```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

     Mounted at /content/drive
```

```
! unzip /content/drive/MyDrive/lab2data/bd.zip -d /content/drive/MyDrive/lab2data
! unzip /content/drive/MyDrive/lab2data/cl.zip -d /content/drive/MyDrive/lab2data

     Archive:  /content/drive/MyDrive/lab2data/bd.zip
     replace /content/drive/MyDrive/lab2data/bd/bd_test.h5? [y]es, [n]o, [A]ll, [N]one, [r]ename: Archive:  /content/drive/MyDrive/lab2data/c
     replace /content/drive/MyDrive/lab2data/cl/test.h5? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

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
```

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/lab2data/cl/valid.h5"

clean_data_test_filename = "/content/drive/MyDrive/lab2data/cl/test.h5"
poisoned_data_test_filename = "/content/drive/MyDrive/lab2data/bd/bd_test.h5"
```

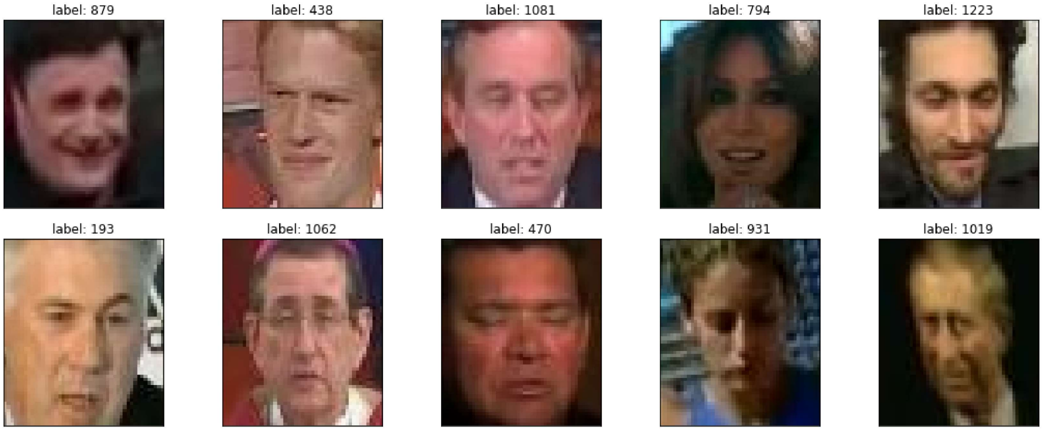
Read the data:

```
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-matplotlib/)
num = 10
np.random.seed(65)
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-matplotlib/)
num = 10
np.random.seed(65)
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/lab2data/bd_net.h5")
B.load_weights("/content/drive/MyDrive/lab2data/bd_weights.h5")

B_clone = keras.models.load_model("/content/drive/MyDrive/lab2data/bd_net.h5")
B_clone.load_weights("/content/drive/MyDrive/lab2data/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
```

Write code to implement pruning defense

```
## To-do ##

models_saved = np.zeros(3,dtype=bool)

# 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(feature_maps_cl,axis=(0,1,2))

# Store the indices of average activation values (averageActivationsCl) in increasing order
idxToPrune = np.argsort(averageActivationsCl)


# Get the conv_3 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-of-a-layer-in-keras)
lastConvLayerWeights = B_clone.layers[5].get_weights()[0]
lastConvLayerBiases  = B_clone.layers[5].get_weights()[1]

for chIdx in idxToPrune:

  # Prune one channel at a time
  # Hint: Replace all values in channel 'chIdx' of lastConvLayerWeights and lastConvLayerBiases with 0
  lastConvLayerWeights[:,:,:,chIdx] = 0
  lastConvLayerBiases[chIdx] = 0

  # Update weights and biases of B_clone
  # Hint: Use the set_weights() method
  B_clone.layers[5].set_weights([lastConvLayerWeights, lastConvLayerBiases])

  # Evaluate the updated model's (B_clone) clean validation accuracy
  cl_label_p_valid = np.argmax(B_clone(cl_x_valid), axis=1)
  clean_accuracy_valid = np.mean(np.equal(cl_label_p_valid, cl_y_valid)) * 100
  # If drop in clean_accuracy_valid is just greater (or equal to) than the desired threshold compared to clean_accuracy, then save B_clone as
  if (clean_accuracy-clean_accuracy_valid >=2 and not models_saved[0]):
    print("The accuracy drops at least 2%,model saved")
    B_clone.save('modelX2.h5')
    models_saved[0] = 1
  if (clean_accuracy-clean_accuracy_valid >=4 and not models_saved[1]):
    print("The accuracy drops at least 4%, model saved")
    B_clone.save('modelX4.h5')
    models_saved[1] = 1
  if (clean_accuracy-clean_accuracy_valid >=10 and not models_saved[2]):
    print("The accuracy drops at least 10%,model saved")
    B_clone.save('modelX10.h5')
    models_saved[2] = 1
    # Save B_clone as B_prime and break
    break
```

```
361/361 [==============================] - 1s 2ms/step
The accuracy drops at least 2%,model saved
The accuracy drops at least 4%, model saved
The accuracy drops at least 10%,model saved
```

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_primeX2 = keras.models.load_model("/content/modelX2.h5")
B_primeX2.load_weights("/content/modelX2.h5")
B_primeX4 = keras.models.load_model("/content/modelX4.h5")
B_primeX4.load_weights("/content/modelX4.h5")
B_primeX10 = keras.models.load_model("/content/modelX10.h5")
B_primeX10.load_weights("/content/modelX10.h5")
```

Check performance of the repaired model on the test data:

```
cl_label_pX2 = np.argmax(B_primeX2.predict(cl_x_test), axis=1)
clean_accuracy_B_primeX2 = np.mean(np.equal(cl_label_pX2, cl_y_test))*100
print('Clean Classification accuracy for B_primeX2:', clean_accuracy_B_primeX2)
```

```
cl_label_pX4 = np.argmax(B_primeX4.predict(cl_x_test), axis=1)
clean_accuracy_B_primeX4 = np.mean(np.equal(cl_label_pX4, cl_y_test))*100
print('Clean Classification accuracy for B_primeX4:', clean_accuracy_B_primeX4)
cl_label_pX10 = np.argmax(B_primeX10.predict(cl_x_test), axis=1)
clean_accuracy_B_primeX10 = np.mean(np.equal(cl_label_pX10, cl_y_test))*100
print('Clean Classification accuracy for B_primeX10:', clean_accuracy_B_primeX10)

bd_label_pX2 = np.argmax(B_primeX2.predict(bd_x_test), axis=1)
asr_B_primeX2 = np.mean(np.equal(bd_label_pX2, bd_y_test))*100
print('Attack Success Rate for B_prime_X_2:', asr_B_primeX2)
bd_label_pX4 = np.argmax(B_primeX4.predict(bd_x_test), axis=1)
asr_B_primeX4 = np.mean(np.equal(bd_label_pX4, bd_y_test))*100
print('Attack Success Rate for B_prime_X_4:', asr_B_primeX4)
bd_label_pX10 = np.argmax(B_primeX10.predict(bd_x_test), axis=1)
asr_B_primeX10 = np.mean(np.equal(bd_label_pX10, bd_y_test))*100
print('Attack Success Rate for B_prime_X_10:', asr_B_primeX10)
```

```
401/401 [==============================] - 1s 3ms/step
Clean Classification accuracy for B_primeX2: 95.90023382696803
401/401 [==============================] - 1s 3ms/step
Clean Classification accuracy for B_primeX4: 92.29150428682775
401/401 [==============================] - 1s 3ms/step
Clean Classification accuracy for B_primeX10: 84.54403741231489
401/401 [==============================] - 1s 2ms/step
Attack Success Rate for B_prime_X_2: 100.0
401/401 [==============================] - 1s 2ms/step
Attack Success Rate for B_prime_X_4: 99.98441153546376
401/401 [==============================] - 1s 3ms/step
Attack Success Rate for B_prime_X_10: 77.20966484801247
```

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 [==============================] - 2s 3ms/step
Clean Classification accuracy for B: 98.62042088854248
401/401 [==============================] - 1s 3ms/step
Attack Success Rate for B: 100.0
```

Create repaired network

```
# Repaired network repaired_net
repaired_netX2 = G(B, B_primeX2)
repaired_netX4 = G(B, B_primeX4)
repaired_netX10 = G(B, B_primeX10)
```

Check the performance of the repaired_net on the test data

```
cl_label_pX2 = np.argmax(repaired_netX2(cl_x_test), axis=1)
clean_accuracy_repaired_net_X_2 = np.mean(np.equal(cl_label_pX2, cl_y_test))*100
print('Clean Classification accuracy for repaired netX2:', clean_accuracy_repaired_net_X_2)
cl_label_pX4 = np.argmax(repaired_netX4(cl_x_test), axis=1)
clean_accuracy_repaired_net_X_4 = np.mean(np.equal(cl_label_pX4, cl_y_test))*100
print('Clean Classification accuracy for repaired netX4:', clean_accuracy_repaired_net_X_4)
cl_label_pX10 = np.argmax(repaired_netX10(cl_x_test), axis=1)
clean_accuracy_repaired_net_X_10 = np.mean(np.equal(cl_label_pX10, cl_y_test))*100
print('Clean Classification accuracy for repaired netX10:', clean_accuracy_repaired_net_X_10)


bd_label_pX2 = np.argmax(repaired_netX2(bd_x_test), axis=1)
asr_repaired_netX2 = np.mean(np.equal(bd_label_pX2, bd_y_test))*100
print('Attack Success Rate for repaired netX2:', asr_repaired_netX2)
bd_label_pX4 = np.argmax(repaired_netX4(bd_x_test), axis=1)
asr_repaired_netX4 = np.mean(np.equal(bd_label_pX4, bd_y_test))*100
print('Attack Success Rate for repaired netX4:', asr_repaired_netX4)
bd_label_pX10 = np.argmax(repaired_netX10(bd_x_test), axis=1)
asr_repaired_netX10 = np.mean(np.equal(bd_label_pX10, bd_y_test))*100
print('Attack Success Rate for repaired netX10:', asr_repaired_netX10)
```

```
Clean Classification accuracy for repaired netX2: 95.74434918160561
Clean Classification accuracy for repaired netX4: 92.1278254091972
Clean Classification accuracy for repaired netX10: 84.3335931410756
Attack Success Rate for repaired netX2: 100.0
Attack Success Rate for repaired netX4: 99.98441153546376
Attack Success Rate for repaired netX10: 77.20966484801247
```

✓ 9s completed at 22:01 ● ×