0] # Full batch: GD

model, losses = train_torch_model(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train_torch,
    y_train=y_train_ohe_torch,
    lr=lr,
    tolerance=tolerance
)
```

```
import matplotlib.pyplot as plt
plt.plot([x[0] for x in losses],[x[1] for x in losses])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('loss vs epochs')
#######
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]

Epoch: 0, Loss: 0.14073053002357483
Epoch: 5000, Loss: 0.039423793554306O3
Break! Early stopping at epoch 6200
```

[102]:
```
Text(0.5, 1.0, 'loss vs epochs')
```



[103]:
```
# print accuracies of model
print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test)
```

```
Train accuracy: 0.9290187891440501
Test accuracy: 0.9361111111111111
```

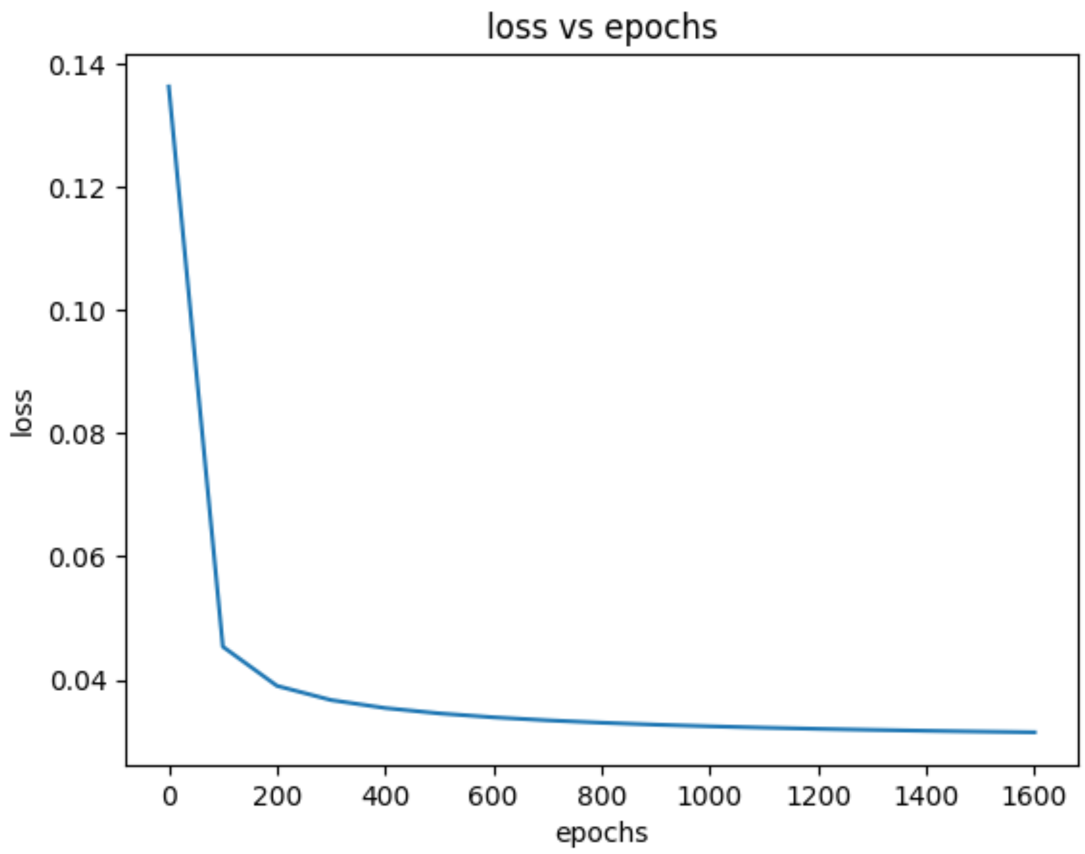
#2. MSE Loss and Stochastic Gradient Descent (SGD)

- Use `nn.MSELoss`.
- Use `batch_size=64` for training
- Also plot the graph of loss vs number of epochs.

[104]:
```python
#######
model = Linear_Model()
criterion = torch.nn.MSELoss()
batch_size = 64 # Small BatchL: SGD
model, losses = train_torch_model(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train_torch,
    y_train=y_train_ohe_torch,
    lr=lr,
    tolerance=tolerance
)

import matplotlib.pyplot as plt
plt.plot([x[0] for x in losses],[x[1] for x in losses])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('loss vs epochs')
#######
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]

Epoch: 0, Loss: 0.13629525899887085
Break! Early stopping at epoch 1678
```

[104]:
```
Text(0.5, 1.0, 'loss vs epochs')
```



[105]:
```python
# print accuracies of model
print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test)
```

```
Train accuracy: 0.9450243562978428
Test accuracy: 0.95
```

Cross-Entropy (CE) Loss with Linear Model

- Instead of using MSE Loss, we will use a much more natural loss function for the multi-class logistic regression task which is the Cross Entropy Loss.
- CE loss converts the model scores of each class into a probability.
- This model penalizes both choosing the wrong class as well as uncertainty (choosing the right class with low probability).
- And we will use the same linear model defined in **(c)**.

Note: The Cross Entropy Loss for multiclass classification is the mean of the negative log likelihood of the output logits after softmax:

$$L = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\underbrace{- y^{(i)} \underbrace{log \underbrace{\frac{e^{\hat{y}^{(i)}}}{\sum_{j=1}^{p} e^{\hat{y}^{(j)}}}}_{\text{Softmax}}}_{\text{LogSoftmax}}}_{\text{Negative Log Likelihood (NLL)}}}_{\text{Cross Entropy (CE) Loss}},$$

where $y^{(i)}$ is the ground truth, and $\hat{y}^{(k)}$ (also called as *logits*) represent the outputs of the last linear layer of the model.

#3. CE Loss and GD

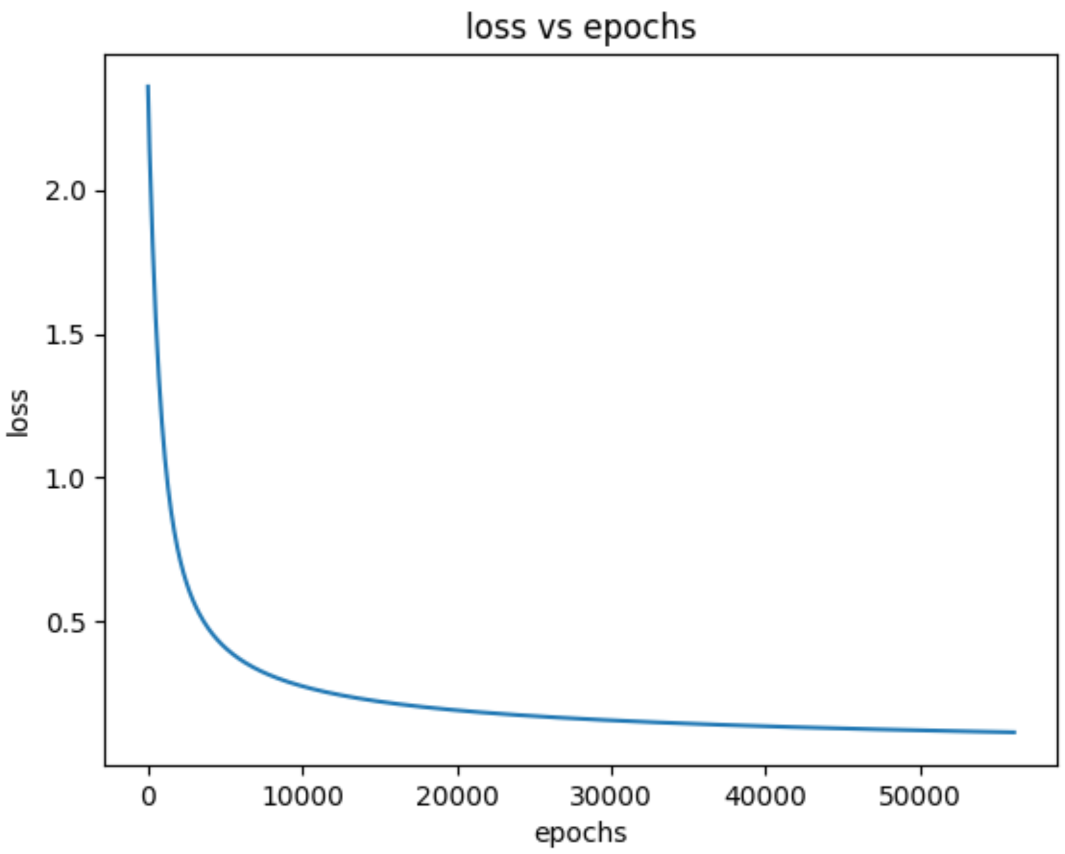
- Instead of `nn.MSELoss`, train the linear model with `nn.CrossEntropyLoss`.
- Use **full-batch**.
- Also plot the graph between loss and number of epochs.

[106]:
```
#######
model = Linear_Model()
criterion = torch.nn.CrossEntropyLoss()
batch_size = X_train_torch.shape[0]
model, losses = train_torch_model(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train_torch,
    y_train=y_train_ohe_torch,
    lr=lr,
    tolerance=tolerance
)

import matplotlib.pyplot as plt
plt.plot([x[0] for x in losses],[x[1] for x in losses])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('loss vs epochs')
#######
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]

Epoch: 0, Loss: 2.3591091632843018
Epoch: 5000, Loss: 0.40864360332489014
Epoch: 10000, Loss: 0.27448174357414246
Epoch: 15000, Loss: 0.22115346789360046
Epoch: 20000, Loss: 0.1908445954322815
Epoch: 25000, Loss: 0.17064359784126282
Epoch: 30000, Loss: 0.15589269995689392
Epoch: 35000, Loss: 0.1444670855998993
Epoch: 40000, Loss: 0.13524574041366577
Epoch: 45000, Loss: 0.12757539749145508
```

```
Epoch: 50000, Loss: 0.12104677408933364
Epoch: 55000, Loss: 0.11538895219564438
Break! Early stopping at epoch 56144
```

[106]:
```
Text(0.5, 1.0, 'loss vs epochs')
```



[107]:
```
# print accuracies of model
print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test)
```

```
Train accuracy: 0.9798190675017397
Test accuracy: 0.9638888888888889
```

#4. CE Loss and SGD

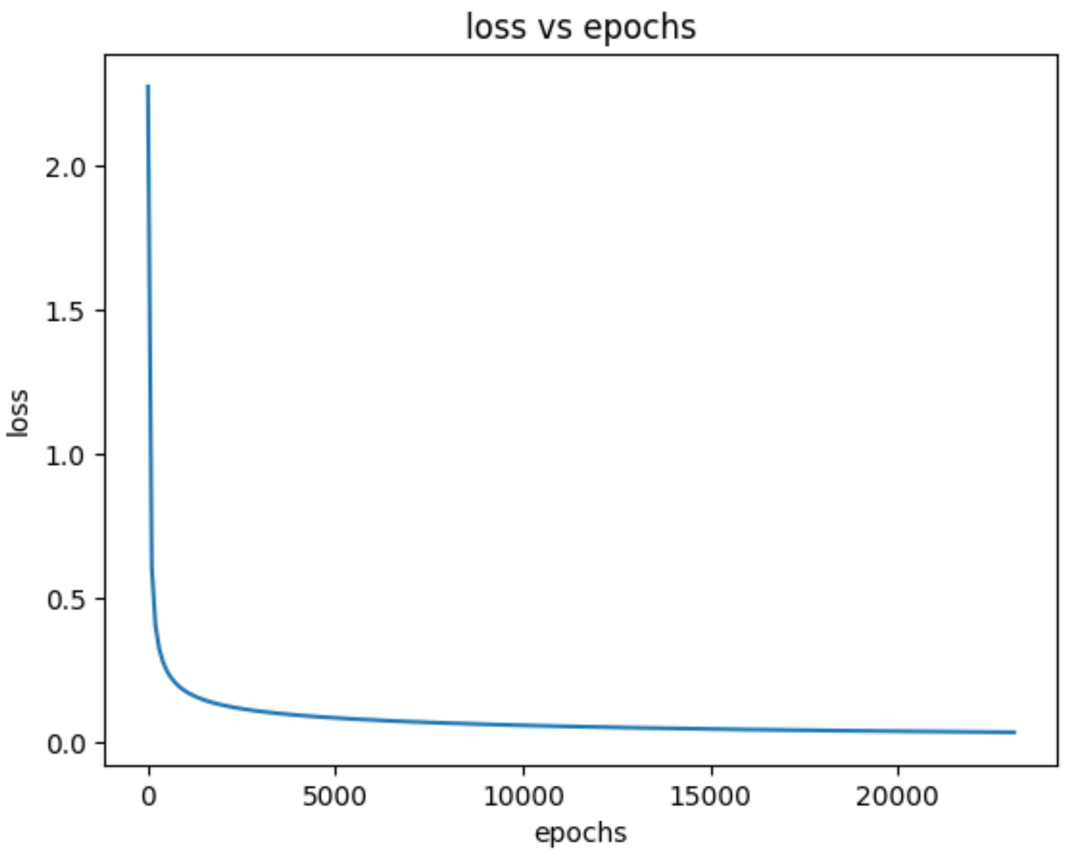
- Use a different batch size, `batch_size=64` with the new loss and repeat the previous part #1.
- Also plot the graph of loss vs epochs.

[108]:
```
#######
model = Linear_Model()
criterion = torch.nn.CrossEntropyLoss()
```

```
batch_size = 64
model, losses = train_torch_model(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train_torch,
    y_train=y_train_ohe_torch,
    lr=lr,
    tolerance=tolerance
)

import matplotlib.pyplot as plt
plt.plot([x[0] for x in losses],[x[1] for x in losses])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('loss vs epochs')
#######
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]

Epoch: 0, Loss: 2.277090311050415
Epoch: 5000, Loss: 0.0869542807340622
Epoch: 10000, Loss: 0.061202846467494965
Epoch: 15000, Loss: 0.04852709919214249
Epoch: 20000, Loss: 0.04068692773580551
Break! Early stopping at epoch 23106
```

```
[108]: Text(0.5, 1.0, 'loss vs epochs')
```



[109]:
```
# print accuracies of model
print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test)
```

```
Train accuracy: 0.9979123173277662
Test accuracy: 0.9694444444444444
```

Training a neural network model in PyTorch

- We will train a neural network in pytorch with two hidden layers of sizes 32 and 16 neurons.
- We will use non-linear ReLU activations to effectively make it a non-linear model.
- We will use this neural network model for multi-class classification with Cross Entropy Loss.

Note: The neural network model output can be represented mathematically as below: $\hat{y}^{(i)}_{10\times1} = W^{(3)}_{10\times16}\sigma(W^{(2)}_{16\times32}\sigma(W^{(1)}_{32\times64}\mathbf{x}^{(i)}_{64\times1} + \mathbf{b}^{(1)}_{32\times1}) + \mathbf{b}^{(2)}_{16\times1}) + \mathbf{b}^{(3)}_{10\times1}$, where σ represents ReLU activation, $W^{(i)}$ is the weight of the i^{th} linear layer, and $\mathbf{b}^{(i)}$ is the layer's bias. We use the subscript to denote the dimension for clarity.

#5. Define the 2 hidden-layer Neural Network (NN)

[110]:
```
#######
# Define a neural network model class using torch.nn
class NN_Model(torch.nn.Module):
  def __init__(self):
    super(NN_Model, self).__init__()
    # Initalize various layers of model as instructed below
    # 1. initialize three linear layers: num_features -> 32, 32 -> 16, 16 ->␣
 ↪num_targets
    self.fc1 = torch.nn.Linear(num_features, 32)
    self.fc2 = torch.nn.Linear(32, 16)
    self.fc3 = torch.nn.Linear(16, num_classes)

    # 2. initialize RELU
    self.relu = torch.nn.ReLU()

  def forward(self, X):
    # 3. define the feedforward algorithm of the model and return the final␣
 ↪output
    # Apply non-linear ReLU activation between subsequent layers
    x1 = self.relu(self.fc1(X)) # Layer 1 -> Layer 2
    x2 = self.relu(self.fc2(x1)) # Layer 2 -> Layer 3
    output = self.fc3(x2)
    return output

#######

print(X_train_torch.shape)
print(num_features)
print(num_classes)
```

```
torch.Size([1437, 64])
64
10
```

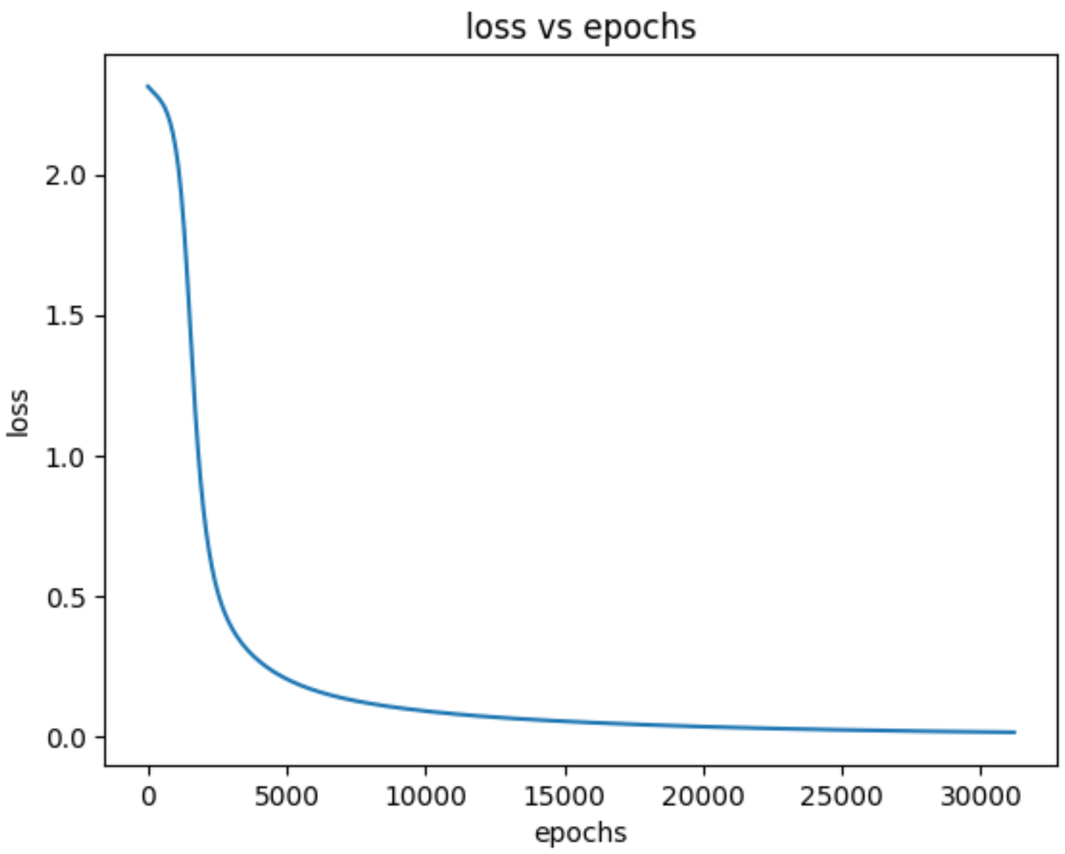
#6. NN with CE Loss and GD

- Use Cross Entropy Loss.
- Use full-batch and plot the graph of loss vs number of epochs.
- Note that you can re-use the training function `train_torch_model` (from part (b)).

[135]:
```
#######
model = NN_Model()
criterion = torch.nn.CrossEntropyLoss()
batch_size = X_train_torch.shape[0]
model, losses = train_torch_model(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train_torch,
    y_train=y_train_ohe_torch,
    lr=lr,
    tolerance=tolerance
)

import matplotlib.pyplot as plt
plt.plot([x[0] for x in losses],[x[1] for x in losses])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('loss vs epochs')
#######
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]

Epoch: 0, Loss: 2.3116660118103027
Epoch: 5000, Loss: 0.20603753626346588
Epoch: 10000, Loss: 0.09224589914083481
Epoch: 15000, Loss: 0.055876996368169785
Epoch: 20000, Loss: 0.03684386983513832
Epoch: 25000, Loss: 0.025357631966471672
Epoch: 30000, Loss: 0.01810634322464466
Break! Early stopping at epoch 31236
```

```
[135]: Text(0.5, 1.0, 'loss vs epochs')
```



[136]:
```
# print accuracies of model
print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test)
```

```
Train accuracy: 0.9993041057759221
Test accuracy: 0.9555555555555556
```

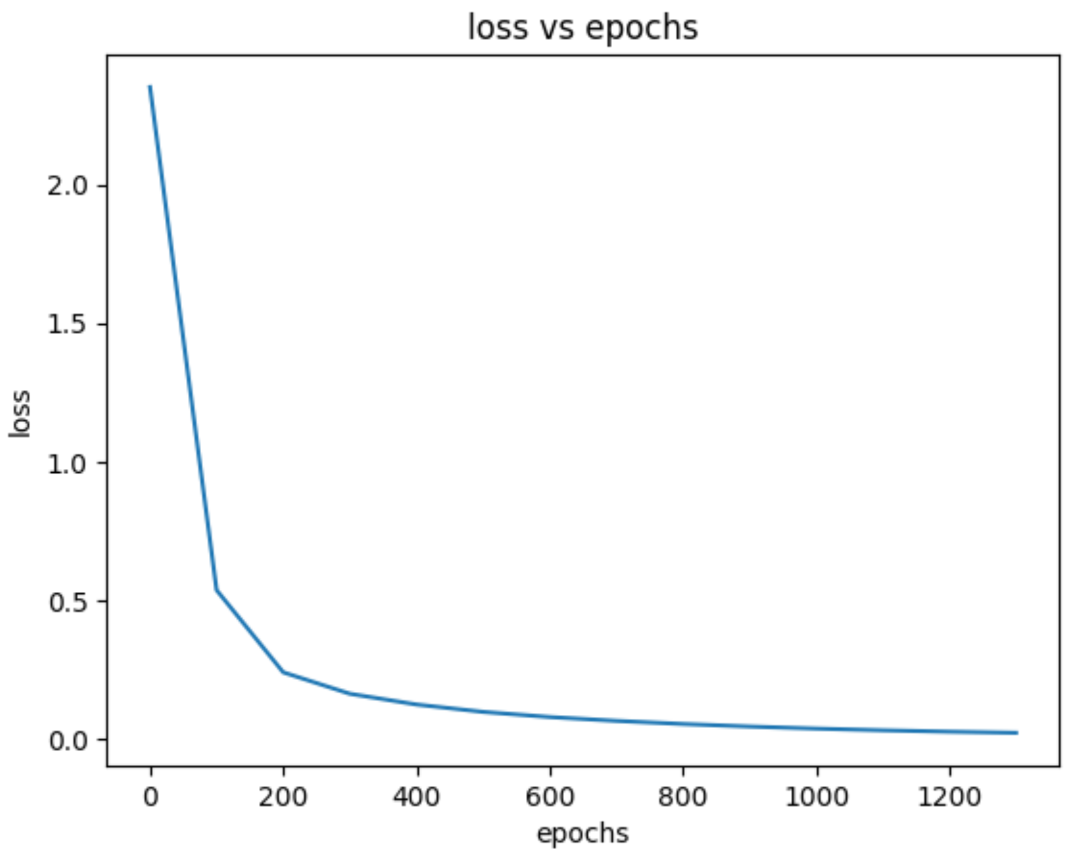
#7. NN with CE Loss and SGD

- Re-train the above model with `batch_size=64`.
- Also plot the graph of loss vs epochs.

[128]:
```
#######
model = NN_Model()
criterion = torch.nn.CrossEntropyLoss()
batch_size = 64
model, losses = train_torch_model(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train_torch,
    y_train=y_train_ohe_torch,
    lr=lr,
    tolerance=tolerance
)

import matplotlib.pyplot as plt
plt.plot([x[0] for x in losses],[x[1] for x in losses])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('loss vs epochs')
#######
```

```
  0%|          | 0/100000 [00:00<?, ?it/s]

Epoch: 0, Loss: 2.35133957862854
Break! Early stopping at epoch 1312
```

```
[128]: Text(0.5, 1.0, 'loss vs epochs')
```



[129]:
```
# print accuracies of model
print_accuracies_torch(model, X_train_torch, X_test_torch, y_train, y_test)
```

```
Train accuracy: 0.9993041057759221
Test accuracy: 0.9638888888888889
```

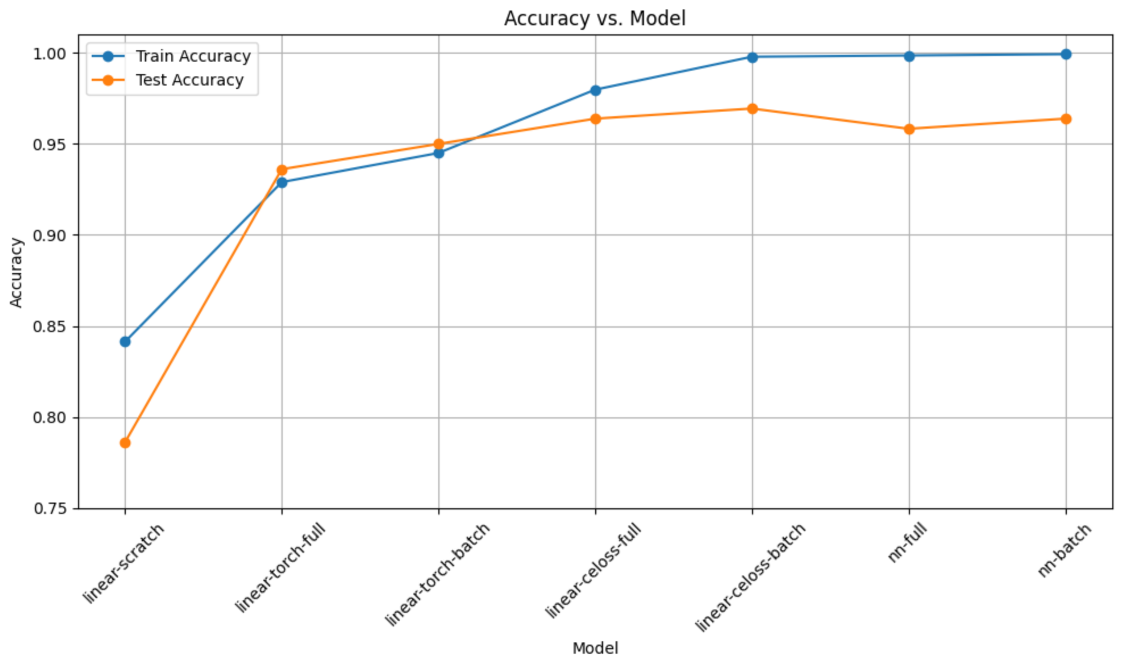
1.0.6 *(e) Analyze the results* (10pt)

In the above few examples, we performed several experiments with different batch size and loss functions. Now it's time to analyze our observations from the results.

Recall that we trained the following models in this tutorial:

1. Linear Model - Scratch + MSE Loss + Full Batch (GD)
2. Linear Model - PyTorch + MSE Loss + Full Batch
3. Linear Model - PyTorch + MSE Loss + Mini Batch (SGD)
4. Linear Model - PyTorch + CE Loss + Full Batch
5. Linear Model - PyTorch + CE Loss + Mini Batch
6. NN Model - PyTorch + CE Loss + Full Batch
7. NN Model - PyTorch + CE Loss + Mini Batch

[2]:
```
import matplotlib.pyplot as plt

xlabels = ['linear-scratch', 'linear-torch-full', 'linear-torch-batch',
           'linear-celoss-full', 'linear-celoss-batch', 'nn-full', 'nn-batch']
train_accuracies = [0.8413361169102297, 0.9290187891440501, 0.9450243562978428,
                    0.9798190675017397, 0.9979123173277662, 0.9986082115518441,␣
 ↪0.9993041057759221]
test_accuracies = [0.7861111111111111, 0.9361111111111111, 0.95,
                   0.9638888888888889, 0.9694444444444444, 0.9555555555555556,␣
 ↪0.9638888888888889]

plt.figure(figsize=(10, 6))
plt.plot(xlabels, train_accuracies, marker='o', label='Train Accuracy')
plt.plot(xlabels, test_accuracies, marker='o', label='Test Accuracy')

plt.xticks(rotation=45)
plt.ylim(0.75, 1.01)
plt.title('Accuracy vs. Model')
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



#1.Analysis (Four Questions)

# Effect of using full vs. batch gradient descent:

Effect: Mini-batch gradient descent helps mitigate overfitting and generally leads to better generalization performance.

Experimental Evidence: Comparisons between (2 vs 3), (4 vs 5), and (6 vs 7) show that, after applying mini-batch gradient descent:

- Both MSE and Cross-Entropy (CE) losses exhibit a clear performance gain in linear models—likely due to improved generalization.
- For neural networks (NN), the improvements are less significant.

Possible Reasons:

- Full-batch gradient descent calculates exact gradients based on the entire dataset, which provides a stable update direction but may get stuck in local optima and is more prone to overfitting. (Both linear model settings using full batch showed signs of overfitting.)
- Mini-batch gradient descent uses small random subsets of data to approximate the gradient, resulting in more frequent updates. The noise introduced by the approximation can help models escape local minima and improve generalization. It may also accelerate convergence in practice.

# Effect of using different loss strategy:

Effect: Cross-Entropy loss is more suitable for classification tasks and tends to result in better test accuracy.

Experimental Evidence:

- (2 vs 4) Full-batch Linear Model: CE outperforms MSE
- (3 vs 5) Mini-batch Linear Model: CE also outperforms MSE

Possible Reasons:

- Cross-Entropy measures the KL divergence between predicted and true label distributions, making it more effective for classification tasks.
- In contrast, Mean Squared Error (MSE) simply measures L2 distance between two vectors. When applied to one-hot encoded labels, it penalizes all output dimensions, even when the prediction is very close to the correct class, leading to suboptimal loss signals.

# Effect of using linear vs. non-linear models:

Effect: Neural networks (non-linear model) are generally better at capturing nonlinear patterns and tend to outperform linear models on high-dimensional or complex data. However, in this experiment, the performance gain is limited (MNIST is a relatively easy task).

Experimental Evidence: - (4 vs 6), (5 vs 7) comparisons show that both linear models and neural networks achieve similar performance on the MNIST classification task.

Possible Reasons: MNIST is relatively simple and highly linearly separable. As a result, linear models already achieve strong performance, and the added nonlinearity of neural networks offers little benefit in this case. This suggests the task does not demand complex feature representations.

# Training time per epoch in different cases:

Observation:

- SGD-based training typically takes longer than GD
- Training linear models tends to take longer than training NNs
- There is no significant time difference between using CE and MSE

Possible Reasons:

- SGD vs GD -> (2 vs 3), (4 vs 5), (6 vs 7): Although SGD introduces noisy updates, it helps models avoid getting trapped in local minima and results in more stable training. Due to its faster early descent (as seen in training curves), it often triggers early stopping after fewer epochs—typically about half as many as full GD. Despite faster per-step computations, the frequent updates in mini-batch training increase total update counts, often resulting in longer overall training time than GD.
- Linear vs NN -> (4 vs 6), (5 vs 7): Neural networks converge more quickly because of their stronger learning capacity, which trigger early stopping sooner than linear models.

PA_Part_II

April 24, 2025

1 *PA - Part II: Advanced Vision Models [Experiments with CIFAR10]* (45pt)

Keywords: Multiclass Image Classification, ResNet, Vision Transformers, CLIP

CIFAR10 * The CIFAR10 database is a large database with images of objects like airplane, bird, cat, dog, etc. * It contains 70,000 labeled images. Each datapoint is a 32×32 pixels RGB image. * To provide a realistic problem, we will use the images at this standard resolution of 32×32.

Agenda: * The PA is split into three parts, the first part dealing with miniature models which we will build from scratch, the second part dealing with modern architectures and the bonus third part dealing with training vision models robust to adversarial attacks. * In this part, we will be moving towards more modern architectures and a more realistic problem, finetuning pretrained model of different architectures on the CIFAR10 dataset and comparing its performance with models trained from scratch (initialized with random weights). * We will also be evaluating CLIP, which uses an architecture which is a backbone or helper model in most modern LLMs for image analysis and even in image generation models. * We will be using PyTorch and HuggingFace for loading and reusing complex model architecture.

Note: * Hardware acceleration (GPU) is **highly recommended** for this part as we will be training comparitively larger models! * If you are running locally, increase the `num_workers` parameter in the dataloaders to speed up computations. As Colab vCPUs have limits, it tends to be slower. * A note on working with GPU: * Take care that whenever declaring new tensors, set `device=device` in parameters. * You can also move a declared torch tensor/model to device using `.to(device)`. * To move a torch model/tensor to cpu, use `.to('cpu')` * Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU). * Run all the cells in order. * Only **add your code** to cells marked with "TODO:" or with "…" * You should not have to change variable names where provided, but you are free to if required for your implementation.

1.0.1 Pre-training and fine-tuning

- In this part, we want to use modern architectures, namely the **ResNet** and **Vision Transformer**, for classification on the CIFAR10 dataset.
- Instead of trying to train these modern architectures from random weights for our task, we can instead use **pre-trained** models that have already been trained on a large amount of data.
- We can then **fine-tune** these models for our specific tasks.

- ImageNet is one such large-scale dataset (~1.3 million images) that is widely used for pre-training neural network models.
- These pretrained models serve as good initializations for various image-related tasks.
- We compare fine-tuned models and models trained from random initialization to analyze the difference in performance of these approaches.

1.0.2 *Setup: Imports and Utils*

```
[ ]: !pip install open_clip_torch
```

```
Collecting open_clip_torch
  Downloading open_clip_torch-2.32.0-py3-none-any.whl.metadata (31 kB)
Requirement already satisfied: torch>=1.9.0 in /usr/local/lib/python3.11/dist-
packages (from open_clip_torch) (2.6.0+cu124)
Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist-
packages (from open_clip_torch) (0.21.0+cu124)
Requirement already satisfied: regex in /usr/local/lib/python3.11/dist-packages
(from open_clip_torch) (2024.11.6)
Collecting ftfy (from open_clip_torch)
  Downloading ftfy-6.3.1-py3-none-any.whl.metadata (7.3 kB)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages
(from open_clip_torch) (4.67.1)
Requirement already satisfied: huggingface-hub in
/usr/local/lib/python3.11/dist-packages (from open_clip_torch) (0.30.2)
Requirement already satisfied: safetensors in /usr/local/lib/python3.11/dist-
packages (from open_clip_torch) (0.5.3)
Requirement already satisfied: timm in /usr/local/lib/python3.11/dist-packages
(from open_clip_torch) (1.0.15)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-
packages (from torch>=1.9.0->open_clip_torch) (3.18.0)
Requirement already satisfied: typing-extensions>=4.10.0 in
/usr/local/lib/python3.11/dist-packages (from torch>=1.9.0->open_clip_torch)
(4.13.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-
packages (from torch>=1.9.0->open_clip_torch) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages
(from torch>=1.9.0->open_clip_torch) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages
(from torch>=1.9.0->open_clip_torch) (2025.3.2)
Collecting nvidia-cuda-nvrtc-cu12==12.4.127 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-runtime-cu12==12.4.127 (from
torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cuda-cupti-cu12==12.4.127 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-
```

```
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in
/usr/local/lib/python3.11/dist-packages (from torch>=1.9.0->open_clip_torch)
(0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch>=1.9.0->open_clip_torch)
(2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch>=1.9.0->open_clip_torch)
(12.4.127)
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch>=1.9.0->open_clip_torch)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-
packages (from torch>=1.9.0->open_clip_torch) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-
packages (from torch>=1.9.0->open_clip_torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.11/dist-packages (from
sympy==1.13.1->torch>=1.9.0->open_clip_torch) (1.3.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.11/dist-
packages (from ftfy->open_clip_torch) (0.2.13)
Requirement already satisfied: packaging>=20.9 in
/usr/local/lib/python3.11/dist-packages (from huggingface-hub->open_clip_torch)
(24.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-
packages (from huggingface-hub->open_clip_torch) (6.0.2)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-
packages (from huggingface-hub->open_clip_torch) (2.32.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
```

```
(from torchvision->open_clip_torch) (2.0.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.11/dist-packages (from torchvision->open_clip_torch)
(11.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from
jinja2->torch>=1.9.0->open_clip_torch) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests->huggingface-
hub->open_clip_torch) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
packages (from requests->huggingface-hub->open_clip_torch) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests->huggingface-
hub->open_clip_torch) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests->huggingface-
hub->open_clip_torch) (2025.1.31)
Downloading open_clip_torch-2.32.0-py3-none-any.whl (1.5 MB)
                          1.5/1.5 MB
26.9 MB/s eta 0:00:00
Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-manylinux2014_x86_64.whl
(363.4 MB)
                          363.4/363.4 MB
3.9 MB/s eta 0:00:00
Downloading nvidia_cuda_cupti_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (13.8 MB)
                          13.8/13.8 MB
39.4 MB/s eta 0:00:00
Downloading nvidia_cuda_nvrtc_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (24.6 MB)
                          24.6/24.6 MB
37.2 MB/s eta 0:00:00
Downloading nvidia_cuda_runtime_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (883 kB)
                          883.7/883.7 kB
27.4 MB/s eta 0:00:00
Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-manylinux2014_x86_64.whl
(664.8 MB)
                          664.8/664.8 MB
2.8 MB/s eta 0:00:00
Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-manylinux2014_x86_64.whl
(211.5 MB)
                          211.5/211.5 MB
6.7 MB/s eta 0:00:00
Downloading nvidia_curand_cu12-10.3.5.147-py3-none-
manylinux2014_x86_64.whl (56.3 MB)
                          56.3/56.3 MB
```

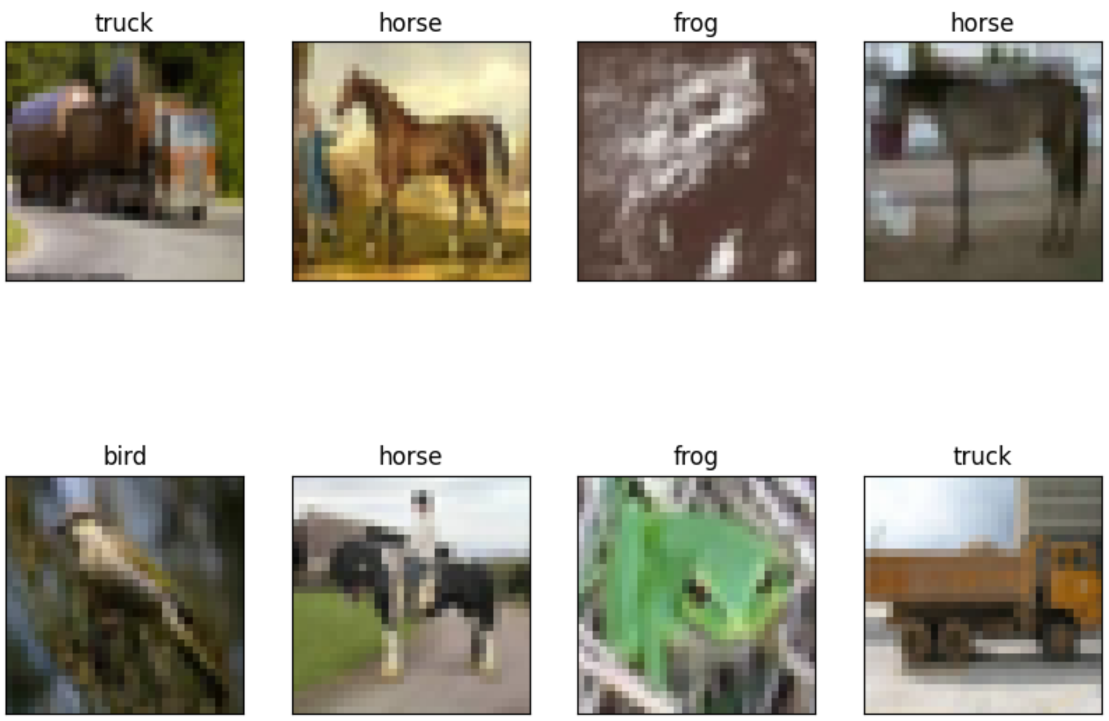
```
13.3 MB/s eta 0:00:00
Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-
manylinux2014_x86_64.whl (127.9 MB)
                          127.9/127.9 MB
7.5 MB/s eta 0:00:00
Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-
manylinux2014_x86_64.whl (207.5 MB)
                          207.5/207.5 MB
5.5 MB/s eta 0:00:00
Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl (21.1 MB)
                          21.1/21.1 MB
43.6 MB/s eta 0:00:00
Downloading ftfy-6.3.1-py3-none-any.whl (44 kB)
                          44.8/44.8 kB
4.4 MB/s eta 0:00:00
Installing collected packages: nvidia-nvjitlink-cu12, nvidia-curand-cu12,
nvidia-cufft-cu12, nvidia-cuda-runtime-cu12, nvidia-cuda-nvrtc-cu12, nvidia-
cuda-cupti-cu12, nvidia-cublas-cu12, ftfy, nvidia-cusparse-cu12, nvidia-cudnn-
cu12, nvidia-cusolver-cu12, open_clip_torch
  Attempting uninstall: nvidia-nvjitlink-cu12
    Found existing installation: nvidia-nvjitlink-cu12 12.5.82
    Uninstalling nvidia-nvjitlink-cu12-12.5.82:
      Successfully uninstalled nvidia-nvjitlink-cu12-12.5.82
  Attempting uninstall: nvidia-curand-cu12
    Found existing installation: nvidia-curand-cu12 10.3.6.82
    Uninstalling nvidia-curand-cu12-10.3.6.82:
      Successfully uninstalled nvidia-curand-cu12-10.3.6.82
  Attempting uninstall: nvidia-cufft-cu12
    Found existing installation: nvidia-cufft-cu12 11.2.3.61
    Uninstalling nvidia-cufft-cu12-11.2.3.61:
      Successfully uninstalled nvidia-cufft-cu12-11.2.3.61
  Attempting uninstall: nvidia-cuda-runtime-cu12
    Found existing installation: nvidia-cuda-runtime-cu12 12.5.82
    Uninstalling nvidia-cuda-runtime-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-runtime-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-nvrtc-cu12
    Found existing installation: nvidia-cuda-nvrtc-cu12 12.5.82
    Uninstalling nvidia-cuda-nvrtc-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-nvrtc-cu12-12.5.82
  Attempting uninstall: nvidia-cuda-cupti-cu12
    Found existing installation: nvidia-cuda-cupti-cu12 12.5.82
    Uninstalling nvidia-cuda-cupti-cu12-12.5.82:
      Successfully uninstalled nvidia-cuda-cupti-cu12-12.5.82
  Attempting uninstall: nvidia-cublas-cu12
    Found existing installation: nvidia-cublas-cu12 12.5.3.2
    Uninstalling nvidia-cublas-cu12-12.5.3.2:
      Successfully uninstalled nvidia-cublas-cu12-12.5.3.2
  Attempting uninstall: nvidia-cusparse-cu12
    Found existing installation: nvidia-cusparse-cu12 12.5.1.3
    Uninstalling nvidia-cusparse-cu12-12.5.1.3:
      Successfully uninstalled nvidia-cusparse-cu12-12.5.1.3
  Attempting uninstall: nvidia-cudnn-cu12
    Found existing installation: nvidia-cudnn-cu12 9.3.0.75
    Uninstalling nvidia-cudnn-cu12-9.3.0.75:
      Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
  Attempting uninstall: nvidia-cusolver-cu12
    Found existing installation: nvidia-cusolver-cu12 11.6.3.83
    Uninstalling nvidia-cusolver-cu12-11.6.3.83:
      Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
Successfully installed ftfy-6.3.1 nvidia-cublas-cu12-12.4.5.8 nvidia-cuda-cupti-
cu12-12.4.127 nvidia-cuda-nvrtc-cu12-12.4.127 nvidia-cuda-runtime-cu12-12.4.127
nvidia-cudnn-cu12-9.1.0.70 nvidia-cufft-cu12-11.2.1.3 nvidia-curand-
cu12-10.3.5.147 nvidia-cusolver-cu12-11.6.1.9 nvidia-cusparse-cu12-12.3.1.170
nvidia-nvjitlink-cu12-12.4.127 open_clip_torch-2.32.0
```

[ ]:

```python
import torch
import torchvision
import torchvision.transforms as transforms
import transformers
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import requests
import timm
import math

batch_size = 100

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

transform = transforms.Compose([
    transforms.Resize(32),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)
```

```
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

def unnormalize(img, mean, std):
    return img.mul_(std.reshape(-1, 1, 1)).add_(mean.reshape(-1, 1, 1))

# utility function to plot gallery of images
def plot_gallery(images, titles, height, width, n_row=2, n_col=4):
    plt.figure(figsize=(2* n_col, 3 * n_row))
    plt.subplots_adjust(bottom=0, left=0.01, right=0.99, top=0.90, hspace=0.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        curr_img = images[i]
        curr_img = unnormalize(curr_img, mean=torch.Tensor([0.485, 0.456, 0.
 ↪406]), std=torch.Tensor([0.229, 0.224, 0.225]))
        plt.imshow(np.transpose(curr_img, (1, 2, 0)))
        plt.title(classes[titles[i]], size=12)
        plt.xticks(())
        plt.yticks(())


# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# visualize some of the images of the CIFAR10 dataset
plot_gallery(images[:8], labels[:8], 32, 32, 2, 4)
```

```
100%|      | 170M/170M [00:12<00:00, 13.6MB/s]
```



1.0.3 *(a) Training framework for CIFAR10* (10pt)

- For the more challenging CIFAR10 dataset, it is more useful to track the training and testing set accuracies during training.
- However, unlike with MNIST and the simple linear models, it is not always possible to load the entire dataset and pass the full batch through the model to compute the accuracy.
- Hence, we need to compute the accuracy in a mini-batch manner in the function `get_accuracy_cifar10`.
  - It takes as input the `model`, a `dataloader` (either for training or test set), and the `batch_size`.
- Following this, implement the training function `train_torch_model_cifar10` (similar to MNIST) but compute and print the training and testing accuracy after every epoch.
  - It should take as input the `model`, `trainloader`, `testloader`, `batch_size`, a predefined `optimizer`, the loss function `criterion`, maximum number of epochs `max_epochs`, and `tolerance`.
- You should be able to use the `train_torch_model` code from Part I by adding support for dataloaders.

[ ]:
```python
def get_accuracy_cifar10(model, dataloader, batch_size):
    correct = 0
    total = 0

    with torch.no_grad(): # Close the Gradient Calculation
      for X_batch, y_batch in dataloader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device) # Ensure the␣
↪tensors to be on the same device
        output = model(X_batch)
        values, predicted_indices = torch.max(output, 1)
        correct += (predicted_indices == y_batch).sum().item()
        total += y_batch.size(0)

    acc = correct / total
    return acc

# Define a function train_torch_model_cifar10
def train_torch_model_cifar10(model, trainloader, testloader, batch_size,␣
↪optimizer, criterion, max_epochs, tolerance, device='cuda'):
  losses = []
  train_acc = []
  test_acc = []
  prev_loss = float('inf')

  #######
  # 3. move model to device
  model.to(device)

  for epoch in tqdm(range(max_epochs)):
    for idx, (X_train_batch, y_train_batch) in enumerate(trainloader):
      # 3. move mini-batch data to device
      X_train_batch, y_train_batch = X_train_batch.to(device), y_train_batch.
↪to(device)

      # 4. reset gradients
      optimizer.zero_grad()

      # 5. prediction on mini-batch data
      pred = model(X_train_batch)

      # 6. calculate loss
      loss = criterion(pred, y_train_batch)

      # 7. backpropagate loss
      loss.backward()

      # 8. perform a single gradient update step
      optimizer.step()

    train_acc_curr = get_accuracy_cifar10(model, trainloader, batch_size)
    test_acc_curr = get_accuracy_cifar10(model, testloader, batch_size)
    #######
    # log loss every 100th epoch and print every 5000th epoch:
```

```
    losses.append((epoch, loss.item()))
    print('Epoch: {}, Loss: {}'.format(epoch, loss.item()))

    print("Train accuracy:", train_acc_curr)
    print("Test accuracy:", test_acc_curr)
    train_acc.append((epoch, train_acc_curr))
    test_acc.append((epoch, test_acc_curr))

    # break if decrease in loss is less than threshold
    if abs(prev_loss - loss.item()) < tolerance:
      print("Break! Early stopping triggere since the loss is less than␣
↪threshold.")
      break
    prev_loss = loss.item()

  # return updated model and logged losses
  return model, losses, train_acc, test_acc
```

1.0.4 *(b) ResNet* (10pt)

#1. Training with random initialization

- Using the defined `train_torch_model_cifar10`, we will first train a model using the ResNet50 architecture from scratch.
- Hyperparameters:
  - Batch size of 100
  - SGD optimizer with learning rate 0.001 and momentum 0.9
  - CE Loss
  - Train the model for a maximum of 15 epochs.
- Use the `torchvision.models.resnet50` function to define the model and set the arguments such that pretrained weights are not used.
- The default `resnet50` model has 1000 output neurons (since ImageNet1k has 1000 classes) in the last layer while we need only 10 for CIFAR10.

We will also compute the training and testing set accuracies after every epoch and plot them.

[ ]:
```
import torchvision.models as models
from torchvision.models import resnet50

model = resnet50(pretrained=False)

model.fc = torch.nn.Linear(model.fc.in_features, 10) # 1000 output neurons ->␣
↪10 for CIFAR10

batch_size = 100
# define optimizer (use torch.optim.SGD (Stochastic Gradient Descent))
# Set learning rate to lr and also set model parameters
criterion = torch.nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

max_epochs = 15
tolerance = 1e-4  # Early Stoping Tolerance

model, losses, train_accs, test_accs = train_torch_model_cifar10(
    model=model,
    trainloader=trainloader,
    testloader=testloader,
    batch_size=batch_size,
    optimizer=optimizer,
    criterion=criterion,
    max_epochs=max_epochs,
    tolerance=tolerance,
    device=device
)
```

```
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=None`.
  warnings.warn(msg)
```

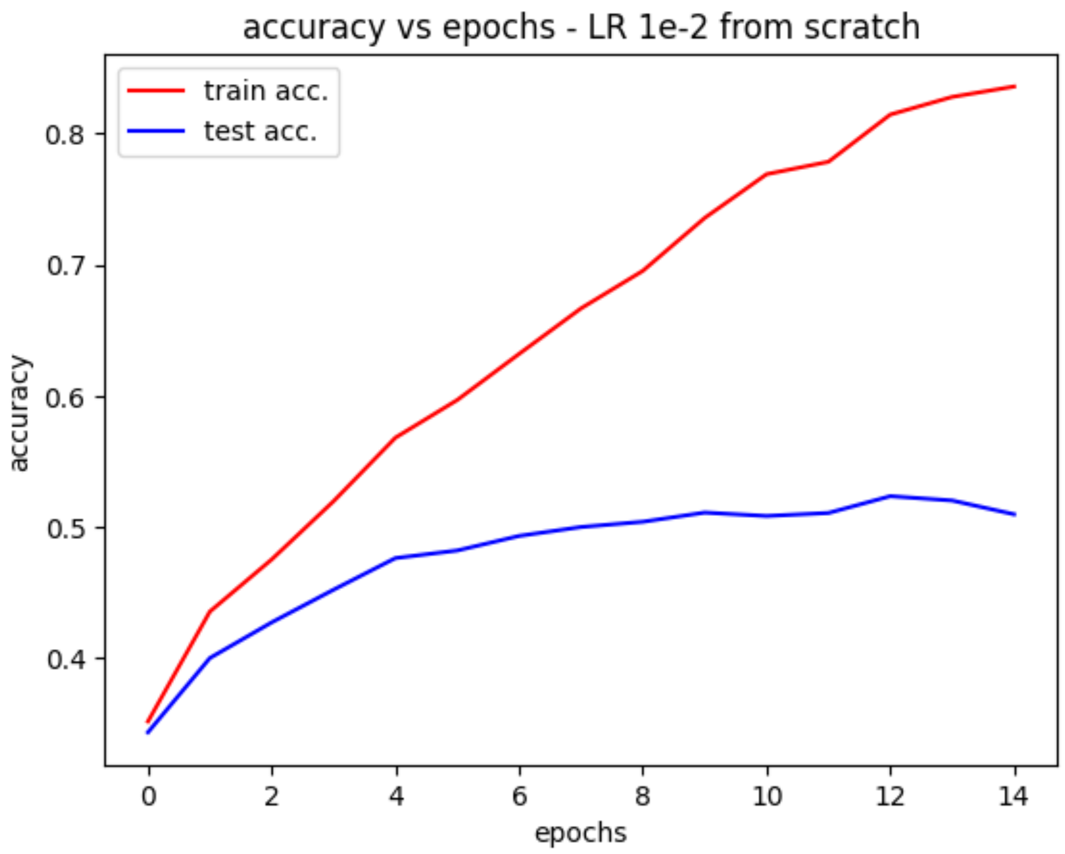
```
  0%|          | 0/15 [00:00<?, ?it/s]
```

```
Epoch: 0, Loss: 2.013981819152832
Train accuracy: 0.35178
Test accuracy: 0.3435
Epoch: 1, Loss: 2.0110676288604736
Train accuracy: 0.43552
Test accuracy: 0.4001
Epoch: 2, Loss: 1.6131242513656616
Train accuracy: 0.47528
Test accuracy: 0.4273
Epoch: 3, Loss: 1.469616413116455
Train accuracy: 0.51966
Test accuracy: 0.4522
Epoch: 4, Loss: 1.641402244567871
Train accuracy: 0.5681
Test accuracy: 0.4763
Epoch: 5, Loss: 1.2013987302780151
Train accuracy: 0.59684
Test accuracy: 0.4821
Epoch: 6, Loss: 1.5383262634277344
Train accuracy: 0.632
```

```
Test accuracy: 0.4932
Epoch: 7, Loss: 0.9974661469459534
Train accuracy: 0.66666
Test accuracy: 0.5
Epoch: 8, Loss: 1.1959586143493652
Train accuracy: 0.6953
Test accuracy: 0.504
Epoch: 9, Loss: 1.1786892414093018
Train accuracy: 0.73562
Test accuracy: 0.511
Epoch: 10, Loss: 0.7713893055915833
Train accuracy: 0.76896
Test accuracy: 0.5084
Epoch: 11, Loss: 0.9026442766189575
Train accuracy: 0.77842
Test accuracy: 0.5107
Epoch: 12, Loss: 0.8003696203231812
Train accuracy: 0.81442
Test accuracy: 0.5235
Epoch: 13, Loss: 0.5874871611595154
Train accuracy: 0.82786
Test accuracy: 0.5202
Epoch: 14, Loss: 0.7248199582099915
Train accuracy: 0.8358
Test accuracy: 0.5097
```

[ ]:
```
plt.plot([x[0] for x in train_accs],[x[1] for x in train_accs], 'r',␣
  ↪label='train acc.')
plt.plot([x[0] for x in test_accs],[x[1] for x in test_accs], 'b', label='test␣
  ↪acc.')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.title('accuracy vs epochs - LR 1e-2 from scratch')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7a3cadf0f410>
```



#2. Training with pretrained initialization

- Using the defined `train_torch_model_cifar10`, we will now train a ResNet50 model initialized from an ImageNet pretrained model.
- We will use the same hyperparameters as the previous part #2.1.
  - Batch size of 100
  - SGD optimizer with learning rate 0.001 and momentum 0.9
  - CE Loss
  - Train the model for a maximum of 15 epochs.
- We will use the same `torchvision.models.resnet50` function to define the model, but set the arguments such that ImageNet1k-v2 pretrained weights are used for initialization.

We will also compute and plot the training and testing set accuracies after every epoch.

[ ]:
```
from torchvision.models import resnet50, ResNet50_Weights

model = resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)

model.fc = torch.nn.Linear(model.fc.in_features, 10)
```

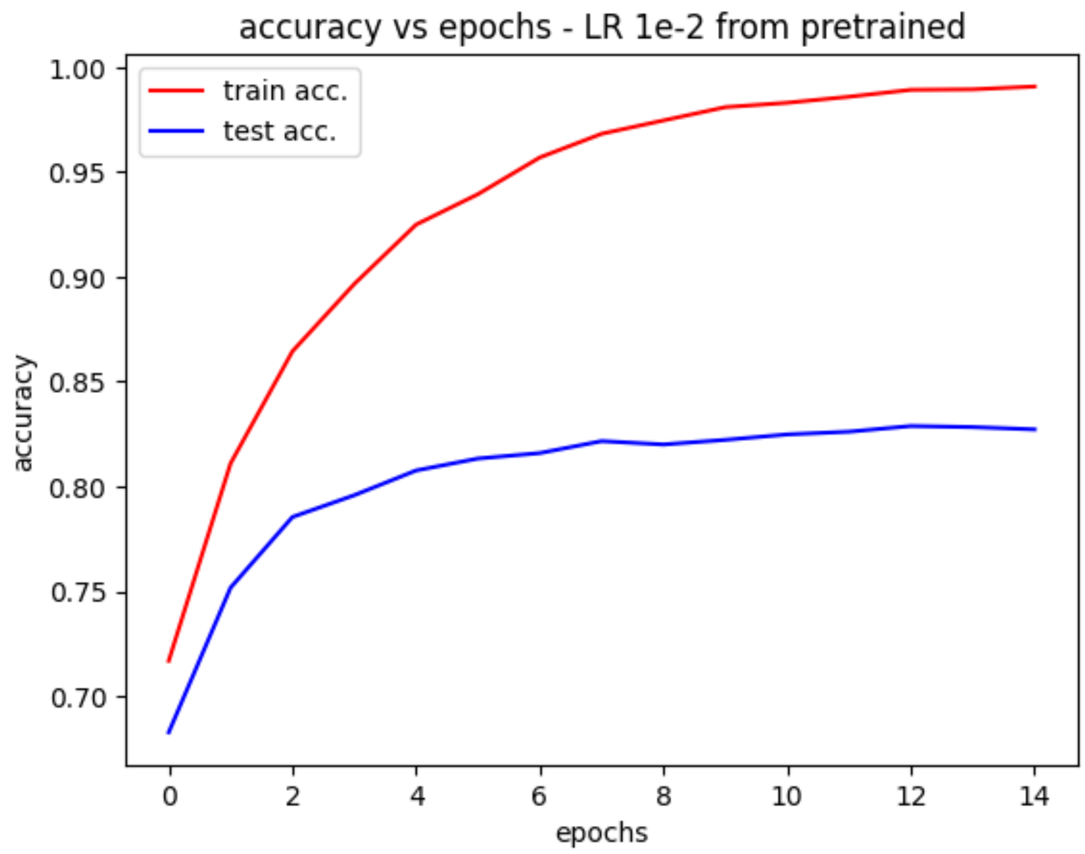
```python
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

max_epochs = 15
tolerance = 1e-4  # Early Stoping Tolerance

batch_size = 100

model, losses, train_accs, test_accs = train_torch_model_cifar10(
    model=model,
    trainloader=trainloader,
    testloader=testloader,
    batch_size=batch_size,
    optimizer=optimizer,
    criterion=criterion,
    max_epochs=max_epochs,
    tolerance=tolerance,
    device=device
)
```

```
  0%|          | 0/15 [00:00<?, ?it/s]

Epoch: 0, Loss: 0.8774340748786926
Train accuracy: 0.71682
Test accuracy: 0.6827
Epoch: 1, Loss: 0.6398633718490601
Train accuracy: 0.811
Test accuracy: 0.7517
Epoch: 2, Loss: 0.5418192744255066
Train accuracy: 0.86444
Test accuracy: 0.7854
Epoch: 3, Loss: 0.41011062264442444
Train accuracy: 0.89656
Test accuracy: 0.7958
Epoch: 4, Loss: 0.2765777111053467
Train accuracy: 0.9249
Test accuracy: 0.8076
Epoch: 5, Loss: 0.1497117131948471
Train accuracy: 0.9394
Test accuracy: 0.8133
Epoch: 6, Loss: 0.20882584154605865
Train accuracy: 0.95692
Test accuracy: 0.8159
Epoch: 7, Loss: 0.0915210172533989
Train accuracy: 0.96822
Test accuracy: 0.8216
Epoch: 8, Loss: 0.19692260026931763
Train accuracy: 0.97458
```

```
Test accuracy: 0.82
Epoch: 9, Loss: 0.1939532458782196
Train accuracy: 0.98092
Test accuracy: 0.8222
Epoch: 10, Loss: 0.05898956209421158
Train accuracy: 0.983
Test accuracy: 0.8248
Epoch: 11, Loss: 0.03971254453063011
Train accuracy: 0.98588
Test accuracy: 0.8261
Epoch: 12, Loss: 0.038184016942977905
Train accuracy: 0.98918
Test accuracy: 0.8288
Epoch: 13, Loss: 0.02974276803433895
Train accuracy: 0.98942
Test accuracy: 0.8283
Epoch: 14, Loss: 0.016494208946824074
Train accuracy: 0.99076
Test accuracy: 0.8272
```

[ ]:
```
plt.plot([x[0] for x in train_accs],[x[1] for x in train_accs], 'r',␣
  ↪label='train acc.')
plt.plot([x[0] for x in test_accs],[x[1] for x in test_accs], 'b', label='test␣
  ↪acc.')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.title('accuracy vs epochs - LR 1e-2 from pretrained')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7a3ca382fa10>
```



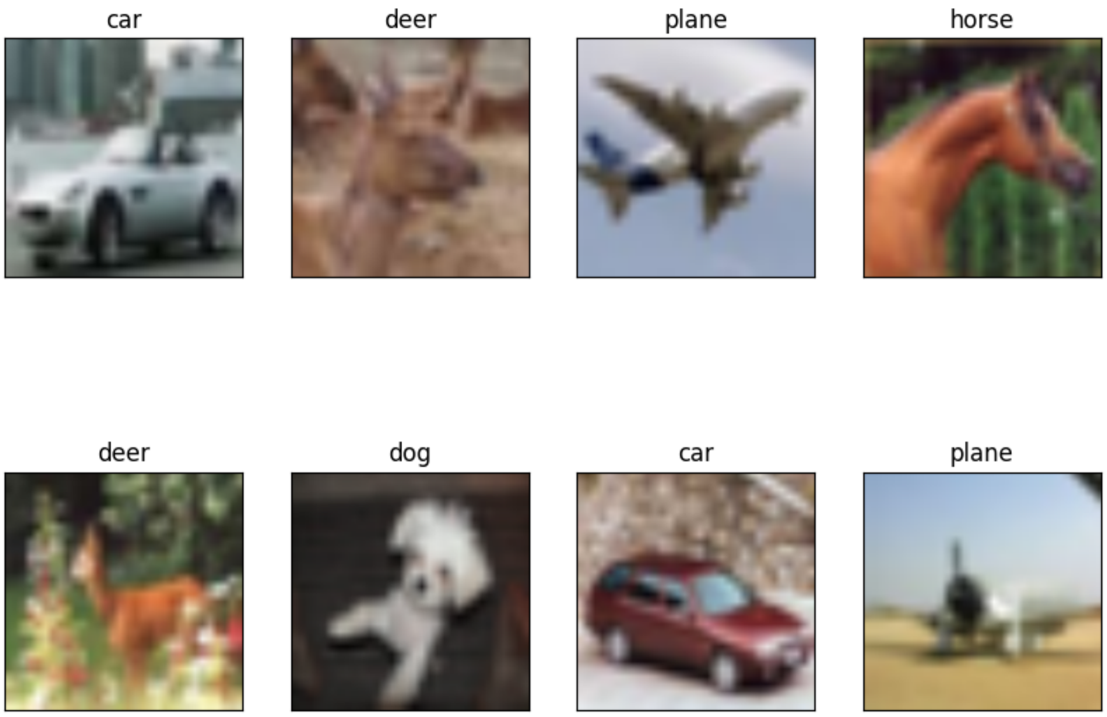
1.0.5 *(c) Vision Transformer* (https://arxiv.org/abs/2010.11929) (10pt)

- This architecture was proposed following the popularization of transformer models for language modeling.
- Vision Transformers can be considered a variation of transformers with image inputs.
- The images are divided as 16x16 patches (the size varies with different models) and a sequence of these patches is passed as the input to the transformer model.

First, we modify the shape of the images to be 224x224 as this is what the ViT model we are using expects.

[ ]:
```
vit_transform = transforms.Compose([
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

vit_trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=vit_transform)
```

```
vit_trainloader = torch.utils.data.DataLoader(vit_trainset,␣
↪batch_size=batch_size,
                                          shuffle=True, num_workers=2)

vit_testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=vit_transform)
vit_testloader = torch.utils.data.DataLoader(vit_testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

dataiter = iter(vit_trainloader)
vit_images, vit_labels = next(dataiter)

# visualize some of the images of the CIFAR10 dataset
plot_gallery(vit_images[:8], vit_labels[:8], 32, 32, 2, 4)
```



#1. Training with random initialization

- Similarly, now we will use the `train_torch_model_cifar10` function to train a Classifier using the Vision Transformer Architecture
- Hyperparameters:
  - Batch size of 100
  - SGD optimizer with learning rate 0.001 and momentum 0.9
  - CE Loss

    - Train the model for a maximum of 7 epochs as training this model is slower than ResNet50.
- Use the `timm/efficientvit_m1.r224_in1k` model either using the `timm` library. We choose this model due to its small number of parameters (3M) which make it easy to train and experiment with.
- By setting the `num_classes` argument, we can customise the number of required classes which is 10, in the case of CIFAR10.

We will also compute the training and testing set accuracies after every epoch and plot them.

Note: This function takes approx 15-30 minutes to run. Ensure it works on a non-GPU runtime first to ensure you don't exhaust resources.

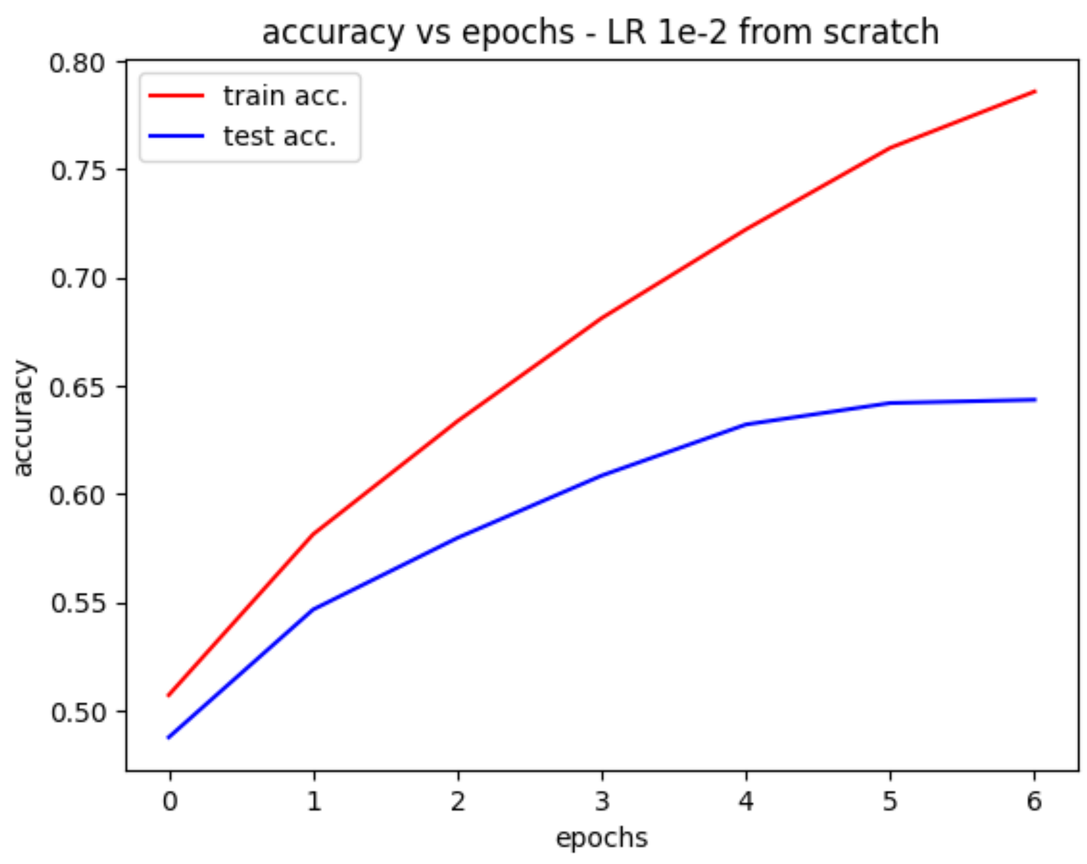
```
[ ]: import torch.nn as nn
import timm

model = timm.create_model('efficientvit_m1.r224_in1k', pretrained = False,␣
 ↪num_classes = 10)

# define optimizer (use torch.optim.SGD (Stochastic Gradient Descent))
# Set learning rate to lr and also set model parameters
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

max_epochs = 7
tolerance = 1e-4  # Early Stoping Tolerance

batch_size = 100

model, losses, train_accs, test_accs = train_torch_model_cifar10(
    model=model,
    trainloader=vit_trainloader,
    testloader=vit_testloader,
    batch_size=batch_size,
    optimizer=optimizer,
    criterion=criterion,
    max_epochs=max_epochs,
    tolerance=tolerance,
    device=device
)
```

```
  0%|          | 0/7 [00:00<?, ?it/s]

Epoch: 0, Loss: 1.6083298921585083
Train accuracy: 0.507
Test accuracy: 0.4875
Epoch: 1, Loss: 1.2575690746307373
Train accuracy: 0.58124
Test accuracy: 0.5465
```

```
Epoch: 2, Loss: 1.1638212203979492
Train accuracy: 0.63348
Test accuracy: 0.5796
Epoch: 3, Loss: 0.8798545002937317
Train accuracy: 0.68112
Test accuracy: 0.6084
Epoch: 4, Loss: 0.8386316895484924
Train accuracy: 0.72212
Test accuracy: 0.632
Epoch: 5, Loss: 0.5688455700874329
Train accuracy: 0.75978
Test accuracy: 0.6419
Epoch: 6, Loss: 0.6319459676742554
Train accuracy: 0.7858
Test accuracy: 0.6434
```

[ ]:
```
plt.plot([x[0] for x in train_accs],[x[1] for x in train_accs], 'r',␣
  ↪label='train acc.')
plt.plot([x[0] for x in test_accs],[x[1] for x in test_accs], 'b', label='test␣
  ↪acc.')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.title('accuracy vs epochs - LR 1e-2 from scratch')
plt.legend()
```

[ ]:
```
<matplotlib.legend.Legend at 0x7840247ce4d0>
```



#2. Training with pretrained initialization

- We will use the same `timm/efficientvit_m1.r224_in1k` model but we will use its pretrained weights on ImageNet using the `pretrained` parameter.
- We will use the same hyperparameters as the previous part #3.1.
  - Batch size of 100
  - SGD optimizer with learning rate 0.001 and momentum 0.9
  - CE Loss
  - Train the model for a maximum of 7 epochs.

We will also compute and plot the training and testing set accuracies after every epoch.

Note: This function takes approx 15-30 minutes to run. Ensure it works on a non-GPU runtime first to ensure you don't exhaust resources.

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

model = timm.create_model('efficientvit_m1.r224_in1k', pretrained = True,␣
 ↪num_classes = 10)
```

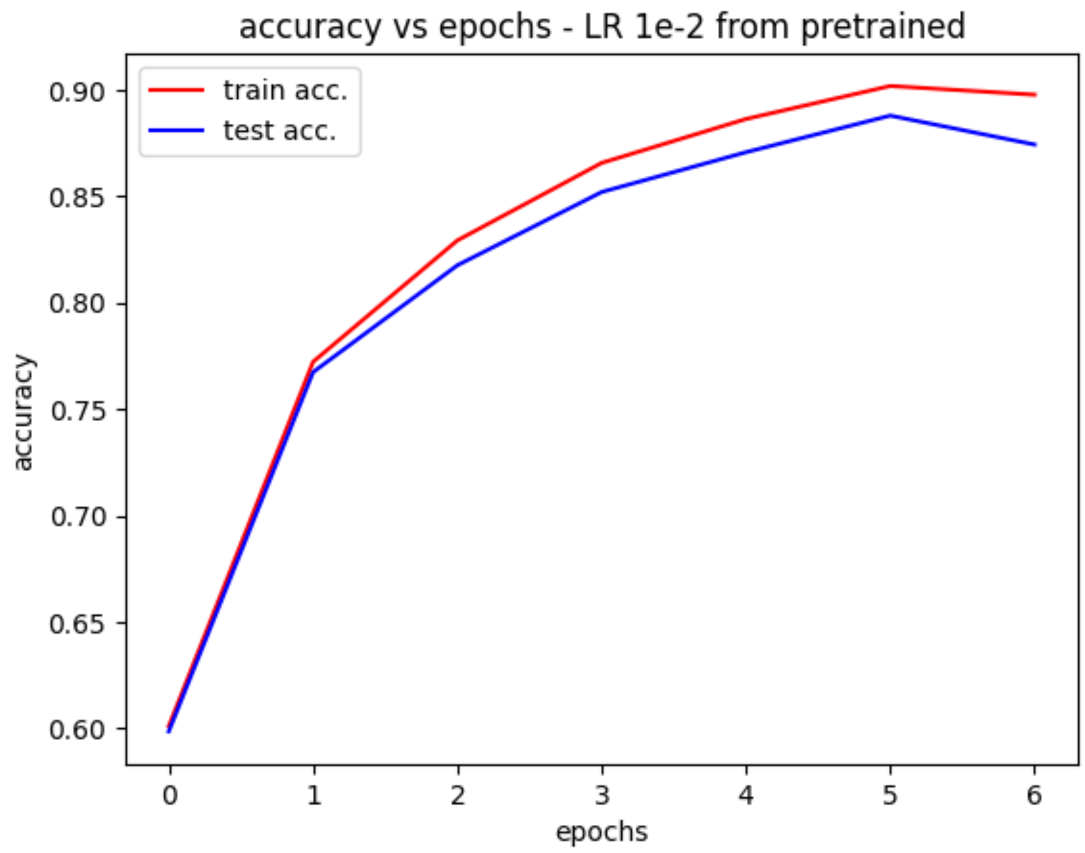
```
# define optimizer (use torch.optim.SGD (Stochastic Gradient Descent))
# Set learning rate to lr and also set model parameters
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

max_epochs = 7
tolerance = 1e-4  # Early Stoping Tolerance

batch_size = 100

model, losses, train_accs, test_accs = train_torch_model_cifar10(
    model=model,
    trainloader=vit_trainloader,
    testloader=vit_testloader,
    batch_size=batch_size,
    optimizer=optimizer,
    criterion=criterion,
    max_epochs=max_epochs,
    tolerance=tolerance,
    device=device
)
```

```
  0%|          | 0/7 [00:00<?, ?it/s]

Epoch: 0, Loss: 1.2248324155807495
Train accuracy: 0.601
Test accuracy: 0.5986
Epoch: 1, Loss: 0.7399659752845764
Train accuracy: 0.77216
Test accuracy: 0.7673
Epoch: 2, Loss: 0.5494275093078613
Train accuracy: 0.82918
Test accuracy: 0.8175
Epoch: 3, Loss: 0.42958521842956543
Train accuracy: 0.86558
Test accuracy: 0.8519
Epoch: 4, Loss: 0.40067949891090393
Train accuracy: 0.8862
Test accuracy: 0.8706
Epoch: 5, Loss: 0.27602532505989075
Train accuracy: 0.90172
Test accuracy: 0.8878
Epoch: 6, Loss: 0.3236459493637085
Train accuracy: 0.89762
Test accuracy: 0.8742
```

[ ]:
```
plt.plot([x[0] for x in train_accs],[x[1] for x in train_accs], 'r',␣
  ↪label='train acc.')
```

```
plt.plot([x[0] for x in test_accs],[x[1] for x in test_accs], 'b', label='test␣
  ↪acc.')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.title('accuracy vs epochs - LR 1e-2 from pretrained')
plt.legend()
```

```
[ ]: <matplotlib.legend.Legend at 0x78402bc34550>
```



1.0.6 *(d) CLIP (Contrastive Language-Image Pretraining - Paper)* (10pt)

- CLIP jointly trains a text-encoder and image-encoder on images and text captions.
- CLIP generates representations (vector embeddings) of both the text and image such that they lie in the same vector space.
- CLIP optimizes the embeddings such that the image and captions are closer in the vector space if they correspond to each other and farther if they do not.
- The image-encoder is a pluggable model - usually a ResNet or Vision Transformer. We will use both.
- For this section, we will not be doing any pretraining or finetuning, rather just loading and evaluating it on CIFAR10 test-set (zero-shot image classification).

Note: This is a simplified explanation. You can read the paper if you need more details. However, this is not essential for the assignment.

[ ]:
```
import open_clip

captions = [f"a photo of a {label}" for label in classes]

clip_testset = torchvision.datasets.CIFAR10(root='./data', download=True,␣
 ↪train=False)
```

Note: We will use the `open_clip` package to run CLIP. You can refer the documentation at https://github.com/mlfoundations/open_clip

We define a new testset for CLIP since it uses PIL images as inputs and does not require tensor conversion

Note: We convert the class labels to a caption format which works better with CLIP.

#1. Predict and accuracy function

- Write the `clip_predict` function that takes in the CLIP `model`, a list of processed `images`, tokenized `texts` and encodes the texts and images.
- The function should then use the encoded features to find and return the predicted label for each of the `images`.
- Complete the `clip_accuracy` function takes in the CLIP `model`, the CLIP image `processor` and text/caption `tokenizer` and torchvision image `dataset` as input
- The function processes the images and texts into vectors in batch-form and uses the `clip_predict` predictions to calculate accuracy over the `dataset`

[ ]:
```
def clip_predict(model, images, texts):

  with torch.no_grad():
      # get the image and text vectors using the model's encode functions
      image_features = model.encode_image(images)
      text_features = model.encode_text(texts)

      # normalize the image and text vectors
      image_features_n = image_features / image_features.norm(dim=-1,␣
 ↪keepdim=True) # Normalized into a unit vector by dividing a L2 Norm
      text_features_n = text_features / text_features.norm(dim=-1, keepdim=True)

      # find the cosine similarity between the image and text vectors and␣
 ↪softmax them to get the probabilities
      logits = image_features @ text_features.T # Original Score: shape␣
 ↪(batch_size, num_classes)
      probs = nn.functional.softmax(logits, dim=-1)

      # calculate predicted label index
      preds = torch.argmax(probs, dim=-1)

  #return predicted labels
  return preds

def clip_accuracy(model, processor, tokenizer, dataset, batch_size,␣
 ↪captions=captions):

  # process the text using the tokenizer
  texts = tokenizer(captions).to(device) # caption -> tensor

  #calculate the number of batches
  number_of_batches = len(dataset) // batch_size

  correct = 0

  for i in range(number_of_batches):

    # define start and end indices
    start = i * batch_size
    end = start + batch_size

    # process the images in the batch using the processor
    original_images = [dataset[j][0] for j in range(start, end)]
    images = torch.stack([processor(img) for img in original_images]).
 ↪to(device) # Process images in the batch into tensors

    # get the labels
    labels = torch.tensor([dataset[j][1] for j in range(start, end)]).
 ↪to(device) # .to(device): Ensure preds and labels are on the same device

    # use the clip_predict function to
    preds = clip_predict(model, images, texts)
    correct += (preds == labels).sum().item()

  return correct / len(dataset)
```

#2. CLIP Model using Vision Transformer

- Use the `open_clip.create_model_and_transforms` function from the `open_clip` library to load the model and image processor.
- Pass the `ViT-B-32` as the model and `openai` as the pretrained weights to use.
- Use the `open_clip.get_tokenizer` with the same model name to get the tokenizer.

[ ]:
```
model, _, processor = open_clip.create_model_and_transforms('ViT-B-32',␣
 ↪pretrained='openai')
tokenizer = open_clip.get_tokenizer('ViT-B-32')
```

- Call the `clip_accuracy` function with appropriate parameters. Optionally, you can pass in custom images and captions to the `clip_predict` function to see interesting outputs.

Note: This function takes approx 15-30 minutes to run. Ensure it works on a non-GPU runtime first to ensure you don't exhaust resources.

```
[ ]: model = model.to(device)

     acc = clip_accuracy(
         model=model,
         processor = processor,
         tokenizer = tokenizer,
         dataset = clip_testset,
         batch_size = 100)

     print(f"CLIP Zero-shot Accuracy on CIFAR-10 using Vision Transformer: {acc:.
      ↪4f}")
```

```
CLIP Zero-shot Accuracy on CIFAR-10 using Vision Transformer: 0.8281
```

Pass in custom images and see the outputs:

```
[ ]: from google.colab import files
     uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving myApple.jpg to myApple.jpg
```

```
[ ]: from PIL import Image

     # Upload an image of an apple
     img = Image.open("myApple.jpg").convert("RGB")
     plt.imshow(img)
     plt.axis("off")
     plt.title("Uploaded Image")
     plt.show()
```

[ ]:
```
# Define the prompt for the uploaded image
prompts = ["a photo of a banana", "a photo of a orange", "a photo of an peach",␣
↪"a photo of an apple"]

# Tokenize prompts and pass it to the device
text_inputs = tokenizer(prompts).to(device)

# Process the image into tensor
img_tensor = processor(img).unsqueeze(0).to(device)

# Do prediction
pred = clip_predict(model, img_tensor, text_inputs)

# Out the predicted class of the uploaded image
print("Predicted class:", prompts[pred.item()])
```

```
Predicted class: a photo of an apple
```

#3. CLIP Model using Resnet50

- We are now going to use a ResNet50 based CLIP-model instead of Vision Transformer a
- Use the same code as in #2 except the model name should be `RN50`

[ ]:
```
model, _, processor = open_clip.create_model_and_transforms("RN50",␣
 ↪pretrained="openai")
tokenizer = open_clip.get_tokenizer('RN50')
```

```
open_clip_model.safetensors:   0%|          | 0.00/408M [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.11/dist-packages/open_clip/factory.py:388: UserWarning:
These pretrained weights were trained with QuickGELU activation but the model
config does not have that enabled. Consider using a model config with a
"-quickgelu" suffix or enable with a flag.
  warnings.warn(
```

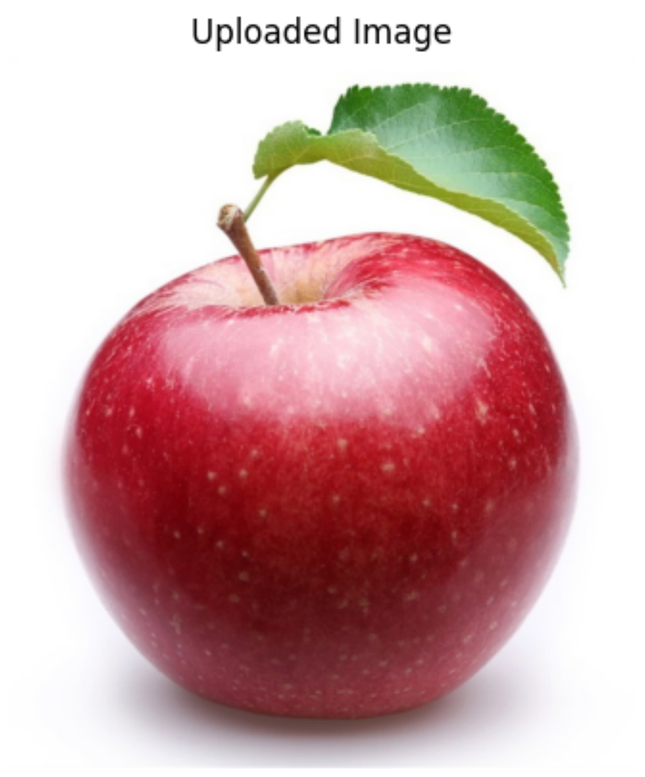
Note: This function takes approx 15-30 minutes to run. Ensure it works on a non-GPU runtime first to ensure you don't exhaust resources.

[ ]:
```
model = model.to(device)

acc = clip_accuracy(
    model=model,
    processor = processor,
    tokenizer = tokenizer,
    dataset = clip_testset,
    batch_size = 100)

print(f"CLIP Zero-shot Accuracy on CIFAR-10 using ResNet50: {acc:.4f}")
```

```
CLIP Zero-shot Accuracy on CIFAR-10 using ResNet50: 0.3522
```

- Call the `clip_predict` function with appropriate parameters. Optionally, you can pass in custom captions to see interesting outputs using CLIP.

[ ]:
```
from PIL import Image

# Upload an image of an apple
img = Image.open("myApple.jpg").convert("RGB")
plt.imshow(img)
plt.axis("off")
plt.title("Uploaded Image")
plt.show()
```



```
[ ]: # Define the prompt for the uploaded image
     prompts = ["a photo of a banana", "a photo of a orange", "a photo of an peach",␣
      ↪"a photo of an apple"]

     # Tokenize prompts and pass it to the device
     text_inputs = tokenizer(prompts).to(device)

     # Process the image into tensor
     img_tensor = processor(img).unsqueeze(0).to(device)

     # Do prediction
     pred = clip_predict(model, img_tensor, text_inputs)

     # Out the predicted class of the uploaded image
     print("Predicted class:", prompts[pred.item()])
```

```
Predicted class: a photo of a banana
```

1.0.7 *(e) Analysis* (5pt)

(Two Questions)

# Effect of pretraining

Effect: Pretraining significantly **improves both training and test performance.** It enables models to converge faster and achieve higher accuracy by starting from more generalizable feature representations learned from some large-scale datasets.

Experimental Evidence:

- After pretraining, all models show substantial accuracy gains within the first few epochs. (Based on the plot)
- CLIP (15 epochs):
  - Final Loss drops dramatically (from 0.72 → 0.02)
  - Train Accuracy: 84% → 99%
  - Test Accuracy: 51% → 83%
- Vision Transformer (7 epochs):
  - Final Loss drops from 0.63 → 0.32
  - Train Accuracy: 79% → 90%
  - Test Accuracy: 64% → 87%

Possible Reasons: Pretraining equips the model with prior knowledge about general visual patterns, which reduces the reliance on large labeled target datasets like CIFAR-10.

# **Effect of architecture** (ResNet v/s ViT v/s CLIP)

Effect: In general, **Vision Transformers (ViT) outperform ResNet** because of their global receptive field and attention mechanisms, and **CLIP-based models, particularly CLIP+ViT**, has the strongest performance due to both pretraining and architectural advantages.

Experimental Evidence:

- ViT vs CLIP:
  - CLIP+ViT shows strong performance on both classification and generalization tasks.
    - CLIP Zero-shot Accuracy on CIFAR-10 with ViT: 0.8281
    - Correctly identifies unseen inputs (e.g., correctly classifies an image of an apple).
  - CLIP+ResNet performs significantly worse in zero-shot settings:
    - CLIP Zero-shot Accuracy on CIFAR-10 with ResNet50: 0.3522
    - Misclassifies the apple image as a banana.

Possible Reasons:

- ViT uses transformer architecture, and the self-attention algorithms is useful on complex datasets. While ResNet is a convolutional model, whose convolutional layers only focus on local patterns in the image, so it may underperform on tasks like zero-shot classification that need deeper semantic understanding.
- CLIP+ViT combines the strength of pretrained multimodal learning with transformer architecture, allowing it to generalize better and better align visual features with language descriptions.

- CLIP+ResNet does benefit from pretraining, but is likely limited by the architectural limitations of convolutional features in aligning with language representations, thus resulting the low zero-shot accuracy.

PA_Part_III

April 24, 2025

1 *PA - Part III: Training a Robust Model - Optional/Bonus* (10pt)

Keywords: Adversarial Robustness Training

About the dataset:
The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.
The MNIST database contains 70,000 labeled images. Each datapoint is a 28×28 pixels grayscale image.

Agenda: * The PA is split into three parts, the first part dealing with miniature models which we will build from scratch, the second part dealing with modern architectures and the bonus third part dealing with training vision models robust to adversarial attacks. * In this part, you will train a 2-hidden layer neural network which is robust to adversarial attacks. * You will train models on adversarial examples generated using FGSM and PGD.

Note: * Hardware acceleration (GPU) is recommended but not required for this part. * A note on working with GPU: * Take care that whenever declaring new tensors, set `device=device` in parameters. * You can also move a declared torch tensor/model to device using `.to(device)`. * To move a torch model/tensor to cpu, use `.to('cpu')` * Keep in mind that all the tensors/model involved in a computation have to be on the same device (CPU/GPU). * Run all the cells in order. * Only **add your code** to cells marked with "TODO:" or with "…" * You should not have to change variable names where provided, but you are free to if required for your implementation.

1.0.1 *Setup: Imports and Utils*

[1]:
```
# install this library
!pip install gdown
```

```
Requirement already satisfied: gdown in /usr/local/lib/python3.11/dist-packages
(5.2.0)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.11/dist-
packages (from gdown) (4.13.4)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-
packages (from gdown) (3.18.0)
Requirement already satisfied: requests[socks] in
/usr/local/lib/python3.11/dist-packages (from gdown) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-packages
```

```
(from gdown) (4.67.1)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.11/dist-
packages (from beautifulsoup4->gdown) (2.7)
Requirement already satisfied: typing-extensions>=4.0.0 in
/usr/local/lib/python3.11/dist-packages (from beautifulsoup4->gdown) (4.13.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests[socks]->gdown) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
packages (from requests[socks]->gdown) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests[socks]->gdown) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests[socks]->gdown)
(2025.1.31)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in
/usr/local/lib/python3.11/dist-packages (from requests[socks]->gdown) (1.7.1)
```

[2]:
```
# imports
import torch
import torch.nn as nn
import numpy as np
import requests
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
import gdown
from zipfile import ZipFile

# set device
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# loading the dataset full MNIST dataset
mnist_train = datasets.MNIST("./data", train=True, download=True,␣
 ↪transform=transforms.ToTensor())
mnist_test = datasets.MNIST("./data", train=False, download=True,␣
 ↪transform=transforms.ToTensor())

mnist_train.data = mnist_train.data.to(device)
mnist_test.data = mnist_test.data.to(device)

mnist_train.targets = mnist_train.targets.to(device)
mnist_test.targets = mnist_test.targets.to(device)

# reshape and min-max scale
X_train =  (mnist_train.data.reshape((mnist_train.data.shape[0], -1))/255).
 ↪to(device)
```

```
y_train = mnist_train.targets
X_test = (mnist_test.data.reshape((mnist_test.data.shape[0], -1))/255).
  ↪to(device)
y_test = mnist_test.targets

# first few examples
example_data = mnist_test.data[:18]/255
example_data_flattened  = example_data.view((example_data.shape[0], -1)).
  ↪to(device) # needed for training
example_labels = mnist_test.targets[:18].to(device)
```

1.0.2 Adversarial training:

- To train robust models, the most intuitive strategy is to train on adversarial examples.
- The adversarial (robust) loss function is defined as: $\min_\theta \frac{1}{|S|} \sum_{x,y \in S} \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$, where θ are the learnable parameters, S is the set of training examples with x representing the input example and y the ground truth label, h_θ is the score function (neural network model), δ is the attack perturbation, and ϵ is the attack budget.
- This is also known as the min-max loss function. The gradient descent step now becomes: $\theta := \theta - \frac{\alpha}{|B|} \sum_{x,y \in B} \nabla_\theta \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$,
  where B is the mini-batch and α is the learning rate.
- Now the question becomes how to solve the inner term: $\nabla_\theta \max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$ of the gradient descent step.
- For this, we can use **Danskin's Theorem**, which states that to compute the (sub)gradient of a function containing a max term, we need to simply
  1. find the maximum and,
  2. compute the normal gradient evaluated at this point.
- This holds only when you have the exact maximum. Note that it is not possible to solve the inner maximization problem exactly (NP-hard). However, the better job we do of solving the inner maximization problem, the closer it seems that Danskin's theorem starts to hold. That is why we can re-use methods such as FGSM/PGD to find approximate worst case examples.
- In other words, we can perform the attack to find $\delta^* = \arg\max_{\|\delta\| \leq \epsilon} \ell(h_\theta(x+\delta), y)$, and then compute this term at the perturbed image: $\nabla_\theta \ell(h_\theta(x+\delta^*), y)$.

In summary, we will create an adversarial example for each datapoint in the mini-batch and use the loss corresponding to these adversarial examples to compute the gradient.

We explore two attacks to get the perturbation - Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD)

- In the Fast Gradient Sign Method (FGSM), the perturbation δ on an input example (e.g. input image) X is given by $\epsilon \cdot \text{sign}(g)$.
  - Here, g is the gradient of the loss function $g := \nabla_\delta \ell(h_\theta(x+\delta), y)$.
  - ℓ is the loss function, more precisely `nn.CrossEntropyLoss`. In the first timestep, this value of δ is 0.
- For the Projected Gradient Descent (PGD) attack, you will create an adversarial example by iteratively performing **steepest descent** with a fixed step size α.
  - The update rule is: $\delta := P(\delta + \alpha\,\text{sign}(\nabla_\delta \ell(h_\theta(x+\delta), y)))$.
  - Here δ is the perturbation, θ are the frozen DNN parameters, x and y is the training example and its ground truth label respectively.
  - h_θ is the score function and ℓ denotes the loss function.
  - P denotes the projection onto a norm ball (l_∞, l_1, l_2, etc.) of interest. For l_∞ ball, this just means clamping the value of δ between $-\epsilon$ and ϵ.

1.0.3 *(a) Setup* (4pt)

- In this part you will create a few adversarial examples using FGSM and PGD attacks. Use an attack budget $\epsilon = 0.05$.

#1. Define the `fgsm` function

- Define a function `fgsm` which takes as input the neural network model (`model`), test examples (`X`), target labels (`y`), and the attack budget (`epsilon`).
- Return the value of the perturbation (δ) after one gradient descent step.

[3]:
```
#######
#TODO:
def fgsm(model, x, y, epsilon):
    x.requires_grad = True
    output = model(x)
    model.zero_grad()
    loss = nn.CrossEntropyLoss()(output, y) # Most Precisely: Cross Entropy loss
    loss.backward()
    delta = epsilon * x.grad.sign() #   =  sign( )
    x_adv = x + delta
    x_adv = torch.clamp(x_adv, 0, 1) # Ensure valid pixel range after adding␣
 ↪perturbation: in MNIST: float between [0,1]
    delta = x_adv - x
    return delta
#######
```

#2. Define the `pgd` function

- Instead of using FGSM, now use Projected Gradient Descent (PGD) with projection on l_∞ ball for the attack.
- Define a function `pgd` that takes as input the neural network model (`model`), training examples (`X`), target labels (`y`), step size (`alpha`), attack budget (`epsilon`), and number of iterations

(num_iter).
- Return the perturbation (δ) after num_iter gradient descent steps.

[4]:
```
#TODO:

def pgd(model, x, y, alpha, epsilon, num_iter):
    # random sampling uniformly between 0 and 1
    delta = torch.rand_like(x) # Every element in tensor delta ~ U(0, 1)
    # bring delta in -eps to eps range
    # how? --> [0, 1] --> [0, 2*eps] --> [-eps, eps]
    delta = delta * 2 * epsilon - epsilon
    delta.requires_grad = True

    for t in (range(num_iter)):
        output = model(torch.clamp(x + delta, 0, 1)) # Ensure valid pixel range␣
 ↪after adding perturbation: in MNIST: float between [0,1]
        loss = nn.CrossEntropyLoss()(output, y)
        model.zero_grad()
        loss.backward()

        delta.data = delta.data + alpha * torch.sign(delta.grad.data) # Update␣
 ↪delta:  + sign(  ( ( + ), ))
        delta.data = torch.clamp(delta, -epsilon, epsilon) # Projection to [- ,␣
 ↪ ] l∞ ball:  := ( + sign(  (  ( + ), )))
        delta.data = torch.clamp(x + delta, 0, 1) # Ensure valid pixel range␣
 ↪after adding perturbation: in MNIST: float between [0,1]
        delta.grad.zero_()

    return delta
```

#3. Define a 2 hidden-layer NN in PyTorch

- Create a 2-hidden-layer neural network model in PyTorch.
- The input should be the size of the flattened MNIST image, and output layer should be of size 10, which is the number of target labels.
- Each of the two hidden layers should be of size 1024 with ReLU activations between each subsequent layer except the last layer.

You can refer to the structure of NN_Model from Part 1 (d) #5

[5]:
```
#######
from torch.nn.functional import relu
#TODO:
class NN_Model(nn.Module):
    def __init__(self):
      super(NN_Model, self).__init__()
      # Initalize various layers of model as instructed below
```

```
        # 1. initialize three linear layers: num_features -> 32, 32 -> 16, 16 ->␣
  ↪num_targets
        self.fc1 = torch.nn.Linear(784, 32)
        self.fc2 = torch.nn.Linear(32, 16)
        self.out = torch.nn.Linear(16, 10)

        # 2. initialize RELU
        self.relu = torch.nn.ReLU()

    def forward(self, X):
        # 3. define the feedforward algorithm of the model and return the final␣
  ↪output
        # Apply non-linear ReLU activation between subsequent layers
        x1 = self.relu(self.fc1(X)) # Hidden Layer 1 -> Hidden Layer 2: ReLU
        x2 = self.relu(self.fc2(x1)) # Hidden Layer 2 -> Output: ReLU
        output = self.out(x2) # Output Layer
        return output

#######
```

#4. Adversarial Training Framework

- Define a function `train_torch_model_adversarial`
  - which takes as input: a PyTorch model (`model`), batch size (`batch_size`), loss function (`criterion`), maximum number of epochs (`max_epochs`), training data (`X_train, y_train`), learning rate (`lr`), tolerance for stopping (`tolerance`), adversarial strategy (`adversarial_strategy: None/'fgsm'/'pgd'`), and attack budget(`epsilon`).
  - Note: use an SGD optimizer with the given learning rate (`lr`) to update all the model parameters.
- If `adversarial_strategy` is `None`, don't train on adversarial examples.
- This function will return a tuple of `(model, losses)`, where `model` is the trained model, and `losses` are a list of tuple of loss logged every epoch.
- The only difference from the function `train_torch_model` that you wrote in Part 1 (c) #2 is that you find the adversarial noise using an attack (based on `adversarial strategy`) and add it to the input before training with it.

[6]:
```
#######
#TODO:
def train_torch_model_adversarial(model, batch_size, criterion, max_epochs, \
                                  X_train, y_train, lr, tolerance, \
                                  adversarial_strategy, epsilon):
    losses = []
    num_batches = X_train.shape[0] // batch_size
    prev_loss = float('inf')
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)

    model.to(device)
```

```
  for epoch in tqdm(range(max_epochs)):
    for i in range(num_batches):
      start = i * batch_size
      end = start + batch_size
      X_train_batch = X_train[start:end]
      y_train_batch = y_train[start:end]

      # Generate Adversarial noise using an attack (based on fgsm/pgd)
      delta = 0

      if adversarial_strategy is not None:
        if adversarial_strategy == 'fgsm':
          delta = fgsm(model, X_train_batch, y_train_batch, epsilon)
        elif adversarial_strategy == 'pgd':
          alpha = 0.01
          num_iter = 40
          delta = pgd(model, X_train_batch, y_train_batch, alpha, epsilon,␣
↪num_iter)

      X_train_batch = X_train_batch + delta
      X_train_batch = torch.clamp(X_train_batch, 0, 1)

      # Reset gradients
      optimizer.zero_grad()

      # Output
      output = model(X_train_batch)

      # Calculate loss
      loss = criterion(output, y_train_batch)

      # Backpropagate loss
      loss.backward()

      # Perform a single gradient update step
      optimizer.step()

  #######

    # log loss every epoch and print every 5 epochs (including 20th epoch)
    # Should we append the average loss of the epoch? or just in the last␣
↪batch?
    losses.append((epoch, loss.item()))
    if epoch % 5 == 0 or epoch == max_epochs - 1:
        print('Epoch: {}, Loss: {}'.format(epoch, loss.item()))
      # break if decrease in loss is less than threshold
      if abs(prev_loss - loss.item()) < tolerance:
        print(f"Break! Early stopping at epoch {epoch}")
        break
      prev_loss = loss.item()

    return model, losses

#######
```

1.0.4 *(b) Training and evaluation with different strategies (3 points)*

#1. Standard, FGSM-based, and PGD-based training

- Train three models:
  - without adversarial training,
  - with adversarial training using `fgsm`
  - with adversarial training using `pgd`.
- Hyperparameters
  - Use attack budget `epsilon` of 0.05.
  - Use a batch-size 512
  - Train for 20 epochs with learning rate 10^{-2}, and early stopping tolerance of 10^{-6}.
  - For `pgd`, use a step-size `alpha=0.01` and number of iterations `num_iter=40` when training.

  Note: PGD implementation will be slow (1hr+) when using a non-GPU runtime.

[7]:
```
#######
#TODO: To put three trained model into a standard training process
trained_models = {}
batch_size = 512
tolerance = 1e-6
lr = 1e-2
max_epochs = 20
epsilon = 0.05

train_with_clean = False

model = NN_Model()
criterion = nn.CrossEntropyLoss()
model, losses = train_torch_model_adversarial(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train,
    y_train=y_train,
    lr=lr,
    tolerance=tolerance,
    adversarial_strategy=None,
    epsilon=epsilon
)
trained_models['None']=(model, losses)
print('Final training loss for None model:', losses[-1][1]) # Record the loss␣
 ↪after training the last epoch

model = NN_Model()
criterion = nn.CrossEntropyLoss()
model, losses = train_torch_model_adversarial(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train,
    y_train=y_train,
    lr=lr,
    tolerance=tolerance,
    adversarial_strategy='fgsm',
    epsilon=epsilon
)
trained_models['fgsm'] = (model, losses)
print('Final training loss for fgsm model:', losses[-1][1])

model = NN_Model()
criterion = nn.CrossEntropyLoss()
model, losses = train_torch_model_adversarial(
    model=model,
    batch_size=batch_size,
    criterion=criterion,
    max_epochs=max_epochs,
    X_train=X_train,
    y_train=y_train,
    lr=lr,
    tolerance=tolerance,
    adversarial_strategy='pgd',
    epsilon=epsilon
)
trained_models['pgd'] = (model, losses)
print('Final training loss for pgd model:', losses[-1][1])

#######
```

```
  0%|          | 0/20 [00:00<?, ?it/s]

Epoch: 0, Loss: 2.260551929473877
```

```
Epoch: 5, Loss: 1.3876010179519653
Epoch: 10, Loss: 0.6220462918281555
Epoch: 15, Loss: 0.4118281602859497
Epoch: 19, Loss: 0.34017112851142883
Final training loss for None model: 0.34017112851142883

  0%|          | 0/20 [00:00<?, ?it/s]

Epoch: 0, Loss: 2.323639392852783
Epoch: 5, Loss: 2.2986323833465576
Epoch: 10, Loss: 2.2233450412750244
Epoch: 15, Loss: 1.7211644649505615
Epoch: 19, Loss: 1.1605569124221802
Final training loss for fgsm model: 1.1605569124221802

  0%|          | 0/20 [00:00<?, ?it/s]

Epoch: 0, Loss: 2.3188986778259277
Epoch: 5, Loss: 2.018470048904419
Epoch: 10, Loss: 0.9669334888458252
Epoch: 15, Loss: 0.6305801868438721
Epoch: 19, Loss: 0.5118045806884766
Final training loss for pgd model: 0.5118045806884766
```

#2. Measuring Standard Performance

- Compute and print the accuracy of each of the three trained models on the clean test dataset.
- You can implement a function similar to the `print_accuracies_torch` function from HW1-Q1 (but compute accuracy only for the test set).

[8]:
```python
#######
from sklearn.metrics import accuracy_score

#TODO:
def get_accuracies_torch(model, X, y):
    predictions_train = model(X)
    y_train_pred = torch.argmax(predictions_train, dim=1).cpu().numpy()
    y = y.cpu().numpy()
    accuracy = accuracy_score(y_train_pred, np.asarray(y, dtype=np.float32))
    return accuracy


for key in trained_models:
    model, _ = trained_models[key]
    acc = get_accuracies_torch(model, X_test, y_test)
    print('Accuracy of {} model on clean test dataset: {}'.format(key, acc))

#######
```

```
Accuracy of None model on clean test dataset: 0.8746
Accuracy of fgsm model on clean test dataset: 0.6588
Accuracy of pgd model on clean test dataset: 0.8334
```

#3. Measuring Adversarial Robustness

- Using the same test dataset, perform adversarial attack to compute robust accuracy for each of the three models. Report the robust accuracy of each of the three models for both:
  - FGSM attack
  - PGD attack
- Note: In total, you need to report 6 robust accuracies here (3 models * 2 attacks).
- To create PGD attack examples, use `alpha=0.01`, `num_iter=40`.

[9]:
```
#######
#TODO:
for key in trained_models:
    model, _ = trained_models[key]
    model.eval()

    # for fgsm
    delta_fgsm = fgsm(model, X_test, y_test, epsilon=0.05)
    X_adv_fgsm = torch.clamp(X_test + delta_fgsm, 0, 1)
    preds_fgsm = torch.argmax(model(X_adv_fgsm), dim=1)
    score = (preds_fgsm == y_test).float().mean().item()
    print('Robust accuracy of {} model on fgsm attack: {}'.format(key, score))

    # for pgd
    delta_pgd = pgd(model, X_test, y_test, alpha=0.01, epsilon=0.05,␣
 ↪num_iter=40)
    X_adv_pgd = torch.clamp(X_test + delta_pgd, 0, 1)
    preds_pgd = torch.argmax(model(X_adv_pgd), dim=1)
    score = (preds_pgd == y_test).float().mean().item()
    print('Robust accuracy of {} model on pgd attack: {}'.format(key, score))

#######
```

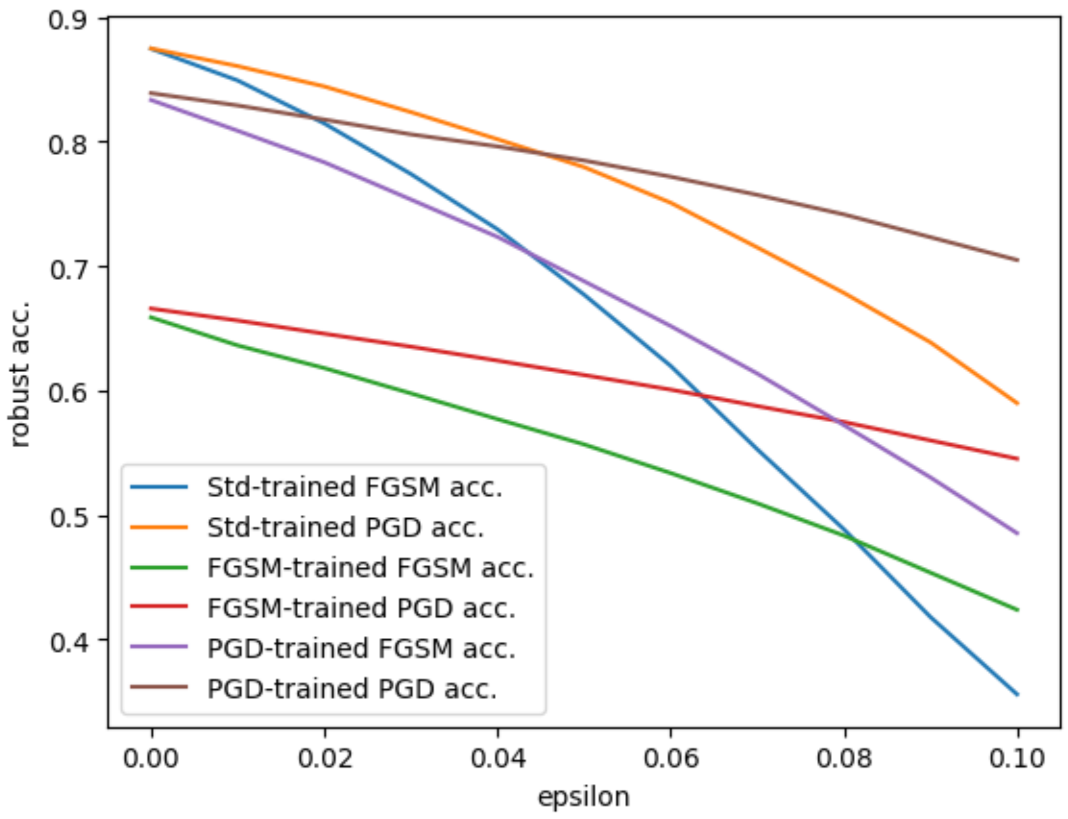
```
Robust accuracy of None model on fgsm attack: 0.656499981880188
Robust accuracy of None model on pgd attack: 0.7795000076293945
Robust accuracy of fgsm model on fgsm attack: 0.5807999968528748
Robust accuracy of fgsm model on pgd attack: 0.6125999689102173
Robust accuracy of pgd model on fgsm attack: 0.7073999643325806
Robust accuracy of pgd model on pgd attack: 0.7849000096321106
```

1.0.5 *(c) Evaluate the robust trained models at different epsilon (1.5 points)*

- Report the robust accuracy of each of the three models (standard-trained, FGSM-trained and PGD-trained from **(b)**) using FGSM and PGD attacks at `epsilon = [0, 0.01, 0.02, ..., 0.09, 0.1]`.
- Plot a (single) graph of robust accuracy vs. `epsilon` (i.e. six curves, since we have 3 models and 2 attacks).

[10]:
```python
from re import X
#######
# TODO:

eps_list = np.arange(0, 0.11, 0.01)

std_model_fgsm = list()
std_model_pgd = list()
fgsm_model_fgsm = list()
fgsm_model_pgd = list()
pgd_model_fgsm = list()
pgd_model_pgd = list()

for eps in eps_list:
    for key in trained_models:
        model, _ = trained_models[key]
        model.eval()

        # FGSM Attack
        delta_fgsm = fgsm(model, X_test, y_test, epsilon=eps)
        X_fgsm = torch.clamp(X_test + delta_fgsm, 0, 1)
        preds = torch.argmax(model(X_fgsm), dim=1)
        acc = (preds == y_test).float().mean().item()

        if key == 'None':
            std_model_fgsm.append(acc)
        elif key == 'fgsm':
            fgsm_model_fgsm.append(acc)
        elif key == 'pgd':
            pgd_model_fgsm.append(acc)

        # PGD Attack
        delta_pgd = pgd(model, X_test, y_test, alpha=0.01, epsilon=eps,␣
 ↪num_iter=40)
        X_pgd = torch.clamp(X_test + delta_pgd, 0, 1)
        preds = torch.argmax(model(X_pgd), dim=1)
        acc = (preds == y_test).float().mean().item()

        if key == 'None':
            std_model_pgd.append(acc)
        elif key == 'fgsm':
            fgsm_model_pgd.append(acc)
        elif key == 'pgd':
            pgd_model_pgd.append(acc)

#######
```

[11]:
```
plt.plot(eps_list, std_model_fgsm, label='Std-trained FGSM acc.')
plt.plot(eps_list, std_model_pgd, label='Std-trained PGD acc.')

plt.plot(eps_list, fgsm_model_fgsm, label='FGSM-trained FGSM acc.')
plt.plot(eps_list, fgsm_model_pgd, label='FGSM-trained PGD acc.')

plt.plot(eps_list, pgd_model_fgsm, label='PGD-trained FGSM acc.')
plt.plot(eps_list, pgd_model_pgd, label='PGD-trained PGD acc.')

plt.xlabel('epsilon')
plt.ylabel('robust acc.')
plt.legend()
plt.show()
```



1.0.6 *(d) Analysis of results (1.5 points)*

Describe and analyze the observations from the graph that you plotted in the previous question **(c)**.

Overall Tendency

1. **Observation:** Across all model types, robust accuracy consistently decreases as epsilon

increases from 0 to 0.1.

- Analysis: It is consistent with our knowledge that, the bigger the perturbation is, the harder for models to maintain accurate predictions.

Diffrent Robust Training Method

1. **Blue line and Orange line Observation:** The robust accuracy of the standard training model drop sharply when epsilon is bigger than 0.02.

- Analysis: The standard model has little robustness, and it will be easily attacked.

2. **Green line and Red line Observation:** FGSM-trained model has lower standard performance compared to the Standard and PGD-trained models. It is robust when facing attack with small epsilon (accuracy slowly decrease), but has a relatively low robust accuracy when the epsilon is big (bigger than 0.08).

- Analysis: The poor initial performance may be due to insufficient training (e.g., only 20 epochs). With extended training (e.g., 40 epochs), the loss converges more effectively, leading to improved standard and robust accuracy. However, the robustness of FGSM-trained models does not generalize well to stronger attacks or larger perturbations, suggesting limited scalability of this method.

3. **Purple line and Brown line Observation:** PGD-trained model is the most robust when facing PGD attack (Brown line); and it is also robust when facing FGSM attack (Purple line), although the robust accuracy declines sightly faster under FGSM attack.

- Analysis: PGD-trained model learns stable features through pgd_attacked_dataset, thus yielding the strongest robustness among three models. And this method is effective at teaching the model to generalize against different perturbation patterns compared to FGSM-trained method.