Deep Learning HW2 - Spring 23

Aniket Sinha - aas10120

Deadline: 10th March 2023

1. (3 points) *Designing convolution filters by hand.* Consider an input 2D image and a 3×3 filter (say w) applied to this image. The goal is to guess good filters which implement each of the following elementary image processing operations.

   (a) Write down the weights of w which acts as a blurring filter, i.e., the output is a blurry form in the input.

   (b) Write down the weights of w which acts as a sharpening filter in the horizontal direction.

   (c) Write down the weights of w which acts as a sharpening filter in the vertical direction.

   (d) Write down the weights of w which acts as a sharpening filter in a diagonal (bottom-left to top-right) direction.

   (e) Give an example of an image operation which cannot be implemented using a 3x3 convolutional filter and briefly explain why

ANSWER:

(a) The **Mean filter** can be used as a blurring filter. The below matrix show the weights of w for a blurring filter

$$w = \frac{1}{9} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

(b) For a sharpening filter in the horizontal direction, **Prewitt or Sobel** operator can be used. The below matrix show the weights of w for a sharpening filter in the horizontal direction.

$$w = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

(c) For a sharpening filter in the vertical direction, **Prewitt** operator can be used. The below matrix show the weights of w for a sharpening filter in the vertical direction.

$$w = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

(d) For a sharpening filter in a diagonal (bottom-left to top-right) direction, the weighrs of w would be:

$$w = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

2. (3 points) *Weight decay.* The use of l_2 regularization for training multilayer neural networks has a special name: weight decay. Assume an arbitrary dataset $(x_i, y_i)_{i=1}^n$ and a loss function $L(w)$ where w is a vector that represents all the trainable weights (and biases).

   (a) Write down the l_2 regularized loss, using a weighting parameter λ for the regularizer.

   (b) Derive the gradient descent update rules for this loss.

   (c) Conclude that in each update, the weights are "shrunk" or "decayed" by a multiplicative factor before applying the descent update.

   (d) What does increasing λ achieve algorithmically, and how should the learning rate be chosen to make the updates stable?

ANSWER:

Given: Dataset: $\{(x_i, y_i)\}_{i=1}^n$ and Loss function $L(w)$

(a) The l_2 regularized loss, $\overline{L}(w)$, using weighting parameter(λ)) is given by:

$$\overline{L}(w) = L(w) + \lambda \parallel w \parallel_2^2$$

(b) Gradient descent equations for the l_2 regularized loss. Here, we are taking the derivative of the above loss function w.r.t. w:

$$\Delta\overline{L}(w) = \Delta L(w) + 2.\lambda.w$$

$$w_{i+1} = w_i - \eta.\Delta\overline{L}(w_i)$$

$$w_{i+1} = w_i - \eta.(\Delta L(w) + 2.\lambda.w)$$

$$w_{i+1} = w_i(1 - 2\eta\lambda) - \eta\Delta L(w)$$

(c) From the above derived equation:

$$w_{i+1} = w_i(1 - 2\eta\lambda) - \eta\Delta L(w)$$

we can see that the weight is being updated by the factor of

$$(1 - 2\eta\lambda)$$

where η is the learning rate and the new weight contains the decaying old weight. Therefore, we can see that the weights are actually getting "shrunk" or "decayed" by a multiplicative factor before applying the descent update.

(d) As per the equation for updating the gradient, we can observe that the optimal value is achieved when the gradient gets closer to zero. Hence, with that in consideration, we can arrive at the following conclusion

$$\mid \Delta w \mid = \mid \Delta L(w) \mid /2\lambda$$

, Based on this, we can infer that

$$\mid \Delta w \mid \propto \frac{1}{2\lambda}$$

.

Therefore, we can deduce from this, that as the value of lambda increases, the weight decreases.

Also, based on the equation $1 - 2\eta\lambda$, we can observe that the relationship

$$\eta \propto \frac{1}{2\lambda}$$

must exist in order to ensure a stable update. Specifically, when we have a larger value of λ, a smaller value of η is required.

3. (2 points) *The IoU metric.* In class we defined the IoU metric (or the Jaccard similarity index) for comparing bounding boxes.

(a) Using elementary properties of sets, argue that the IoU metric between any two pair of bounding boxes is always a non-negative real number in [0, 1].

(b) If we represent each bounding box as a function of the top-left and bottom-right coordinates (assume all coordinates are real numbers) then argue that the IoU metric is non-differentiable and hence cannot be directly optimized by gradient descent.

ANSWER:

(a) Suppose we have two bounding boxes, A and B, where Box A is larger than Box B. Without the loss of generality, we can define the IOU as follows:

$$IOU(A, B) = \frac{\mid A \cap B \mid}{\mid A \cup B \mid}$$

Now, from above, we know that :

$$0 <\mid A \cap B \mid \leq \mid B \mid$$

$$\mid B \mid \leq \mid A \cup B \mid \leq \mid A \mid + \mid B \mid$$

Using the above two equations, we get,

$$\frac{0}{\mid A \mid + \mid B \mid} \leq \frac{\mid A \cap B \mid}{\mid A \cup B \mid} < \frac{\mid B \mid}{\mid B \mid}$$

Hence, we can say that :

$$0 \leq IOU(A, B) \leq 1$$

Therefore,

$$\Rightarrow IOU(A, B) \in [0, 1]$$

(b) Consider two boxes, A and B, defined by their corner points $(x_1, y_1), (x_2, y_2)$ and $(x_3, y_3), (x_4, y_4)$, respectively. Suppose that Box A moves from left to right towards Box B. When the two boxes overlap, we can calculate the intersection as follows:

$$Top - left = (min(x_1, x_3), min(y_1, y_3))$$

$$Bottom - right = (min(x_2, x_4), min(y_2, y_4))$$

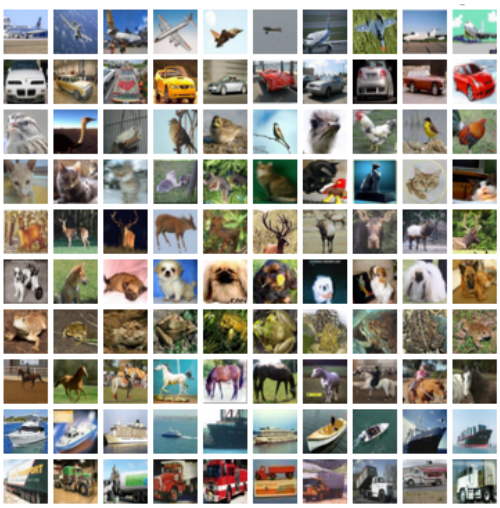
If there is full overlap between the two boxes, the IoU will reach its maximum value, while if there is no overlap, that will result in an IoU close to 0. However, because the numerator of the IoU calculation depends on the non-differentiable minima function, the IoU function itself is also non-differentiable. Hence, gradient descent cannot be used to directly optimize the IoU at every point, as differentiation is not possible for the non-differentiable parts of the function.

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 = data.DataLoader(train_data,   # Load the training data
                                 batch_size=BATCH_SIZE,   # Setting batch size for training data
                                 shuffle=True,   # Shuffling the data for each epoch
                                 num_workers=2);   # Using 2 workers for parallel data loading

valid_iterator = data.DataLoader(valid_data,   # Load the validation data
                                 batch_size=BATCH_SIZE,   # Setting batch size for validation data
                                 shuffle=True,   # Shuffling the data for each epoch
                                 num_workers=2);   # Using 2 workers for parallel data loading

test_iterator = data.DataLoader(test_data,   # Load the test data
                                batch_size=BATCH_SIZE,   # Setting batch size for test data
                                shuffle=True,   # Shuffling the data for each epoch
                                num_workers=2);   # Using 2 workers for parallel data loading
```

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:
  - $3 \rightarrow 64 \rightarrow 192 \rightarrow 384 \rightarrow 256 \rightarrow 256$

  In the end, if everything is correct you should get a feature map of size $2 \times 2 \times 256 = 1024$.
- For the linear layers, the feature sizes are as follows:
  - $1024 \rightarrow 4096 \rightarrow 4096 \rightarrow 10$.

  (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

            # 3 input channels, 64 output channels, 3x3 kernel, padding=1, stride=2
            nn.Conv2d(3,64,kernel_size=3,padding=1,stride=2),
            nn.MaxPool2d(kernel_size=2),   # 2x2 max pooling with stride=2
            nn.ReLU(inplace=True),   # ReLU activation function

            # 64 input channels, 192 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(64,192,kernel_size=3,padding=1,stride=1),
            nn.MaxPool2d(kernel_size=2),   # 2x2 max pooling with stride=2
            nn.ReLU(inplace=True),   # ReLU activation function

            # 192 input channels, 384 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(192,384,kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),   # ReLU activation function

            # 384 input channels, 256 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(384,256,kernel_size=3,padding=1,stride=1),
            nn.ReLU(inplace=True),   # ReLU activation function

            # 256 input channels, 256 output channels, 3x3 kernel, padding=1, stride=1
            nn.Conv2d(256,256,kernel_size=3,padding=1,stride=1),
            nn.MaxPool2d(kernel_size=2),   # 2x2 max pooling with stride=2
            nn.ReLU(inplace=True)   # ReLU activation function

        )

        self.classifier = nn.Sequential(
            # define according to the steps described above

            # Dropout regularization to prevent overfitting
            nn.Dropout(),

            # Fully connected layer with 1024 input features and 4096 output features
            nn.Linear(1024,4096),

            nn.ReLU(inplace=True),   # ReLU activation function

            # Dropout regularization to prevent overfitting
            nn.Dropout(),

            # Fully connected layer with 4096 input features and 4096 output features
            nn.Linear(4096,4096),
            nn.ReLU(inplace=True),   # ReLU activation function

            # Fully connected layer with 4096 input features and 10 output features (for classification)
            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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): ReLU(inplace=True)
    (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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): ReLU(inplace=True)
  )
  (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.

```
# 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.

EPOCHS = 25

# Fill training code here
best_accuracy = 0  # initialize the best validation accuracy to 0
for ep in np.arange(EPOCHS):  # loop over the specified number of epochs
    st_time = time.time()  # record the start time of the epoch
    tr_loss, tr_acc = train(model, train_iterator, optimizer, criterion, device)  # train the model on the training set
    val_loss, validation_accuracy = evaluate(model, valid_iterator, criterion, device)  # evaluate the model on the validatic
    if validation_accuracy > best_accuracy:  # if the validation accuracy is better than the best so far
        torch.save(model, 'best_model.pth')  # save the model as the new best model
        best_accuracy = validation_accuracy  # update the best validation accuracy
    end_time = time.time()  # record the end time of the epoch
    elapsed_mins, elapsed_secs = epoch_time(st_time, end_time)  # calculate the elapsed time of the epoch
    # print the epoch number, elapsed time, training loss/accuracy, validation loss/accuracy
    print("Epoch-{0}[ Elapsed Time: {1}min {2}s] Train [ Loss={3:.2f} Acc={4:.2f} ] Validation [ Loss={5:.2f} Acc={6:.2f} ]"
          .format(ep + 1, elapsed_mins, elapsed_secs, tr_loss, tr_acc, val_loss, validation_accuracy))
```

```
Epoch-1[ Elapsed 0min 29s] Train [ Loss=2.32 Acc=0.24 ] Validation [ Loss=1.59 Acc=0.40 ]
Epoch-2[ Elapsed 0min 28s] Train [ Loss=1.52 Acc=0.44 ] Validation [ Loss=1.33 Acc=0.52 ]
Epoch-3[ Elapsed 0min 28s] Train [ Loss=1.35 Acc=0.51 ] Validation [ Loss=1.19 Acc=0.58 ]
Epoch-4[ Elapsed 0min 28s] Train [ Loss=1.25 Acc=0.55 ] Validation [ Loss=1.17 Acc=0.59 ]
Epoch-5[ Elapsed 0min 29s] Train [ Loss=1.16 Acc=0.59 ] Validation [ Loss=1.09 Acc=0.62 ]
Epoch-6[ Elapsed 0min 28s] Train [ Loss=1.11 Acc=0.61 ] Validation [ Loss=1.05 Acc=0.63 ]
Epoch-7[ Elapsed 0min 28s] Train [ Loss=1.05 Acc=0.63 ] Validation [ Loss=0.99 Acc=0.66 ]
Epoch-8[ Elapsed 0min 28s] Train [ Loss=1.00 Acc=0.64 ] Validation [ Loss=0.94 Acc=0.67 ]
Epoch-9[ Elapsed 0min 29s] Train [ Loss=0.97 Acc=0.66 ] Validation [ Loss=0.95 Acc=0.66 ]
Epoch-10[ Elapsed 0min 28s] Train [ Loss=0.93 Acc=0.68 ] Validation [ Loss=0.91 Acc=0.68 ]
Epoch-11[ Elapsed 0min 28s] Train [ Loss=0.90 Acc=0.68 ] Validation [ Loss=0.88 Acc=0.69 ]
Epoch-12[ Elapsed 0min 28s] Train [ Loss=0.88 Acc=0.69 ] Validation [ Loss=0.83 Acc=0.71 ]
Epoch-13[ Elapsed 0min 29s] Train [ Loss=0.84 Acc=0.71 ] Validation [ Loss=0.82 Acc=0.72 ]
Epoch-14[ Elapsed 0min 28s] Train [ Loss=0.82 Acc=0.71 ] Validation [ Loss=0.81 Acc=0.72 ]
Epoch-15[ Elapsed 0min 28s] Train [ Loss=0.80 Acc=0.72 ] Validation [ Loss=0.83 Acc=0.71 ]
Epoch-16[ Elapsed 0min 28s] Train [ Loss=0.79 Acc=0.73 ] Validation [ Loss=0.77 Acc=0.74 ]
Epoch-17[ Elapsed 0min 28s] Train [ Loss=0.76 Acc=0.74 ] Validation [ Loss=0.77 Acc=0.75 ]
Epoch-18[ Elapsed 0min 27s] Train [ Loss=0.75 Acc=0.74 ] Validation [ Loss=0.77 Acc=0.75 ]
Epoch-19[ Elapsed 0min 27s] Train [ Loss=0.74 Acc=0.75 ] Validation [ Loss=0.77 Acc=0.74 ]
Epoch-20[ Elapsed 0min 28s] Train [ Loss=0.71 Acc=0.76 ] Validation [ Loss=0.74 Acc=0.76 ]
Epoch-21[ Elapsed 0min 27s] Train [ Loss=0.69 Acc=0.76 ] Validation [ Loss=0.76 Acc=0.75 ]
Epoch-22[ Elapsed 0min 27s] Train [ Loss=0.70 Acc=0.76 ] Validation [ Loss=0.74 Acc=0.75 ]
Epoch-23[ Elapsed 0min 28s] Train [ Loss=0.67 Acc=0.77 ] Validation [ Loss=0.73 Acc=0.76 ]
Epoch-24[ Elapsed 0min 27s] Train [ Loss=0.66 Acc=0.77 ] Validation [ Loss=0.72 Acc=0.76 ]
Epoch-25[ Elapsed 0min 27s] Train [ Loss=0.65 Acc=0.78 ] Validation [ Loss=0.72 Acc=0.76 ]
```

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.

model=torch.load("best_model.pth") #loading the best performing model
test_loss,test_accuracy=evaluate(model,test_iterator,criterion,device) #evaluating it on test dataset
print("Test [ Loss=%.2f Acc=%.2f} ]" % (test_loss,test_accuracy)) #print the test loss and accuracy
```

```
Test [ Loss=0.72 Acc=0.75} ]
```

```
def get_predictions(model, iterator, device):

    model.eval()

    labels = []
    probs = []

    # Q4: Fill code here.

    #I'm using a PyTorch model to predict on data from a loader, concatenating the
    #targets and predictions into tensors, and moving them to device.
    #The torch.no_grad() block disables gradient computation for efficiency.

    with torch.no_grad():

        for (x, y) in iterator:
            # moving x,y to device
            x = x.to(device)
            y = y.to(device)
            #forward pass through the model
            y_pred, _ = model(x)
            # add target to list of targets
            labels.append(y)
            # add predicted output to list of predictions
            probs.append(y_pred)

    labels = torch.cat(labels, dim = 0) # concatenate list of targets into one tensor
    probs = torch.cat(probs, dim = 0) # concatenating list of predictions into one tensor
    labels = labels.to(device)
    probs = probs.to(device)

    return labels, probs
```

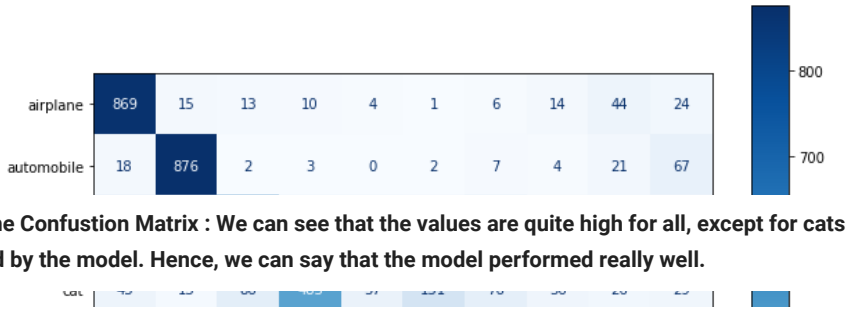
```
labels, probs = get_predictions(model, test_iterator, device) # getting predictions using the model and test data
labels = labels.cpu().numpy() #converting to numpy array
```

```
pred_labels = torch.argmax(probs, 1).cpu().numpy()
```

```
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)
```



From the Confustion Matrix : We can see that the values are quite high for all, except for cats and dogs. Even then, the images were correctly labelled by the model. Hence, we can say that the model performed really well.

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.)