ENGN8536 CLab2 Report

Shreya Chawla (u7195872)

Q1: The code for Q1 is in Appendix A. The data is first loaded from the “ *./engn/Dataset*” directory into train and test by importing “*os*”. The train is then split into train and validation sets using sklearn library’s *train_test_split()* function with split=0.1. This ensures that the number of train, validation and testing samples are 18000, 2000 and 4000 each respectively.

The data is labelled according to the file name by splitting the filename string. These labels are stored in a list and returned by *label_data()* function for further use. The cat image files are labelled by 0 and dog image files by 1.

Next, the data is transformed - train data using *train_image_transform()* function and the rest with *image_transform_all()* function as required. The *torchvision.transforms* module is used to make this possible.
The test and validation images are first resized to 224x224 size. To make the test and validation data normalized for it to range between -1 and 1, mean of [0.5,0.5,0.5] and standard deviation of [0.5,0.5,0.5] is used.
Training images are randomly horizontally flipped (over the vertical axis) to increase the dataset. They are padded, normalized and randomly cropped.

The image tensors and their respective labels are stacked together in a tensor. Next, dataloaders of a given *batch_size* with shuffled data are created for the three sets and returned by the *get_data_loaders()* function.

Q2: *CNNModel* class is built with the given architecture: (Appendix B)

```
CNNModel(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batch1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batch2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=200704, out_features=1024, bias=True)
  (batch3): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu3): ReLU()
  (fc2): Linear(in_features=1024, out_features=1, bias=True)
)
```

BCE Loss with Logits (*BCEWithLogitsLoss*) is set as the criterion for improving model and *Adam* as the optimizer for the model.
The visualize function in visualize.py (Appendix C) visualizes the 4 plots as shown in Fig 1.

The hyperparameters were tuned to improve the results also the architecture was modified to decrease overfitting. The model is trained for 15 *epochs* with *learning rate* as 0.002. The *batch_size* is 16. For all the training, testing and validation purposes, to compare results, a seed of 10 was used for both torch and cuda.

(a) Training accuracy vs epochs

(b) Training loss vs epochs

(c) Validation accuracy vs epochs

(d) Validation loss vs epochs

Fig 1: The 4 plots for Q3.4 - (a) Training accuracy vs. epochs, (b) Training loss vs. epochs, (c) Validation accuracy vs epochs and (d) Validation loss vs epochs.

The modified CNN model is as follows:

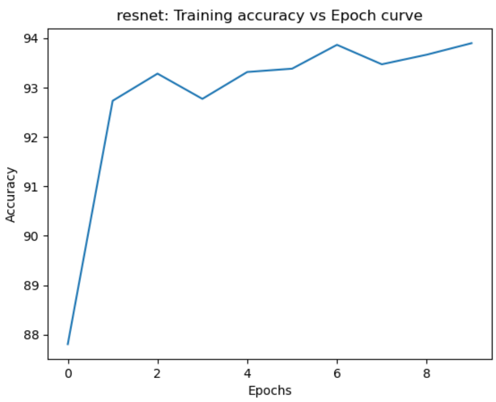
```
CNNModel(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batch4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu4): ReLU()
  (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batch1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (batch2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=200704, out_features=1024, bias=True)
  (batch3): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu3): ReLU()
  (drop): Dropout(p=0.15, inplace=False)
```

  (fc2): Linear(in_features=1024, out_features=1, bias=True)
)

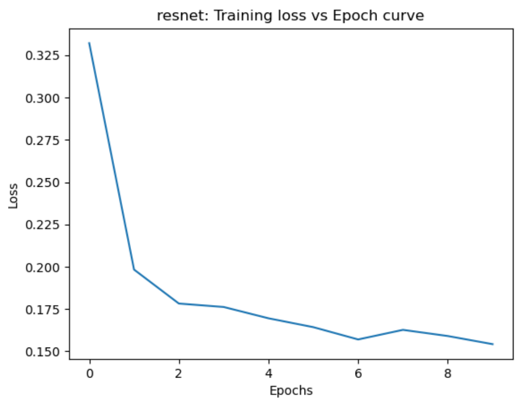
Figure 1 shows that the model has converged during training to a stable accuracy and low loss. The validation accuracy oscillates between 76% and 80% due to the use of mini-batch training but overall it can be considered to have stabilized at around 78%. Although several changes were made, the validation loss is still around 0.7 indicating that the model is overfitting. *Dropout*, and *batchnorm* layers were added to add regularization but it did not have as much effect as expected. Regularization and batch normalization are known to crub overfitting. Fig 1 is the result after adding these layers (Appendix B). Upon testing on the test set, the accuracy was 80.173% with a loss of 0.78. The accuracy is of our model is quite high.

Q3: Using *torchvision*, *ResNet-18* pre-trained model is loaded. All the layers of this model are frozen. The final fully connected layer (*fc*) is modified such that it’s output dimension is 1. This modified final (*fc)* layer is unfrozen (by default when layer is defined, it is unfrozen), which is to be fine-tuned. These modifications to the pre-trained model are made under *ModifiedResnet18 class* in *models.py* script (code in Appendix B).

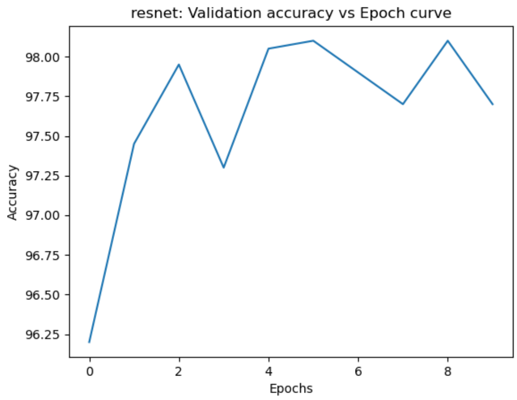
An object of the *ModifiedResnet18* class is trained on the dataset. The learning rate is set very low (*lr* = 0.0001) with a small *batch_size* of 8 looped over 10 *epochs*. A small learning rate is needed to fine-tune pretrained parameters as the model was trained on ImageNet dataset which has lots of cat and dog images. Best results were found for these hyperparameters.



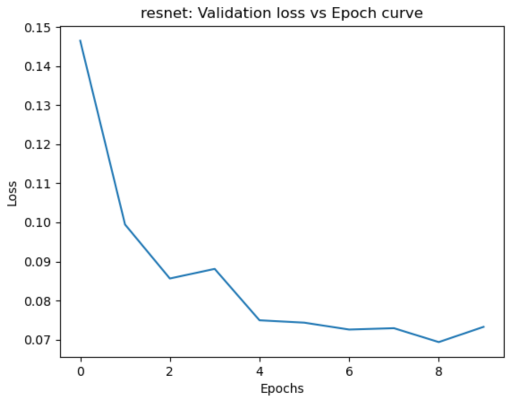
(c) Training accuracy vs epochs



(d) Training loss vs epochs



(c) Validation accuracy vs epochs



(d) Validation loss vs epochs

Fig 2: The 4 plots for Q3.4 - (a) Training accuracy vs. epochs, (b) Training loss vs. epochs, (c) Validation accuracy vs epochs and (d) Validation loss vs epochs.

The plots for this model are in Fig 2. From Fig 2 (a) it is observed that the training has converged with 93.827% accuracy. The train loss in Fig 2 (b) shows that the trained model has not been underfit for sure. The validation accuracy (Fig 2 (c)) oscillates between 97.5% and 98% hence the model performs well on the validation set on which was used to set the hyperparameters for this model. The loss curve in Fig 2 (d) is similar to Fig 2 (b). Hence, the model is stable and has converged to a local minima. The test and validation accuracy as well as loss surpass the training results proving that the model is neither underfit nor overfit. The Average test_accuracy is 97.683 and test_loss is 0.076. Thus, it performs very well on unseen data.
When the learning rate was set higher (lr = 0.05 or lr = 0.01) then the model would overfit as the validation loss would then be much higher than train loss. This can be attributed to the fact that high learning rate would mean a big step but since the unfrozen layer is randomly initialized, it might take wrong step and hence lead to poor generalization. Also large batch size learning (batch_size = 100) also lead to overfitting of the model.

Appendix - Code

A. Code in pre_process.py file (for Q1 - custom data loader)

```
from torchvision import transforms
from PIL import Image
import os
from torch.utils.data import TensorDataset, DataLoader
import torch.tensor
from sklearn.model_selection import train_test_split
import glob


def data_loaders(batch_size):
    """
```

```
    Function to load, read, split, transform image dataset into required
tensors
    :param batch_size: size of each batch for data loader
    :return: train, val and test data loaders
    """
    # Get list of files in train and test directories
    BASE_DIR   = "./engn8536/Dataset/"
    TRAIN_DIR  = os.path.join(BASE_DIR, 'cat-dog-train/')
    TEST_DIR   = os.path.join(BASE_DIR, 'cat-dog-test/')
    train_list = glob.glob(os.path.join(TRAIN_DIR, '*.tif'))
    test_list  = glob.glob(os.path.join(TEST_DIR, '*.tif'))

    # Split train_files into train and validation sets
    train_list, val_list = train_test_split(train_list, test_size=0.1)

    def get_data_loaders(data_list, train=False):
        """
        Helper function to get customized data loaders from list of data
paths
        :param train: if True, then transform data as for training else
for val or test
        :param data_list: list of paths of image data
        :return:
        """
        # Get labels
        data_labels = label_data(data_list)
        # Apply transforms on images and get tensors
        if train:
            data_list = train_image_transform(data_list)
        else:
            data_list = image_transform_all(data_list)
        # Pair image tensors with their label in a tensor dataset
        data = TensorDataset(torch.stack(data_list),
torch.tensor(data_labels))
        # Shuffle the data and get data loaders of specified batch_size
        data_loader = DataLoader(dataset=data, batch_size=batch_size,
shuffle=True)
        return data_loader

    train = get_data_loaders(train_list, True)
    val   = get_data_loaders(val_list, False)
    test  = get_data_loaders(test_list, False)

    print('Loaded images into custom data loaders...')

    return train, val, test


def label_data(files_path):
    """
```

```
    Generates a list of labels according to filenames - 0 denotes label
'cat' and 1 denotes label 'dog'
    :param files_path: path of image files folder
    :return: list of labels
    """
    labels = []
    for filename in files_path:
        label = filename.split('/')[4].split('.')[0]
        if 'cat' == label:
            labels.append(0.0)
        elif 'dog' == label:
            labels.append(1.0)
    return labels


def image_transform_all(dir_path):
    """
    Function to resize image and normalize data when loading data
    :param dir_path: directory path of images
    :return: transformed image tensor
    """
    image_tensor = []
    for image_path in dir_path:
        image = Image.open(image_path)
        transform = transforms.Compose([
            transforms.Resize(224),
            transforms.ToTensor(),
            transforms.Normalize([0.5] * 3, [0.5] * 3)
            ])
        image_tensor.append(transform(image))
    return image_tensor


def train_image_transform(dir_path):
    """
    Perform data augmentation and transform when training - random flip,
padding, crop images
    :param dir_path: directory path of images
    :return: transformed image tensor
    """
    image_tensor = []
    for image_path in dir_path:
        image = Image.open(image_path)
        transform = transforms.Compose([
            transforms.Resize(224),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.Pad(4),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.5] * 3, [0.5] * 3)
            ])
```

```
        transformed_image = transform(image)
        image_tensor.append(transformed_image)
    return image_tensor
```

B. Code in models.py file (for (1) Q2 - contains the CNN model (after modification); (2) Q3 - the ResNet18 pre-trained model)

```
from torch.nn import Module, Conv2d, BatchNorm2d, ReLU, MaxPool2d,
Linear, BatchNorm1d
from torchvision.models import resnet18


# Create CNN Model - Q2
class CNNModel(Module):
def __init__(self, num_classes):
    super(CNNModel, self).__init__()

    self.conv1 = Conv2d(3, 32, kernel_size=(3, 3), padding=1)
    self.batch4 = BatchNorm2d(32)
    self.relu4 = ReLU()

    self.conv2 = Conv2d(32, 32, kernel_size=(3, 3), padding=1)
    self.batch1 = BatchNorm2d(32)
    self.relu1 = ReLU()

    self.pool1 = MaxPool2d(2, stride=2)

    self.conv3 = Conv2d(32, 64, kernel_size=(3, 3), padding=1)
    self.batch2 = BatchNorm2d(64)
    self.relu2 = ReLU()

    self.pool2 = MaxPool2d(2, stride=2)

    self.fc1 = Linear(64 * 56 * 56, 1024)
    self.batch3 = BatchNorm1d(1024)
    self.relu3 = ReLU()

    self.fc2 = Linear(1024, num_classes)
    self.drop = Dropout(p=0.15)

def forward(self, x):
    out = self.conv1(x)
    out = self.batch4(out)
    out = self.relu4(out)
    out = self.conv2(out)
    out = self.batch1(out)
    out = self.relu1(out)
    out = self.pool1(out)
    out = self.conv3(out)
```

```
    out = self.batch2(out)
    out = self.relu2(out)
    out = self.pool2(out)
    out = out.view(-1, 64 * 56 * 56)
    out = self.fc1(out)
    out = self.batch3(out)
    out = self.relu3(out)
    out = self.drop(out)
    out = self.fc2(out)
    return out


# Modified pre-trained ResNet 18 model for Q3
class ModifiedResnet18(Module):
    def __init__(self, num_classes):
        super().__init__()
        self.model = resnet18(pretrained=True)
        self.freeze()
        # Modify last layer for fine tuning
        self.model.fc = Linear(in_features=512,
out_features=num_classes, bias=True)

    def forward(self, x):
        return self.model(x)

    def freeze(self):
        # Freeze all layers
        for param in self.model.parameters():
            param.requires_grad = False
```

C. Code in visualize.py file (for (1) Q2 and (2) Q3 - visualizing the 4 plots as required)

```
import matplotlib.pyplot as plt
import numpy as np

def plot_learning_curve(y_arr, title, loss_or_acc=""):
    """
    Function to plot a learning curve given train_loss and val_loss
    :param y_arr: array storing training loss or training accuracy or
validation loss or validation accuracy
    :param title: string storing what type of value is stored in y_arr
to be used in plot title
    :param loss_or_acc: string ylabel for plot, values: "Loss" or
"Accuracy"
    """
    # Get number of epochs
    epochs = np.arange(0,len(y_arr))
    # Plot the curve and set title, legend, labels appropriately
    plt.plot(epochs, y_arr)
```

```
    plt.title(title+" vs Epoch curve")
    plt.xlabel("Epochs")
    plt.ylabel(loss_or_acc)
    plt.savefig(title.replace(" ", "_").replace(":", "")+'.png')
    plt.close()


def visualize(model_name, train_loss, train_accuracy, val_loss,
val_accuracy):
    """
    Function to plot
    :param model_name: string of name of model being visualized
    :param train_loss:
    :param train_accuracy:
    :param val_loss:
    :param val_accuracy:
    :return:
    """
    plot_learning_curve(train_loss, model_name+': Training loss',
'Loss')
    plot_learning_curve(train_accuracy, model_name+': Training
accuracy', 'Accuracy')
    plot_learning_curve(val_loss, model_name+': Validation loss',
'Loss')
    plot_learning_curve(val_accuracy, model_name+': Validation
accuracy', 'Accuracy')
```

D. Code in main.py file (for (1) Q2 and (2) Q3 - this is where the actual training, validation and testing takes place right from loading data, splitting it, defining model object, to visualizing results)

```
import torch.optim as optim
from torch.nn import BCEWithLogitsLoss, Linear
from tqdm.auto import tqdm
import sys
from torchvision.models import resnet18

from models import *
from pre_process import *
from visualize_plots import *


def get_y_pred(output):
    y_pred = []
    for out in output:
        if out > 0:
            y_pred.append(1)
        else:
            y_pred.append(0)
```

```
    y_pred = torch.tensor(y_pred).to(device)
    return y_pred


def training(model, optimizer, criterion, train_dl):
    model.train()
    total = 0
    running_loss = 0
    correct = 0
    for image, label in train_dl:
        image = image.to(device)
        label = label.to(device)

        output = model(image)
        y_pred = get_y_pred(output)
        loss = criterion(output.squeeze(), label)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total += label.size(0)
        correct += sum(y_pred == label)
        running_loss += loss.item()

    epoch_train_accuracy = 100 * correct / total
    epoch_train_loss = running_loss / len(train_dl)

    return epoch_train_accuracy, epoch_train_loss


def validation(model, criterion, val_dl):
    model.eval()
    total = 0
    running_loss = 0
    correct = 0
    with torch.no_grad():
        for image, label in val_dl:
            image = image.to(device)
            label = label.to(device)

            val_output = model(image)
            y_pred = get_y_pred(val_output)
            loss = criterion(val_output.squeeze(), label)

            total += label.size(0)
            correct += sum(y_pred == label)
            running_loss += loss.item()

            epoch_val_accuracy = 100 * correct / total
            epoch_val_loss = running_loss / len(val_dl)
```

```
    return epoch_val_accuracy, epoch_val_loss


def testing(model, criterion, test_dl):
    model.eval()
    acc = []
    losses = []
    total = 0
    running_loss = 0
    correct = 0
    with torch.no_grad():
        for image, label in test_dl:
            image = image.to(device)
            label = label.to(device)

            test_output = model(image)
            y_pred = get_y_pred(test_output)
            loss = criterion(test_output.squeeze(), label)

            total += label.size(0)
            correct += sum(y_pred == label)
            running_loss += loss.item()

            acc.append(100 * correct / total)
            losses.append(loss.item())

    return acc, losses


def get_acc_loss_and_plot(model_name, model, optimizer, batch_size):
    # Get train, val, test data loaders
    train_dl, val_dl, test_dl = data_loaders(batch_size)

    # Define loss criterion: Binary Cross Entropy Loss with Logits
    criterion = BCEWithLogitsLoss()

    train_loss, train_accuracy, val_loss, val_accuracy = [], [], [], []
    for epoch in tqdm(range(num_epochs)):
        # Train model
        acc, loss = training(model, optimizer, criterion, train_dl)
        train_accuracy.append(acc)
        train_loss.append(loss)
        # if epoch % 10 == 0:
        print('Epoch : {}, train accuracy : {}, train loss :
{}'.format(epoch + 1, acc, loss))

        # Validate results
        acc, loss = validation(model, criterion, val_dl)
        val_accuracy.append(acc)
        val_loss.append(loss)
        # if epoch % 10 == 0:
```

```
        print('Epoch : {}, val_accuracy : {}, val_loss :
{}'.format(epoch + 1, acc, loss))

    # Test model on unseen data
    acc, loss = testing(model, criterion, test_dl)
    test_acc = acc
    test_loss = loss
    print('Average test_accuracy : {}, test_loss : {}'.format(
        sum(test_acc) / len(test_acc), sum(test_loss) /
len(test_loss)))

    # Plot and save training and validation loss and accuracy
    visualize(model_name, train_loss, train_accuracy, val_loss,
val_accuracy)


########################################################################

# Check if GPU is available
device = 'cuda' if torch.cuda.is_available() else 'cpu'
torch.manual_seed(10)
if device == 'cuda':
    torch.cuda.manual_seed_all(10)
# Number of classes in our dataset
num_classes = 1

# Definition of hyperparameters for CNN model
batch_size = 16
num_epochs = 15
lr = 0.002

# Store all print statements in a txt file
# sys.stdout = open("all_print_stmt.txt", "w")

print("CNN will be trained")
# Get training, val, testing loss and accuracy and their plots for both
models
# Create CNN model
cnn = CNNModel(num_classes)
cnn.to(device)
# Define optimizer for cnn model
optimizer = optim.Adam(cnn.parameters(), lr=lr)
# Get loss and accuracy plots for train, val, test sets
get_acc_loss_and_plot("cnn", cnn, optimizer, batch_size=batch_size)

########################################################################

print("Modified ResNet-18 will be trained")

# Definition of hyperparameters for CNN model
batch_size = 8
```

```
num_epochs = 10
lr = 0.0001

modified_resnet18 = ModifiedResnet18(num_classes)
modified_resnet18.to(device)

# Define optimizer
optimizer = optim.Adam(modified_resnet18.parameters(), lr=lr)
# Get loss and accuracy plots for train, val, test sets
get_acc_loss_and_plot("resnet", modified_resnet18, optimizer,
batch_size=batch_size)
# sys.stdout.close()
```