
CPSC 6200: COMPUTER SECURITY PRINCIPLES

Homework 6

Saurabh Sharma
11/21/2023

LAB1

1) Lab Environment Setup

I ran the code on the provided google collab as it already has all the prerequisites installed. First we clone the GitHub repository in the output below and this imports everything into google collab.

```
✓ 2s !git clone https://github.com/nishantvishwamitra/CyberbullyingLab1.git

Cloning into 'CyberbullyingLab1'...
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 25 (delta 7), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (25/25), 3.32 MiB | 6.97 MiB/s, done.
Resolving deltas: 100% (7/7), done.

Next, we install transformers which offer us some tools we can use

✓ 5s [4] !pip install transformers

Requirement already satisfied: transformers in /usr/local/lib/python3.
Requirement already satisfied: filelock in /usr/local/lib/python3.
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in /us
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/pytho
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/p
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/pytho
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib
Requirement already satisfied: requests in /usr/local/lib/python3
```

Now we install all the dependencies such as pandas, numpy, torch libraries.

```
✓ 6s [5] import pandas as pd
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertModel
from transformers import AdamW

import matplotlib.pyplot as plt

from tqdm.notebook import tqdm
```

1.1) Dataset selection

After selecting the dataset, we split it into training, testing and validation sets.

```
# let's divide the dataset into non-cyberbullying and cyberbullying samples
o_class = main_df.loc[main_df.label == 0, :]
l_class = main_df.loc[main_df.label == 1, :]

# let's create train, val and test splits
train_val = main_df.iloc[:int(main_df.shape[0] * .80)]
test_df = main_df.iloc[int(main_df.shape[0] * .80):]
train_df = train_val.iloc[:int(train_val.shape[0] * .80)]
val_df = train_val.iloc[int(train_val.shape[0] * .80):]

#print(train.shape, val.shape, test.shape)
print('\nTraining set:\n', train_df.label.value_counts())
print('\nValidation set:\n', val_df.label.value_counts())
print('\nTest set:\n', test_df.label.value_counts())
```

Training set:

0	7871
1	550

Name: label, dtype: int64

Validation set:

0	1955
1	151

Name: label, dtype: int64

Test set:

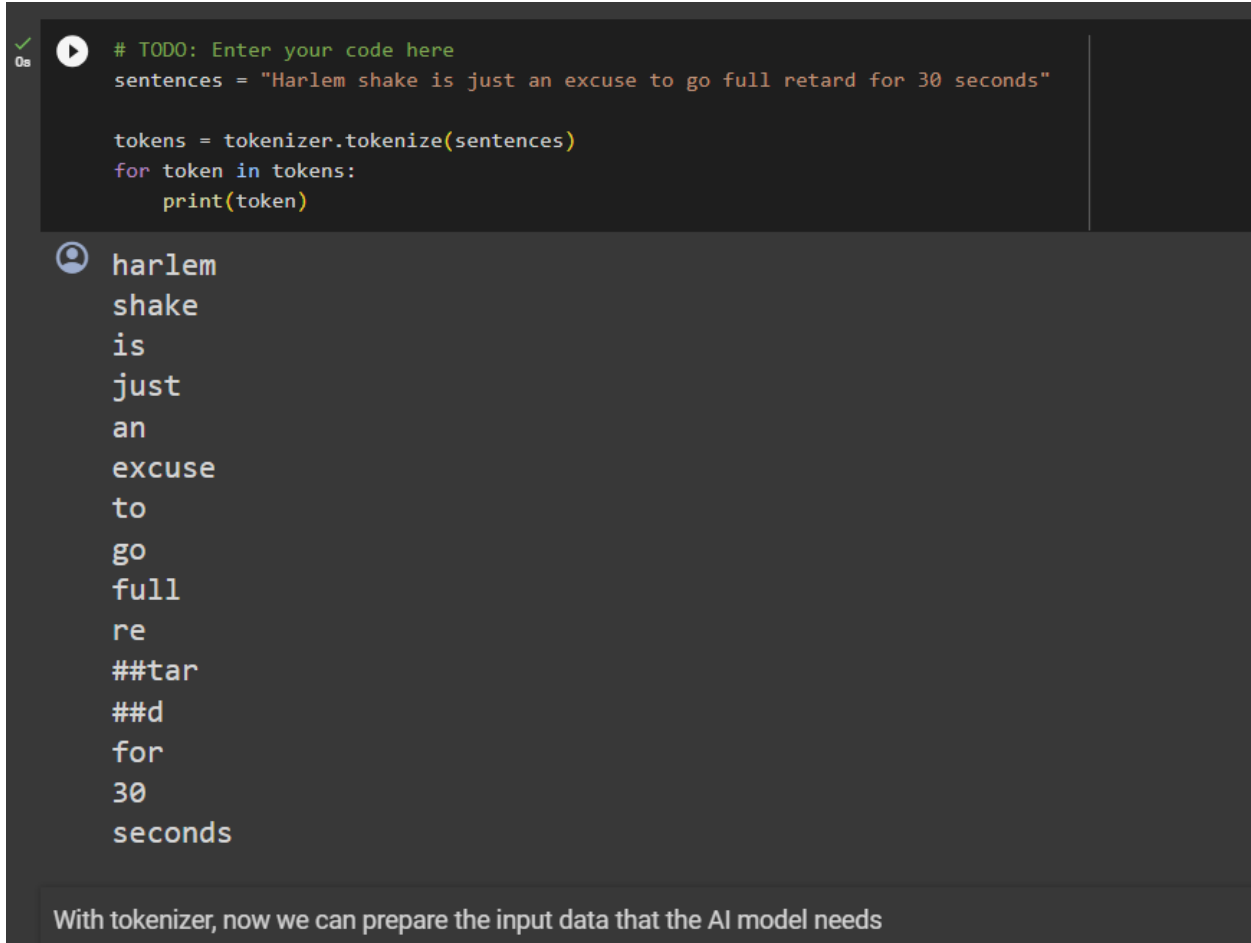
0	2465
1	167

Name: label, dtype: int64

We can see that the training set has 7851+550 data points, testing has 2465+167 data points while validation has 1955+151 data points. This dataset split is very skewed so we will preprocess the data to ensure the number of cyberbullying and noncyberbullying texts are equal.

1.2) Preprocessing data

To process the text we input the function in `tokenizer.tokenize()` function and it automatically processes the sentence and generates tokens as below.



The screenshot shows a code editor with a dark background. At the top, there is a green checkmark and a play button icon. Below it, the code is as follows:

```
# TODO: Enter your code here
sentences = "Harlem shake is just an excuse to go full retard for 30 seconds"

tokens = tokenizer.tokenize(sentences)
for token in tokens:
    print(token)
```

Below the code, the output is displayed as a list of tokens, each on a new line:

```
harlem
shake
is
just
an
excuse
to
go
full
re
##tar
##d
for
30
seconds
```

At the bottom of the editor, there is a text box with the following text:

With tokenizer, now we can prepare the input data that the AI model needs

1.3) Model Training

In the output below we can see that we achieved the final accuracy of 97.77% before hyperparameter tuning. We trained the model over 5 epochs.

```
✓ [20] 100% 35/35 [00:25<00:00, 1.94it/s]
| Epoch: 01 | Train Loss: 0.538 | Train Acc: 70.21% | Val. Loss: 0.483 | Val. Acc: 83.79% |
100% 35/35 [00:20<00:00, 2.04it/s]
| Epoch: 02 | Train Loss: 0.324 | Train Acc: 88.24% | Val. Loss: 0.397 | Val. Acc: 84.20% |
100% 35/35 [00:20<00:00, 2.06it/s]
| Epoch: 03 | Train Loss: 0.192 | Train Acc: 93.12% | Val. Loss: 0.446 | Val. Acc: 86.47% |
100% 35/35 [00:20<00:00, 2.04it/s]
| Epoch: 04 | Train Loss: 0.110 | Train Acc: 96.37% | Val. Loss: 0.676 | Val. Acc: 84.20% |
100% 35/35 [00:20<00:00, 2.03it/s]
| Epoch: 05 | Train Loss: 0.079 | Train Acc: 97.77% | Val. Loss: 0.513 | Val. Acc: 83.97% |

Task 2: After training, what is the training accuracy that your model achieves?

0s [24] #TODO: add your code below to print the final training accuracy out
print(f'|Train Acc: {train_acc*100:.2f}% ||')

|Train Acc: 97.77% ||
```

1.4) Model evaluation

Now we run the model that we have just trained against the test set so that we get a better understanding of the performance of our model in real world.

```
Task 3: Let's review the previous code then finish the next code cell

✓ [25] #TODO: complete the code below
2s test_loss, test_acc = evaluate(model, test_data_loader, criterion)
print(f'| Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}% |')

| Test Loss: 0.600 | Test Acc: 81.57% |
```

We can see that the test accuracy is less than the training accuracy which is normal. We achieved a test accuracy of 81.57% which is decent.

1.5) Deploy and run Custom samples

We ran three custom samples and ran them through our model and we can see that the model classified these texts correctly and with pretty high accuracy. First two texts are toxic and are correctly classified as cyberbullying while the third text is really nice and the model says its noncyberbullying text with high confidence as well. This indicates our model is running well. We can test it further and it might not work correctly, to solve we can improve it further by hyperparameter tuning.

```

#TODO: complete the code below
text1 = 'you guys are a bunch of losers, fuck you'
ret1 = predict_cb(text1)
print("Sample prediction: ", ret1[2], f'Confidence: {ret1[0].item() * 100:.2f}%')

print("=====")

text2 = ' I\'m never going to see your little pathetic self again'
ret2 = predict_cb(text2)
print("Sample prediction: ", ret2[2], f'Confidence: {ret2[0].item() * 100:.2f}%')

print("=====")

text3 = 'She looks really nice today!'
ret3 = predict_cb(text3)
print("Sample prediction: ", ret3[2], f'Confidence: {ret3[0].item() * 100:.2f}%')

```

Sample prediction: Cyberbullying detected. Confidence: 99.63%
 =====
 Sample prediction: Cyberbullying detected. Confidence: 99.54%
 =====
 Sample prediction: Cyberbullying not detected. Confidence: 99.75%

We can see our model failing below.

```

#@title Input your own sentence to check the prediction.
My_Sentence = "we are in the lab1, and you are not stupid\\"" #@param {type:"string"}

ret = predict_cb(My_Sentence)
print("====The Model Prediciton is====")
print("The input sentence is: ", ret[2], f'Confidence: {ret[0].item() * 100:.2f}%')

```

=====The Model Prediciton is=====
 The input sentence is: Cyberbullying detected. Confidence: 99.34%

1.20) Hyperparameter tuning

Now in order to improve the performance of our model we will try to adjust the hyperparameters of our model so that it learns and then performs better.

1.21) Number of epochs

Number of epochs is the number of complete iterations of the dataset. By increasing the number of epochs we train our model over the training model again and again. This minimizes the loss up to a point but if we overdo it the model can overfit the dataset and it won't perform well in real world.

Let us try to find the optimal value.

Training for 2 epochs

100%  35/35 [00:20<00:00, 2.04it/s]

| Epoch: 01 | Train Loss: 0.556 | Train Acc: 70.03% | Val. Loss: 0.434 | Val. Acc: 84.11% |

100%  35/35 [00:20<00:00, 2.06it/s]

| Epoch: 02 | Train Loss: 0.286 | Train Acc: 89.73% | Val. Loss: 0.422 | Val. Acc: 83.17% |

Test Loss: 0.423 | Test Acc: 84.42% |

training complete!

Training for 4 epochs

100%  35/35 [00:20<00:00, 2.06it/s]

| Epoch: 01 | Train Loss: 0.518 | Train Acc: 72.35% | Val. Loss: 0.387 | Val. Acc: 85.22% |

100%  35/35 [00:20<00:00, 2.03it/s]

| Epoch: 02 | Train Loss: 0.243 | Train Acc: 90.09% | Val. Loss: 0.484 | Val. Acc: 83.57% |

100%  35/35 [00:20<00:00, 2.06it/s]

| Epoch: 03 | Train Loss: 0.087 | Train Acc: 97.23% | Val. Loss: 0.475 | Val. Acc: 85.22% |

100%  35/35 [00:21<00:00, 1.91it/s]

| Epoch: 04 | Train Loss: 0.028 | Train Acc: 99.02% | Val. Loss: 0.745 | Val. Acc: 83.57% |

Test Loss: 0.746 | Test Acc: 83.73% |

Training for 7 epochs

100%  35/35 [00:20<00:00, 2.01it/s]

| Epoch: 01 | Train Loss: 0.542 | Train Acc: 71.25% | Val. Loss: 0.418 | Val. Acc: 81.83% |

100%  35/35 [00:20<00:00, 2.05it/s]

| Epoch: 02 | Train Loss: 0.263 | Train Acc: 90.45% | Val. Loss: 0.445 | Val. Acc: 85.36% |

100%  35/35 [00:20<00:00, 2.05it/s]

| Epoch: 03 | Train Loss: 0.139 | Train Acc: 95.45% | Val. Loss: 0.608 | Val. Acc: 83.88% |

100%  35/35 [00:20<00:00, 2.04it/s]

| Epoch: 04 | Train Loss: 0.100 | Train Acc: 96.79% | Val. Loss: 0.570 | Val. Acc: 84.64% |

100%  35/35 [00:20<00:00, 2.05it/s]

| Epoch: 05 | Train Loss: 0.042 | Train Acc: 98.48% | Val. Loss: 0.602 | Val. Acc: 84.82% |

100%  35/35 [00:20<00:00, 2.02it/s]

| Epoch: 06 | Train Loss: 0.016 | Train Acc: 99.73% | Val. Loss: 0.835 | Val. Acc: 83.08% |

100%  35/35 [00:20<00:00, 2.07it/s]

| Epoch: 07 | Train Loss: 0.015 | Train Acc: 99.55% | Val. Loss: 0.810 | Val. Acc: 84.82% |

Test Loss: 0.745 | Test Acc: 83.93% |

```
training complete!
Training for 10 epochs
100% ██████████ 35/35 [00:20<00:00, 2.01it/s]
| Epoch: 01 | Train Loss: 0.537 | Train Acc: 72.53% | Val. Loss: 0.391 | Val. Acc: 83.66% |
100% ██████████ 35/35 [00:20<00:00, 2.05it/s]
| Epoch: 02 | Train Loss: 0.248 | Train Acc: 90.48% | Val. Loss: 0.544 | Val. Acc: 84.20% |
100% ██████████ 35/35 [00:20<00:00, 2.03it/s]
| Epoch: 03 | Train Loss: 0.154 | Train Acc: 94.29% | Val. Loss: 0.584 | Val. Acc: 83.79% |
100% ██████████ 35/35 [00:20<00:00, 2.07it/s]
| Epoch: 04 | Train Loss: 0.110 | Train Acc: 96.16% | Val. Loss: 0.461 | Val. Acc: 86.47% |
100% ██████████ 35/35 [00:20<00:00, 2.03it/s]
| Epoch: 05 | Train Loss: 0.041 | Train Acc: 98.93% | Val. Loss: 0.682 | Val. Acc: 85.22% |
100% ██████████ 35/35 [00:20<00:00, 2.06it/s]
| Epoch: 06 | Train Loss: 0.025 | Train Acc: 99.29% | Val. Loss: 0.636 | Val. Acc: 86.16% |
100% ██████████ 35/35 [00:20<00:00, 2.05it/s]
| Epoch: 07 | Train Loss: 0.015 | Train Acc: 99.64% | Val. Loss: 0.717 | Val. Acc: 85.54% |
100% ██████████ 35/35 [00:20<00:00, 2.01it/s]
| Epoch: 08 | Train Loss: 0.021 | Train Acc: 99.46% | Val. Loss: 0.703 | Val. Acc: 86.16% |
100% ██████████ 35/35 [00:20<00:00, 2.07it/s]
| Epoch: 09 | Train Loss: 0.011 | Train Acc: 99.64% | Val. Loss: 0.702 | Val. Acc: 85.45% |
100% ██████████ 35/35 [00:20<00:00, 2.05it/s]
| Epoch: 10 | Train Loss: 0.010 | Train Acc: 99.73% | Val. Loss: 0.737 | Val. Acc: 85.76% |
Test Loss: 0.765 | Test Acc: 85.71% |
```

Here we can clearly see that as we increase the number of epochs the training accuracy as well as validation accuracy increases. We tried 2,4,7,10 number of epochs and we got the best result with 10 epochs. It didn't overfit the data because even on the testing dataset it was performing well. Hence we should train it for 10 epochs for optimal performance.

1.22) Learning rate

Learning rate determines how quickly we update the weights of our model. We have experimented with different learning rates. It should be between 0 and 1. We tried **0.1, 0.01, 1e-3, 1e-5, 1e-8** learning rates and discovered that the sweet spot is somewhere around 1e-05.

```
Learning rate: 0.1
100% ██████████ 35/35 [00:19<00:00, 2.12it/s]
| Epoch: 01 | Train Loss: 90.240 | Train Acc: 48.42% | Val. Loss: 11.601 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:19<00:00, 2.17it/s]
| Epoch: 02 | Train Loss: 31.134 | Train Acc: 50.68% | Val. Loss: 20.247 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:19<00:00, 2.16it/s]
| Epoch: 03 | Train Loss: 31.750 | Train Acc: 51.25% | Val. Loss: 9.070 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:18<00:00, 2.20it/s]
| Epoch: 04 | Train Loss: 24.501 | Train Acc: 49.23% | Val. Loss: 20.501 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:19<00:00, 2.14it/s]
| Epoch: 05 | Train Loss: 21.198 | Train Acc: 49.17% | Val. Loss: 21.185 | Val. Acc: 50.00% |
Test Loss: 21.494 | Test Acc: 49.27% |
```



```
Training complete!
Learning rate: 0.01
100% ██████████ 35/35 [00:21<00:00, 2.16it/s]
| Epoch: 01 | Train Loss: 5.689 | Train Acc: 50.77% | Val. Loss: 4.695 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:19<00:00, 2.16it/s]
| Epoch: 02 | Train Loss: 4.842 | Train Acc: 47.83% | Val. Loss: 1.694 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:19<00:00, 2.15it/s]
| Epoch: 03 | Train Loss: 3.685 | Train Acc: 50.62% | Val. Loss: 4.140 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:19<00:00, 2.18it/s]
| Epoch: 04 | Train Loss: 3.290 | Train Acc: 48.87% | Val. Loss: 1.058 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:19<00:00, 2.11it/s]
| Epoch: 05 | Train Loss: 2.962 | Train Acc: 51.55% | Val. Loss: 2.176 | Val. Acc: 50.00% |
Test Loss: 2.207 | Test Acc: 49.27% |
```

```
Training complete!
Learning rate: 0.001
100% ██████████ 35/35 [00:20<00:00, 2.08it/s]
| Epoch: 01 | Train Loss: 1.089 | Train Acc: 49.38% | Val. Loss: 0.703 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:20<00:00, 2.07it/s]
| Epoch: 02 | Train Loss: 0.825 | Train Acc: 49.23% | Val. Loss: 0.786 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:20<00:00, 2.01it/s]
| Epoch: 03 | Train Loss: 0.782 | Train Acc: 52.11% | Val. Loss: 0.810 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:20<00:00, 2.08it/s]
| Epoch: 04 | Train Loss: 0.801 | Train Acc: 47.83% | Val. Loss: 0.836 | Val. Acc: 50.00% |
100% ██████████ 35/35 [00:20<00:00, 2.05it/s]
| Epoch: 05 | Train Loss: 0.790 | Train Acc: 47.98% | Val. Loss: 0.719 | Val. Acc: 50.00% |
Test Loss: 0.722 | Test Acc: 49.27% |
```

```
Training complete!
Learning rate: 1e-05
100% ██████████ 35/35 [00:20<00:00, 2.02it/s]
| Epoch: 01 | Train Loss: 0.502 | Train Acc: 73.42% | Val. Loss: 0.377 | Val. Acc: 85.98% |
100% ██████████ 35/35 [00:20<00:00, 2.06it/s]
| Epoch: 02 | Train Loss: 0.243 | Train Acc: 90.45% | Val. Loss: 0.462 | Val. Acc: 82.23% |
100% ██████████ 35/35 [00:20<00:00, 2.04it/s]
| Epoch: 03 | Train Loss: 0.156 | Train Acc: 95.18% | Val. Loss: 0.446 | Val. Acc: 84.20% |
100% ██████████ 35/35 [00:20<00:00, 2.03it/s]
| Epoch: 04 | Train Loss: 0.066 | Train Acc: 97.50% | Val. Loss: 0.516 | Val. Acc: 83.26% |
100% ██████████ 35/35 [00:20<00:00, 2.07it/s]
| Epoch: 05 | Train Loss: 0.023 | Train Acc: 99.46% | Val. Loss: 0.571 | Val. Acc: 85.04% |
Test Loss: 0.634 | Test Acc: 83.36% |
```

```
Training complete!
Learning rate: 1e-08
100% ██████████ 35/35 [00:20<00:00, 2.06it/s]
| Epoch: 01 | Train Loss: 0.720 | Train Acc: 49.79% | Val. Loss: 0.721 | Val. Acc: 50.40% |
100% ██████████ 35/35 [00:20<00:00, 2.00it/s]
| Epoch: 02 | Train Loss: 0.720 | Train Acc: 48.01% | Val. Loss: 0.719 | Val. Acc: 50.40% |
100% ██████████ 35/35 [00:20<00:00, 2.06it/s]
| Epoch: 03 | Train Loss: 0.724 | Train Acc: 47.98% | Val. Loss: 0.718 | Val. Acc: 50.71% |
100% ██████████ 35/35 [00:20<00:00, 2.06it/s]
| Epoch: 04 | Train Loss: 0.717 | Train Acc: 49.97% | Val. Loss: 0.716 | Val. Acc: 51.34% |
100% ██████████ 35/35 [00:20<00:00, 2.02it/s]
| Epoch: 05 | Train Loss: 0.721 | Train Acc: 51.10% | Val. Loss: 0.715 | Val. Acc: 51.65% |
Test Loss: 0.710 | Test Acc: 49.39% |
```

We got the best results with $1e-5$. If we were to send this to a production system we can do more fine-grained testing and figure out the exact learning rate for maximum test accuracy but this is not needed now. We proved that the optimal learning rate impacts the test accuracy hugely and most optimal value for us is $1e-05$.

1.27) Discussion

Hyperparameter tuning significantly impacts AI model training, enhancing performance, generalization, and convergence speed. Before tuning, the model exhibited suboptimal performance, struggling with overfitting or slow convergence. However, after adjusting hyperparameters like learning rate and batch size, the model showed improved accuracy, faster convergence, and better generalization to unseen data. Tuning helps strike a balance between underfitting and overfitting, making the model more robust, efficient, and capable of handling diverse datasets. Ultimately, proper hyperparameter tuning is essential for maximizing a model's effectiveness and real-world applicability.

2) CYBERBULLY DETECTION ON IMAGE

Now we will create AI models to detect cyberbullying detection in images. We load the dataset mentioned in the file and also load the pre-trained model.

```
[6]
def __len__(self):
    return len(self.samples)

def __getitem__(self, idx):
    if torch.is_tensor(idx):
        idx = idx.tolist()

    img_name, pose_name, aux_name, label = self.samples[idx]
    image = io.imread(img_name)

    aux = pickle.load(open(aux_name + '.p', 'rb'))
    aux = torch.tensor(aux)

    # drop the alpha channel for some images
    if image.shape == (224, 224):
        # handle grayscale images
        image = np.stack([image, image, image], axis=2)

    if image.shape == (224, 224, 4):
        image = image[:, :, :3]

    image = image.transpose((2, 0, 1)) # C X H X W
    pose = io.imread(pose_name)
    if pose.shape != (224, 224):
        pose = pose[:, :, 0]
    pose = np.expand_dims(pose, axis = 0)
    image = np.concatenate((image, pose), axis = 0)
    sample = {'image': torch.from_numpy(image.copy()).float() / 255, 'aux': aux, 'label': label}
    return sample

[7] test_set = PosesDataset('cyberbullying_data/cyberbullying_data_splits_clean/test/', 'cyberbullying_data/cyberbullying_poses/test/', 'cyberbullying_data/cyberbullying_data_auxes/test/')
test_loader = torch.utils.data.DataLoader(test_set, batch_size = 1, shuffle = True)
```

2.1) Generate result report containing: Accuracy, Precision, Recall, and F1-Score

We use the values of false positives, false negatives, true positives and true negatives to calculate the result report as per the formulas given in

<https://machinelearningmastery.com/how-to-calculate-precision-recall-f1-and-more-for-deep-learning-models/>

```
[16] # TODO: Complete the following code to calculate the accuracy, precision, recall and F1 score.
acc = (tp + tn) / (tp + tn + fp + fn)
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f1 = 2*tp / (2*tp + fp + fn)

print('The accuracy for test dataset is: {}'.format(acc * 100))
print('The precision for test dataset is: {}'.format(precision * 100))
print('The recall for test dataset is: {}'.format(recall * 100))
print('The f1 score for test dataset is: {}'.format(f1 * 100))

The accuracy for test dataset is: 85.0%
The precision for test dataset is: 88.88888888888889%
The recall for test dataset is: 80.0%
The f1 score for test dataset is: 84.21052631578947%
```

2.2) Generating the Confusion matrix.

Now we generate the confusion matrix by getting the values of y_{true} i.e. the ground truths and y_{pred} i.e. the values predicted by the model.

Below code shows how we calculated it.

```
# Complete the following code to get the confusion matrix for the test set
# get the confusion matrix for the test set
from sklearn.metrics import confusion_matrix

y_true = []
y_pred = []

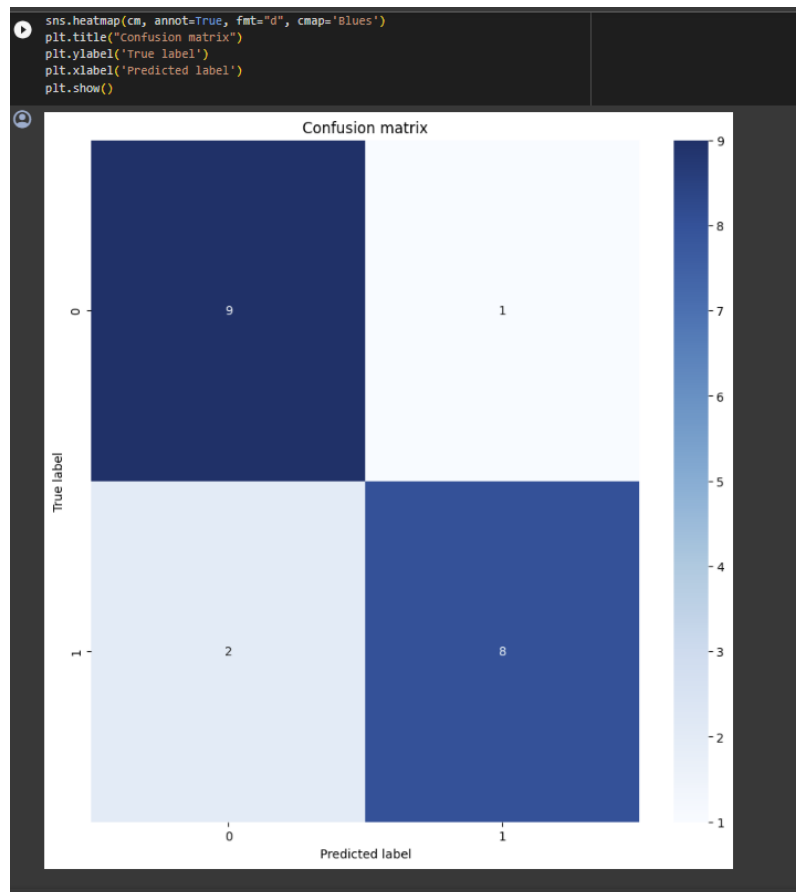
# TODO: Write the code to get the y_true and y_pred lists for the test set
# your code here:
with torch.no_grad():
    for i_v, data_v in enumerate(test_loader):
        x_test, y_test, aux_test = data_v['image'], data_v['label'], data_v['aux']

        y_true.append(int(y_test))
        x_test, y_test, aux_test = x_test.to(device), y_test.to(device, dtype = torch.long), aux_test.to(device, dtype = torch.float)
        y_test_ = model(x_test, aux_test) # forward pass for the fine-tuned model
        _, predicted = torch.max(y_test_.data, 1)

        y_pred.append(int(predicted))

# get the confusion matrix
cm = confusion_matrix(y_true, y_pred)
# cm
```

The Confusion matrix is shown below.



2.3) Fine-tuning the model with the training dataset

I completed the given code so that we can tune the hyperparameters for optimal performance. The completed code is shown below

```
# TODO: complete the following code by replace "___", to mimic fine-tune (further training) the model
ft_model.train()

epochs = 10
for epoch in range(epochs):
    for i, data in enumerate(train_loader):
        inputs = data['image'].to(device)
        aux = data['aux'].to(device)
        labels = data['label'].to(device)
        optimizer.zero_grad() # zero the parameter gradients
        outputs = ft_model(inputs, aux) # forward pass
        loss = criterion(outputs, labels) # compute loss via comparing model's outputs and our predefined labels
        loss.backward() # backward pass
        optimizer.step() # update weights
        running_loss.append(loss.item()) # save loss
        _, predicted = torch.max(outputs.data, 1) # get predictions
        total += labels.size(0) # update total
        correct += (predicted == labels).sum().item() # update correct predictions
        if i % 50 == 0: # print every 50 mini-batches
            print('Epoch: %d, Iteration: %d, Loss: %.4f, Accuracy: %.4f' % (epoch, i, loss.item(), correct / total))
            correct, incorrect, total = 0.0, 0.0, 0.0 # reset correct, incorrect, and total. hint: float is better than int

# Note:
# This code is a very basic version that helps us keep training the model with the training set,
# Recall from the last lecture, we can have a validation set to help us decide when to stop training
# the model.
# If you are interested, you can try to split the training set into training set and validation set,
# and use the validation set to help you decide when to stop training the model
```

```
Epoch: 0, Iteration: 0, Loss: 0.3969, Accuracy: 0.8750
/usr/local/lib/python3.10/dist-packages/PIL/Image.py:996: UserWarning: Palette images with Transparency expressed in bytes should be converted to RGBA images
  warnings.warn(
Epoch: 0, Iteration: 50, Loss: 0.3133, Accuracy: 0.8717
Epoch: 0, Iteration: 100, Loss: 0.5633, Accuracy: 0.8717
Epoch: 0, Iteration: 150, Loss: 0.3966, Accuracy: 0.8817
Epoch: 0, Iteration: 200, Loss: 0.5633, Accuracy: 0.8550
Epoch: 1, Iteration: 0, Loss: 0.3133, Accuracy: 0.9444
Epoch: 1, Iteration: 50, Loss: 0.3966, Accuracy: 0.8500
Epoch: 1, Iteration: 100, Loss: 0.3966, Accuracy: 0.8933
Epoch: 1, Iteration: 150, Loss: 0.3133, Accuracy: 0.8650
Epoch: 1, Iteration: 200, Loss: 0.5633, Accuracy: 0.8700
Epoch: 2, Iteration: 0, Loss: 0.3966, Accuracy: 0.8889
Epoch: 2, Iteration: 50, Loss: 0.4799, Accuracy: 0.8667
Epoch: 2, Iteration: 100, Loss: 0.5633, Accuracy: 0.8783
Epoch: 2, Iteration: 150, Loss: 0.3133, Accuracy: 0.8567
Epoch: 2, Iteration: 200, Loss: 0.4799, Accuracy: 0.8800
Epoch: 3, Iteration: 0, Loss: 0.3966, Accuracy: 0.8333
Epoch: 3, Iteration: 50, Loss: 0.3966, Accuracy: 0.8650
Epoch: 3, Iteration: 100, Loss: 0.6466, Accuracy: 0.8783
Epoch: 3, Iteration: 150, Loss: 0.3133, Accuracy: 0.8717
Epoch: 3, Iteration: 200, Loss: 0.3966, Accuracy: 0.8683
Epoch: 4, Iteration: 0, Loss: 0.4799, Accuracy: 0.7222
Epoch: 4, Iteration: 50, Loss: 0.5633, Accuracy: 0.8633
Epoch: 4, Iteration: 100, Loss: 0.3133, Accuracy: 0.8800
Epoch: 4, Iteration: 150, Loss: 0.4799, Accuracy: 0.8683
Epoch: 4, Iteration: 200, Loss: 0.3966, Accuracy: 0.8700
Epoch: 5, Iteration: 0, Loss: 0.5633, Accuracy: 0.7778
Epoch: 5, Iteration: 50, Loss: 0.3966, Accuracy: 0.8683
Epoch: 5, Iteration: 100, Loss: 0.3966, Accuracy: 0.8600
Epoch: 5, Iteration: 150, Loss: 0.4799, Accuracy: 0.8683
Epoch: 5, Iteration: 200, Loss: 0.4799, Accuracy: 0.8850
Epoch: 6, Iteration: 0, Loss: 0.3133, Accuracy: 1.0000
Epoch: 6, Iteration: 50, Loss: 0.4799, Accuracy: 0.8700
Epoch: 6, Iteration: 100, Loss: 0.4799, Accuracy: 0.8867
Epoch: 6, Iteration: 150, Loss: 0.5633, Accuracy: 0.8733
Epoch: 6, Iteration: 200, Loss: 0.4799, Accuracy: 0.8467
Epoch: 7, Iteration: 0, Loss: 0.3133, Accuracy: 1.0000
Epoch: 7, Iteration: 50, Loss: 0.3133, Accuracy: 0.8883
Epoch: 7, Iteration: 100, Loss: 0.5633, Accuracy: 0.8400
Epoch: 7, Iteration: 150, Loss: 0.4799, Accuracy: 0.8767
Epoch: 7, Iteration: 200, Loss: 0.3133, Accuracy: 0.8733
Epoch: 8, Iteration: 0, Loss: 0.3966, Accuracy: 0.8889
Epoch: 8, Iteration: 50, Loss: 0.3966, Accuracy: 0.8550
Epoch: 8, Iteration: 100, Loss: 0.7299, Accuracy: 0.8650
Epoch: 8, Iteration: 150, Loss: 0.4799, Accuracy: 0.8733
Epoch: 8, Iteration: 200, Loss: 0.6466, Accuracy: 0.8883
Epoch: 9, Iteration: 0, Loss: 0.5633, Accuracy: 0.7222
Epoch: 9, Iteration: 50, Loss: 0.3133, Accuracy: 0.8617
Epoch: 9, Iteration: 100, Loss: 0.3966, Accuracy: 0.8733
Epoch: 9, Iteration: 150, Loss: 0.5633, Accuracy: 0.8700
Epoch: 9, Iteration: 200, Loss: 0.3133, Accuracy: 0.8767
```

We can see that we got an accuracy of 87.67% after fine-tuning the model.

2.4) Printing fine-tuned model's results

The below code shows how to print the results of the fine-tuned model.

```
[32] ft_model.eval()
# TODO: write code to evaluate the model on the test set
total = 0
correct = 0
with torch.no_grad(): # Disable gradient calculation during evaluation
    for i, data in enumerate(test_loader):
        inputs = data['image'].to(device)
        aux = data['aux'].to(device)
        labels = data['label'].to(device)

        outputs = ft_model(inputs, aux) # Forward pass

        _, predicted = torch.max(outputs.data, 1) # Get predictions
        total += labels.size(0) # Update total number of samples

        correct += (predicted == labels).sum().item() # Update number of correct predictions

# Calculate accuracy
accuracy = correct / total
print("Accuracy on the test set:", accuracy)

Accuracy on the test set: 0.5
```

We discovered that we only got 50% accuracy when we used our fine-tuned model on the test dataset which is significantly worse than the previous model.

The reason for this is that our model has overfitted the data, what it means is that it has memorized the training dataset without generalizing enough and that's why it's performing so badly on the test dataset which it hasn't seen.

2.5) Testing the image with the fine-tuned model and seeing the result.

Let's test our fine-tuned model on the image shown below.



We find its index using code below and then we run the model against it.

```
# TODO: find the picture_index of the chosen image by comparing with the previous visualization cell
picture_index = 0

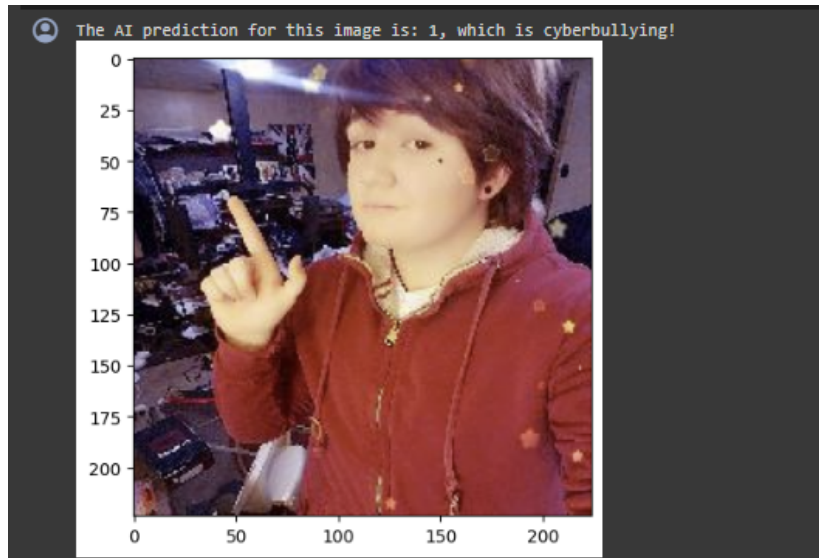
for i in range(len(test_set)):
    print(type(test_set[i]['image']), type(img))
    if test_set[i]['image'] == transforms.toTensor()(img).squeeze(0):
        picture_index=i
        break
instance = test_set[picture_index]

# check if the prediction is correct
instance_image, instance_label, instance_aux = instance['image'].to(device), torch.tensor(instance['label']).to(device, dtype = torch.long), instance['aux'].to(device, dtype = torch.float)

# TODO: get the prediction for the image
output = ft_model(instance_image.unsqueeze(0), instance_aux.unsqueeze(0)).data
_, prediction = torch.max(output.data, 1)
predict_label = "cyberbullying" if prediction.item()==1 else "non-cyberbullying"
comparision = "correct" if prediction==instance_label else "not correct"

print("The AI prediction for this image is: {}, which is {}".format(predict_label, comparision))
```

Running the code we get the following result.



Hence we understand that our model is working correctly after fine tuning.