



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

ИКБ направление «Киберразведка и противодействие угрозам с применением технологий искусственного интеллекта» 10.04.01

Кафедра КБ-4 «Интеллектуальные системы информационной безопасности»

Лабораторная работа №2
по дисциплине

«Анализ защищенности систем искусственного интеллекта»

Группа:
ББМО-01-22
Выполнил:
Воронов А.И.

Проверил:
Спирин А.А.

Москва 2023

Задание 1

Установим библиотеку ART, которая необходима для выполнения задания.

```
[ ] # установим библиотеку ART
!pip install adversarial-robustness-toolbox

Collecting adversarial-robustness-toolbox
  Downloading adversarial_robustness_toolbox-1.16.0-py3-none-any.whl (1.6 MB)
    1.6/1.6 MB 8.4 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from adversarial-robustness-toolbox) (1.23.5)
Requirement already satisfied: scipy>=1.4.1 in /usr/local/lib/python3.10/dist-packages (from adversarial-robustness-toolbox) (1.11.4)
Collecting scikit-learn<1.2.0,>=0.22.2 (from adversarial-robustness-toolbox)
  Downloading scikit_learn-1.1.3-cp310-cp310-manylinux_2_17_x86_64_manylinux2014_x86_64.whl (30.5 MB)
    30.5/30.5 MB 35.9 MB/s eta 0:00:00
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from adversarial-robustness-toolbox) (1.16.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from adversarial-robustness-toolbox) (67.7.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from adversarial-robustness-toolbox) (4.66.1)
Requirement already satisfied: joblib>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn<1.2.0,>=0.22.2->adversarial-robustness-toolbox) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn<1.2.0,>=0.22.2->adversarial-robustness-toolbox) (3.2.0)
Installing collected packages: scikit-learn, adversarial-robustness-toolbox
  Attempting uninstall: scikit-learn
    Found existing installation: scikit-learn 1.2.2
    Uninstalling scikit-learn-1.2.2:
      Successfully uninstalled scikit-learn-1.2.2
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source of the following dependency conflicts.
bigframes 0.17.0 requires scikit-learn>=1.2.2, but you have scikit-learn 1.1.3 which is incompatible.
Successfully installed adversarial-robustness-toolbox-1.16.0 scikit-learn-1.1.3
```

Импортируем необходимые библиотеки.

```
[ ] # импортируем необходимые библиотеки
import cv2
import os
import torch
import random
import pickle
import zipfile
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
from keras.applications import ResNet50
from keras.applications import VGG16
from keras.applications.resnet50 import preprocess_input
from keras.preprocessing import image
from keras.models import load_model, save_model
from keras.layers import Dense, Flatten, GlobalAveragePooling2D
from keras.models import Model
from keras.optimizers import Adam
from keras.losses import categorical_crossentropy
from keras.metrics import categorical_accuracy
from keras.callbacks import ModelCheckpoint, EarlyStopping, TensorBoard
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D, AvgPool2D, BatchNormalization, Reshape, Lambda
from art.estimators.classification import KerasClassifier
from art.attacks.evasion import FastGradientMethod, ProjectedGradientDescent
%matplotlib inline
```

Далее нам нужно подключить гугл диск для возможности корректно и быстро работать с датасетом. Разархивируем датасет.

Подключаем гугл диск через вкладку "файлы".

```
[ ] !ls
```

```
drive sample_data
```

```
[ ] # разархивируем датасет, который находится на подключенном гугл диске
zip_file = '/content/drive/MyDrive/dataset/archive.zip'
z = zipfile.ZipFile(zip_file, 'r')
z.extractall()

print(os.listdir())

['.config', 'Test.csv', 'Test', 'train', 'Meta.csv', 'Meta', 'Train.csv', 'meta', 'drive', 'test', 'Train', 'sample_data']
```

Далее задаем пути к разархивированным данным.

```
[ ] # задаем пути к разархивированным данным
data_path = '/content'
train_data_path = os.path.join(data_path, 'Train')
test_data_path = os.path.join(data_path, 'Test')
meta_data_path = os.path.join(data_path, 'Meta')
```

Прочитаем данные и выполним предварительную обработку изображений из тестового набора.

```
# прочитаем и выполним предварительную обработку изображений из
# тренировочного набора данных
data = []
labels = []
class_count = 43
for i in range(class_count):
    img_path = os.path.join(train_data_path, str(i))
    for img in os.listdir(img_path):
        img = image.load_img(img_path + '/' + img, target_size=(32, 32))
        img_array = image.img_to_array(img)
        img_array = img_array / 255
        data.append(img_array)
        labels.append(i)
data = np.array(data)
labels = np.array(labels)
labels = to_categorical(labels, 43)
# отобразим первый элемент
print("data[0]:\n", data[0])
```

Вывод блока кода:

```
data[0]:
[[[0.627451  0.11764706 0.09019608]
 [0.5921569  0.13333334 0.11764706]
 [0.57254905 0.12156863 0.09411765]
 ...
 [0.5529412  0.10196079 0.07058824]
```

```

[0.53333336 0.10588235 0.07058824]
[0.54509807 0.10196079 0.07450981]]

[[0.62352943 0.12156863 0.08627451]
 [0.6313726 0.11764706 0.08235294]
 [0.6313726 0.12156863 0.08627451]
 ...
 [0.5882353 0.44313726 0.40784314]
 [0.5019608 0.44313726 0.38039216]
 [0.60784316 0.49019608 0.46666667]]

[[0.627451 0.1254902 0.08627451]
 [0.62352943 0.12156863 0.07450981]
 [0.62352943 0.1254902 0.07843138]
 ...
 [0.19215687 0.2 0.2 ]
 [0.13333334 0.14901961 0.15294118]
 [0.09803922 0.10196079 0.10980392]]

...

[[0.16078432 0.16862746 0.11372549]
 [0.17254902 0.1764706 0.11764706]
 [0.2 0.21176471 0.16862746]
 ...
 [0.12156863 0.12941177 0.10588235]
 [0.10980392 0.11372549 0.07843138]
 [0.10588235 0.10588235 0.07843138]]

[[0.16078432 0.15294118 0.11372549]
 [0.14117648 0.13333334 0.09803922]
 [0.11764706 0.12941177 0.09411765]
 ...
 [0.12941177 0.12941177 0.08627451]
 [0.10588235 0.10588235 0.07843138]
 [0.11764706 0.10980392 0.09803922]]

[[0.16078432 0.17254902 0.13333334]
 [0.19215687 0.19215687 0.15294118]
 [0.21568628 0.17254902 0.14901961]
 ...
 [0.10980392 0.10980392 0.07450981]
 [0.09803922 0.10980392 0.08235294]
 [0.10980392 0.11764706 0.09411765]]]

```

Отообразим первый элемент в виде картинки.


```
# создаем модель глубокого обучения для классификации изображений (ResNet50)
model = Sequential()
model.add(ResNet50(include_top = False, pooling = 'avg'))
model.add(Dropout(0.1))
model.add(Dense(256, activation="relu"))
model.add(Dropout(0.1))
model.add(Dense(43, activation = 'softmax'))
model.layers[2].trainable = False
# отобразим итоговую сводку по модели
print(model.summary())
```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 [=====] - 1s 0us/step
Model: "sequential"

Layer (type)	Output Shape	Param #
resnet50 (Functional)	(None, 2048)	23587712
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 256)	524544
dropout_1 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 43)	11051

=====

Total params: 24123307 (92.02 MB)
Trainable params: 23545643 (89.82 MB)
Non-trainable params: 577664 (2.20 MB)

None

Обучим эту модель в течение 5 эпох, используем оптимизатор Adam и функцию потерь categorical_crossentropy. Также сохраним историю обучения модели для возможности последующего анализа.

```
# обучаем модель в течение 5 эпох, используем оптимизатор Adam и
# функцию потерь categorical_crossentropy
model.compile(loss='categorical_crossentropy', optimizer="adam", metrics=['accuracy'])
# сохраним историю обучения для последующего анализа на графиках
history = model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs = 5, batch_size = 64)
```

Epoch 1/5
429/429 [=====] - 69s 67ms/step - loss: 0.9785 - accuracy: 0.7307 - val_loss: 4.0010 - val_accuracy: 0.1506
Epoch 2/5
429/429 [=====] - 23s 54ms/step - loss: 0.1974 - accuracy: 0.9455 - val_loss: 0.9168 - val_accuracy: 0.7715
Epoch 3/5
429/429 [=====] - 23s 55ms/step - loss: 0.1393 - accuracy: 0.9618 - val_loss: 0.2020 - val_accuracy: 0.9407
Epoch 4/5
429/429 [=====] - 23s 53ms/step - loss: 0.0995 - accuracy: 0.9749 - val_loss: 0.1593 - val_accuracy: 0.9592
Epoch 5/5
429/429 [=====] - 26s 61ms/step - loss: 0.0800 - accuracy: 0.9800 - val_loss: 0.2102 - val_accuracy: 0.9421

Считаем данные из csv в датафрейм, в ней содержится оригинальная метка класса и путь к изображению.

```

test = pd.read_csv("Test.csv")
test_imgs = test['Path'].values
data = []

for img in test_imgs:
    img = image.load_img(img, target_size=(32, 32))
    img_array = image.img_to_array(img)
    img_array = img_array / 255
    data.append(img_array)

data = np.array(data)
y_test = test['ClassId'].values.tolist()
y_test = np.array(y_test)
y_test = to_categorical(y_test, 43)

```

Создадим модель VGG16 и обучим по аналогии с предыдущей, также сохраним историю.

```

# по аналогии с предыдущей, создаем модель для классификации изображений (VGG16)
model2 = Sequential()
model2.add(VGG16(include_top=False, pooling = 'avg'))
model2.add(Dropout(0.1))
model2.add(Dense(256, activation="relu"))
model2.add(Dropout(0.1))
model2.add(Dense(43, activation = 'softmax'))
model2.layers[2].trainable = False
# отобразим итоговую сводку по модели
print(model2.summary())

```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_data_format.h5
58889256/58889256 [=====] - 1s 0us/step
Model: "sequential_1"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 512)	14714688
dropout_2 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 256)	131328
dropout_3 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 43)	11051
Total params: 14857067 (56.68 MB)		
Trainable params: 14725739 (56.17 MB)		
Non-trainable params: 131328 (513.00 KB)		
None		

```
# обучаем модель в течение 5 эпох, используем оптимизатор Adam и
# функцию потерь categorical_crossentropy
model2.compile(loss='categorical_crossentropy', optimizer="adam", metrics=['accuracy'])
# сохраним историю обучения для последующего анализа на графиках
history2 = model2.fit(x_train, y_train, validation_data=(x_val, y_val), epochs = 5, batch_size = 64)

Epoch 1/5
429/429 [=====] - 28s 49ms/step - loss: 3.2036 - accuracy: 0.1161 - val_loss: 2.0396 - val_accuracy: 0.3183
Epoch 2/5
429/429 [=====] - 17s 40ms/step - loss: 1.1366 - accuracy: 0.6176 - val_loss: 0.5541 - val_accuracy: 0.8053
Epoch 3/5
429/429 [=====] - 17s 40ms/step - loss: 0.3636 - accuracy: 0.8740 - val_loss: 0.2725 - val_accuracy: 0.9174
Epoch 4/5
429/429 [=====] - 19s 43ms/step - loss: 0.1799 - accuracy: 0.9497 - val_loss: 0.1737 - val_accuracy: 0.9572
Epoch 5/5
429/429 [=====] - 18s 41ms/step - loss: 0.1107 - accuracy: 0.9723 - val_loss: 0.0923 - val_accuracy: 0.9759

# сохраняем модель для последующего использования
save_model(model2, 'VGG16.h5')

<ipython-input-15-19742678d690>:2: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This file format is con
save_model(model2, 'VGG16.h5')
```

Далее выполним оценку производительности обеих моделей на тестовом наборе данных.

```
# выполним оценку производительности двух моделей на тестовом наборе данных
history_test = model.fit(x_val, y_val, epochs=5, batch_size=64, validation_data=(x_val, y_val))
history2_test = model2.fit(x_val, y_val, epochs=5, batch_size=64, validation_data=(x_val, y_val))

Epoch 1/5
184/184 [=====] - 13s 72ms/step - loss: 0.1274 - accuracy: 0.9687 - val_loss: 0.1082 - val_accuracy: 0.9700
Epoch 2/5
184/184 [=====] - 10s 56ms/step - loss: 0.0863 - accuracy: 0.9790 - val_loss: 0.0856 - val_accuracy: 0.9757
Epoch 3/5
184/184 [=====] - 11s 62ms/step - loss: 0.0448 - accuracy: 0.9886 - val_loss: 0.1389 - val_accuracy: 0.9621
Epoch 4/5
184/184 [=====] - 12s 65ms/step - loss: 0.0569 - accuracy: 0.9862 - val_loss: 0.0222 - val_accuracy: 0.9935
Epoch 5/5
184/184 [=====] - 12s 65ms/step - loss: 0.0322 - accuracy: 0.9919 - val_loss: 0.0554 - val_accuracy: 0.9878
Epoch 1/5
184/184 [=====] - 11s 60ms/step - loss: 0.1239 - accuracy: 0.9690 - val_loss: 0.1064 - val_accuracy: 0.9714
Epoch 2/5
184/184 [=====] - 9s 50ms/step - loss: 0.1893 - accuracy: 0.9613 - val_loss: 0.1196 - val_accuracy: 0.9700
Epoch 3/5
184/184 [=====] - 9s 50ms/step - loss: 0.1120 - accuracy: 0.9738 - val_loss: 0.0914 - val_accuracy: 0.9775
Epoch 4/5
184/184 [=====] - 9s 51ms/step - loss: 0.1701 - accuracy: 0.9623 - val_loss: 0.0620 - val_accuracy: 0.9872
Epoch 5/5
184/184 [=====] - 9s 48ms/step - loss: 0.0643 - accuracy: 0.9866 - val_loss: 0.0236 - val_accuracy: 0.9942
```



```

from tabulate import tabulate
# создаем и выводим таблицу, которая показывает точность
# обеих моделей на тренировочном, валидационном и тестовом наборе данных
train_accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
test_accuracy = history_test.history['accuracy']

train_accuracy2 = history2_test.history['accuracy']
val_accuracy2 = history2_test.history['val_accuracy']
test_accuracy2 = history2_test.history['accuracy']

table = [
    ["Model", "Training Accuracy", "Validation Accuracy", "Test Accuracy"],
    ["Resnet50", train_accuracy[4]*100, val_accuracy[4]*100, test_accuracy[4]*100],
    ["VGG16", train_accuracy2[4]*100, val_accuracy2[4]*100, test_accuracy2[4]*100]
]

table1 = tabulate(table, headers="firstrow", tablefmt="grid")
print(table1)

```

Model	Training Accuracy	Validation Accuracy	Test Accuracy
Resnet50	97.9961	94.2107	99.1924
VGG16	98.6568	99.4219	98.6568

Как видно из таблицы, модели показывают приблизительно схожие результаты, немного лучше оказалась модель VGG16 (по валидации).

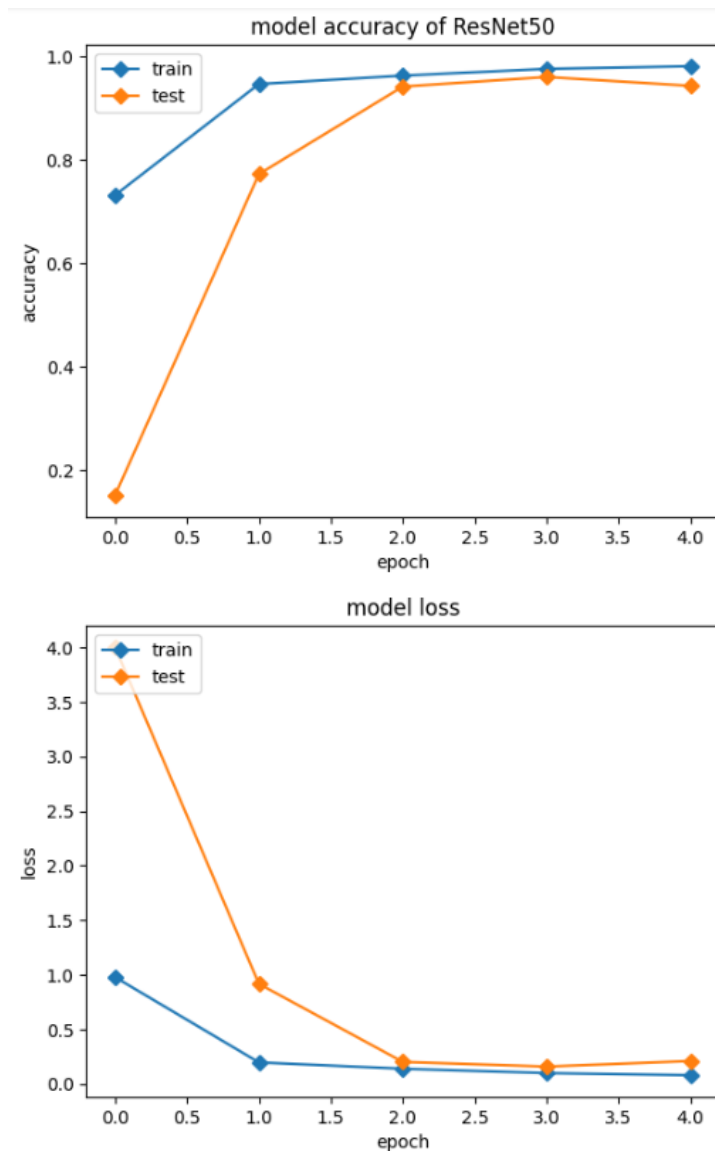
Построим строим два графика процесса обучения модели ResNet50, графики отражают зависимость метрики от эпохи для тренировочного и тестового наборов.

```

# график точности
plt.plot(history.history['accuracy'], marker='D')
plt.plot(history.history['val_accuracy'], marker='D')
plt.title('model accuracy of ResNet50')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# график потерь
plt.plot(history.history['loss'], marker='D')
plt.plot(history.history['val_loss'], marker='D')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

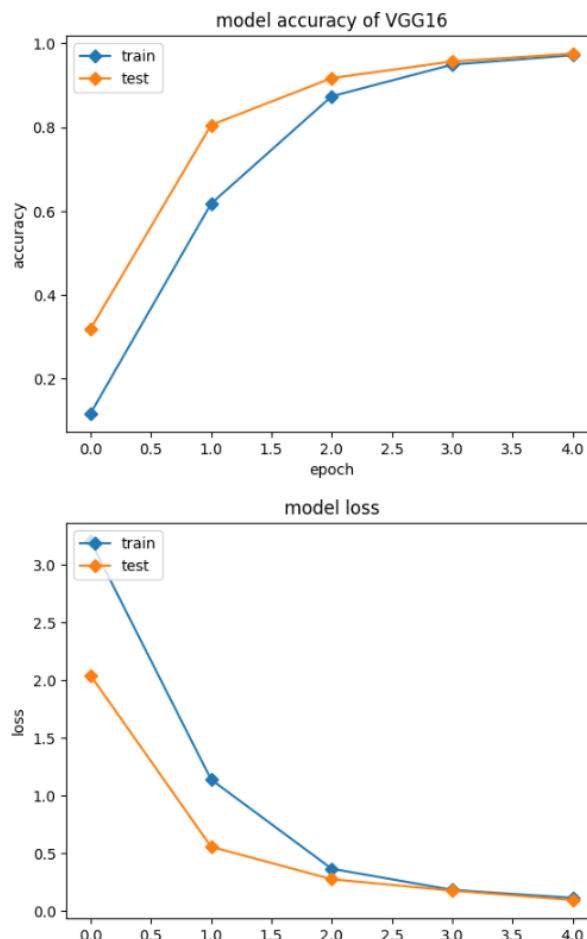
```



Аналогично построим графики для VGG16.

```
# график точности
plt.plot(history2.history['accuracy'], marker='D')
plt.plot(history2.history['val_accuracy'], marker='D')
plt.title('model accuracy of VGG16')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# график потерь
plt.plot(history2.history['loss'], marker='D')
plt.plot(history2.history['val_loss'], marker='D')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



Задание 2

Загрузим модель из предыдущего задания и возьмем тысячу первых элементов из тестового множества, создаем классификатор ART. Также для проведения операций сразу, без построения графа вычислений выполним «tf.compat.v1.disable_eager_execution()».

```
# загрузим модель из предыдущего задания и берем тысячу первых элементов
# из тестового множества, создаем классификатор ART
tf.compat.v1.disable_eager_execution() # для проведения операций сразу, без
# построения графа вычислений

model=load_model('ResNet50.h5')
x_test = data[:1000]
y_test = y_test[:1000]
classifier = KerasClassifier(model=model, clip_values=(np.min(x_test), np.max(x_test)))

WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/keras/src/layers/normalization/batch_
Instructions for updating:
Colocations handled automatically by placer.
```

Создадим атаку FGSM. Проходимся по диапазону значений eps, который представляет размер шага, с которым FGSM изменяет оригинальные данные для создания адверсариальных параметров.

```

# создаем атаку FGSM
attack_fgsm = FastGradientMethod(estimator=classifier, eps=0.3)
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
true_accuracies = [] # для точности оригинальных данных
adv_accuracises_fgsm = []
true_losses = [] # для потерь на оригинальных данных
adv_losses_fgsm = []

# проходимся по диапазону значений eps, который представляет размер шага,
# с которым FGSM изменяет оригинальные данные для создания
# адверсариальных параметров
for eps in eps_range:
    attack_fgsm.set_params(**{'eps': eps}) # установка нового значения eps
    print(f"Eps: {eps}")
    x_test_adv = attack_fgsm.generate(x_test, y_test) # генерация адверсариальных
    # примеров для тестового набора данных
    loss, accuracy = model.evaluate(x_test_adv, y_test) # оценка потерь и точности
    adv_accuracises_fgsm.append(accuracy)
    adv_losses_fgsm.append(loss)
    print(f"Adv Loss: {loss}")
    print(f"Adv Accuracy: {accuracy}")
    loss, accuracy = model.evaluate(x_test, y_test)
    true_accuracies.append(accuracy)
    true_losses.append(loss)
    print(f"True Loss: {loss}")
    print(f"True Accuracy: {accuracy}")

```

Сохраним эту атаку для дальнейшего анализа.

```

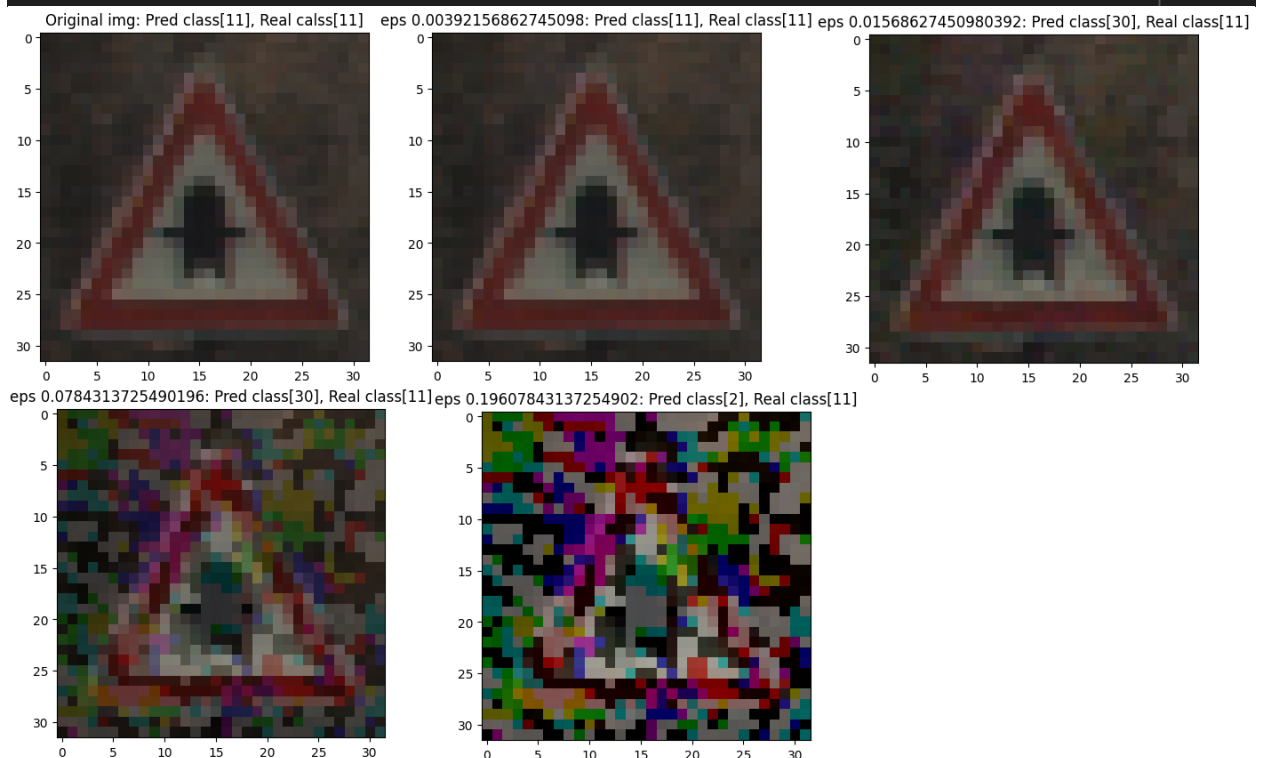
# сохраним атаку FGSM для дальнейшего анализа
adv_losses_fgsm = np.array(adv_losses_fgsm)
adv_accuracises_fgsm = np.array(adv_accuracises_fgsm)
np.save("adv_losses_fgsm_ResNet50", adv_losses_fgsm)
np.save("adv_accuracises_fgsm_ResNet50", adv_accuracises_fgsm)

```

Отообразим исходные и адверсариальные изображения для разных значений eps.

```
# отображаем исходные и адверсариальные изображения для разных значений eps
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
pred = np.argmax(model.predict(x_test[4:5]))
plt.figure(4)
plt.title(f"Original img: Pred class[{pred}], Real class[{np.argmax(y_test[4])}]")
plt.imshow(x_test[4])
plt.show()
i = 1

# проходимся по каждому eps из заданного диапазона
for eps in eps_range:
    attack_fgsm.set_params(**{'eps': eps})
    x_test_adv = attack_fgsm.generate(x_test, y_test)
    pred = np.argmax(model.predict(x_test_adv[4:5]))
    plt.figure(i)
    plt.title(f"eps {eps}: Pred class[{pred}], Real class[{np.argmax(y_test[4])}]")
    plt.imshow(x_test_adv[4])
    plt.show()
    i += 1
```



Как видно, ошибки предсказания из-за наложенного шума начались со значения 2/225. (в первых тестах с 4/255 и 5/255).

Прделаем те же действия для атаки PGD.

```
# теперь реализуем атаку PGD для той же модели, создаем атаку по аналогии с
# предыдущей
tf.compat.v1.disable_eager_execution()
model=load_model('ResNet50.h5')
x_test = data[:1000]
y_test = y_test[:1000]
classifier = KerasClassifier(model=model, clip_values=(np.min(x_test), np.max(x_test)))
```

```

# создаем атаку PGD
attack_pgd = ProjectedGradientDescent(estimator=classifier, eps=0.3, max_iter=4, verbose=False)
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
true_accuracies = [] # для точности оригинальных данных
adv_accuracises_pgd = []
true_losses = [] # для потерь на оригинальных данных
adv_losses_pgd = []

# проходимся по диапазону значений eps, который представляет размер шага,
# с которым PGD изменяет оригинальные данные для создания
# адверсариальных параметров
for eps in eps_range:
    attack_pgd.set_params(**{'eps': eps})
    print(f"Eps: {eps}")
    x_test_adv = attack_pgd.generate(x_test, y_test)
    loss, accuracy = model.evaluate(x_test_adv, y_test)
    adv_accuracises_pgd.append(accuracy)
    adv_losses_pgd.append(loss)
    print(f"Adv Loss: {loss}")
    print(f"Adv Accuracy: {accuracy}")
    loss, accuracy = model.evaluate(x_test, y_test)
    true_accuracies.append(accuracy)
    true_losses.append(loss)
    print(f"True Loss: {loss}")
    print(f"True Accuracy: {accuracy}")

# сохраним атаку PGD для дальнейшего анализа с помощью графика
adv_losses_pgd = np.array(adv_losses_pgd)
adv_accuracises_pgd = np.array(adv_accuracises_pgd)
np.save("adv_losses_pgd_ResNet50", adv_losses_pgd)
np.save("adv_accuracises_pgd_ResNet50", adv_accuracises_pgd)

```

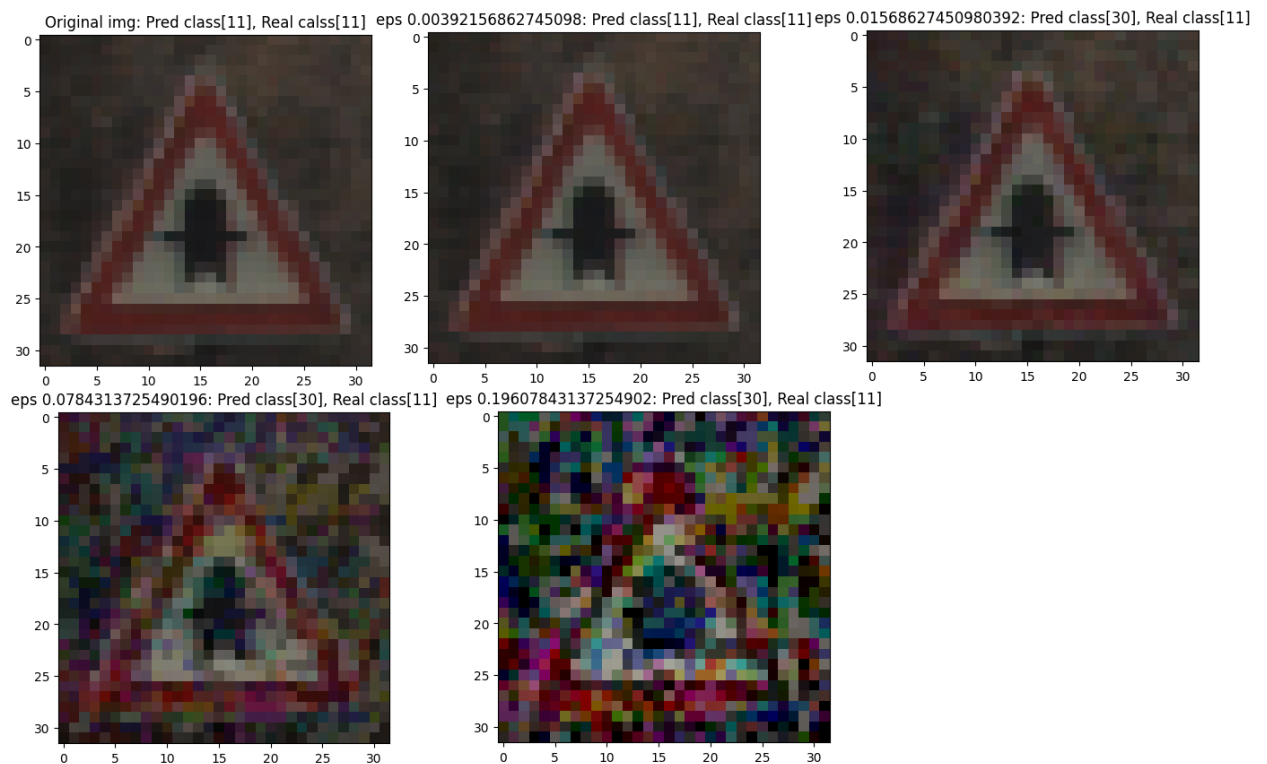
Отобразим исходные и адверсариальные изображения для разных значений eps.

```

# отображаем исходные и адверсариальные изображения для разных значений eps
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
pred = np.argmax(model.predict(x_test[4:5]))
plt.figure(4)
plt.title(f"Original img: Pred class[{pred}], Real class[{np.argmax(y_test[4])}]")
plt.imshow(x_test[4])
plt.show()
i = 1

# проходимся по каждому eps из заданного диапазона
for eps in eps_range:
    attack_pgd.set_params(**{'eps': eps})
    x_test_adv = attack_pgd.generate(x_test, y_test)
    pred = np.argmax(model.predict(x_test_adv[4:5]))
    plt.figure(i)
    plt.title(f"eps {eps}: Pred class[{pred}], Real class[{np.argmax(y_test[4])}]")
    plt.imshow(x_test_adv[4])
    plt.show()
    i += 1

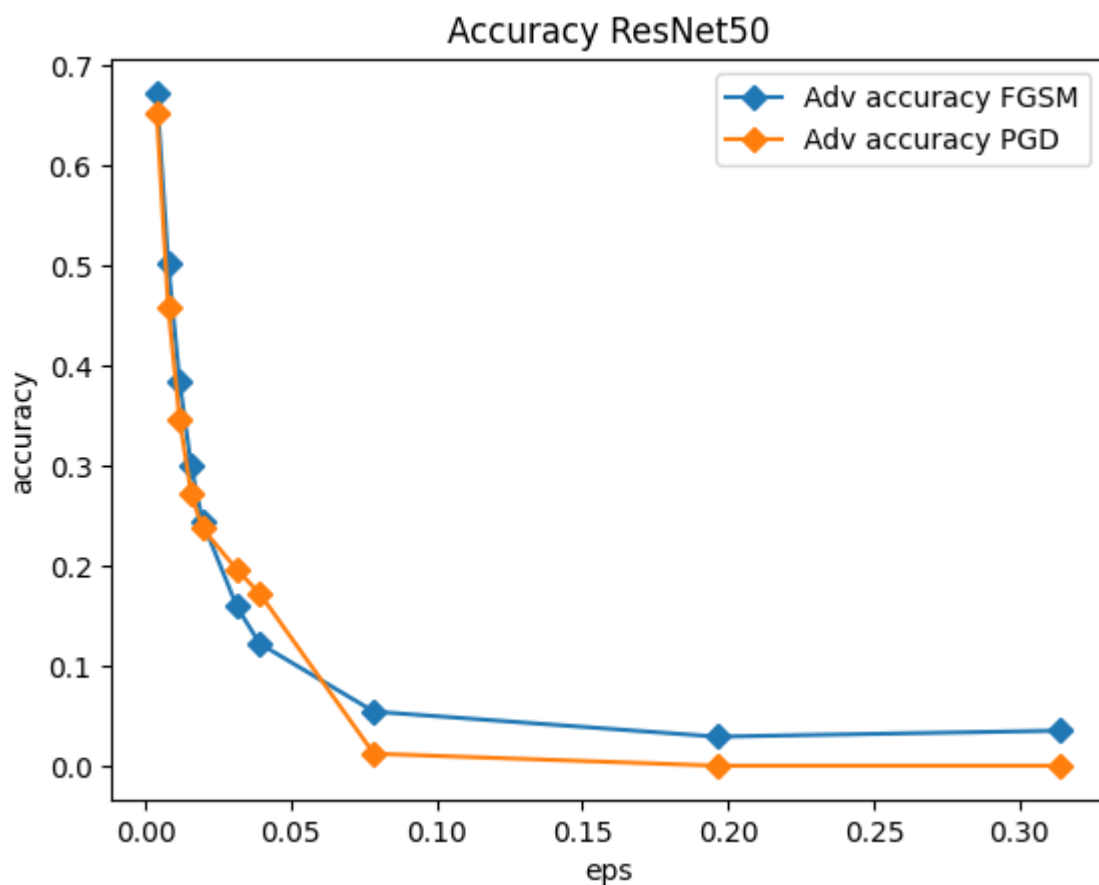
```



Предсказания стали ложными при параметре $2/255$. (Запускал блокнот несколько раз, до этого значения были $5/255$, $6/255$).

Загружаем ранее сохраненный массив адверсариальных точностей для атак FGSM и PDG и строим график зависимости адверсариальной точности от значения ϵ .

```
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
# загружаем ранее сохраненный массив адверсариальных точностей для атак
# FGSM и PDG
adv_accuracises_fgsm = np.load("adv_accuracises_fgsm_ResNet50.npy")
adv_accuracises_pgd = np.load("adv_accuracises_pgd_ResNet50.npy")
# строим график зависимости адверсариальной точности от значения eps
# для атак PDG и FGSM
plt.figure(0)
plt.plot(eps_range, adv_accuracises_fgsm, label="Adv accuracy FGSM", marker='D')
plt.plot(eps_range, adv_accuracises_pgd, label="Adv accuracy PGD", marker='D')
plt.title("Accuracy ResNet50")
plt.xlabel("eps")
plt.ylabel("accuracy")
plt.legend()
plt.show()
```



Исходя из графика можно сказать, что с увеличением значения ϵ , атака PGD сильнее снижает точность, чем FGSM, хотя сначала разница почти незаметна.

Прделаем аналогичные действия для модели VGG16.


```

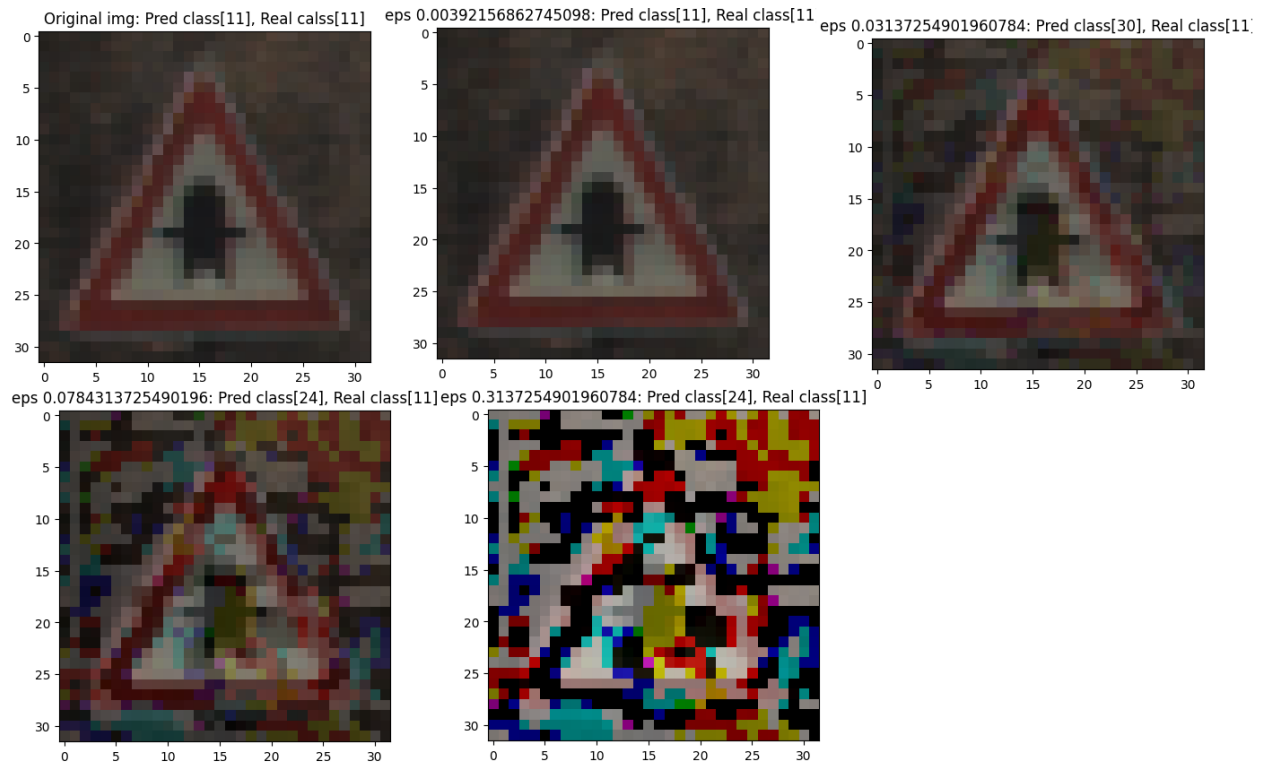
# создаем атаку FGSM по аналогии с VGG16
attack_fgsm = FastGradientMethod(estimator=classifier, eps=0.3)
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
true_accuracies = [] # для точности оригинальных данных
adv_accuracises_fgsm = []
true_losses = [] # для потерь на оригинальных данных
adv_losses_fgsm = []

# прохилдимся по диапазону значений eps, который представляет размер шага,
# с которым FGSM изменяет оригинальные данные для создания
# адверсариальных параметров
for eps in eps_range:
    attack_fgsm.set_params(**{'eps': eps})
    print(f"Eps: {eps}")
    x_test_adv = attack_fgsm.generate(x_test, y_test)
    loss, accuracy = model.evaluate(x_test_adv, y_test)
    adv_accuracises_fgsm.append(accuracy)
    adv_losses_fgsm.append(loss)
    print(f"Adv Loss: {loss}")
    print(f"Adv Accuracy: {accuracy}")
    loss, accuracy = model.evaluate(x_test, y_test)
    true_accuracies.append(accuracy)
    true_losses.append(loss)
    print(f"True Loss: {loss}")
    print(f"True Accuracy: {accuracy}")

# отображаем исходные и адверсариальные изображения для разных значений eps
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
pred = np.argmax(model.predict(x_test[4:5]))
plt.figure(0)
plt.title(f"Original img: Pred class[{pred}], Real calss[{np.argmax(y_test[4])}]")
plt.imshow(x_test[4])
plt.show()
i = 1

# проходимся по каждому eps из заданного диапазона
for eps in eps_range:
    attack_fgsm.set_params(**{'eps': eps})
    x_test_adv = attack_fgsm.generate(x_test, y_test)
    pred = np.argmax(model.predict(x_test_adv[4:5]))
    plt.figure(i)
    plt.title(f"eps {eps}: Pred class[{pred}], Real class[{np.argmax(y_test[4])}]")
    plt.imshow(x_test_adv[4])
    plt.show()
    i += 1

```



Был выдан ложный результат при значении eps 8/255. Стоит отметить, что скорость намного выше, чем у ResNet50.

Прделаем такой же алгоритм для PDG атаки.

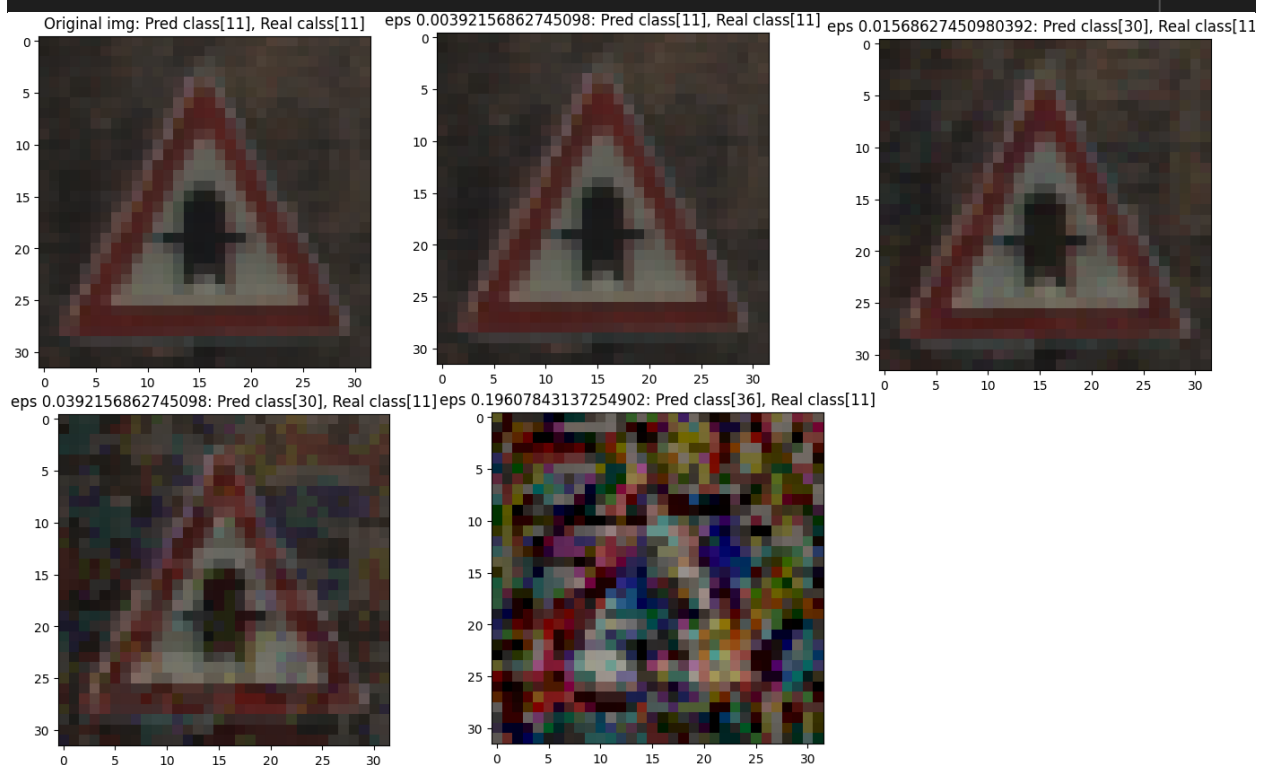
```
[ ] # реализуем атаку PGD для модели VGG16
tf.compat.v1.disable_eager_execution()
model=load_model('VGG16.h5')
x_test = data[:1000]
y_test = y_test[:1000]
classifier = KerasClassifier(model=model, clip_values=(np.min(x_test), np.max(x_test)))

[ ] # создаем атаку PGD по аналогии с ResNet50
attack_pgd = ProjectedGradientDescent(estimator=classifier, eps=0.3, max_iter=4, verbose=False)
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
true_accuracies = [] # для точности оригинальных данных
adv_accuracies_pgd = []
true_losses = [] # для потерь на оригинальных данных
adv_losses_pgd = []

# проходимся по диапазону значений eps, который представляет размер шага,
# с которым PGD изменяет оригинальные данные для создания
# адверсариальных параметров
for eps in eps_range:
    attack_pgd.set_params(**{'eps': eps})
    print(f"Eps: {eps}")
    x_test_adv = attack_pgd.generate(x_test, y_test)
    loss, accuracy = model.evaluate(x_test_adv, y_test)
    adv_accuracies_pgd.append(accuracy)
    adv_losses_pgd.append(loss)
    print(f"Adv Loss: {loss}")
    print(f"Adv Accuracy: {accuracy}")
    loss, accuracy = model.evaluate(x_test, y_test)
    true_accuracies.append(accuracy)
    true_losses.append(loss)
    print(f"True Loss: {loss}")
    print(f"True Accuracy: {accuracy}")
```

```
# отображаем исходные и адверсариальные изображения для разных значений eps
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]
pred = np.argmax(model.predict(x_test[4:5]))
plt.figure(0)
plt.title(f"Original img: Pred class[{pred}], Real class[{np.argmax(y_test[4])}]")
plt.imshow(x_test[4])
plt.show()
i = 1

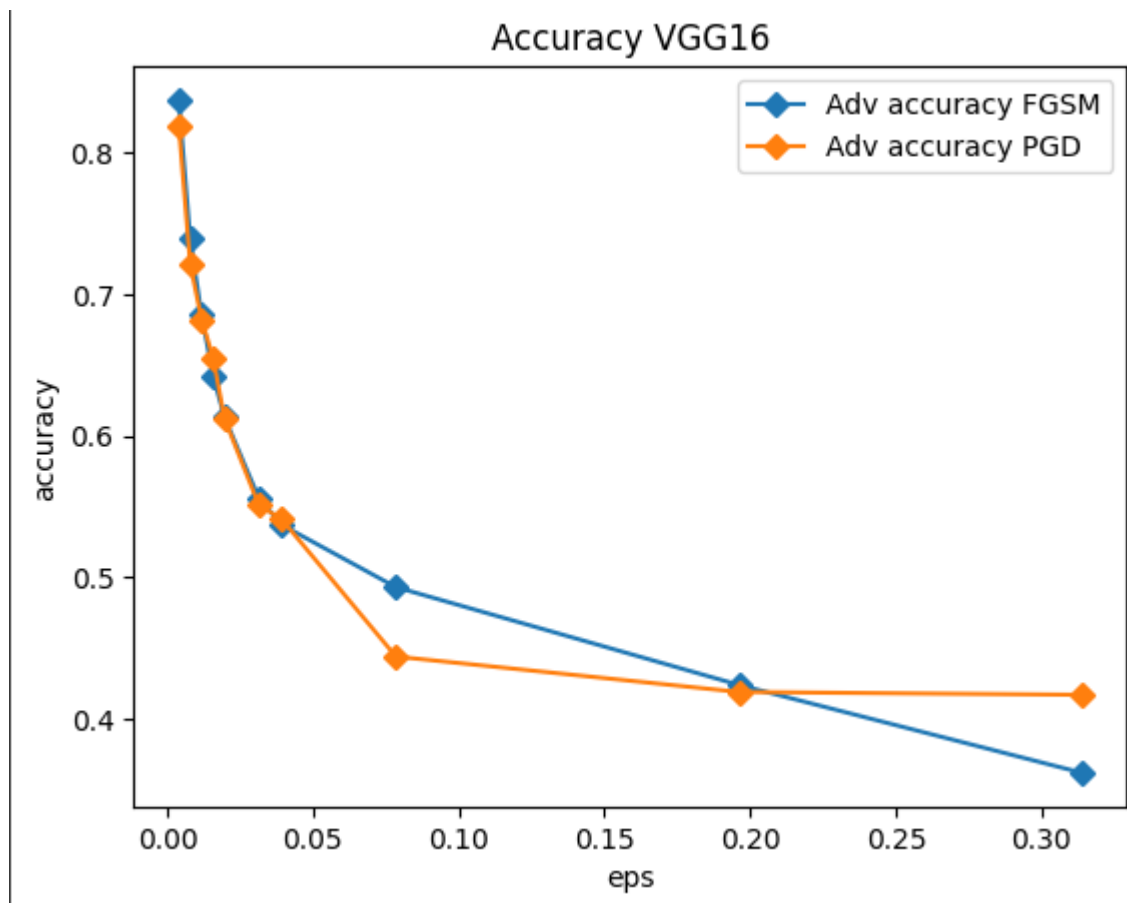
# проходимся по каждому eps из заданного диапазона
for eps in eps_range:
    attack_pgd.set_params(**{'eps': eps})
    x_test_adv = attack_pgd.generate(x_test, y_test)
    pred = np.argmax(model.predict(x_test_adv[4:5]))
    plt.figure(i)
    plt.title(f"eps {eps}: Pred class[{pred}], Real class[{np.argmax(y_test[4])}]")
    plt.imshow(x_test_adv[4])
    plt.show()
    i += 1
```



Ошибка предсказания произошла при значении ϵ 4/255. (В случае VGG16, в предыдущих запусках блокнота, атаки ощутимо сильнее, чем ResNet50.)

Также было замечено, что атаки в случае VGG16 отрабатывают ощутимо быстрее.

Загружаем ранее сохраненный массив адверсариальных точностей для атак FGSM и PDG и строим график зависимости адверсариальной точности от значения ϵ .



В случае VGG16, при атаках PGD и FGSM точность сначала падает одинаково, но с повышением значений ϵ в какой-то момент точность при атаке PGD начинает падать сильнее, но при максимальном значении ϵ точность сильнее всего упала при атаке FGSM.

Отобразим таблицу со значениями точности для обеих моделей.

Model	Original accuracy	eps = 1/255	eps = 2/255	eps = 3/255	eps = 4/255	eps = 5/255	eps = 8/255	eps = 10/255	eps = 20/255	eps = 50/255	eps = 80/255
Resnet50 FGSM	97.9961	67.2	50.2	38.3	30	24.4	15.9	12.2	5.4	2.9	3.5
Resnet50 PGD	97.9961	65.2	45.8	34.6	27.2	23.7	19.6	17.2	1.2	0	0
VGG16 FGSM	98.6568	83.7	73.9	68.5	64.2	61.3	55.6	53.7	49.3	42.4	36.2
VGG16 PGD	98.6568	81.9	72.1	68.2	65.5	61.2	55.1	54.1	44.4	41.9	41.7

По таблице видно, что точность выше при всех значениях ϵ у модели VGG16.

Задание 3

Создадим две целевых атаки, загружаем тестовый набор данных из Test.csv и извлекаем изображения с меткой 14. Преобразуем изображения в массив чисел и нормализуем.

```

# создадим две целевых атаки
# загружаем тестовый набор данных из Test.csv и извлекаем изображения с меткой 14
# Преобразуем изображения в массив чисел и нормализуем
test = pd.read_csv("Test.csv")
test_imgs = test['Path'].values
data = []
y_test = []
labels = test['ClassId'].values.tolist()
i = -1

for img in test_imgs:
    i += 1
    if labels[i] != 14:
        continue
    img = image.load_img(img, target_size=(32, 32))
    img_array = image.img_to_array(img)
    img_array = img_array / 255
    data.append(img_array)
    y_test.append(labels[i])
data = np.array(data)
y_test = np.array(y_test)
y_test = to_categorical(y_test, 43)

```

Реализуем целевую атаку FGSM. Сгенерируем адверсариальные примеры и оценим точность модели на адверсариальных, примерах и на исходных тестовых данных.

```

# реализуем целевую атаку FGSM
# сгенерируем адверсариальные примеры и оценим точность модели на адверсариальных
# примерах и на исходных тестовых данных
model=load_model('ResNet50.h5')
tf.compat.v1.disable_eager_execution()
t_class = 1
t_class = to_categorical(t_class, 43)
t_classes = np.tile(t_class, (270, 1))
x_test = data
classifier = KerasClassifier(model=model, clip_values=(np.min(x_test), np.max(x_test)))
attack_fgsm = FastGradientMethod(estimator=classifier, eps=0.2, targeted=True, batch_size=64)
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]

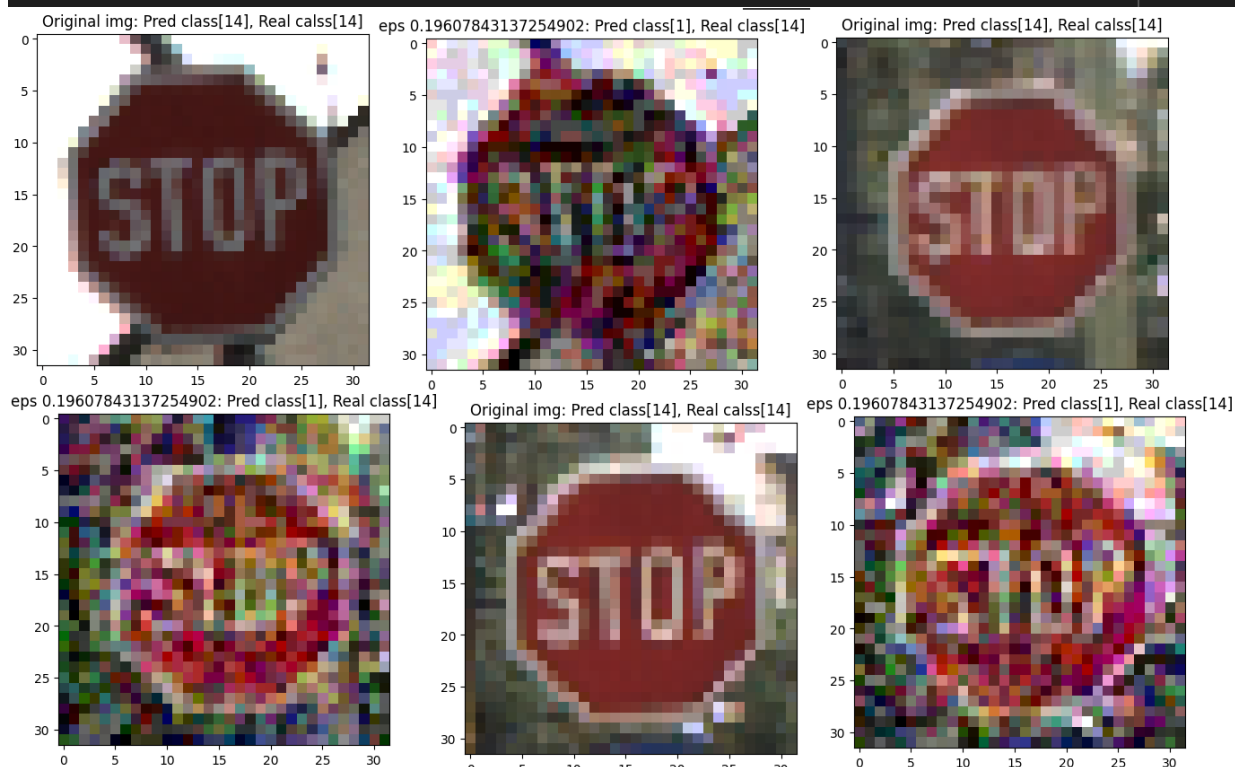
for eps in eps_range:
    attack_fgsm.set_params(**{'eps': eps})
    print(f"Eps: {eps}")
    x_test_adv = attack_fgsm.generate(x_test, t_classes)
    loss, accuracy = model.evaluate(x_test_adv, y_test)
    print(f"Adv Loss: {loss}")
    print(f"Adv Accuracy: {accuracy}")
    loss, accuracy = model.evaluate(x_test, y_test)
    print(f"True Loss: {loss}")
    print(f"True Accuracy: {accuracy}")

```

Определим значение eps, которое лучше всего покажет себя и отобразим разные изображения для визуализации действия атаки.

```
# тут играемся со значениями eps, лучше всего себя показывает 10/255
eps = 10/255
attack_fgsm.set_params(**{'eps': eps})
x_test_adv = attack_fgsm.generate(x_test, t_classes)

# отобразим 5 разных изображений для визуализации действия атаки
range = [0, 10, 20, 30, 40]
i = 0
for index in range:
    plt.figure(i)
    pred = np.argmax(model.predict(x_test[index:index+1]))
    plt.title(f"Original img: Pred class[{pred}], Real class[{np.argmax(y_test[index])}]")
    plt.imshow(x_test[index])
    plt.show()
    i += 1
    pred = np.argmax(model.predict(x_test_adv[index:index+1]))
    plt.figure(i)
    plt.title(f"eps {eps}: Pred class[{pred}], Real class[{np.argmax(y_test[index])}]")
    plt.imshow(x_test_adv[index])
    plt.show()
```



Целевая атака FGSM достигает своего пика на $\epsilon = 10/255$ в нашем случае, при больших значениях ϵ атака хоть и будет давать больше неточности при предсказании, но это будут разные классы, в большинстве случаев отличные от первого (знак стоп), который мы указали. Можно сделать вывод, что FGSM не очень подходит для целевых атак.

Прделаем те же действия для PGD атаки.

```

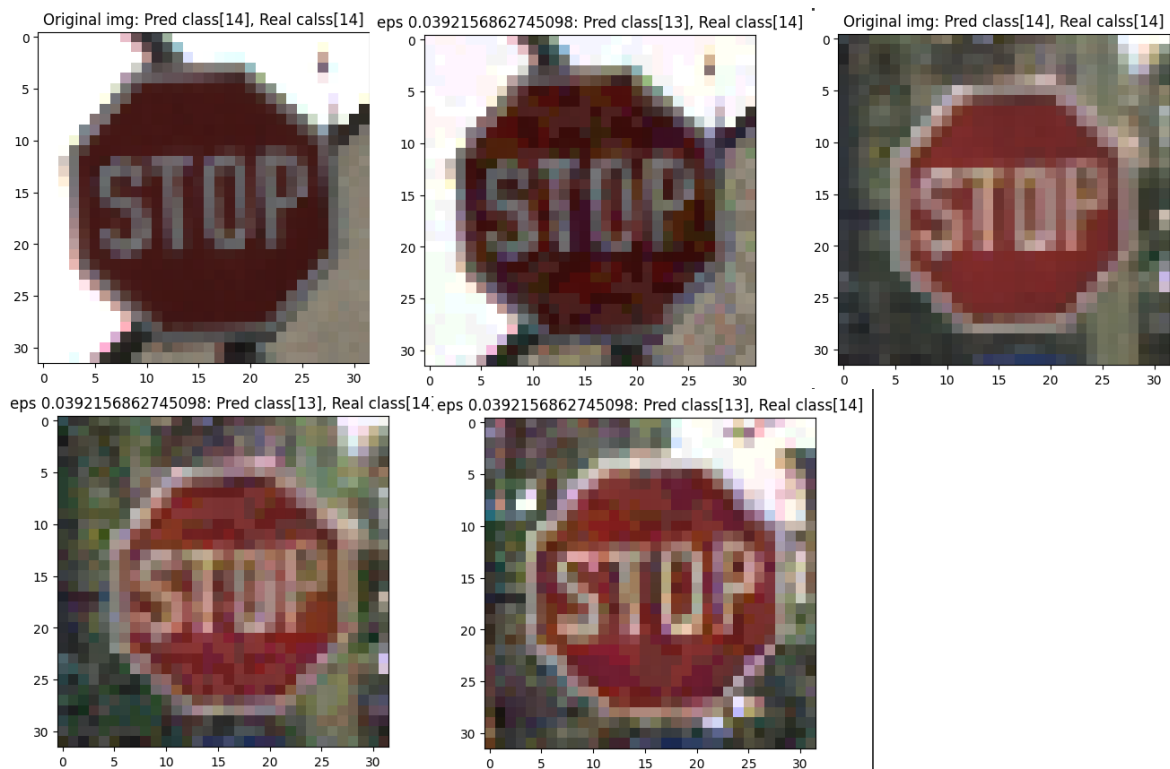
# реализуем целевую атаку PGD
# сгенерируем адверсариальные примеры и оценим точность модели на адверсариальных
# примерах и на исходных тестовых данных
model=load_model('ResNet50.h5')
classifier = KerasClassifier(model=model, clip_values=(np.min(x_test), np.max(x_test)))
attack_pgd = ProjectedGradientDescent(estimator=classifier, eps=0.3, max_iter=4, verbose=False, targeted=True)
eps_range = [1/255, 2/255, 3/255, 4/255, 5/255, 8/255, 10/255, 20/255, 50/255, 80/255]

for eps in eps_range:
    attack_pgd.set_params(**{'eps': eps})
    print(f"Eps: {eps}")
    x_test_adv = attack_pgd.generate(x_test, t_classes)
    loss, accuracy = model.evaluate(x_test_adv, y_test)
    print(f"Adv Loss: {loss}")
    print(f"Adv Accuracy: {accuracy}")
    loss, accuracy = model.evaluate(x_test, y_test)
    print(f"True Loss: {loss}")
    print(f"True Accuracy: {accuracy}")

# играемся с параметрами eps
eps = 10/255
attack_pgd.set_params(**{'eps': eps})
x_test_adv = attack_pgd.generate(x_test, t_classes)

# отобразим 5 разных изображений для визуализации действия атаки
range = [0, 10, 20, 30, 40]
i = 0
for index in range:
    plt.figure(i)
    pred = np.argmax(model.predict(x_test[index:index+1]))
    plt.title(f"Original img: Pred class[{pred}], Real class[{np.argmax(y_test[index])}]")
    plt.imshow(x_test[index])
    plt.show()
    i += 1
    pred = np.argmax(model.predict(x_test_adv[index:index+1]))
    plt.figure(i)
    plt.title(f"eps {eps}: Pred class[{pred}], Real class[{np.argmax(y_test[index])}]")
    plt.imshow(x_test_adv[index])
    plt.show()

```



Атака PGD достигает отличных значений при eps 50/255, при таком значении очень много требуемых результатов.

Отобразим таблицу со значениями точности при двух атаках.

Искажение	FGSM	PGD
$\epsilon=1/255$	85%	97%
$\epsilon=2/255$	76%	93%
$\epsilon=3/255$	62%	87%
$\epsilon=4/255$	52%	86%
$\epsilon=5/255$	45%	79%
$\epsilon=8/255$	19%	76%
$\epsilon=10/255$	13%	75%
$\epsilon=20/255$	4%	38%
$\epsilon=50/255$	1%	5%
$\epsilon=80/255$	1%	3%

Как видим, атака PGD дольше сохраняет точность, чем FGSM. При этом PGD намного лучше подходит для целевых атак, так как на больших значениях eps выдает лучший требуемый (класс 1 - знак стоп) результат, чем FGSM.