A Deep Learning Visual Content Based Recommender System to Defend Adversarial Attacks- Configuration Manual

MSc Research Project
MSc in Data Analytics

Komal -
Student ID: x20207034

School of Computing
National College of Ireland

Supervisor: Dr. Paul Stynes
Dr. Musfira Jilani
Dr. Pramod Pathak

Student Name:	Komal -
Student ID:	x20207034
Programme:	MSc in Data Analytics
Year:	2022
Module:	MSc Research Project
Supervisor:	Dr. Paul Stynes, Dr. Musfira Jilani, Dr. Pramod Pathak
Submission Due Date:	15/08/2022
Project Title:	A Deep Learning Visual Content Based Recommender System to Defend Adversarial Attacks- Configuration Manual
Word Count:	1080
Page Count:	12

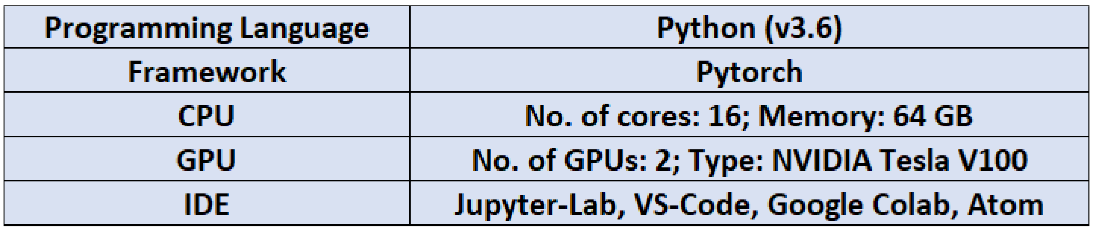
Signature:	Komal -
Date:	14th August 2022

A Deep Learning Visual Content Based Recommender System to Defend Adversarial Attacks- Configuration Manual

Komal -
x20207034

1 Initial Environment Setting

The table below provides a description of the setup's software requirements:



Programming Language	Python (v3.6)
Framework	Pytorch
CPU	No. of cores: 16; Memory: 64 GB
GPU	No. of GPUs: 2; Type: NVIDIA Tesla V100
IDE	Jupyter-Lab, VS-Code, Google Colab, Atom

Figure 1: Software Requirement Description

1.1 Google Colab Environment Set-up

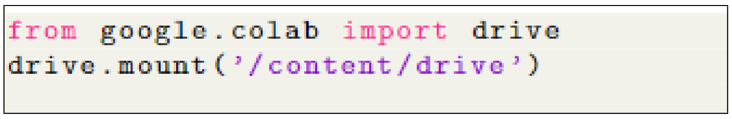
The steps listed below can be used to configure code execution in the Google Colab environment:

1. Use gmail to login to access your account.

2. Upload the whole dataset folder by opening the Google Drive tab. Given the scale of the dataset, this could take some time to complete.

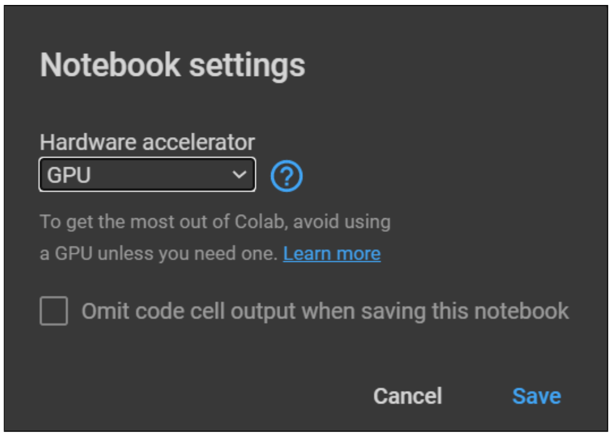
3. To run the code, open the Google Colab's notebook and select the GPU environment.

4. Use the code mentioned in below image to mount the drive in the google colab environment:



```
from google.colab import drive
drive.mount('/content/drive')
```

5. Open the first python (*.ipynb*) notebook from the artefacts once the dataset has been uploaded and mounted, and then begin the cell execution one at a time. To prevent mistakes, run the notebooks in order.


Notebook settings

Hardware accelerator: GPU

To get the most out of Colab, avoid using a GPU unless you need one. Learn more

☐ Omit code cell output when saving this notebook

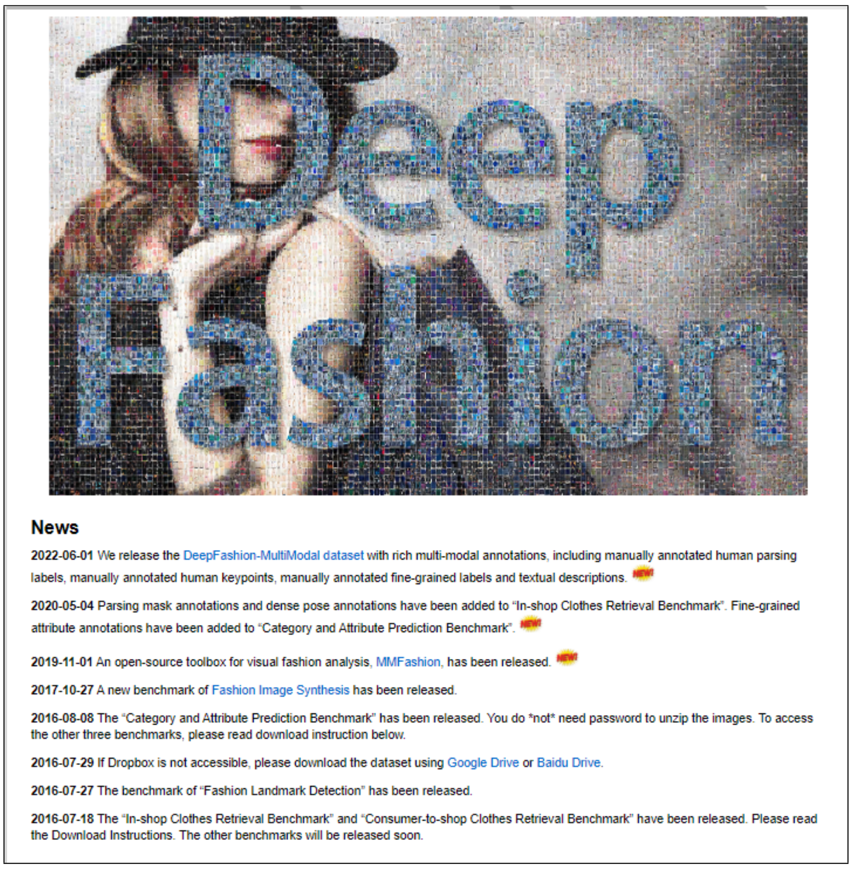
Figure 2: Configuring Google Colab's GPU-enabled environment

2 Data Pre-processing & EDA

2.1 Data Preparation

2.1.1 Initial Set-up

1. Download DeepFashion dataset files from the official website Dataset_Link(figure 3).



News

2022-06-01 We release the DeepFashion-MultiModal dataset with rich multi-modal annotations, including manually annotated human parsing labels, manually annotated human keypoints, manually annotated fine-grained labels and textual descriptions.

2020-05-04 Parsing mask annotations and dense pose annotations have been added to "In-shop Clothes Retrieval Benchmark". Fine-grained attribute annotations have been added to "Category and Attribute Prediction Benchmark".

2019-11-01 An open-source toolbox for visual fashion analysis, MMFashion, has been released.

2017-10-27 A new benchmark of Fashion Image Synthesis has been released.

2016-08-08 The "Category and Attribute Prediction Benchmark" has been released. You do *not* need password to unzip the images. To access the other three benchmarks, please read download instruction below.

2016-07-29 If Dropbox is not accessible, please download the dataset using Google Drive or Baidu Drive.

2016-07-27 The benchmark of "Fashion Landmark Detection" has been released.

2016-07-18 The "In-shop Clothes Retrieval Benchmark" and "Consumer-to-shop Clothes Retrieval Benchmark" have been released. Please read the Download Instructions. The other benchmarks will be released soon.

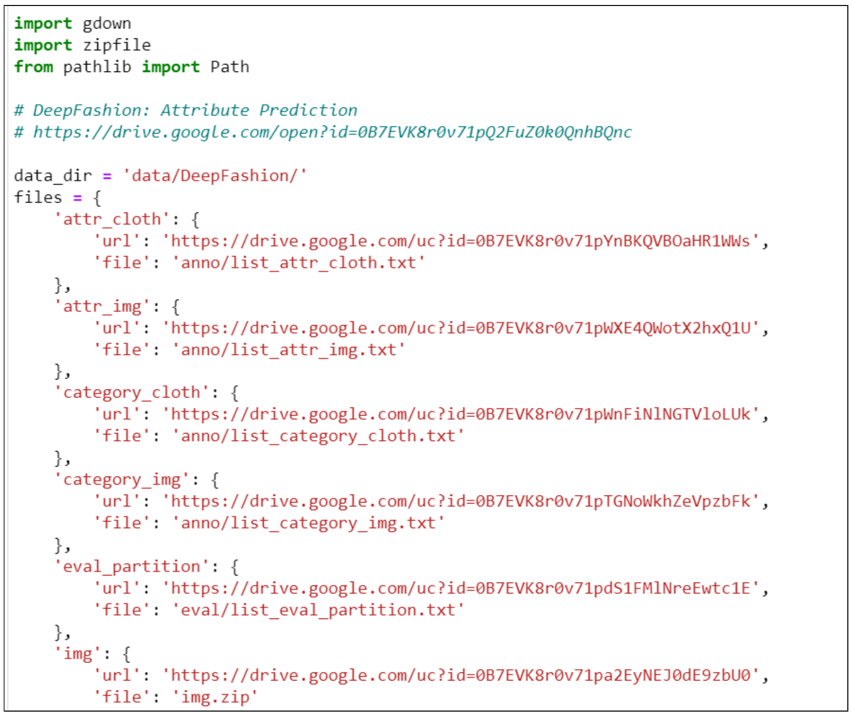
Figure 3: DeepFashion Dataset

2. Simply upload the data to Google Drive for pre-processing, then proceed as directed in Section 1.1. Save the data files in a folder with the name data for the model training and subsequent procedures..

2.2 Data Pre-processing

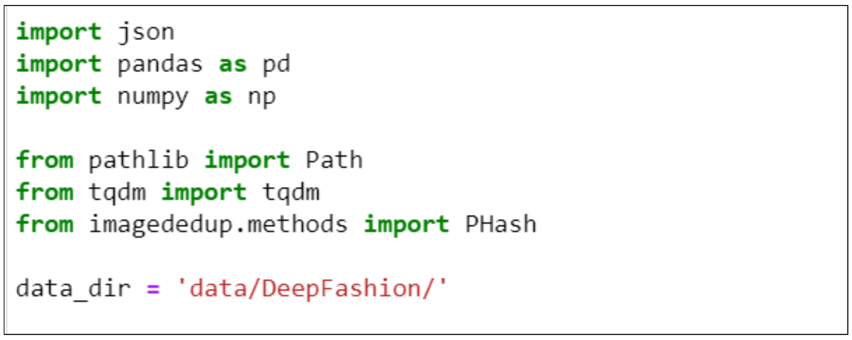
Follow the steps listed below for pre-processing:

- Import necessary python libraries to download and unzip the dataset files.



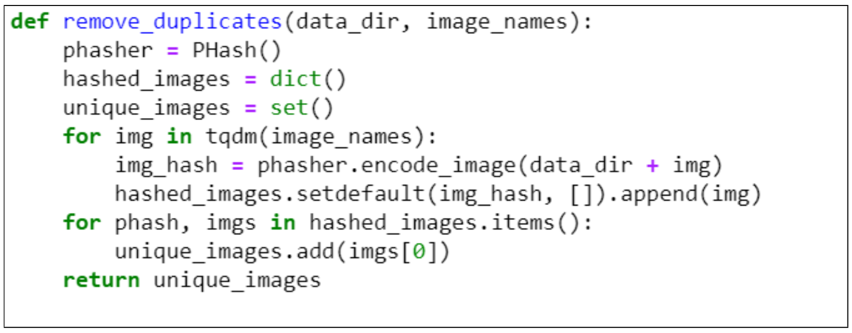
```
import gdown
import zipfile
from pathlib import Path

# DeepFashion: Attribute Prediction
# https://drive.google.com/open?id=0B7EVK8r0v71pQ2FuZ0k0QnhBQnc

data_dir = 'data/DeepFashion/'
files = {
    'attr_cloth': {
        'url': 'https://drive.google.com/uc?id=0B7EVK8r0v71pYnBKQVBOaHR1WWs',
        'file': 'anno/list_attr_cloth.txt'
    },
    'attr_img': {
        'url': 'https://drive.google.com/uc?id=0B7EVK8r0v71pWXE4QWotX2hxQ1U',
        'file': 'anno/list_attr_img.txt'
    },
    'category_cloth': {
        'url': 'https://drive.google.com/uc?id=0B7EVK8r0v71pWnFiNlNGTVloLUk',
        'file': 'anno/list_category_cloth.txt'
    },
    'category_img': {
        'url': 'https://drive.google.com/uc?id=0B7EVK8r0v71pTGNoWkhZeVpzbFk',
        'file': 'anno/list_category_img.txt'
    },
    'eval_partition': {
        'url': 'https://drive.google.com/uc?id=0B7EVK8r0v71pdS1FMlNreEwtc1E',
        'file': 'eval/list_eval_partition.txt'
    },
    'img': {
        'url': 'https://drive.google.com/uc?id=0B7EVK8r0v71pa2EyNEJ0dE9zbU0',
        'file': 'img.zip'
```

- For data preparation, import necessary libraries.



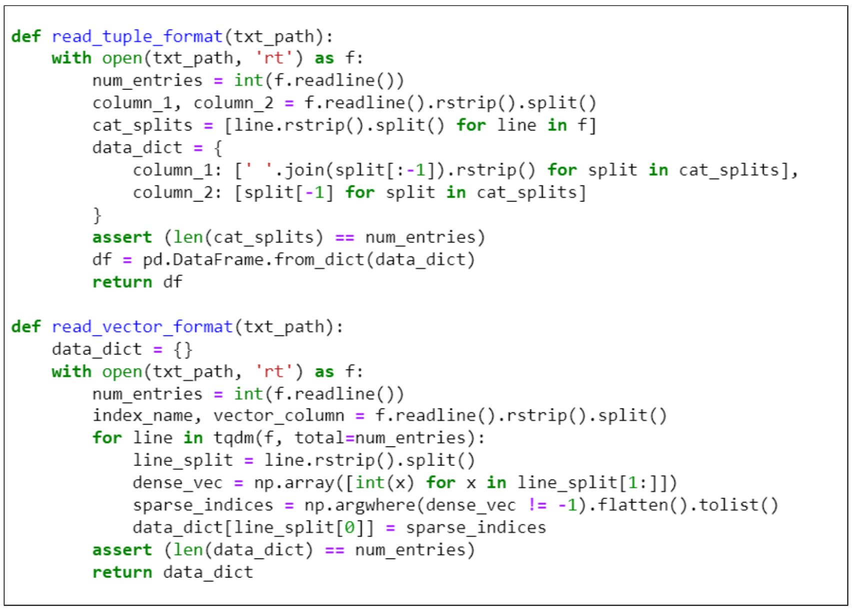
```
import json
import pandas as pd
import numpy as np

from pathlib import Path
from tqdm import tqdm
from imagededup.methods import PHash

data_dir = 'data/DeepFashion/'
```

- After importing data files and required libraries. It is necessary to remove the irrelevant data from our files. Below code is used to remove the duplicate values in our dataset.



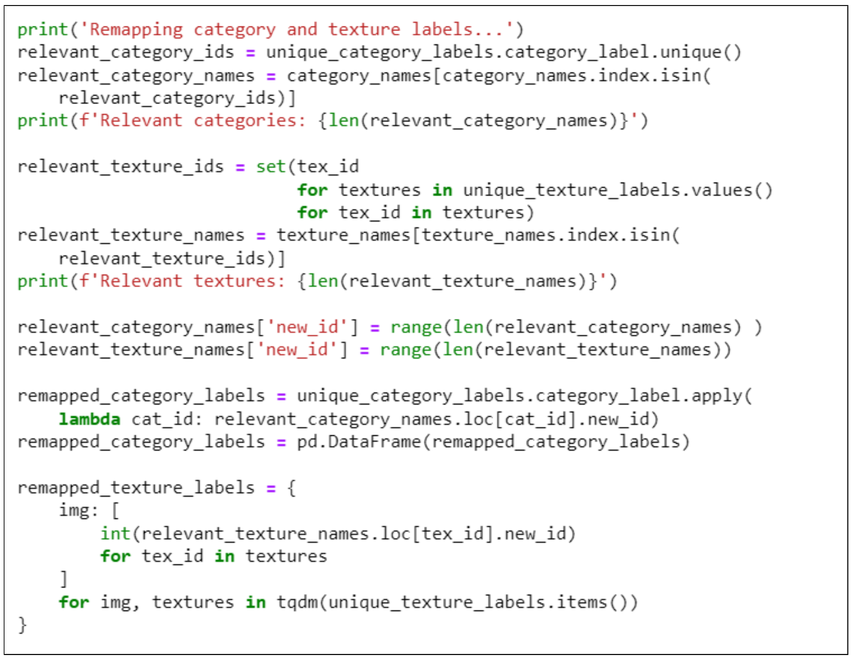
```python
def remove_duplicates(data_dir, image_names):
    phasher = PHash()
    hashed_images = dict()
    unique_images = set()
    for img in tqdm(image_names):
        img_hash = phasher.encode_image(data_dir + img)
        hashed_images.setdefault(img_hash, []).append(img)
    for phash, imgs in hashed_images.items():
        unique_images.add(imgs[0])
    return unique_images
```

- After filtering out duplicate values, two functions are used to parse DeepFashion category and attributes to python dictionaries as shown in below figure.



```python
def read_tuple_format(txt_path):
    with open(txt_path, 'rt') as f:
        num_entries = int(f.readline())
        column_1, column_2 = f.readline().rstrip().split()
        cat_splits = [line.rstrip().split() for line in f]
        data_dict = {
            column_1: [' '.join(split[:-1]).rstrip() for split in cat_splits],
            column_2: [split[-1] for split in cat_splits]
        }
        assert (len(cat_splits) == num_entries)
        df = pd.DataFrame.from_dict(data_dict)
        return df

def read_vector_format(txt_path):
    data_dict = {}
    with open(txt_path, 'rt') as f:
        num_entries = int(f.readline())
        index_name, vector_column = f.readline().rstrip().split()
        for line in tqdm(f, total=num_entries):
            line_split = line.rstrip().split()
            dense_vec = np.array([int(x) for x in line_split[1:]])
            sparse_indices = np.argwhere(dense_vec != -1).flatten().tolist()
            data_dict[line_split[0]] = sparse_indices
        assert (len(data_dict) == num_entries)
        return data_dict
```

- Remapping category and texture lables of the images for that below code can be used.



```python
print('Remapping category and texture labels...')
relevant_category_ids = unique_category_labels.category_label.unique()
relevant_category_names = category_names[category_names.index.isin(
    relevant_category_ids)]
print(f'Relevant categories: {len(relevant_category_names)}')

relevant_texture_ids = set(tex_id
                           for textures in unique_texture_labels.values()
                           for tex_id in textures)
relevant_texture_names = texture_names[texture_names.index.isin(
    relevant_texture_ids)]
print(f'Relevant textures: {len(relevant_texture_names)}')

relevant_category_names['new_id'] = range(len(relevant_category_names) )
relevant_texture_names['new_id'] = range(len(relevant_texture_names))

remapped_category_labels = unique_category_labels.category_label.apply(
    lambda cat_id: relevant_category_names.loc[cat_id].new_id)
remapped_category_labels = pd.DataFrame(remapped_category_labels)

remapped_texture_labels = {
    img: [
        int(relevant_texture_names.loc[tex_id].new_id)
        for tex_id in textures
    ]
    for img, textures in tqdm(unique_texture_labels.items())
}
```

- After data pre-processing is done, important libraries will be imported for further steps.

```
import os
import torch
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from PIL import Image
from torch.utils.data import Dataset
from torchvision import transforms

from enum import Enum
```

- Dataset was divided into train and test data using below code.

```
class Split(Enum):
    TEST = 'test'
    TRAIN = 'train'
    BOTH = 'both'

    def __str__(self):
        return self.value
```

- To understand the data better, few visualizations were done. In the below code, we analysed categories label and plotted top 10 category labels.

```
plt.style.use('seaborn')

category_counts = remapped_category_labels.category_label.value_counts().sort_values()
category_counts.index = category_counts.index.map(lambda x: relevant_category_names.category_name[x])
ax = category_counts.plot.barh(figsize=(20,10))
ax.set_title('Example counts for all categories', fontsize=16)
plt.show()

ax = category_counts[-10:].plot.barh(figsize=(20,10))
ax.set_title('Example counts for top 10 categories', fontsize=16)
tikzplotlib.save('../results/figures/top10_categories.tex')
```

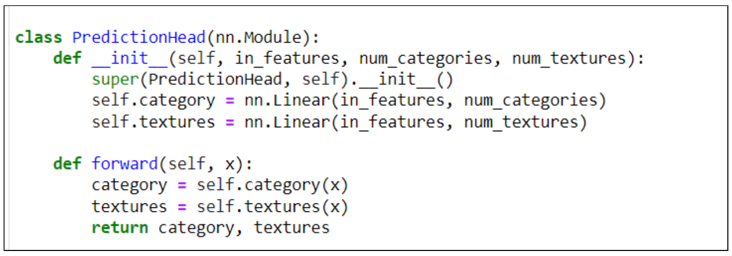
- Using below code, texture labels were analyzed and top 10 texture lables were plotted.

```
texture_counts = pd.Series([val for vals in remapped_texture_labels.values() for val in vals]).value_counts().sort_values()
texture_counts.index = texture_counts.index.map(lambda x: relevant_texture_names.attribute_name[x])
ax = texture_counts.plot.barh(figsize=(20,10))
ax.set_title('Label counts for all textures', fontsize=16)
plt.show()

ax = texture_counts[-50:].plot.barh(figsize=(20,10))
ax.set_title('Label counts for top 50 textures', fontsize=16)
plt.show()

ax = texture_counts[-10:].plot.barh(figsize=(20,10))
ax.set_title('Label counts for top 10 textures', fontsize=16)
tikzplotlib.save('../results/figures/top10_textures.tex')
```

3 Training the Image-Based Recommender System

The image based recommender system was trained next using a pre-trained CNN *MobileNetV2*. The steps followed have been described below:

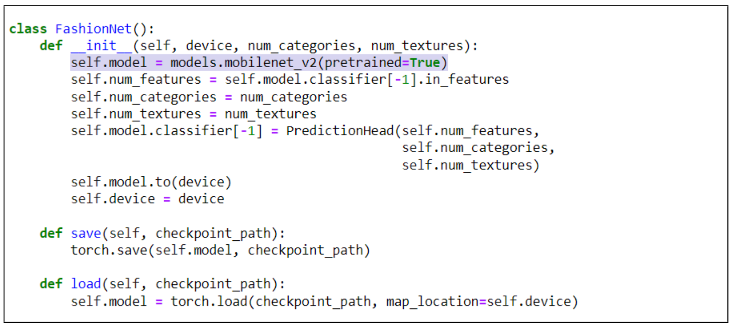
- The layers in the model were initialized first using Python classes and objects. The two variables *category* and *textures* were also initialized (as shown in the picture below) to be used later.



```python
class PredictionHead(nn.Module):
    def __init__(self, in_features, num_categories, num_textures):
        super(PredictionHead, self).__init__()
        self.category = nn.Linear(in_features, num_categories)
        self.textures = nn.Linear(in_features, num_textures)

    def forward(self, x):
        category = self.category(x)
        textures = self.textures(x)
        return category, textures
```

Figure 4

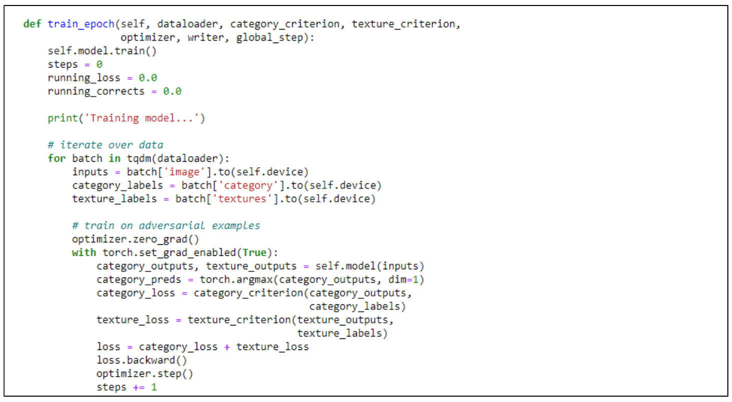
- The pre-trained model MobileNetV2 was then initialized with a classifier layer. The features, categories and textures were passed into the final classifier layer to correctly categorize a test image. The functions to save and load the model were also defined within this class. For saving the model, *torch.save()* function was used from the pytorch library



```python
class FashionNet():
    def __init__(self, device, num_categories, num_textures):
        self.model = models.mobilenet_v2(pretrained=True)
        self.num_features = self.model.classifier[-1].in_features
        self.num_categories = num_categories
        self.num_textures = num_textures
        self.model.classifier[-1] = PredictionHead(self.num_features,
                                                   self.num_categories,
                                                   self.num_textures)
        self.model.to(device)
        self.device = device

    def save(self, checkpoint_path):
        torch.save(self.model, checkpoint_path)

    def load(self, checkpoint_path):
        self.model = torch.load(checkpoint_path, map_location=self.device)
```

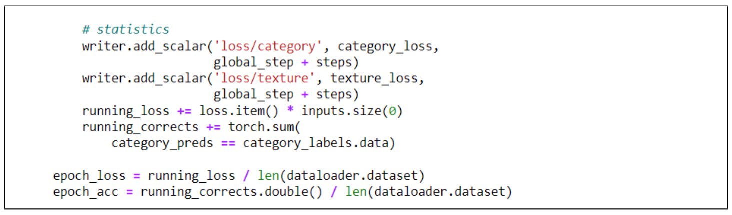
Figure 5

- After the the pytorch pre-trained model was trained and iterated over the data. The model was also trained on the adversarial examples (that were created in the later stages of the project)[figure 6]. The loss and accuracy values were saved into the logically named variables (figure 7).



```python
def train_epoch(self, dataloader, category_criterion, texture_criterion,
                optimizer, writer, global_step):
    self.model.train()
    steps = 0
    running_loss = 0.0
    running_corrects = 0.0

    print('Training model...')

    # iterate over data
    for batch in tqdm(dataloader):
        inputs = batch['image'].to(self.device)
        category_labels = batch['category'].to(self.device)
        texture_labels = batch['textures'].to(self.device)

        # train on adversarial examples
        optimizer.zero_grad()
        with torch.set_grad_enabled(True):
            category_outputs, texture_outputs = self.model(inputs)
            category_preds = torch.argmax(category_outputs, dim=1)
            category_loss = category_criterion(category_outputs,
                                               category_labels)
            texture_loss = texture_criterion(texture_outputs,
                                             texture_labels)
            loss = category_loss + texture_loss
            loss.backward()
            optimizer.step()
            steps += 1
```

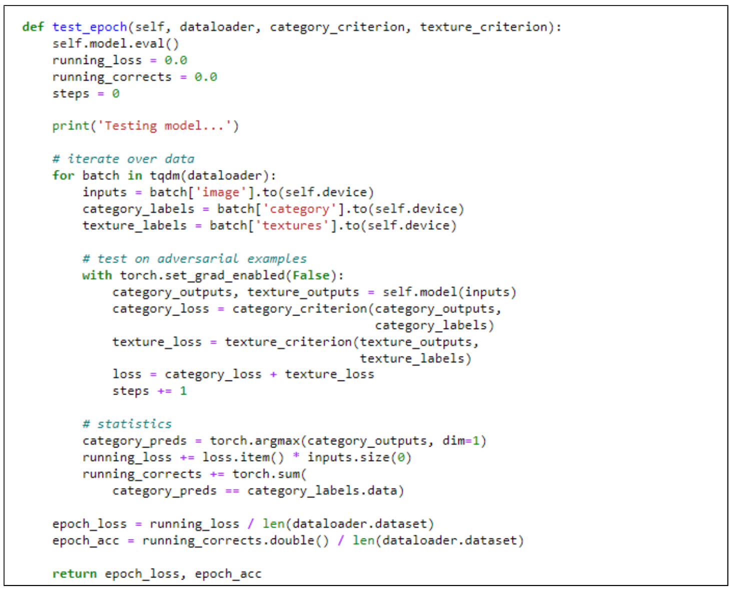
Figure 6: Training the MobileNetV2



```python
        # statistics
        writer.add_scalar('loss/category', category_loss,
                          global_step + steps)
        writer.add_scalar('loss/texture', texture_loss,
                          global_step + steps)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(
            category_preds == category_labels.data)

    epoch_loss = running_loss / len(dataloader.dataset)
    epoch_acc = running_corrects.double() / len(dataloader.dataset)
```

Figure 7: Saving the loss and accuracy values for the training dataset

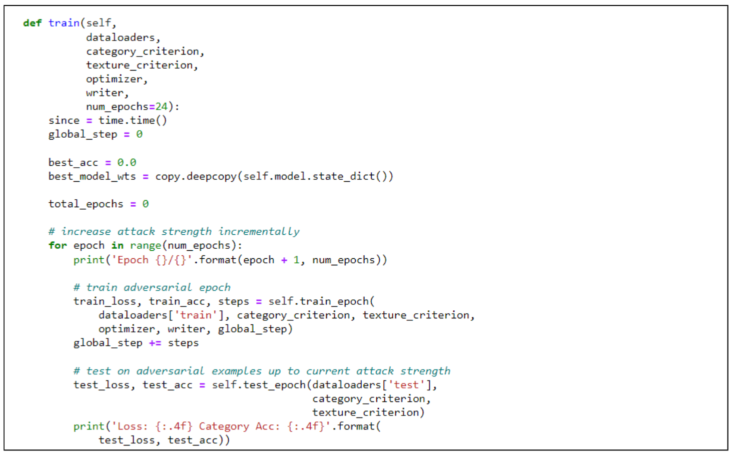
- In this next step , we define the *test_epoch* function to repeat the same procedure as in previous step for the test images.



```python
def test_epoch(self, dataloader, category_criterion, texture_criterion):
    self.model.eval()
    running_loss = 0.0
    running_corrects = 0.0
    steps = 0

    print('Testing model...')

    # iterate over data
    for batch in tqdm(dataloader):
        inputs = batch['image'].to(self.device)
        category_labels = batch['category'].to(self.device)
        texture_labels = batch['textures'].to(self.device)

        # test on adversarial examples
        with torch.set_grad_enabled(False):
            category_outputs, texture_outputs = self.model(inputs)
            category_loss = category_criterion(category_outputs,
                                               category_labels)
            texture_loss = texture_criterion(texture_outputs,
                                             texture_labels)
            loss = category_loss + texture_loss
            steps += 1

        # statistics
        category_preds = torch.argmax(category_outputs, dim=1)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(
            category_preds == category_labels.data)

    epoch_loss = running_loss / len(dataloader.dataset)
    epoch_acc = running_corrects.double() / len(dataloader.dataset)

    return epoch_loss, epoch_acc
```

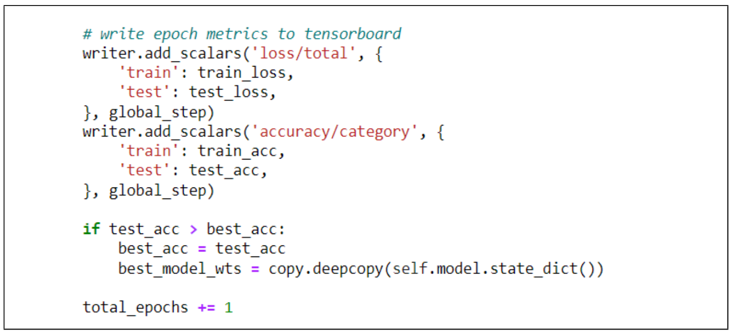
Figure 8: Testing the MobileNetV2

- In this last step, we train and test our model again after executing the adversarial attacks. We train the models in an iterative manner and increase the strength of the attacks in steps. We then reset the model to best weights and print the best category accuracy. As an additional step, we also write the epoch metrics to tensorboard to create a dashboard of our results.



```python
def train(self,
          dataloaders,
          category_criterion,
          texture_criterion,
          optimizer,
          writer,
          num_epochs=24):
    since = time.time()
    global_step = 0

    best_acc = 0.0
    best_model_wts = copy.deepcopy(self.model.state_dict())

    total_epochs = 0

    # increase attack strength incrementally
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch + 1, num_epochs))

        # train adversarial epoch
        train_loss, train_acc, steps = self.train_epoch(
            dataloaders['train'], category_criterion, texture_criterion,
            optimizer, writer, global_step)
        global_step += steps

        # test on adversarial examples up to current attack strength
        test_loss, test_acc = self.test_epoch(dataloaders['test'],
                                              category_criterion,
                                              texture_criterion)
        print('Loss: {:.4f} Category Acc: {:.4f}'.format(
            test_loss, test_acc))
```

Figure 9: Training the model after executing attacks



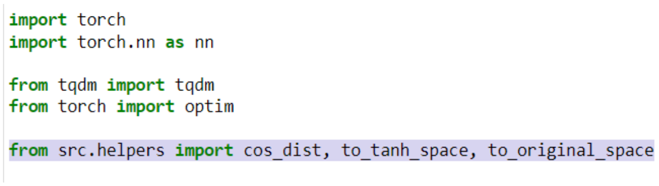
```python
        # write epoch metrics to tensorboard
        writer.add_scalars('loss/total', {
            'train': train_loss,
            'test': test_loss,
        }, global_step)
        writer.add_scalars('accuracy/category', {
            'train': train_acc,
            'test': test_acc,
        }, global_step)

        if test_acc > best_acc:
            best_acc = test_acc
            best_model_wts = copy.deepcopy(self.model.state_dict())

        total_epochs += 1
```

Figure 10: Writing the results to create a tensorboard dashboard

4 Executing The Adversarial Attacks

In this research, we execute three different types of gradient attack methods namely Fast-Gradient Sign Method(FGSM), Projected Gradient Descent (PGD) and Carlini and Wagner Method (C&W method). We execute the attacks in two stages. In the first stage, we clone the adversarial images and execute simulated attacks (FGSM, PGD and CW). The details for all of these attacks are provided below:

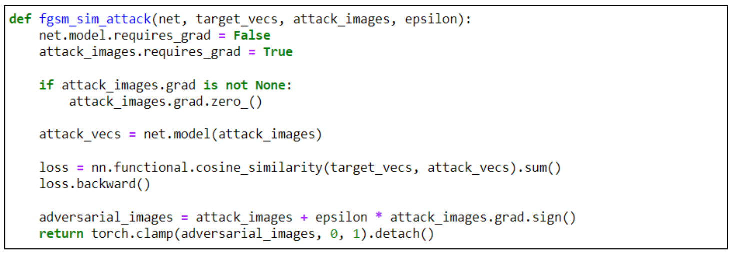
[Note: We use some of the helper functions for calculating the cosine distance between the images and for projecting the images to and from the feature space These helper functions were defined in the helpers.py file inside the src folder.]



```python
import torch
import torch.nn as nn

from tqdm import tqdm
from torch import optim

from src.helpers import cos_dist, to_tanh_space, to_original_space
```

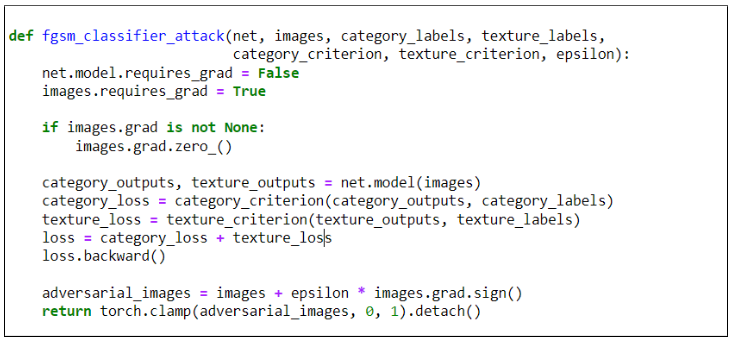
Figure 11: Using helper functions

1. **Fast Gradient Sign Method:** We assess the effectiveness of this method in minimizing the cosine distances between the image embeddings produced by MobileNetV2 feature extractor. We execute the attack for different value of k and *epsilon*.



```python
def fgsm_sim_attack(net, target_vecs, attack_images, epsilon):
    net.model.requires_grad = False
    attack_images.requires_grad = True

    if attack_images.grad is not None:
        attack_images.grad.zero_()

    attack_vecs = net.model(attack_images)

    loss = nn.functional.cosine_similarity(target_vecs, attack_vecs).sum()
    loss.backward()

    adversarial_images = attack_images + epsilon * attack_images.grad.sign()
    return torch.clamp(adversarial_images, 0, 1).detach()
```

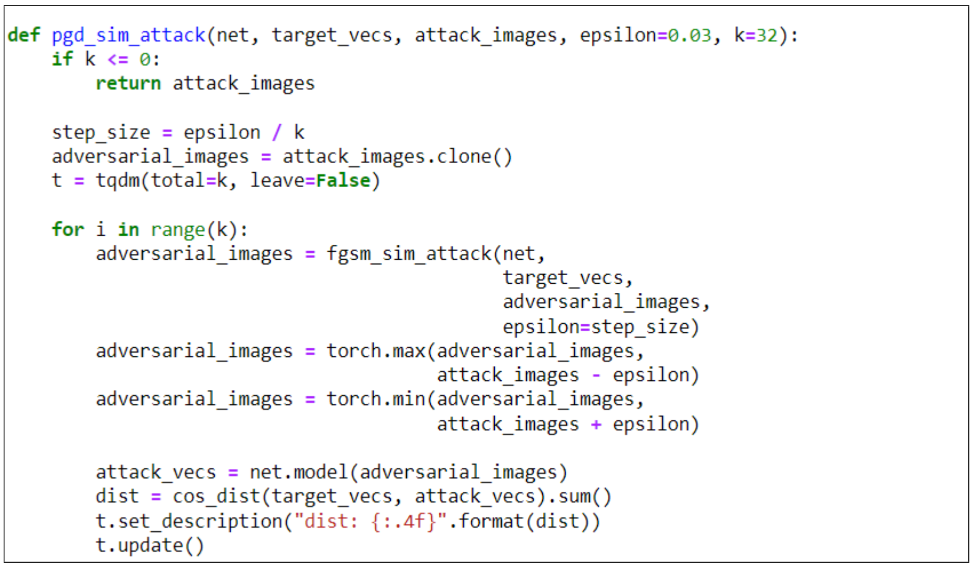
Figure 12



```python
def fgsm_classifier_attack(net, images, category_labels, texture_labels,
                           category_criterion, texture_criterion, epsilon):
    net.model.requires_grad = False
    images.requires_grad = True

    if images.grad is not None:
        images.grad.zero_()

    category_outputs, texture_outputs = net.model(images)
    category_loss = category_criterion(category_outputs, category_labels)
    texture_loss = texture_criterion(texture_outputs, texture_labels)
    loss = category_loss + texture_loss
    loss.backward()

    adversarial_images = images + epsilon * images.grad.sign()
    return torch.clamp(adversarial_images, 0, 1).detach()
```

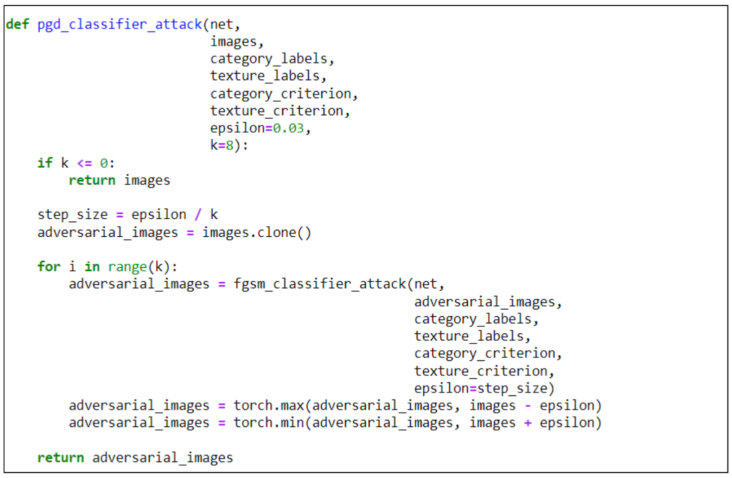
Figure 13

2. **Projected Gradient Descent Method:** For the PGD attack, we use a step size of $\alpha = \frac{\epsilon}{k}$, where k denotes the no. of iterations. We use different values of k to optimize our adversarial objective.



```python
def pgd_sim_attack(net, target_vecs, attack_images, epsilon=0.03, k=32):
    if k <= 0:
        return attack_images

    step_size = epsilon / k
    adversarial_images = attack_images.clone()
    t = tqdm(total=k, leave=False)

    for i in range(k):
        adversarial_images = fgsm_sim_attack(net,
                                             target_vecs,
                                             adversarial_images,
                                             epsilon=step_size)
        adversarial_images = torch.max(adversarial_images,
                                       attack_images - epsilon)
        adversarial_images = torch.min(adversarial_images,
                                       attack_images + epsilon)

        attack_vecs = net.model(adversarial_images)
        dist = cos_dist(target_vecs, attack_vecs).sum()
        t.set_description("dist: {:.4f}".format(dist))
        t.update()
```

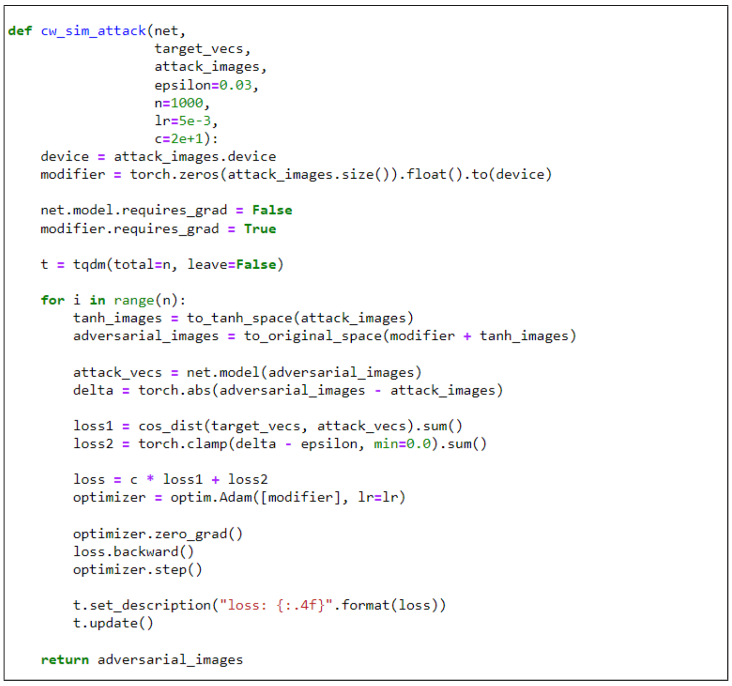
Figure 14



```python
def pgd_classifier_attack(net,
                          images,
                          category_labels,
                          texture_labels,
                          category_criterion,
                          texture_criterion,
                          epsilon=0.03,
                          k=8):
    if k <= 0:
        return images

    step_size = epsilon / k
    adversarial_images = images.clone()

    for i in range(k):
        adversarial_images = fgsm_classifier_attack(net,
                                                    adversarial_images,
                                                    category_labels,
                                                    texture_labels,
                                                    category_criterion,
                                                    texture_criterion,
                                                    epsilon=step_size)
        adversarial_images = torch.max(adversarial_images, images - epsilon)
        adversarial_images = torch.min(adversarial_images, images + epsilon)

    return adversarial_images
```

Figure 15

3. **Carlini and Wagner Method:** In the last step, we executed the strongest attack of them all, the CW attack. For this attack, we don't maximize the misclassification, but instead we use an Adam Optimizer with a learning rate of 0.005 and 1000 steps for achieving our adversary objective.

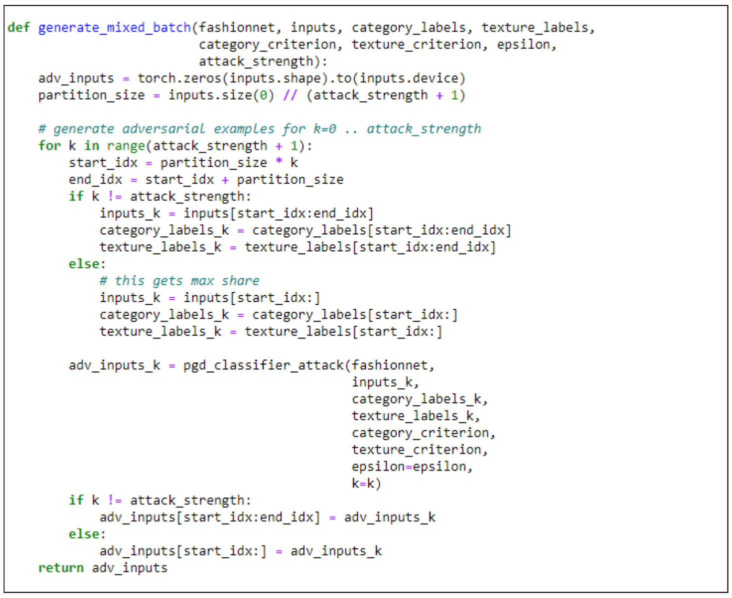


```python
def cw_sim_attack(net,
                  target_vecs,
                  attack_images,
                  epsilon=0.03,
                  n=1000,
                  lr=5e-3,
                  c=2e+1):
    device = attack_images.device
    modifier = torch.zeros(attack_images.size()).float().to(device)

    net.model.requires_grad = False
    modifier.requires_grad = True

    t = tqdm(total=n, leave=False)

    for i in range(n):
        tanh_images = to_tanh_space(attack_images)
        adversarial_images = to_original_space(modifier + tanh_images)

        attack_vecs = net.model(adversarial_images)
        delta = torch.abs(adversarial_images - attack_images)

        loss1 = cos_dist(target_vecs, attack_vecs).sum()
        loss2 = torch.clamp(delta - epsilon, min=0.0).sum()

        loss = c * loss1 + loss2
        optimizer = optim.Adam([modifier], lr=lr)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        t.set_description("loss: {:.4f}".format(loss))
        t.update()

    return adversarial_images
```

Figure 16

5 Defenses

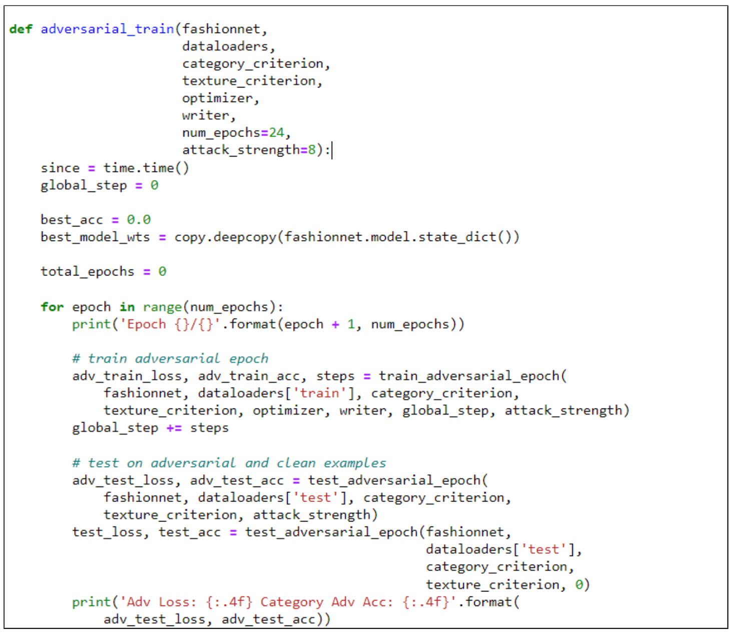
In the last stage of this research, we train our feature extractor through two different defense mechanisms - Adversarial training and Curriculum Adversarial training. We start by generating a mixed batch of texture and category labels (figure 17)



```python
def generate_mixed_batch(fashionnet, inputs, category_labels, texture_labels,
                         category_criterion, texture_criterion, epsilon,
                         attack_strength):
    adv_inputs = torch.zeros(inputs.shape).to(inputs.device)
    partition_size = inputs.size(0) // (attack_strength + 1)

    # generate adversarial examples for k=0 .. attack_strength
    for k in range(attack_strength + 1):
        start_idx = partition_size * k
        end_idx = start_idx + partition_size
        if k != attack_strength:
            inputs_k = inputs[start_idx:end_idx]
            category_labels_k = category_labels[start_idx:end_idx]
            texture_labels_k = texture_labels[start_idx:end_idx]
        else:
            # this gets max share
            inputs_k = inputs[start_idx:]
            category_labels_k = category_labels[start_idx:]
            texture_labels_k = texture_labels[start_idx:]

        adv_inputs_k = pgd_classifier_attack(fashionnet,
                                             inputs_k,
                                             category_labels_k,
                                             texture_labels_k,
                                             category_criterion,
                                             texture_criterion,
                                             epsilon=epsilon,
                                             k=k)
        if k != attack_strength:
            adv_inputs[start_idx:end_idx] = adv_inputs_k
        else:
            adv_inputs[start_idx:] = adv_inputs_k
    return adv_inputs
```

Figure 17: Batch Mixing before performing adversarial training

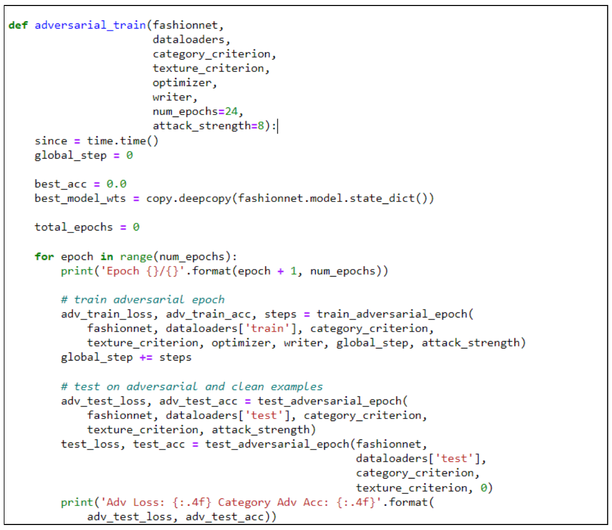
1. **Adversarial Training:** Our goal during the adversarial training is to increase the probability of mis-classification of the texture and category of the items. We performed AT for 24 epochs. It is then tested on adversarial and clean examples and the evaluation metrics (loss, accuracy and success rates) were stored in appropriate pytorch objects.



```python
def adversarial_train(fashionnet,
                      dataloaders,
                      category_criterion,
                      texture_criterion,
                      optimizer,
                      writer,
                      num_epochs=24,
                      attack_strength=8):
    since = time.time()
    global_step = 0

    best_acc = 0.0
    best_model_wts = copy.deepcopy(fashionnet.model.state_dict())

    total_epochs = 0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch + 1, num_epochs))

        # train adversarial epoch
        adv_train_loss, adv_train_acc, steps = train_adversarial_epoch(
            fashionnet, dataloaders['train'], category_criterion,
            texture_criterion, optimizer, writer, global_step, attack_strength)
        global_step += steps

        # test on adversarial and clean examples
        adv_test_loss, adv_test_acc = test_adversarial_epoch(
            fashionnet, dataloaders['test'], category_criterion,
            texture_criterion, attack_strength)
        test_loss, test_acc = test_adversarial_epoch(fashionnet,
                                                     dataloaders['test'],
                                                     category_criterion,
                                                     texture_criterion, 0)
        print('Adv Loss: {:.4f} Category Adv Acc: {:.4f}'.format(
            adv_test_loss, adv_test_acc))
```

Figure 18: Adversarial Training

2. **Curriculum Adversarial Training:** The curriculum adversarial training (CAT) is another defense mechanism that we used. We implemented this defense method by training the model using projected gradient descent attacks with k=8 and $\epsilon = 0.03$. We train the model until overfitting for a given attack strength.

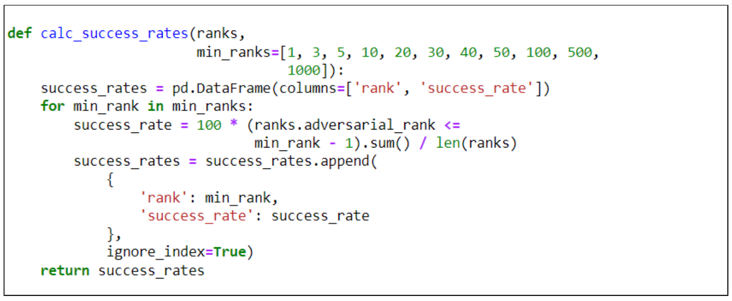


```python
def adversarial_train(fashionnet,
                      dataloaders,
                      category_criterion,
                      texture_criterion,
                      optimizer,
                      writer,
                      num_epochs=24,
                      attack_strength=8):
    since = time.time()
    global_step = 0

    best_acc = 0.0
    best_model_wts = copy.deepcopy(fashionnet.model.state_dict())

    total_epochs = 0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch + 1, num_epochs))

        # train adversarial epoch
        adv_train_loss, adv_train_acc, steps = train_adversarial_epoch(
            fashionnet, dataloaders['train'], category_criterion,
            texture_criterion, optimizer, writer, global_step, attack_strength)
        global_step += steps

        # test on adversarial and clean examples
        adv_test_loss, adv_test_acc = test_adversarial_epoch(
            fashionnet, dataloaders['test'], category_criterion,
            texture_criterion, attack_strength)
        test_loss, test_acc = test_adversarial_epoch(fashionnet,
                                                     dataloaders['test'],
                                                     category_criterion,
                                                     texture_criterion, 0)
        print('Adv Loss: {:.4f} Category Adv Acc: {:.4f}'.format(
            adv_test_loss, adv_test_acc))
```

Figure 19: Curriculum Adversarial Training

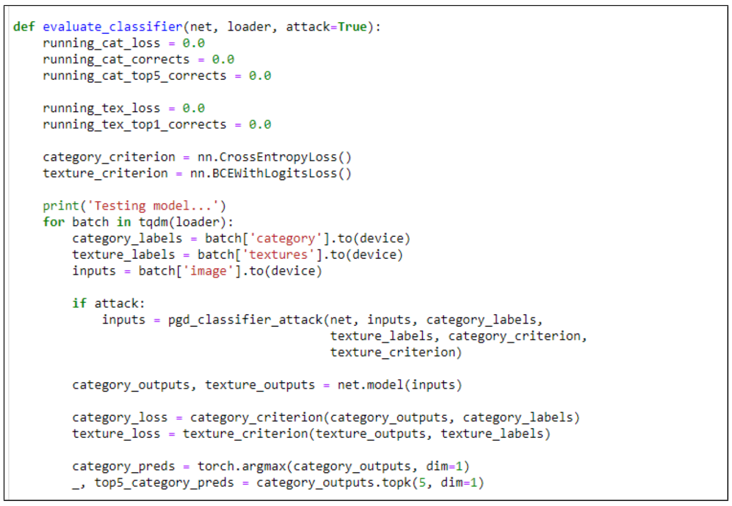
6 Evaluation

The gradient based attack models were evaluated using success rate metric for different values of ranks and the results were appended into a pandas dataframe. The classifier was also evaluated on the basis of loss and accuracy metrics.



```python
def calc_success_rates(ranks,
                       min_ranks=[1, 3, 5, 10, 20, 30, 40, 50, 100, 500,
                                  1000]):
    success_rates = pd.DataFrame(columns=['rank', 'success_rate'])
    for min_rank in min_ranks:
        success_rate = 100 * (ranks.adversarial_rank <=
                              min_rank - 1).sum() / len(ranks)
        success_rates = success_rates.append(
            {
                'rank': min_rank,
                'success_rate': success_rate
            },
            ignore_index=True)
    return success_rates
```

Figure 20: Evaluating Adversarial Attacks



```python
def evaluate_classifier(net, loader, attack=True):
    running_cat_loss = 0.0
    running_cat_corrects = 0.0
    running_cat_top5_corrects = 0.0

    running_tex_loss = 0.0
    running_tex_top1_corrects = 0.0

    category_criterion = nn.CrossEntropyLoss()
    texture_criterion = nn.BCEWithLogitsLoss()

    print('Testing model...')
    for batch in tqdm(loader):
        category_labels = batch['category'].to(device)
        texture_labels = batch['textures'].to(device)
        inputs = batch['image'].to(device)

        if attack:
            inputs = pgd_classifier_attack(net, inputs, category_labels,
                                           texture_labels, category_criterion,
                                           texture_criterion)

        category_outputs, texture_outputs = net.model(inputs)

        category_loss = category_criterion(category_outputs, category_labels)
        texture_loss = texture_criterion(texture_outputs, texture_labels)

        category_preds = torch.argmax(category_outputs, dim=1)
        _, top5_category_preds = category_outputs.topk(5, dim=1)
```

Figure 21: Evaluating CNN classifier using loss and accuracy