Designing a backdoor detector for BadNets trained on the YouTube Face dataset using the pruning defense.

```
# All necessary imports
import os
import tarfile
import requests
import re
import sys
import warnings
warnings.filterwarnings('ignore')
import h5py
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras import backend as K
from keras.models import Model
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
import matplotlib.font_manager as font_manager
import cv2
```

Define function to load the data

```
# Load data
def data_loader(filepath):
    data = h5py.File(filepath, 'r')
    x_data = np.array(data['data'])
    y_data = np.array(data['label'])
    x_data = x_data.transpose((0,2,3,1))
    return x_data, y_data
```

Model Architecture

```
def Net():
    # define input
    x = keras.Input(shape=(55, 47, 3), name='input')
    # feature extraction
    conv_1 = keras.layers.Conv2D(20, (4, 4), activation='relu', name='conv_1')(x)
    pool_1 = keras.layers.MaxPooling2D((2, 2), name='pool_1')(conv_1)
    conv_2 = keras.layers.Conv2D(40, (3, 3), activation='relu', name='conv_2')(pool_1)
    pool_2 = keras.layers.MaxPooling2D((2, 2), name='pool_2')(conv_2)
    conv_3 = keras.layers.Conv2D(60, (3, 3), activation='relu', name='conv_3')(pool_2)
    pool_3 = keras.layers.MaxPooling2D((2, 2), name='pool_3')(conv_3)
    # first interpretation model
    flat_1 = keras.layers.Flatten()(pool_3)
```

```
    fc_1 = keras.layers.Dense(160, name='fc_1')(flat_1)
    # second interpretation model
    conv_4 = keras.layers.Conv2D(80, (2, 2), activation='relu', name='conv_4')(pool_3)
    flat_2 = keras.layers.Flatten()(conv_4)
    fc_2 = keras.layers.Dense(160, name='fc_2')(flat_2)
    # merge interpretation
    merge = keras.layers.Add()([fc_1, fc_2])
    add_1 = keras.layers.Activation('relu')(merge)
    drop = keras.layers.Dropout(0.5)
    # output
    y_hat = keras.layers.Dense(1283, activation='softmax', name='output')(add_1)
    model = keras.Model(inputs=x, outputs=y_hat)
    # summarize layers
    #print(model.summary())
    # plot graph
    #plot_model(model, to_file='model_architecture.png')

    return model
```

Follow instructions under [Data Section]() to download the datasets.

We will be using the clean validation data (valid.h5) from cl folder to design the defense and clean test data (test.h5 from cl folder) and sunglasses poisoned test data (bd_test.h5 from bd folder) to evaluate the models.

```
## To-do ##
# After downloading the datasets, provide corresponding filepaths below

clean_data_valid_filename =  '/content/drive/MyDrive/lab3/data/cl/valid.h5'

clean_data_test_filename = '/content/drive/MyDrive/lab3/data/cl/test.h5'
poisoned_data_test_filename =  '/content/drive/MyDrive/lab3/data/bd/bd_test.h5'
```

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

```
    Mounted at /content/drive
```

Read the data:

```
cl_x_valid, cl_y_valid = data_loader(clean_data_valid_filename)

cl_x_test, cl_y_test = data_loader(clean_data_test_filename)
bd_x_test, bd_y_test = data_loader(poisoned_data_test_filename)
```

Visualizing the clean test data

```
# Plot some images from the validation set (see https://mrdatascience.com/how-to-plot-mnist-digits-using-matplotl
num = 10
```

```
np.random.seed(45)
randIdx = [np.random.randint(10000) for i in range(num)]
num_row = 2
num_col = 5# plot images
fig, axes = plt.subplots(num_row, num_col, figsize=(3*num_col,3*num_row))
for i in range(num):
    ax = axes[i//num_col, i%num_col]
    ax.imshow(cl_x_test[randIdx[i]].astype('uint8'))
    ax.set_title('label: {:.0f}'.format(cl_y_test[randIdx[i]]))
    ax.set_xticks([])
    ax.set_yticks([])
plt.tight_layout()
plt.show()
```

Visualizing the sunglasses poisioned test data

```
# Plot some images from the validation set (see https://mrdatascience.com/how-to-plot-mnist-digits-using-matplotl
num = 10
np.random.seed(45)
randIdx = [np.random.randint(10000) for i in range(num)]
num_row = 2
num_col = 5# plot images
fig, axes = plt.subplots(num_row, num_col, figsize=(3*num_col,3*num_row))
for i in range(num):
    ax = axes[i//num_col, i%num_col]
    ax.imshow(bd_x_test[randIdx[i]].astype('uint8'))
    ax.set_title('label: {:.0f}'.format(bd_y_test[randIdx[i]]))
    ax.set_xticks([])
    ax.set_yticks([])
```

```
plt.tight_layout()
plt.show()
```

Load the backdoored model.

The backdoor model and its weights can be found here

```
## To-do ##

# First create clones of the original badnet model (by providing the model filepath below)
# The result of repairing B_clone will be B_prime

B = keras.models.load_model( '/content/drive/MyDrive/lab3/models/bd_net.h5')
B.load_weights( '/content/drive/MyDrive/lab3/models/bd_weights.h5')

B_clone = keras.models.load_model(( '/content/drive/MyDrive/lab3/models/bd_net.h5'))
B_clone.load_weights( '/content/drive/MyDrive/lab3/models/bd_weights.h5')
```

Output of the original badnet accuracy on the validation data:

```
# Get the original badnet model's (B) accuracy on the validation data
cl_label_p = np.argmax(B(cl_x_valid), axis=1)
clean_accuracy = np.mean(np.equal(cl_label_p, cl_y_valid)) * 100

print("Clean validation accuracy before pruning {0:3.6f}".format(clean_accuracy))
K.clear_session()
```

```
Clean validation accuracy before pruning 98.649000
```

```
B.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input (InputLayer)             [(None, 55, 47, 3)]  0           []

 conv_1 (Conv2D)                (None, 52, 44, 20)   980         ['input[0][0]']

 pool_1 (MaxPooling2D)          (None, 26, 22, 20)   0           ['conv_1[0][0]']

 conv_2 (Conv2D)                (None, 24, 20, 40)   7240        ['pool_1[0][0]']

 pool_2 (MaxPooling2D)          (None, 12, 10, 40)   0           ['conv_2[0][0]']

 conv_3 (Conv2D)                (None, 10, 8, 60)    21660       ['pool_2[0][0]']

 pool_3 (MaxPooling2D)          (None, 5, 4, 60)     0           ['conv_3[0][0]']

 conv_4 (Conv2D)                (None, 4, 3, 80)     19280       ['pool_3[0][0]']

 flatten_1 (Flatten)            (None, 1200)         0           ['pool_3[0][0]']

 flatten_2 (Flatten)            (None, 960)          0           ['conv_4[0][0]']

 fc_1 (Dense)                   (None, 160)          192160      ['flatten_1[0][0]']

 fc_2 (Dense)                   (None, 160)          153760      ['flatten_2[0][0]']

 add_1 (Add)                    (None, 160)          0           ['fc_1[0][0]',
                                                                  'fc_2[0][0]']

 activation_1 (Activation)      (None, 160)          0           ['add_1[0][0]']

 output (Dense)                 (None, 1283)         206563      ['activation_1[0][0]']

==================================================================================================
Total params: 601,643
Trainable params: 601,643
Non-trainable params: 0
__________________________________________________________________________________________________
```

Write code to implement pruning defense

```
## To-do ##

# Redefine model to output right after the last pooling layer ("pool_3")
intermediate_model = Model(inputs=B.inputs, outputs=B.get_layer('pool_3').output)

# Get feature map for last pooling layer ("pool_3") using the clean validation data and intermediate_model
feature_maps_cl =intermediate_model.predict(cl_x_valid)

# Get average activation value of each channel in last pooling layer ("pool_3")
averageActivationsCl = np.mean(np.array(feature_maps_cl), axis=0)
```

```
361/361 [==============================] - 12s 32ms/step
```

```
# Store the indices of average activation values (averageActivationsCl) in increasing order
idxToPrune = np.argsort(np.sum(averageActivationsCl, axis=(0, 1)))
print(idxToPrune)
```

```
[ 0 26 27 30 31 33 34 36 37 38 25 39 41 44 45 47 48 49 50 53 55 40 24 59
  9  2 12 13 17 14 15 23  6 51 32 22 21 20 19 43 58  3 42  1 29 16 56 46
  5  8 11 54 10 28 35 18  4  7 52 57]
```

```
print(np.sort(np.sum(averageActivationsCl, axis=(0, 1))))
```

```
[0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
 0.0000000e+00 6.0581528e-02 1.2481733e-01 2.6643127e-01 3.0013326e-01
 8.7959361e-01 1.6707796e+00 3.6756399e+00 4.8763165e+00 8.5526381e+00
 1.0146558e+01 1.0615954e+01 1.1531752e+01 1.7157463e+01 2.1179457e+01
 3.1309752e+01 3.2705936e+01 3.7080822e+01 4.0579247e+01 4.2212044e+01
 4.3960865e+01 7.2382416e+01 8.2977959e+01 9.6880943e+01 9.7297119e+01
 1.0173821e+02 1.0290282e+02 1.0738125e+02 1.2407619e+02 1.6446213e+02]
```

```
# Get the conv_4 layer weights and biases from the original network that will be used for prunning
# Hint: Use the get_weights() method (https://stackoverflow.com/questions/43715047/how-do-i-get-the-weights-c
conv3_layer = B.get_layer('conv_3')
lastConvLayerWeights ,lastConvLayerBiases  =  conv3_layer.get_weights()
bclone_conv3_layer = B_clone.get_layer('conv_3')
print(len(lastConvLayerWeights),len(lastConvLayerBiases))
```

```
3 60
```

```
n = len(idxToPrune)
n
```

```
60
```

```
acc=[]
asr=[]
for idx in range(30,n):
  cur_idx = idxToPrune[idx]
  lastConvLayerWeights[:,:,:,cur_idx] = 0
  lastConvLayerBiases[idx] = 0
  bclone_conv3_layer.set_weights([lastConvLayerWeights,lastConvLayerBiases])
  print('epoch',idx+1)
  cl_label_p_valid = np.argmax(B_clone.predict(cl_x_valid), axis=1)
  clean_accuracy_valid = np.mean(np.equal(cl_label_p_valid, cl_y_valid)) * 100
  acc.append(clean_accuracy_valid)
  posion_p_valid = np.argmax(B_clone.predict(bd_x_test), axis=1)
  attack_accuracy_valid = np.mean(np.equal(posion_p_valid, bd_y_test)) * 100
  asr.append(attack_accuracy_valid)
  print('Accuracy:', clean_accuracy_valid)
```

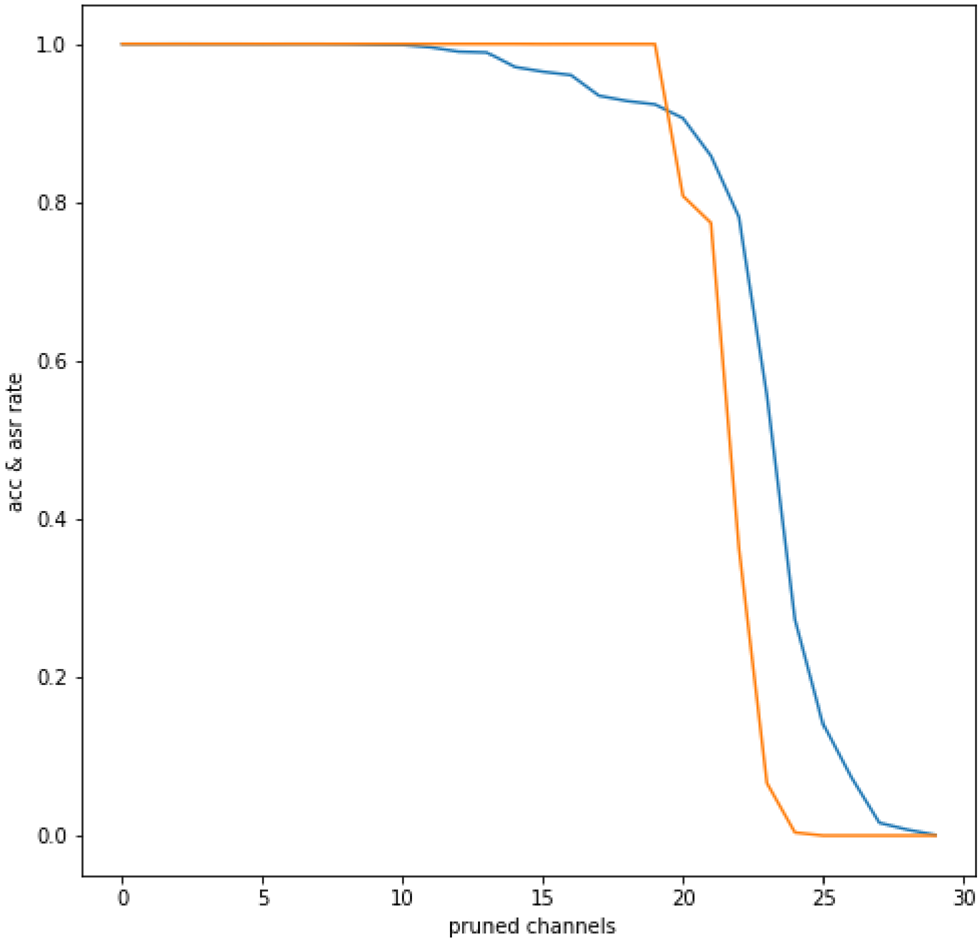
```
    print('Attack Success Rate:', attack_accuracy_valid)
    if idx == 45:
      B_clone.save('/content/drive/MyDrive/lab3/models/B1'+ '_2' +'.h5')
    if idx == 47:
     B_clone.save('/content/drive/MyDrive/lab3/models/B1'+ '_4' +'.h5')
    if idx == 51:
      B_clone.save('/content/drive/MyDrive/lab3/models/B1'+ '_10' +'.h5')
```

```
401/401 [==============================] - 12s 31ms/step
Accuracy: 91.53026760197453
Attack Success Rate: 99.98441153546376
epoch 50
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 91.10591495626569
Attack Success Rate: 99.97661730319564
epoch 51
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 89.38252359920325
Attack Success Rate: 80.77162899454405
epoch 52
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 84.68866372217893
Attack Success Rate: 77.36554949337491
epoch 53
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 77.05031609941976
Attack Success Rate: 36.30553390491036
epoch 54
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 54.845414393348925
Attack Success Rate: 6.632891660171474
epoch 55
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 26.907421841170866
Attack Success Rate: 0.3897116134060795
epoch 56
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 15s 37ms/step
Accuracy: 13.87373343725643
Attack Success Rate: 0.0
epoch 57
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 7.2659565255044605
Attack Success Rate: 0.0
epoch 58
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 1.5934874859270804
Attack Success Rate: 0.0
epoch 59
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
```

```
Accuracy: 0.7361219364337057
Attack Success Rate: 0.0
epoch 60
361/361 [==============================] - 11s 31ms/step
401/401 [==============================] - 12s 31ms/step
Accuracy: 0.0779423226812159
Attack Success Rate: 0.0
```

```
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 8))
plt.plot(acc / acc[0], label='acc')
plt.plot(asr / asr[0], label='asr')
plt.xlabel(' pruned channels')
plt.ylabel('acc & asr rate')
```

```
Text(0, 0.5, 'acc & asr rate')
```



If I prune from the index 0 to the all total 60, the colab will crush at epcho 30 - 40. But we can observe that there is no big change before first 30 prunes, to avoid session crush I will start the loop from 30 to 60.

From the result we can see, when accuracy drop to 2% is epcho 45, 4% is epcho 47,10% is epcho 51. Then we will start to save the model.

```
# for idx in range(30,n):
#   cur_idx = idxToPrune[idx]
```

```
#   lastConvLayerWeights[:,:,:,cur_idx] = 0
#   lastConvLayerBiases[idx] = 0
#   bclone_conv3_layer.set_weights([lastConvLayerWeights,lastConvLayerBiases])
#   print('epoch',idx+1)
#   if i == 45:
#      B_clone.save('/content/drive/MyDrive/cyber/lab3/models/B1'+ '_2' +'.h5')
#   if i == 48:
#     B_clone.save('/content/drive/MyDrive/cyber/lab3/models/B1'+ '_4' +'.h5')
#   if i == 52:
#      B_clone.save('/content/drive/MyDrive/cyber/lab3/models/B1'+ '_10' +'.h5')
```

Now we need to combine the models into a repaired goodnet G that outputs the correct class if the test input is clean and class N+1 if the input is backdoored. One way to do it is to "subclass" the models in Keras:

```
#https://stackoverflow.com/questions/64983112/keras-vertical-ensemble-model-with-condition-in-between
class G(tf.keras.Model):
   def __init__(self, B, B_prime):
      super(G, self).__init__()
      self.B = B
      self.B_prime = B_prime

   def predict(self,data):
      y = np.argmax(self.B(data), axis=1)
      y_prime = np.argmax(self.B_prime(data), axis=1)
      tmpRes = np.array([y[i] if y[i] == y_prime[i] else 1283 for i in range(y.shape[0])])
      res = np.zeros((y.shape[0],1284))
      res[np.arange(tmpRes.size),tmpRes] = 1
      return res

   # For small amount of inputs that fit in one batch, directly using call() is recommended for faster execution,
   # e.g., model(x), or model(x, training=False) is faster then model.predict(x) and do not result in
   # memory leaks (see for more details https://www.tensorflow.org/api_docs/python/tf/keras/Model#predict)
   def call(self,data):
      y = np.argmax(self.B(data), axis=1)
      y_prime = np.argmax(self.B_prime(data), axis=1)
      tmpRes = np.array([y[i] if y[i] == y_prime[i] else 1283 for i in range(y.shape[0])])
      res = np.zeros((y.shape[0],1284))
      res[np.arange(tmpRes.size),tmpRes] = 1
      return res
```

However, Keras prevents from saving this kind of subclassed model as HDF5 file since it is not serializable. However, we still can use this architecture for model evaluation.

Load the saved B_prime model

```
## To-do ##
# Provide B_prime model filepath below
```

```
B_prime = keras.models.load_model("/content/drive/MyDrive/lab3/models/B1_2.h5")
# B_prime.load_weights("")
```

Check performance of the repaired model on the test data:

```
cl_label_p = np.argmax(B_prime.predict(cl_x_test), axis=1)
clean_accuracy_B_prime = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for B_prime:', clean_accuracy_B_prime)

bd_label_p = np.argmax(B_prime.predict(bd_x_test), axis=1)
asr_B_prime = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for B_prime:', asr_B_prime)
```

```
401/401 [==============================] - 22s 54ms/step
Clean Classification accuracy for B_prime: 95.57287607170693
401/401 [==============================] - 14s 35ms/step
Attack Success Rate for B_prime: 99.97661730319564
```

Check performance of the original model on the test data:

```
cl_label_p = np.argmax(B.predict(cl_x_test), axis=1)
clean_accuracy_B = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for B:', clean_accuracy_B)

bd_label_p = np.argmax(B.predict(bd_x_test), axis=1)
asr_B = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for B:', asr_B)
```

```
401/401 [==============================] - 13s 32ms/step
Clean Classification accuracy for B: 98.62042088854248
401/401 [==============================] - 13s 31ms/step
Attack Success Rate for B: 100.0
```

Create repaired network

```
# Repaired network repaired_net
repaired_net = G(B, B_prime)
```

Check the performance of the repaired_net on the test data

```
cl_label_p = np.argmax(repaired_net(cl_x_test), axis=1)
clean_accuracy_repaired_net = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for repaired net:', clean_accuracy_repaired_net)

bd_label_p = np.argmax(repaired_net(bd_x_test), axis=1)
asr_repaired_net = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for repaired net:', asr_repaired_net)
```

```
Clean Classification accuracy for repaired net: 95.40919719407638
Attack Success Rate for repaired net: 99.97661730319564
```

For 4%:

```
B_prime4 = keras.models.load_model("/content/drive/MyDrive/lab3/models/B1_4.h5")
```

```
cl_label_p = np.argmax(B_prime4.predict(cl_x_test), axis=1)
clean_accuracy_B_prime = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for B_prime:', clean_accuracy_B_prime)

bd_label_p = np.argmax(B_prime4.predict(bd_x_test), axis=1)
asr_B_prime = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for B_prime:', asr_B_prime)
```

```
401/401 [==============================] - 20s 49ms/step
Clean Classification accuracy for B_prime: 92.33047544816836
401/401 [==============================] - 13s 33ms/step
Attack Success Rate for B_prime: 99.98441153546376
```

```
cl_label_p = np.argmax(B.predict(cl_x_test), axis=1)
clean_accuracy_B = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for B:', clean_accuracy_B)

bd_label_p = np.argmax(B.predict(bd_x_test), axis=1)
asr_B = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for B:', asr_B)
```

```
401/401 [==============================] - 13s 33ms/step
Clean Classification accuracy for B: 98.62042088854248
401/401 [==============================] - 13s 32ms/step
Attack Success Rate for B: 100.0
```

```
# Repaired network repaired_net
repaired_net4 = G(B, B_prime4)
```

```
cl_label_p = np.argmax(repaired_net4(cl_x_test), axis=1)
clean_accuracy_repaired_net = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for repaired net:', clean_accuracy_repaired_net)

bd_label_p = np.argmax(repaired_net4(bd_x_test), axis=1)
asr_repaired_net = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for repaired net:', asr_repaired_net)
```

```
Clean Classification accuracy for repaired net: 92.19017926734216
Attack Success Rate for repaired net: 99.98441153546376
```

for 10 %:

```
B_prime10 = keras.models.load_model("/content/drive/MyDrive/lab3/models/B1_10.h5")
```

```
cl_label_p = np.argmax(B_prime10.predict(cl_x_test), axis=1)
clean_accuracy_B_prime = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for B_prime:', clean_accuracy_B_prime)

bd_label_p = np.argmax(B_prime10.predict(bd_x_test), axis=1)
asr_B_prime = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for B_prime:', asr_B_prime)
```

```
401/401 [==============================] - 13s 33ms/step
Clean Classification accuracy for B_prime: 84.94154325798908
401/401 [==============================] - 17s 42ms/step
Attack Success Rate for B_prime: 77.36554949337491
```

```
cl_label_p = np.argmax(B.predict(cl_x_test), axis=1)
clean_accuracy_B = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for B:', clean_accuracy_B)

bd_label_p = np.argmax(B.predict(bd_x_test), axis=1)
asr_B = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for B:', asr_B)
```

```
401/401 [==============================] - 14s 35ms/step
Clean Classification accuracy for B: 98.62042088854248
401/401 [==============================] - 14s 35ms/step
Attack Success Rate for B: 100.0
```

```
# Repaired network repaired_net
repaired_net10 = G(B, B_prime)
```

```
cl_label_p = np.argmax(repaired_net10(cl_x_test), axis=1)
clean_accuracy_repaired_net = np.mean(np.equal(cl_label_p, cl_y_test))*100
print('Clean Classification accuracy for repaired net:', clean_accuracy_repaired_net)

bd_label_p = np.argmax(repaired_net10(bd_x_test), axis=1)
asr_repaired_net = np.mean(np.equal(bd_label_p, bd_y_test))*100
print('Attack Success Rate for repaired net:', asr_repaired_net)
```

```
Clean Classification accuracy for repaired net: 95.40919719407638
Attack Success Rate for repaired net: 99.97661730319564
```

46s completed at 8:24 PM