Question 4

AlexNet

In this problem, you are asked to train a deep convolutional neural network to perform image classification. In fact, this is a slight variation of a network called *AlexNet*. This is a landmark model in deep learning, and arguably kickstarted the current (and ongoing, and massive) wave of innovation in modern AI when its results were first presented in 2012. AlexNet was the first real-world demonstration of a *deep* classifier that was trained end-to-end on data and that outperformed all other ML models thus far.

We will train AlexNet using the CIFAR10 dataset, which consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. The classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

A lot of the code you will need is already provided in this notebook; all you need to do is to fill in the missing pieces, and interpret your results.

Warning : AlexNet takes a good amount of time to train (~1 minute per epoch on Google Colab). So please budget enough time to do this homework.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim.lr_scheduler import _LRScheduler
import torch.utils.data as data

import torchvision.transforms as transforms
import torchvision.datasets as datasets

from sklearn import decomposition
from sklearn import manifold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import numpy as np

import copy
import random
import time

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Loading and Preparing the Data

Our dataset is made up of color images but three color channels (red, green and blue), compared to MNIST's black and white images with a single color channel. To normalize our data we need to calculate the means and standard deviations for each of the color channels independently, and normalize them.

```
ROOT = '.data'
train_data = datasets.CIFAR10(root = ROOT,
                              train = True,
                              download = True)
```

```
Files already downloaded and verified
```

```
# Compute means and standard deviations along the R,G,B channel

means = train_data.data.mean(axis = (0,1,2)) / 255
stds = train_data.data.std(axis = (0,1,2)) / 255
```

Next, we will do data augmentation. For each training image we will randomly rotate it (by up to 5 degrees), flip/mirror with probability 0.5, shift by +/-1 pixel. Finally we will normalize each color channel using the means/stds we calculated above.

```
train_transforms = transforms.Compose([
                           transforms.RandomRotation(5),
                           transforms.RandomHorizontalFlip(0.5),
                           transforms.RandomCrop(32, padding = 2),
                           transforms.ToTensor(),
                           transforms.Normalize(mean = means,
                                                std = stds)
```

```
                       ])

test_transforms = transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize(mean = means,
                                                std = stds)
                       ])
```

Next, we'll load the dataset along with the transforms defined above.

We will also create a validation set with 10% of the training samples. The validation set will be used to monitor loss along different epochs, and we will pick the model along the optimization path that performed the best, and report final test accuracy numbers using this model.

```
train_data = datasets.CIFAR10(ROOT,
                              train = True,
                              download = True,
                              transform = train_transforms)

test_data = datasets.CIFAR10(ROOT,
                             train = False,
                             download = True,
                             transform = test_transforms)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
VALID_RATIO = 0.9

n_train_examples = int(len(train_data) * VALID_RATIO)
n_valid_examples = len(train_data) - n_train_examples

train_data, valid_data = data.random_split(train_data,
                                           [n_train_examples,
n_valid_examples])

valid_data = copy.deepcopy(valid_data)
valid_data.dataset.transform = test_transforms
```

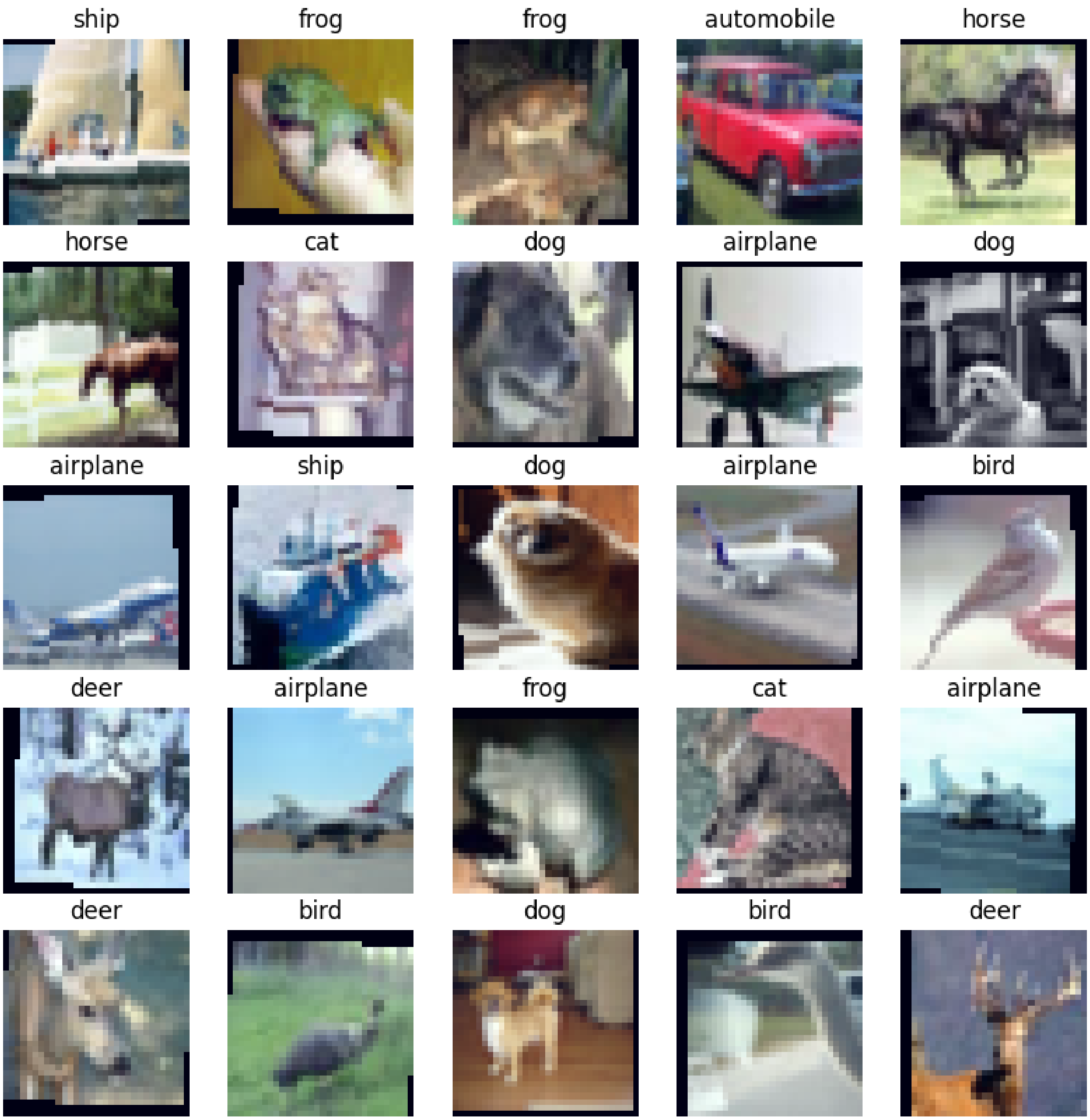
Now, we'll create a function to plot some of the images in our dataset to see what they actually look like.

Note that by default PyTorch handles images that are arranged `[channel, height, width]`, but `matplotlib` expects images to be `[height, width, channel]`, hence we need to permute the dimensions of our images before plotting them.

```
def plot_images(images, labels, classes, normalize = False):

    n_images = len(images)
    rows = int(np.sqrt(n_images))
    cols = int(np.sqrt(n_images))

    fig = plt.figure(figsize = (10, 10))

    for i in range(rows*cols):

        ax = fig.add_subplot(rows, cols, i+1)

        image = images[i]

        if normalize:
            image_min = image.min()
            image_max = image.max()
            image.clamp_(min = image_min, max = image_max)
            image.add_(-image_min).div_(image_max - image_min + 1e-5)

        ax.imshow(image.permute(1, 2, 0).cpu().numpy())
        ax.set_title(classes[labels[i]])
        ax.axis('off')
```

One point here: `matplotlib` is expecting the values of every pixel to be between $[0,1)$, however our normalization will cause them to be outside this range. By default `matplotlib` will then clip these values into the $[0,1)$ range. This clipping causes all of the images to look a bit weird - all of the colors are oversaturated. The solution is to normalize each image between [0,1].

```
N_IMAGES = 25

images, labels = zip(*[(image, label) for image, label in
                           [train_data[i] for i in range(N_IMAGES)]])

classes = test_data.classes

plot_images(images, labels, classes, normalize = True)
```



We'll be normalizing our images by default from now on, so we'll write a function that does it for us which we can use whenever we need to renormalize an image.

```
def normalize_image(image):
    image_min = image.min()
    image_max = image.max()
    image.clamp_(min = image_min, max = image_max)
    image.add_(-image_min).div_(image_max - image_min + 1e-5)
    return image
```

The final bit of the data processing is creating the iterators. We will use a large. Generally, a larger batch size means that our model trains faster but is a bit more susceptible to overfitting.

```
# Q1: Create data loaders for train_data, valid_data, test_data
# Use batch size 256


BATCH_SIZE = 256

train_iterator = torch.utils.data.DataLoader(train_data,
batch_size=BATCH_SIZE, shuffle=True)#Added the iterable wraps for the
datasets

valid_iterator = torch.utils.data.DataLoader(valid_data,
batch_size=BATCH_SIZE, shuffle=True)

test_iterator = torch.utils.data.DataLoader(test_data,
batch_size=BATCH_SIZE, shuffle=True)
```

Defining the Model

Next up is defining the model.

AlexNet will have the following architecture:

- There are 5 2D convolutional layers (which serve as *feature extractors*), followed by 3 linear layers (which serve as the *classifier*).

- All layers (except the last one) have `ReLU` activations. (Use `inplace=True` while defining your ReLUs.)

- All convolutional filter sizes have kernel size 3 x 3 and padding 1.

- Convolutional layer 1 has stride 2. All others have the default stride (1).

- Convolutional layers 1,2, and 5 are followed by a 2D maxpool of size 2.

- Linear layers 1 and 2 are preceded by Dropouts with Bernoulli parameter 0.5.

- For the convolutional layers, the number of channels is set as follows. We start with 3 channels and then proceed like this:

  - 3 → 64 → 192 → 384 → 256 → 256

  In the end, if everything is correct you should get a feature map of size $ 2\times2 \ times 256 = 1024$.

- For the linear layers, the feature sizes are as follows:

  - 1024 → 4096 → 4096 → 10.

  (The 10, of course, is because 10 is the number of classes in CIFAR-10).

```
class AlexNet(nn.Module):
    def __init__(self, output_dim):
        super().__init__()

        self.features = nn.Sequential(
            # Define according to the steps described above
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
#Added layers for AlexNet
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
        )

        self.classifier = nn.Sequential(
            # define according to the steps described above
            nn.Dropout(p=0.5),
            nn.Linear(1024, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 10),
        )

    def forward(self, x):
        x = self.features(x)
        h = x.view(x.shape[0], -1)
        x = self.classifier(h)
        return x, h
```

We'll create an instance of our model with the desired amount of classes.

```
OUTPUT_DIM = 10
model = AlexNet(OUTPUT_DIM)
```

Training the Model

We first initialize parameters in PyTorch by creating a function that takes in a PyTorch module, checking what type of module it is, and then using the `nn.init` methods to actually initialize the parameters.

For convolutional layers we will initialize using the *Kaiming Normal* scheme, also known as *He Normal*. For the linear layers we initialize using the *Xavier Normal* scheme, also known as *Glorot Normal*. For both types of layer we initialize the bias terms to zeros.

```
def initialize_parameters(m):
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight.data, nonlinearity = 'relu')
        nn.init.constant_(m.bias.data, 0)
    elif isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight.data, gain =
nn.init.calculate_gain('relu'))
        nn.init.constant_(m.bias.data, 0)
```

We apply the initialization by using the model's `apply` method. If your definitions above are correct you should get the printed output as below.

```
model.apply(initialize_parameters)
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=1024, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

We then define the loss function we want to use, the device we'll use and place our model and criterion on to our device.

```
optimizer = optim.Adam(model.parameters(), lr = 1e-3)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
criterion = nn.CrossEntropyLoss()

model = model.to(device)
criterion = criterion.to(device)

# This is formatted as code
```

We define a function to calculate accuracy...

```
def calculate_accuracy(y_pred, y):
    top_pred = y_pred.argmax(1, keepdim = True)
    correct = top_pred.eq(y.view_as(top_pred)).sum()
    acc = correct.float() / y.shape[0]
    return acc
```

As we are using dropout we need to make sure to "turn it on" when training by using `model.train()`.

```
def train(model, iterator, optimizer, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for (x, y) in iterator:

        x = x.to(device)
        y = y.to(device)

        optimizer.zero_grad()

        y_pred, _ = model(x)

        loss = criterion(y_pred, y)

        acc = calculate_accuracy(y_pred, y)

        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

We also define an evaluation loop, making sure to "turn off" dropout with `model.eval()`.

```
def evaluate(model, iterator, criterion, device):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for (x, y) in iterator:

            x = x.to(device)
            y = y.to(device)

            y_pred, _ = model(x)

            loss = criterion(y_pred, y)

            acc = calculate_accuracy(y_pred, y)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Next, we define a function to tell us how long an epoch takes.

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Then, finally, we train our model.

Train it for 25 epochs (using the train dataset). At the end of each epoch, compute the validation loss and keep track of the best model. You might find the command `torch.save` helpful.

At the end you should expect to see validation losses of ~76% accuracy.

```python
# Q3: train your model here for 25 epochs.
# Print out training and validation loss/accuracy of the model after each epoch
# Keep track of the model that achieved best validation loss thus far.
import time

EPOCHS = 25
model_copy = copy.deepcopy(model)
accuracy_ref = -1 #Lowest the accuracy can go is 0, so this lets the accuracy reference and new model be updated on the first epoch
# Fill training code here
for epoch in range(EPOCHS):
    start_time = time.time()
    loss, accuracy =train(model, train_iterator, optimizer, criterion, device) #Training data
    print(f'Epoch {epoch}, Train Loss {loss}, Train Accuracy {accuracy}')
    valid_loss, valid_accuracy =evaluate(model, valid_iterator, criterion, device) #Validation data
    print(f'Epoch {epoch}, Valid Loss {valid_loss}, Valid Accuracy {valid_accuracy}')
    if valid_accuracy > accuracy_ref: #Copies the new model if the validation accuracy is better than the previous copy
        accuracy_ref = valid_accuracy
        model_copy = copy.deepcopy(model)
        print(f'Model Copied')
    end_time = time.time()
    min, sec = epoch_time(start_time, end_time)
    print(f'Epoch took {min} minutes and {sec} seconds')
```

```
Epoch 0, Train Loss 2.3859293684363365, Train Accuracy 0.21515447443181818
Epoch 0, Valid Loss 1.6082023859024048, Valid Accuracy 0.40342371314764025
Model Copied
Epoch took 2 minutes and 32 seconds
Epoch 1, Train Loss 1.5127338414842433, Train Accuracy 0.43745649859986524
Epoch 1, Valid Loss 1.3785551190376282, Valid Accuracy 0.48405330926179885
Model Copied
Epoch took 2 minutes and 30 seconds
Epoch 2, Train Loss 1.3497779613191432, Train Accuracy 0.5120134942910888
Epoch 2, Valid Loss 1.2114853024482728, Valid Accuracy 0.5644990801811218
Model Copied
Epoch took 2 minutes and 20 seconds
Epoch 3, Train Loss 1.252753977071155, Train Accuracy 0.5508149858902801
Epoch 3, Valid Loss 1.1492135107517243, Valid Accuracy
0.5816061586141587
Model Copied
Epoch took 2 minutes and 21 seconds
Epoch 4, Train Loss 1.1764032918621192, Train Accuracy
0.5811496803706343
Epoch 4, Valid Loss 1.1288484692573548, Valid Accuracy
0.6064223349094391
Model Copied
Epoch took 2 minutes and 27 seconds
Epoch 5, Train Loss 1.1137279759753833, Train Accuracy
0.607915483076464
Epoch 5, Valid Loss 1.041766142845154, Valid Accuracy
0.6327435672283173
Model Copied
Epoch took 2 minutes and 25 seconds
Epoch 6, Train Loss 1.0533894764428788, Train Accuracy
0.6306143467399207
Epoch 6, Valid Loss 1.0123582810163498, Valid Accuracy
0.6456916362047196
Model Copied
Epoch took 2 minutes and 34 seconds
Epoch 7, Train Loss 1.016191769729961, Train Accuracy
0.6427503549917177
Epoch 7, Valid Loss 0.9771516680717468, Valid Accuracy
0.656031709909439
Model Copied
Epoch took 2 minutes and 51 seconds
Epoch 8, Train Loss 0.9675288193605163, Train Accuracy
0.6617116477679122
Epoch 8, Valid Loss 0.9208255469799042, Valid Accuracy
0.6822150737047196
Model Copied
Epoch took 3 minutes and 7 seconds
Epoch 9, Train Loss 0.9362695372917436, Train Accuracy
0.673189808360555
Epoch 9, Valid Loss 0.9574713259935379, Valid Accuracy 0.6697265625
Epoch took 2 minutes and 47 seconds
Epoch 10, Train Loss 0.915364300662821, Train Accuracy
0.67897372150963
Epoch 10, Valid Loss 0.8721305072307587, Valid Accuracy
0.6986787676811218
Model Copied
Epoch took 2 minutes and 43 seconds
Epoch 11, Train Loss 0.8793735033409162, Train Accuracy
0.6945791904899207
Epoch 11, Valid Loss 0.8667667210102081, Valid Accuracy
0.6991842836141586
Model Copied
```

```
Epoch took 2 minutes and 40 seconds
Epoch 12, Train Loss 0.8457419658926401, Train Accuracy
0.7068705609576269
Epoch 12, Valid Loss 0.8426105201244354, Valid Accuracy
0.7082375913858414
Model Copied
Epoch took 2 minutes and 43 seconds
Epoch 13, Train Loss 0.8229072757742621, Train Accuracy
0.7161807529628277
Epoch 13, Valid Loss 0.8680290371179581, Valid Accuracy
0.7025850176811218
Epoch took 2 minutes and 53 seconds
Epoch 14, Train Loss 0.8056459250775251, Train Accuracy
0.7222514203326269
Epoch 14, Valid Loss 0.7988647848367691, Valid Accuracy
0.7249540448188782
Model Copied
Epoch took 2 minutes and 49 seconds
Epoch 15, Train Loss 0.7857820635492151, Train Accuracy
0.7266770242290064
Epoch 15, Valid Loss 0.7764606267213822, Valid Accuracy
0.7301700353622437
Model Copied
Epoch took 2 minutes and 44 seconds
Epoch 16, Train Loss 0.7603419257158582, Train Accuracy
0.73777432536537
Epoch 16, Valid Loss 0.7669012516736984, Valid Accuracy
0.7389131426811218
Model Copied
Epoch took 2 minutes and 50 seconds
Epoch 17, Train Loss 0.7526960955424742, Train Accuracy
0.74021573161537
Epoch 17, Valid Loss 0.7977280527353287, Valid Accuracy
0.7365923702716828
Epoch took 2 minutes and 35 seconds
Epoch 18, Train Loss 0.7299478788944808, Train Accuracy
0.74827237224037
Epoch 18, Valid Loss 0.7806057006120681, Valid Accuracy
0.7405560672283172
Model Copied
Epoch took 2 minutes and 42 seconds
Epoch 19, Train Loss 0.7180966392836787, Train Accuracy
0.7520321377299048
Epoch 19, Valid Loss 0.7532464444637299, Valid Accuracy
0.7397977948188782
Epoch took 2 minutes and 41 seconds
Epoch 20, Train Loss 0.695985344323245, Train Accuracy
0.7610040838745508
Epoch 20, Valid Loss 0.7738403081893921, Valid Accuracy
```

```
0.7301470577716828
Epoch took 2 minutes and 45 seconds
Epoch 21, Train Loss 0.687265569852157, Train Accuracy
0.7641637074676427
Epoch 21, Valid Loss 0.7499783217906952, Valid Accuracy
0.7522977948188782
Model Copied
Epoch took 2 minutes and 44 seconds
Epoch 22, Train Loss 0.6752578406171366, Train Accuracy
0.7682998935607347
Epoch 22, Valid Loss 0.7266848772764206, Valid Accuracy
0.7558938413858414
Model Copied
Epoch took 2 minutes and 48 seconds
Epoch 23, Train Loss 0.6537438587031581, Train Accuracy
0.7748322087255392
Epoch 23, Valid Loss 0.7215611875057221, Valid Accuracy
0.7567210465669632
Model Copied
Epoch took 2 minutes and 48 seconds
Epoch 24, Train Loss 0.6516660617833788, Train Accuracy
0.7765660512853753
Epoch 24, Valid Loss 0.6996587306261063, Valid Accuracy
0.7605583637952804
Model Copied
Epoch took 2 minutes and 49 seconds
```

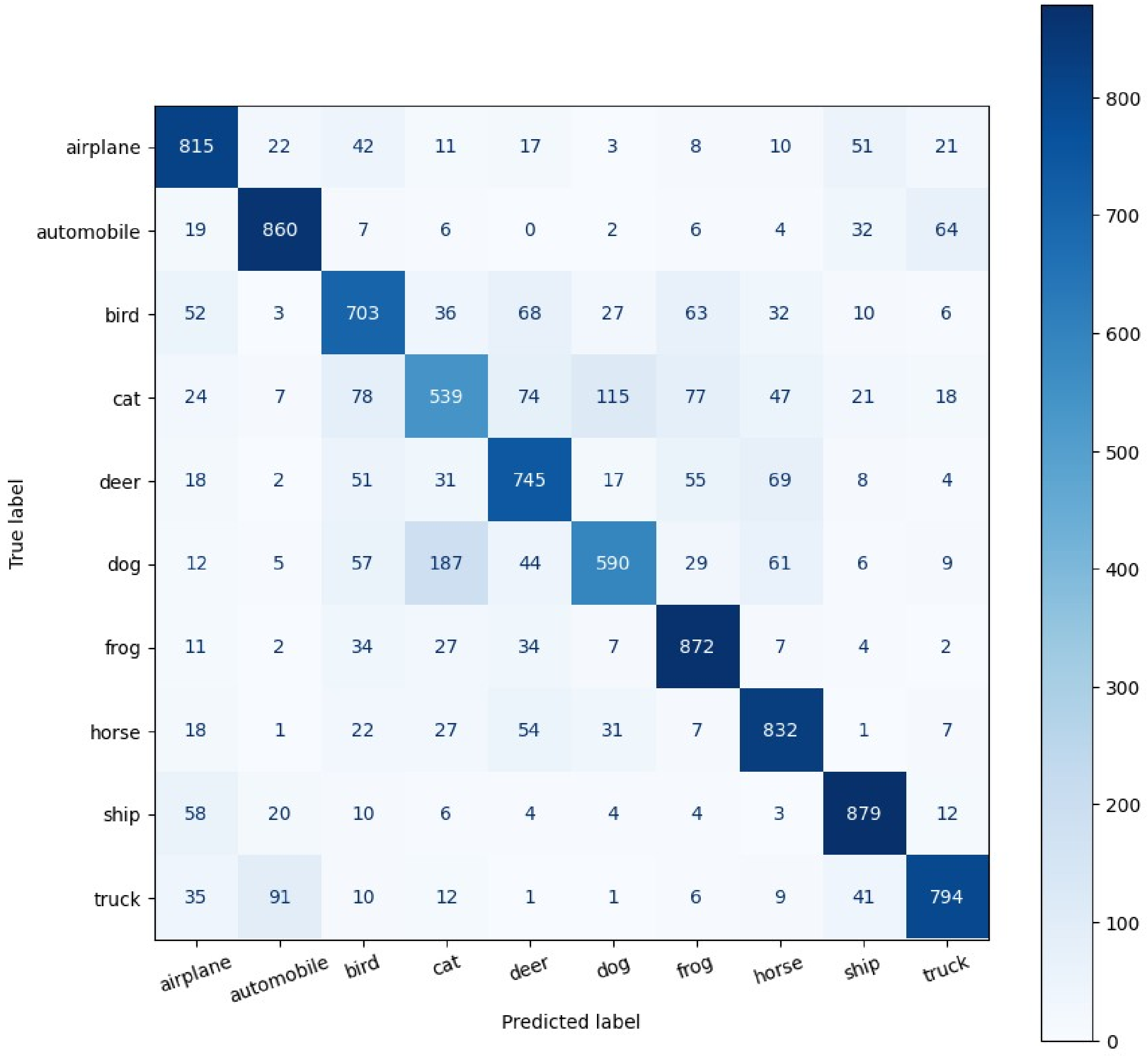
Evaluating the model

We then load the parameters of our model that achieved the best validation loss. You should expect to see ~75% accuracy of this model on the test dataset.

Finally, plot the confusion matrix of this model and comment on any interesting patterns you can observe there. For example, which two classes are confused the most?

```
# Q4: Load the best performing model, evaluate it on the test dataset,
and print test accuracy.

# Also, print out the confusion matrox.

def get_predictions(model, iterator, device):

    model.eval()
    with torch.no_grad():
        labels = []
        probs = []

    # Q4: Fill code here.
```

```
        for (x, y) in iterator:
            x, y = x.to(device), y.to(device)
            predicted_output = model(x) #Prediction on test data
            labels.append(y)
            probs.append(predicted_output[0]) #Ignore the second
tensor



        labels = torch.cat(labels, dim = 0) #Converts all of the
tensors into 1 big tensor
        probs = torch.cat(probs, dim = 0)

    return labels, probs

labels, probs = get_predictions(model_copy, test_iterator, device)
#New model is used

pred_labels = torch.argmax(probs, 1) #Gets index or largest
possibility

print(torch.eq(labels, pred_labels)) #Matches the prediction and label
tensor values

tensor([False,  True,  True,  ...,  True,  True, False])

torch.eq(labels, pred_labels).sum() #Since it's boolean, I can just
sum it all up to get the amount of matches

tensor(7629)

torch.eq(labels, pred_labels).sum()/len(torch.eq(labels, pred_labels))
#Percentage that was predicted correctly at 76%

tensor(0.7629)

def plot_confusion_matrix(labels, pred_labels, classes):

    fig = plt.figure(figsize = (10, 10));
    ax = fig.add_subplot(1, 1, 1);
    cm = confusion_matrix(labels, pred_labels);
    cm = ConfusionMatrixDisplay(cm, display_labels = classes);
    cm.plot(values_format = 'd', cmap = 'Blues', ax = ax)
    plt.xticks(rotation = 20)

plot_confusion_matrix(labels, pred_labels, classes)
```



Conclusion

That's it! As a side project (this is not for credit and won't be graded), feel free to play around with different design choices that you made while building this network.

- Whether or not to normalize the color channels in the input.
- The learning rate parameter in Adam.
- The batch size.
- The number of training epochs.
- (and if you are feeling brave -- the AlexNet architecture itself.)