CSCI 5922 -Neural Networks and Deep Learning
Lab - 4 Solutions

Gowri Shankar Raju Kurapati
Student ID 110568555

October 27, 2022

1 Validation

1.1 Methods

Our goal in this assignment was to create models that predict whether or not a question is answerable based on the image provided with the question. We used the visual question answering (**VizWiz -VQA**) dataset from VizWiz for this purpose. There were **20,523 training** image and question pairs and **4319 validation** image and question pairs in the dataset. The main VQA challenge consisted of two tasks which include generating an appropriate answer to a question and classifying whether or not the visual question could be answered. However, for our task, we concentrate on predicting whether or not the visual question attached to the image is answerable. The images in the dataset were captured by blind people using a recorded spoken question. Ten crowd-sourced responses to each visual question are also included. The two output classes are **"0" if the question cannot be answered** using the image and **"1" if it can be**.
To train our neural networks, we must first establish fundamental methods that will allow us to exploit important information from both the source images and the input question. This was accomplished through the use of multimodal feature extraction from the dataset having different modalities, in our case which are image and language. So we extract image features and text features via questions related to that image. For image features, we consider large pre-trained image classification networks for visual features which are represented as pixel intensities or outputs of feature extraction layers. The text features for the question are represented as a series of word vector embeddings. Given the dual nature of the modalities included in the given dataset, we pre-process the image and questions to independent feature vectors before **concatenating them as a single feature vector** and putting them through the neural network learning phases of training and validation.

We use three different methods in our experiment for the classification task of the question being answerable from the corresponding image. The three models use the same **text features where BERT-BASE tokens** of the text (padded to the maximum length of input for BERT-BASE which is 512) are passed through the BERT-BASE itself, where t**he embedding (768-dimensional vector) corresponding output to <CLS> token is considered, which is assumed to be a "sentence embedding"**. The three models differ only in the image feature vector used. We considered **resNet50, vgg16, and vgg19 for generating features for the image**. We use the penultimate layer (avg pool layer) outputs of the model as the image features. The penultimate layer has dimensions of 2048, 4096 (flatten) & 4096(flatten) for resNet50, vgg16, and vgg19 respectively. Since the images in the dataset are all of different sizes and since the pre-trained image models (resNet50, vgg16, and vgg19

are used for this experiment) expect to have images of a certain size, the source images were initially resized to (224, 224,3), where 3 denotes for 3 RGB channels. As stated previously, we concatenate the considered sentence embedding & image feature embedding to form the joint embedding for image and question and use this joint embedding for classification.
The **joint representation embedding is passed through an MLP layer** which has three dense layers with 1024 and 512 nodes in the first and second hidden layers. The input layer dimension is 768 + 2048 , 768 + 4096, 768 + 4096 for the models using restNet50, vgg16 and vgg19 respectively. The output layer is 1 dimension followed by a softmax layer to give the confidence score. Note that the **BERT Layer and Image Model used are frozen** and the only **trainable parameters are the model parameters of the dense layers of MLP**. We have **dropout at the rate of 0.2** as a regularizer for the MLP.
The model is trained with **Adam optimizer** with a **learning rate of 0.001** and **batch size of 32** for **10 epochs**. All the experiments are run on CUDA-enabled GPU which is **NVIDIA GeForce GTX 1050Ti**. Each Model has been trained on 1500, 2000, and 2500 examples, and **1000 samples are used for validation**. We evaluated all three models (*which differ only in the model used for image feature extraction*) against the average precision on the validation dataset to get the best-performing model.

1.2 Results

Models Summary

Model	Text Features	Image Features
Model- v1	BERT-BASE	resNet50
Model- v2	BERT-BASE	vgg16
Model- v3	BERT-BASE	vgg19

Table 1: Models Summary. Pre-trained models for the extraction of Text Features and Image Features are specified for each model used in the experiment. The MLP used for classification is the same across all the models specified with the same hyper-parameters in the above section.

Average Precision Scores (out of 100)

Model/ No of Training Samples	v1	v2	v3
1500	52.91	58.87	62.98
2000	53.68	60.79	64.06
2500	**56.76**	**62.43**	**67.68**

Table 2: The Average Precision scores against the validation dataset v1, v2, and v3 models when trained on 1500, 2000 & 2500 samples. Note that we have considered only 1000 images for validation.

Validation Accuracies

Model/ No of Training Samples	v1	v2	v3
1500	52.06	57.87	59.35
2000	54.37	58.63	60.06
2500	**56.66**	**61.82**	**62.01**

Table 3: The Validation Accuracy scores for the v1, v2, and v3 models when trained on 1500, 2000 & 2500 samples. Note that we have considered only 1000 images for validation.

1.3 Analysis

The Average Precision score is the statistic used to compare models. Average Precision presents the precision-recall curve and computes the weighted mean of precision with the increase in recall from the previous threshold used as the weight which is obtained after training the neural network model on the provided dataset. Because we are attempting to establish the extent to which our questions in the context of the image may be answered, we are utilizing average precision scores. In Classification, values are often categorized as 0 or 1 depending on a threshold point for calculating Precision. This defeats our goal of establishing the model's confidence in predicting whether or not the questions can be answered. Nevertheless, we still compare the accuracy trends as well.

When considering the Average Precision scores from Table 2, we see that the v3 model trained on 2500 samples gives us the highest score for average precision on the validation set. This can be explained by the fact that the v3 model uses vgg19 which is a deep network compared to vgg16 which is used in v2. Also, the image feature vector used for v2/ v3 models is a 4096-dimensional vector compared to a 2048-dimensional vector used in v1. Since a high dimensional vector can capture much more information, especially for rich data such as images, v3 is supposed to be better at capturing image information which reflects in a better joint representation of text and image and finally in predicting the confidence scores.

Also, we look at the average precision scores for the three models trained on the same number of training samples, there is an increasing trend in scores from restNet50 to vgg16 to vgg19. Though all three image models are of comparable size, the difference between v1 and v2/v3 stems from the dimension of the image feature vector used as explained above. Note that, this trend is seen across the category of the number of training samples used.

The average precision scores increases as the number of training samples increases. This can be inferred when looked the columns of Table 2. For every model, we can clearly observe an increasing score trend as the number of samples it is trained on increases. This is expected as the model is able to learn more information from large training samples and thereby is able to perform better on the same validation dataset used across all the models trained on the different number of samples.
Due to time and resource constraints, the experiment was limited to just 2500 samples. The hypothesis is that the validation accuracy increases as the number of training samples increases. Since only around 12.5 % of the total training samples are used for the experiment, the average precision score is around 68 %. But this can be expected to increase by a significant margin when the percentage of training samples used for training increases to 25% and above.

The same similar trends can be observed in terms of validation accuracies (Table 3) as well. The same hypothesis and explanations hold for this validation accuracy metric as well.

2 Test Dataset

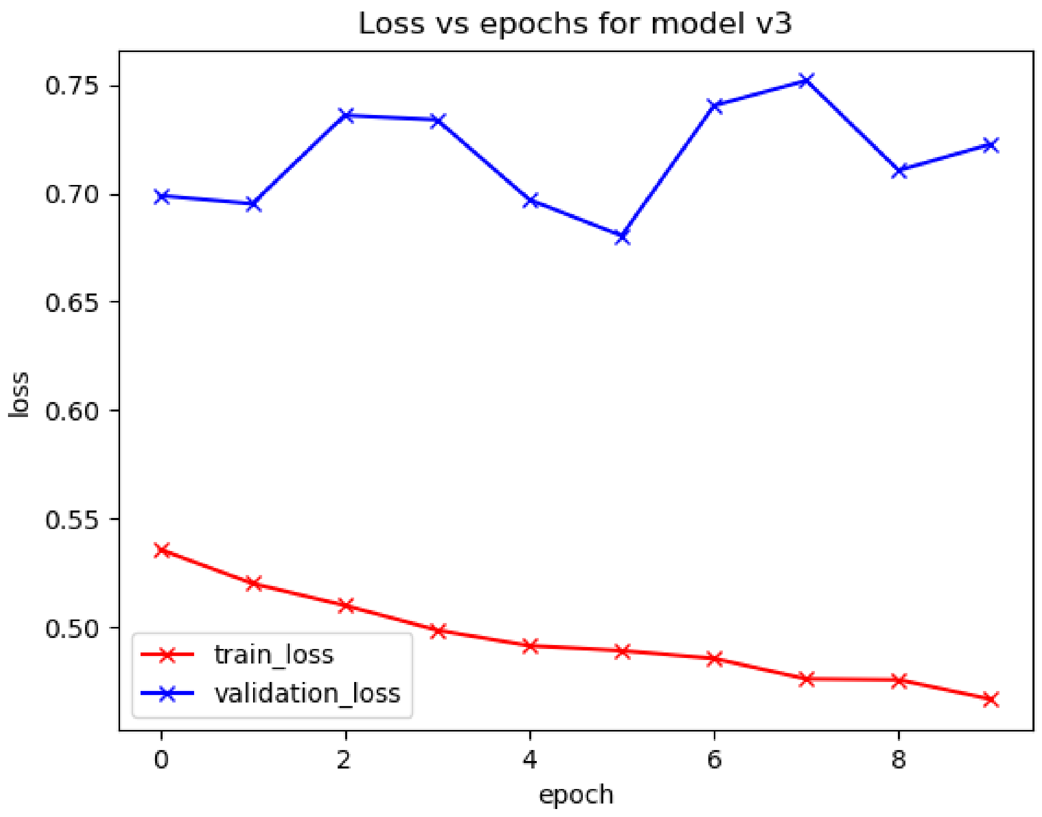
2.1 Methods

We use the test dataset from the same Viz-Wiz VQA that we used for the training and validation phases. The provided dataset had 8000 test images and question pairings but the target labels for answerable are not given. We employ the same setup approaches to leverage important information from both the source image and the question to train our neural network models. This was accomplished through the use of multimodal feature representation. In the same way that we obtained the training and validation datasets, we preprocess the images and questions to individual feature vectors. Our test dataset was created by concatenating these two sequences. When it comes to the size of the samples in the test dataset, we do not use all of the samples in this annotation file. We used only 1000 samples for testing purposes.

From Table 2, we see that the best-performing model is v3 trained on 2500 examples. Due to time constraints, the same model is exactly used. No additional data is trained on v3. Since the test dataset present in our experiments does not have true label predictions, we use the predictions from the best neural network model which is v3 trained on 2500 training samples, and store them in a 'results.csv' file with no headers.

2.2 Results

Figure 1: Loss Curve for v3 trained on 2500 Training Samples



The loss curve for model v3 when being trained on 2500 samples for 10 epochs is plotted above.

2.3 Analysis

As we can observe from Figure 1, the training loss gradually decreases as the number of epochs increases. But when the validation loss is observed, it shows a different trend, it decreases to its lowest at 5 the epoch and then starts to increase intermittently again. Maybe, the model is being overfitted from the 5 epoch and that is why the model parameters for the highest validation score are considered for scoring the test dataset.

3 CODE

```
import torch
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
torch.__version__
```

```
torch.cuda.empty_cache()
torch.cuda.is_available()
```

```
import torchvision
import matplotlib.pyplot as plt
import json
import cv2
import numpy as np
from copy import deepcopy
from skimage import io
import matplotlib.pyplot as plt
import time
```

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

```
#1 Part 1: Set up the dataset
img_dir = "https://vizwiz.cs.colorado.edu//VizWiz_visualization_img/"
split = "train"
annotation_file = "https://vizwiz.cs.colorado.edu/VizWiz_final/vqa_data/Annotations/%s.jso
print(annotation_file)
```

```
#1 Part 1: Set up the dataset
img_dir = "https://vizwiz.cs.colorado.edu//VizWiz_visualization_img/"
split = "val"
annotation_file_val = "https://vizwiz.cs.colorado.edu/VizWiz_final/vqa_data/Annotations/%s
print(annotation_file_val)
```

```
#1 Part 1: Set up the dataset
img_dir = "https://vizwiz.cs.colorado.edu//VizWiz_visualization_img/"
split = "test"
annotation_file_test = "https://vizwiz.cs.colorado.edu/VizWiz_final/vqa_data/Annotations/%
print(annotation_file_test)
```

```
##3. Read the file to extract each dataset example with label
import requests
import numpy as np
split_data = requests.get(annotation_file, allow_redirects=True)
data = split_data.json()
```

```
split_data_test = requests.get(annotation_file_test, allow_redirects=True)
data_test = split_data_test.json()
```

```
df_test = pd.DataFrame(data_test)
```

```
df_test.head(5)
```

```
split_data_val = requests.get(annotation_file_val, allow_redirects=True)
data_val = split_data_val.json()
```

```
import pandas as pd
df = pd.DataFrame(data)
df_val = pd.DataFrame(data_val)
```

```
df_val.head(5)
```

```
df['question'].map(lambda x: len(x)).min()
```

```
df.head()
df["image"][0]
```

```
import torch.nn as nn
import torchvision.models as models
import torchvision.transforms as transforms
from torch.autograd import Variable
from PIL import Image
```

```
# Load the pretrained model
model = models.vgg19(pretrained=True)
model.fc = nn.Identity()
```

```
# Load the pretrained model
model = models.resnet50(pretrained=True)
model.fc = nn.Identity()
```

```
# Load the pretrained model
model = models.vgg16(pretrained=True)
model.fc = nn.Identity()
```

```
scaler = transforms.Resize((224, 224))
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
to_tensor = transforms.ToTensor()
```

```
import urllib.request as urllib
def getImgVector(image_name):
    # 1. Load the image with Pillow library
    img = Image.open(image_name)
    # 2. Create a PyTorch Variable with the transformed image
    t_img = Variable(normalize(to_tensor(scaler(img))).unsqueeze(0))
    model.eval()
    with torch.no_grad():
      vf = model(t_img)

    return (img,vf.reshape([-1,]))
```

```
img_val = 'val/val/'
def getImgVectorVal(image_name):
    # 1. Load the image with Pillow library
    img = Image.open(img_val + image_name)
    # 2. Create a PyTorch Variable with the transformed image
    t_img = Variable(normalize(to_tensor(scaler(img))).unsqueeze(0))
    model.eval()
    with torch.no_grad():
      vf = model(t_img)

    return (img,vf.reshape([-1,]))
```

```
a, b = getImgVector(img_dir+"VizWiz_train_00000143.jpg")
```

```
b.shape
```

```
from torch.utils.data import TensorDataset, DataLoader, Dataset
class VizWizDataset(Dataset):

    def __init__(self, annotation, trainVal, num):

        print(annotation)
        self.s = requests.get(annotation, allow_redirects=True)
        self.d = self.s.json()
        self.d1 = self.d[0:num]
        self.df = pd.DataFrame(self.d1)
        self.img_dir = ""+ trainVal + "/" + trainVal + "/"
        self.trainVal = trainVal
        #print(self.img_dir)
        #print(self.d[0])
        ##self.landmarks_frame = pd.read_csv(csv_file)
        #self.transform = transform

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()
        img , imgfv  = getImgVector(self.img_dir + self.df["image"][idx])
          # Detectron expects BGR images
        #img_bgr = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        # Image BGR
        img_bgr = torch.tensor([1])
        question = self.df["question"][idx]
        if self.trainVal != "test" :
            label = self.df["answerable"][idx]
        else:
            #Always 2
            label = 2
        return (img, imgfv, img_bgr, question, label)

annotation_file_val


dataset = VizWizDataset(annotation_file, "train", 2500)
dataset_val= VizWizDataset(annotation_file_val, "val", 1000)


dataset_test= VizWizDataset(annotation_file_test, "test", 1000)


# Display text and label.
print('\nFirst iteration of data set: ', next(iter(dataset)), '\n')
print(next(iter(dataset)))


# Display text and label.
print('\nFirst iteration of val set: ', next(iter(dataset_val)), '\n')
print(next(iter(dataset_val)))


!pip3 install transformers
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')


from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence

#Images are of various shape
max_len = df['question'].map(lambda x: len(x)).max()
def collate_batch(batch):
    img_list, imgfv_list , img_bgr_list, question_list, label_list = [], [],[], [],[]
    for (img, imgfv, img_bgr, question, label) in batch:
        label_list.append(label)
        encoded = encoded_dict = tokenizer.encode_plus(
                        question,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = max_len,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )
        tok_ques = encoded_dict["input_ids"].reshape([-1,])
        question_list.append(tok_ques)
        #print(img.shape)
        img_list.append(img)
        img_bgr_list.append(img_bgr)
        imgfv_list.append(imgfv)

    return  img_list, torch.stack(imgfv_list), img_bgr_list, torch.stack(question_list), t

context = "I am the king hell ?"
encoded_dict = tokenizer.encode_plus(
                        context,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = 10,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                   )

print(encoded_dict)
print(encoded_dict["input_ids"].reshape([-1,]))
#print(context_tokens)
print(tokenizer.decode(encoded_dict['input_ids'].reshape([-1,])))


BATCH_SIZE = 32
train_dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, collate_fn=collate_batch)
valid_dataloader = DataLoader(dataset_val, batch_size=BATCH_SIZE, collate_fn=collate_batch


BATCH_SIZE = 32
test_dataloader = DataLoader(dataset_test, batch_size=BATCH_SIZE, collate_fn=collate_batch


dataiter = iter(test_dataloader)
print(dataiter.next())


from tqdm import tqdm
import torch.optim as optim
import torch.nn.functional as F


import torch.nn as nn

from transformers import BertTokenizer, BertModel

bert = BertModel.from_pretrained('bert-base-uncased').to(device=device)


@torch.no_grad()
def accuracy(outputs, labels):
    #_, preds = torch.round(torch.sigmoid(outputs))
     #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(outputs))
    correct = (rounded_preds == labels).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return torch.tensor(acc)
```

```
@torch.no_grad()
def evaluate(model, data_loader, testOrValidation):
    model.eval()
    outputs = [model.testOrValidation_step(batch, testOrValidation) for batch in data_load
    return model.at_testOrValidation_epoch_end(outputs,testOrValidation)

class Utils(nn.Module):
    def training_step(self, batch):
        images, imagesfv, images_bgr, questions, true_labels = batch
        questions = questions.to(device=device)
        imagesfv = imagesfv.to(device = device)
        true_labels = true_labels.float().to(device=device)
        pred_labels = self(questions,imagesfv,images_bgr).to(device=device)
        #print(true_labels)
        #print(pred_labels)
        loss = criterion(pred_labels, true_labels).to(device=device)
        accur = accuracy(pred_labels,true_labels)
        return loss,accur

    def testOrValidation_step(self, batch, testOrValidation):
        images, imagesfv, images_bgr, questions, true_labels = batch
        questions = questions.to(device=device)
        imagesfv = imagesfv.to(device = device)
        true_labels = true_labels.float().to(device=device)
        pred_labels = self(questions,imagesfv,images_bgr)
        loss = criterion(pred_labels, true_labels)
        accur= accuracy(pred_labels, true_labels)
        return {testOrValidation +'_loss': loss.detach(), testOrValidation+'_accuracy': ac

    def at_testOrValidation_epoch_end(self, outputs, testOrValidation):
        batch_losses = [x[testOrValidation+'_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        batch_accuracies = [x[testOrValidation+'_accuracy'] for x in outputs]
        epoch_accuracy = torch.stack(batch_accuracies).mean()
        return {testOrValidation +'_loss': epoch_loss.item(), testOrValidation+'_accuracy'

    def at_epoch_end(self, epoch, result):
        print("Epoch :",epoch + 1)
        print(f'Train Accuracy:{result["train_accuracy"]*100:.2f}% Validation Accuracy:{re
        print(f'Train Loss:{result["train_loss"]:.4f} Validation Loss:{result["validation_

def linear_block(in_f, out_f, enable_dropOut, activation_function):
    if(enable_dropOut):
        return nn.Sequential(
             nn.Linear(in_f, out_f),
             nn.Dropout(0.2),
             activation_function
            )
    else:
        return nn.Sequential(
             nn.Linear(in_f, out_f),
             activation_function
            )
```

```
class VisL(Utils):
    def __init__(self,
                 bert,
                 enable_dropOut):

        super().__init__()

        self.bert = bert

        embedding_dim = bert.config.to_dict()['hidden_size']

        self.Linear1 = linear_block(1768, 1024,enable_dropOut,nn.ReLU().to(device = device
        self.Linear2 = linear_block(1024, 512, enable_dropOut,nn.ReLU().to(device = device
        self.Linear3 = nn.Linear(512, 1)

    def forward(self, text, imvf, img_bgr):

        #text = [batch size, sent len]
        self.bert.eval()
        with torch.no_grad():
            embedded = self.bert(text)[0]

        #embedded = [batch size, sent len, emb dim]
        append_dim = imvf.shape[1]
        batch_size = embedded.shape[0]
        emb_dim = embedded.shape[2]
        embedded= embedded[:, 0, :]
        emdedded = torch.empty((batch_size,emb_dim+append_dim))
        x = torch.concat((embedded,imvf),1).to(device = device)
        #print(x.shape)
        x = self.Linear1(x.to(device = device))
        x = self.Linear2(x.to(device = device))
        x = self.Linear3(x.to(device = device))
        return x.reshape([-1,])

model_main = VisL(bert, True).to(device = device)


with torch.no_grad():
    model_main.eval()
    a = model_main.forward(torch.tensor([[ 101, 1045, 2572, 1996, 2332, 3109, 1029,  102,
    print(a)


torch.cuda.is_available()


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model_main):,} trainable parameters')


for name, param in model_main.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model_main):,} trainable parameters')


for name, param in model_main.named_parameters():
    if param.requires_grad:
        print(name)


def plot_accuracies(history, model_name):
    Validation_accuracies = [x['validation_accuracy'] for x in history]
    Training_Accuracies = [x['train_accuracy'] for x in history]
    plt.figure()
    plt.plot(Training_Accuracies, '-rx')
    plt.plot(Validation_accuracies, '-bx')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.legend(['Training', 'Validation'])
    plt.title('Acc vs epochs for model ' + model_name);
    plt.savefig(f'{model_name}_acc.png')
    plt.close()

def plot_losses(history,model_name):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['validation_loss'] for x in history]
    plt.figure()
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs epochs '+ model_name);
    plt.savefig(f'{model_name}_loss.png')
    plt.close()


from tqdm import tqdm
def trainNetwork(model, train_loader, validation_loader,epochs,learning_rate):
    t1 = time.time()
    best_valid = None
    history = []
    optimizer = torch.optim.Adam(model.parameters(), learning_rate,weight_decay=0.0005)
    for epoch in range(epochs):
        # Training
        model.train()
        train_losses = []
        train_accuracy = []
        for batch in tqdm(train_loader):
            loss,accu = model.training_step(batch)
            train_losses.append(loss)
            train_accuracy.append(accu)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation
        result = evaluate(model, validation_loader,"validation")
        result['train_loss'] = torch.stack(train_losses).mean().item()
        result['train_accuracy'] = torch.stack(train_accuracy).mean().item()
        model.at_epoch_end(epoch, result)
        if(best_valid == None or best_valid<result['validation_accuracy']):
            best_valid=result['validation_accuracy']
            torch.save(model.state_dict(), 'Lab4_P1'+'.pth')
        history.append(result)
    t2 = time.time()
    timeToTrain = t2 - t1
    #print(f'TIme to train {model.name} : {timeToTrain}')
    return (history, timeToTrain)
```

```
learning_rate = 0.001
batch_size = 64
epochs = 10
criterion = nn.BCEWithLogitsLoss().to(device=device)
(listAccuracies, trainingTime) = trainNetwork(model_main, train_dataloader, valid_dataload
# trainingTimes.append(trainingTime)
plot_accuracies(listAccuracies, "v3")
plot_losses(listAccuracies, "v3")
# result = evaluate(model, testLoader, "test")
# testAccuracies.append(result)
```

```
!pip install torchmetrics
```

```
from torchmetrics.classification import BinaryAveragePrecision
preds = torch.tensor([0, 0.5, 0.7, 0.8])
target = torch.tensor([0, 1, 1, 0])
metric = BinaryAveragePrecision(thresholds=None)
print(metric(preds, target))

metric = BinaryAveragePrecision(thresholds=5)
print(metric(preds, target))
```

```
modelMain1 = VisL(bert, True)
modelMain1.load_state_dict(torch.load('Lab4_P1'+'.pth'))
modelMain1.eval()
```

```
modelMain1.to(device = device)
```

```
result = evaluateValidation(modelMain1, valid_dataloader,"validation")
```

```
@torch.no_grad()
def evaluateValidation(model, data_loader, testOrValidation):
    model.eval()
    true_labels = torch.tensor([]).to(device=device)
    pred_labels = torch.tensor([]).to(device=device)
    for batch in data_loader:
        (outputs, tl) = validation_step(model,batch,"validation")
        pred_labels = torch.cat((pred_labels,outputs)).to(device=device)
        true_labels = torch.cat((true_labels,tl)).to(device=device)
    return (pred_labels, true_labels)
```

```
def validation_step(model, batch, testOrValidation):
    images, imagesfv, images_bgr, questions, true_labels = batch
    questions = questions.to(device=device)
    imagesfv = imagesfv.to(device = device)
    true_labels = true_labels.float().to(device=device)
    pred_labels = model(questions,imagesfv,images_bgr)
    outputs = torch.sigmoid(pred_labels)
    return (outputs, true_labels)
```

```
(pl,tb) = result
```

```
pl.shape
```

```
tb.shape
```

```
metric = BinaryAveragePrecision(thresholds=None)
print(metric(pl, tb))
```

```
@torch.no_grad()
def evaluateTest(model, data_loader, testOrValidation):
    model.eval()
    pred_labels = torch.tensor([]).to(device=device)
    for batch in data_loader:
        (outputs, tl) = validation_step(model,batch,"test")
        pred_labels = torch.cat((pred_labels,outputs)).to(device=device)
    return pred_labels
```

```
def validation_step(model, batch, testOrValidation):
    images, imagesfv, images_bgr, questions, true_labels = batch
    questions = questions.to(device=device)
    imagesfv = imagesfv.to(device = device)
    true_labels = true_labels.float().to(device=device)
    pred_labels = model(questions,imagesfv,images_bgr)
    outputs = torch.sigmoid(pred_labels)
    return (outputs, true_labels)
```

```
csvValues = evaluateTest(modelMain1, test_dataloader, "test")
```

```
df = pd.DataFrame(csvValues.cpu())
df.to_csv("results.csv", header = None, index = None)
```