ML Security with the Adversarial Robustness Toolbox

Part 1 — Attacking Machine Learning Models

Kedion · Follow
17 min read · Apr 26, 2022

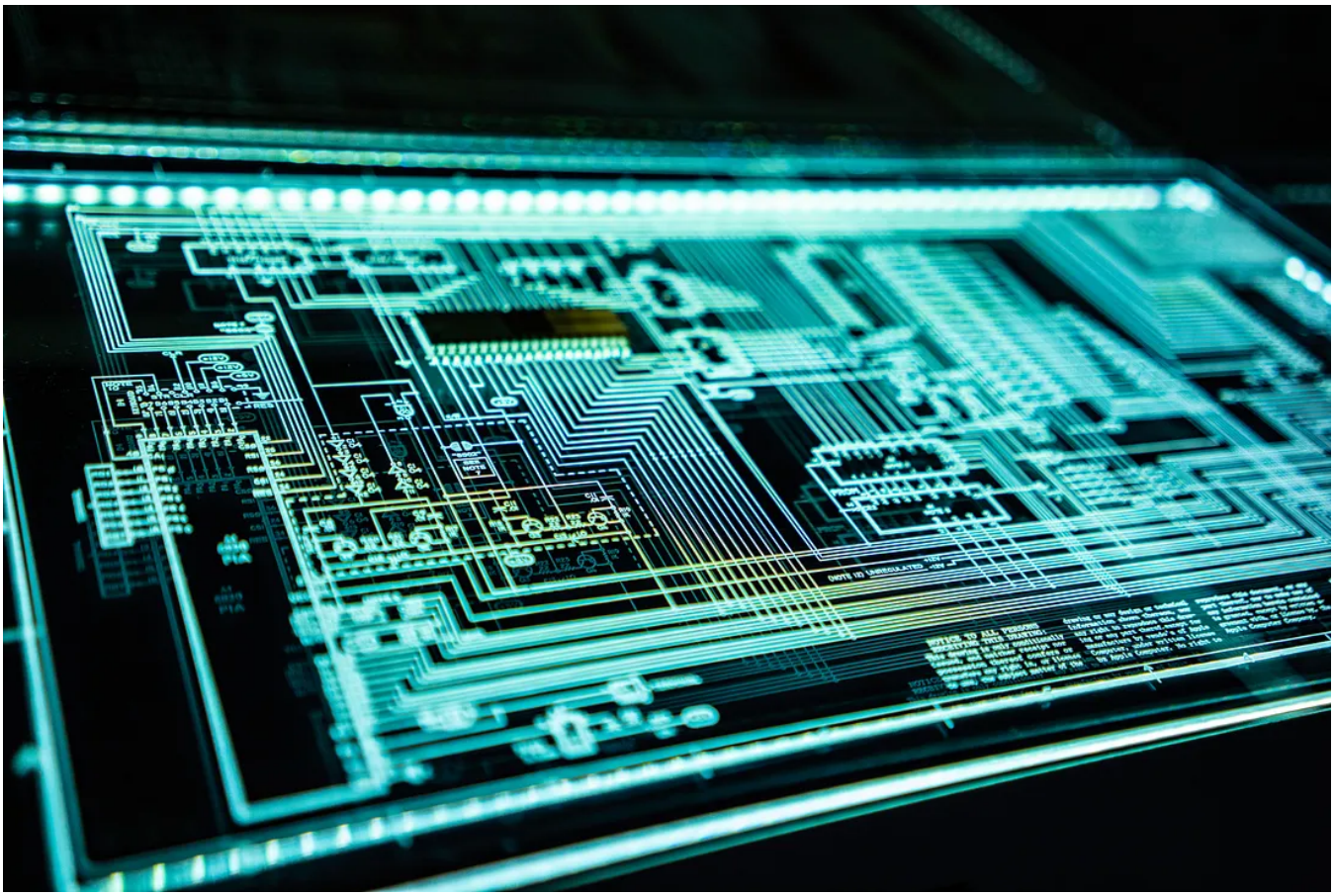


Photo by Adi Goldstein

Written by Tigran Avetisyan.

Developing machine learning models and putting them into production can be very challenging. However, successfully deploying an ML pipeline is just part of the story — you also need to think about keeping it secure.

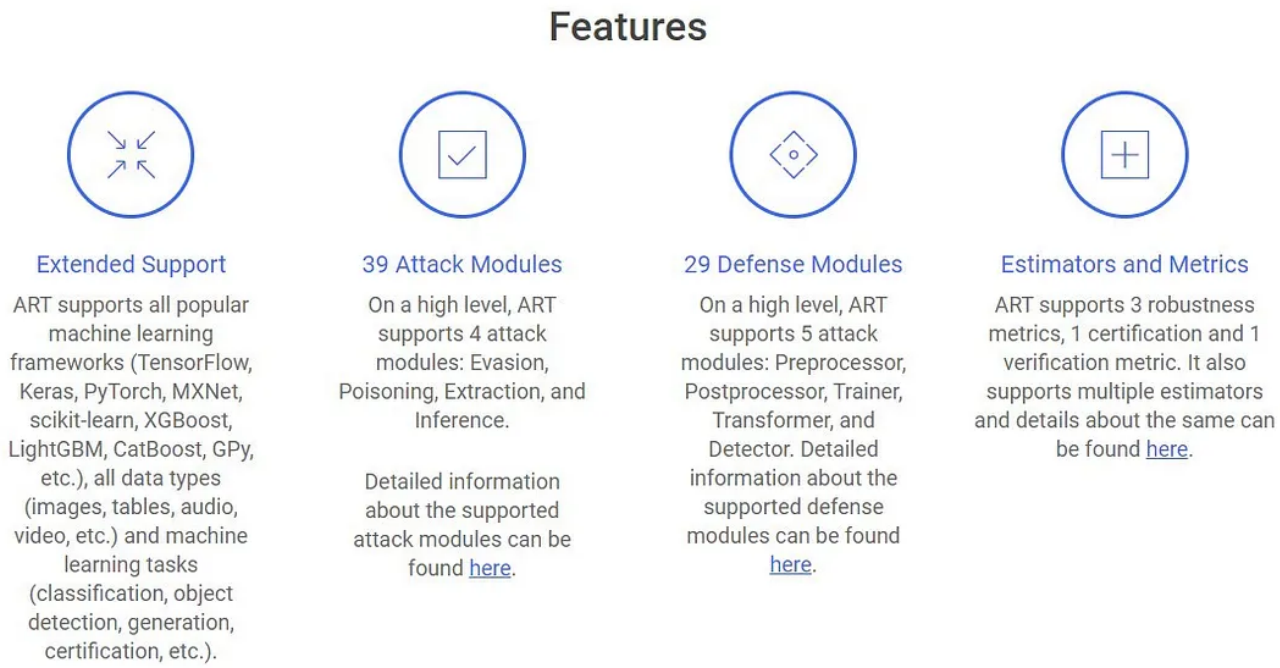
Machine learning models are used in a wide range of areas, like finance, medicine, or surveillance, and can be accurate at detecting fraud or filtering out faulty products. In applications like detecting fraud, scammers might be interested in fooling machine learning systems so that they miss scam emails or phishing links.

In this series of tutorials, we are going to have a look at the Adversarial Robustness Toolbox and figure out how it can help you with securing your machine learning pipelines. In PART 1, we will focus on adversarial attacks, and we will be using the MNIST digits dataset along with the TensorFlow/Keras ML framework.

Let's get started!

What is the Adversarial Robustness Toolbox?

The Adversarial Robustness Toolbox, or ART, is a Python framework for machine learning security. ART contains attack and defense tools that can help teams better understand adversarial attacks and develop protection measures based on experimentation.



https://adversarial-robustness-toolbox.org/

ART has 39 attack modules, 29 defense modules, and supports a wide range of machine learning frameworks, including scikit-learn, PyTorch, TensorFlow, and Keras. ART also supports several machine learning tasks (including classification, regression, and generation) and works with all data types (audio, video, images, or tables).

The Adversarial Robustness Toolbox was originally developed and published by IBM. In July 2020, IBM donated ART to the Linux Foundation AI (LFAI). Since then, LFAI has maintained and developed updates for the toolkit.

Attack Types in the Adversarial Robustness Toolbox

Because PART 1 in this series focuses on attacks, let's take a deeper look at the attack types supported by ART.

At a high level, there are 4 types of adversarial attacks implemented in ART:

· **Evasion.** Evasion attacks typically work by perturbing input data to cause a trained model to misclassify it. Evasion is done after training and during inference, i.e. when models are already deployed in production. Adversaries perform evasion attacks to avoid detection by AI systems. As an example, adversaries might run an evasion attack to cause the victim model to miss phishing emails. Evasion attacks might require access to the victim model.

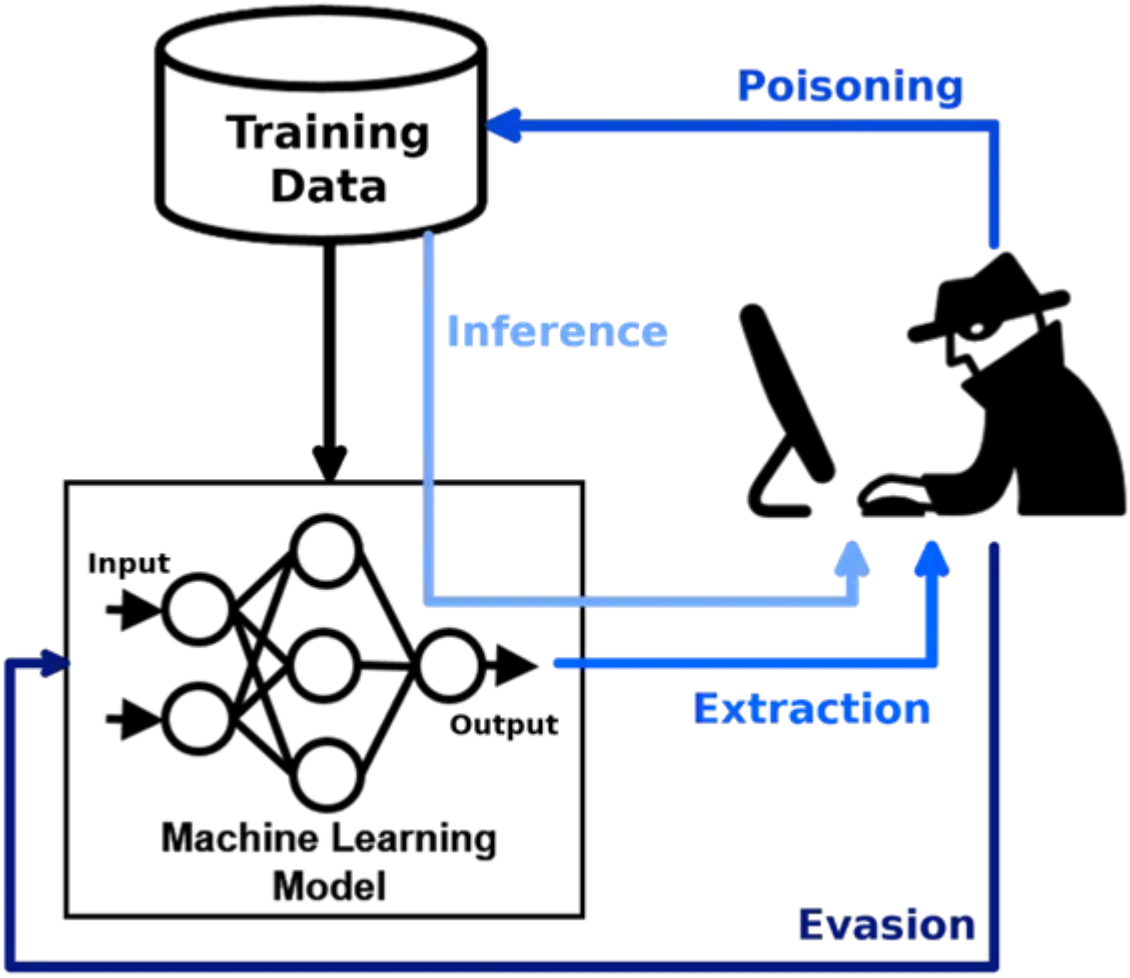
· **Extraction.** Extraction is an attack where an adversary attempts to build a model that is similar or identical to a victim model. In simple words, extraction is the attempt of copying or stealing a machine learning model. Extraction attacks typically require access to the original model, as well as to data that is similar or identical to the data originally used to train the victim model.

· **Inference.** Inference attacks generally aim at reconstructing a part or the entirety of the dataset that was used to train the victim model. Adversaries can use inference attacks to reconstruct entire training samples, separate features, or determine if a sample has been used to train the victim model. Inference attacks typically require access to the victim model. In some cases, attackers might also need to have access to some portion of the data used to train the model.

· **Poisoning.** Poisoning attacks aim to perturb training data to corrupt the victim model during training. Poisoned data contains features (called a backdoor) that trigger the desired output in a trained model. Essentially, the perturbed features

cause the model to overfit to them. As a very simple example (which we'll have a look at in code below), an attacker could poison the digits in the MNIST dataset so that the victim model classifies all digits as 9s. Poisoning attacks require access to the training data of a model before the actual training occurs.

ART's documentation supplies this neat graph that shows how the attacks work at a high level:



https://adversarial-robustness-toolbox.readthedocs.io/en/latest/index.html

For the vast majority of implemented attacks, ART supplies links to the research papers that provide more detail on a given attack. So if you want to learn more about a specific attack, look for paper links in ART's documentation.

Below, we will take a look at each of the attack types supported by ART. We will implement one attack per type, but what you'll see below should transfer to other attacks of the same type as well.

How do attacks on machine learning pipelines happen?

As we pointed out above, adversaries typically need some form of access to your machine learning model or its training data to perform an attack. But assuming that your model is hosted in an environment that an adversary can't reach, how do attacks on ML pipelines even happen? In more practical terms, how can an adversary gain access to your model and training data?

Here are just some of the cases in which adversaries can obtain access to your pipeline and data:

· **You are using a dataset from an unverified source.** An adversary could poison data and publish it somewhere for unsuspecting teams to use. Datasets from untrustworthy sources or datasets that are not verified are at higher risk of being compromised.

· **You are using a model from an unverified source.** Similar to training data, adversaries can publish models for victims to use. They can later use their knowledge of the model to perform attacks on it. Models from suspicious sources might also be pre-trained on poisoned data.

· **You are using a model that is available publicly.** Suppose that you are using an NLP model from Hugging Face. Hugging Face models are available to anyone. An adversary could analyze a publicly available model and use their knowledge to attack other similar models used by other teams. In fact, adversaries don't need to have the exact same model as you do — knowledge of a model that performs the same task as yours can be enough for an attack.

· **An adversary has access to your ML pipeline.** If a malicious insider gains access to your pipeline — by leveraging their position inside the organization, for example — they can get in-depth knowledge about its architecture, weights, and training data. They might also know about the measures that you use to protect your model, which makes malicious insiders arguably the biggest threat to ML pipeline security.

This isn't an exhaustive list, but it should give you an idea of how your model or training data could become compromised.

Prerequisites for Using ART

To be able to follow along with this tutorial, install ART by using this command.

```
pip install adversarial-robustness-toolbox
```

If you are using conda, use this command instead:

```
conda install -c conda-forge adversarial-robustness-toolbox
```

We used ART version 1.10.0, but newer versions should work fine as well.

If necessary, you can install ART just for the specific ML/DL framework that you will be using. You can learn more about this here.

Aside from ART, you will need whichever ML/DL framework you want to attack. We are going to be using TensorFlow, which you can install by using this command:

```
pip install tensorflow
```

Or if you are using conda, this command:

```
conda install -c conda-forge tensorflow
```

We are also going to be using NumPy and Matplotlib, so make sure that you have these libraries as well.

Evasion Attacks in ART

Let's start with evasion attacks in ART. As mentioned earlier, an evasion attack is when the attacker perturbs input at inference time to cause the model to misclassify it.

As of ART version 1.10.0, most of the attacks supported by the framework were evasion attacks.

We will be using the Fast Gradient Method to generate adversarial samples from our test set. You can read more about the Fast Gradient Method in this paper.

Note that you can find the full code for this tutorial here. You can find more examples from the authors of ART here and here.

Importing dependencies

As usual, we start by importing dependencies:

```python
# Importing dependencies
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from art.estimators.classification import KerasClassifier
from art.attacks.evasion import FastGradientMethod
from art.utils import load_dataset

# Disabling eager execution from TF 2
tf.compat.v1.disable_eager_execution()

# Increasing Matplotlib font size
matplotlib.rcParams.update({"font.size": 14})
```

dependencies.py hosted with ❤ by GitHub view raw

Note that we are disabling eager execution in TensorFlow for this guide on line 12. This is because the wrapper class **`art.estimators.classification.KerasClassifier`** doesn't fully support TF 2.

Loading data

To load the MNIST dataset, we are going to use ART's function **`art.utils.load_dataset`**. This function returns tuples for the train and test sets, as well as the minimum and maximum feature values in the dataset.

```python
# Loading the data
(train_images, train_labels), (test_images, test_labels), min, max = load_dataset(name="mnist")
```

The images are already normalized to the **`[0, 1]`** range, while the labels are one-hot encoded. We don't need to do any preprocessing on the dataset.

Training a TensorFlow Keras model

Now, let's create a simple TensorFlow Keras model:

```python
# Function for creating model
def create_model():
    # Defining the model
    model = tf.keras.models.Sequential([
        Conv2D(filters=32, kernel_size=3, activation="relu", input_shape=(28, 28, 1)),
        MaxPool2D(pool_size=2),
        Conv2D(filters=64, kernel_size=3, activation="relu"),
        MaxPool2D(pool_size=2),
        Flatten(),
        Dense(units=10, activation="softmax")
    ])

    # Compiling the model
    model.compile(
        optimizer="adam",
        loss="categorical_crossentropy",
        metrics=["accuracy"]
        )

    # Returning the model
    return model
```

create_model.py hosted with ❤️ by GitHub view raw

And train it:

```python
# Instantiating the model
model = create_model()

# Training the model
model.fit(
    x=train_images,
    y=train_labels,
    epochs=10,
    batch_size=256)
```

train_model.py hosted with ❤ by GitHub view raw

```
Train on 60000 samples
Epoch 1/10
60000/60000 [==============================] - 4s 67us/sample - loss: 0.3924 - accuracy: 0.8961
Epoch 2/10
60000/60000 [==============================] - 2s 33us/sample - loss: 0.0934 - accuracy: 0.9721
Epoch 3/10
60000/60000 [==============================] - 2s 33us/sample - loss: 0.0665 - accuracy: 0.9804
Epoch 4/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0530 - accuracy: 0.9838
Epoch 5/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0467 - accuracy: 0.9858
Epoch 6/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0404 - accuracy: 0.9875
Epoch 7/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0363 - accuracy: 0.9889
Epoch 8/10
60000/60000 [==============================] - 2s 31us/sample - loss: 0.0322 - accuracy: 0.9903
Epoch 9/10
60000/60000 [==============================] - 2s 31us/sample - loss: 0.0288 - accuracy: 0.9912
Epoch 10/10
60000/60000 [==============================] - 2s 32us/sample - loss: 0.0274 - accuracy: 0.9918
```

If training takes too much time for you, you can try to simplify the model. And if you encounter out-of-memory (OOM) issues, reducing the batch size should be able to help.

Defining an evasion attack on our model

As the next step, let's define an evasion attack for our model.

To be able to run attacks on our model, we must wrap it in the `art.estimators.classification.KerasClassifier` class. Here's how this is done:

```
# Creating a classifier by wrapping our TF model in ART's KerasClassifier class
classifier = KerasClassifier(
    model=model,
    clip_values=(min, max)
    )
```

wrap_model.py hosted with ❤ by GitHub view raw

The argument `model=model` indicates our model, while `clip_values=(min, max)` specifies the minimum and maximum values allowed for the features. We are using the values provided by the `art.utils.load_dataset` function.

Instead of `KerasClassifier`, you can also try `TensorFlowV2Classifier`, but note that its usage is different.

We then define the attack by using ART's `FastGradientMethod` class:

```
# Defining an attack using the fast gradient method
attack_fgsm = FastGradientMethod(
    estimator=classifier,
    eps=0.3
    )
```

fast_gradient_attack.py hosted with ❤ by GitHub view raw

`estimator=classifier` shows that the attack will apply to our classifier, while the argument `eps=0.3` essentially defines how strong the attack will be.

We can then generate adversarial samples by calling the attack object's method `generate`, passing to it the target images that we want to perturb:

```
# Generating adversarial images from test images
test_images_adv = attack_fgsm.generate(x=test_images)
```

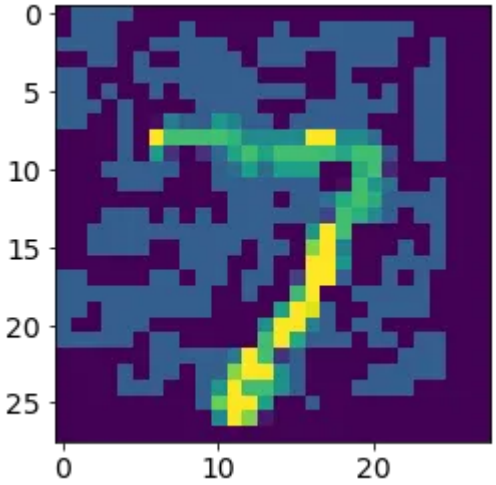
generate_adversarial.py hosted with ❤ by GitHub view raw

Evaluating the effectiveness of the attack

Let's take a look at one adversarial sample:

```
# Viewing an adversarial image
plt.imshow(X=test_images_adv[0])
```

show_adversarial.py hosted with ❤ by GitHub view raw



The fast gradient method applied noise to the clean test images. We can see the effect of the attack by comparing the performance of our model on the clean and adversarial sets:

```
# Evaluating the model on clean images
score_clean = model.evaluate(
    x=test_images,
    y=test_labels
    )

# Evaluating the model on adversarial images
score_adv = model.evaluate(
    x=test_images_adv,
    y=test_labels
    )

# Comparing test losses
print(f"Clean test set loss: {score_clean[0]:.2f} "
      f"vs adversarial set test loss: {score_adv[0]:.2f}")

# Comparing test accuracies
print(f"Clean test set accuracy: {score_clean[1]:.2f} "
      f"vs adversarial test set accuracy: {score_adv[1]:.2f}")
```

evaluate_evasion.py hosted with ❤ by GitHub view raw

```
Clean test set loss: 0.04 vs adversarial set test loss: 5.95Clean
test set accuracy: 0.99 vs adversarial test set accuracy: 0.07
```

The attack has affected the model considerably, making it completely unusable. However, because the perturbations in the image are very evident, it would be very

easy for the victim to figure out that they are being attacked.

Lowering the `eps` value can make the tampering less visible, but the impact on the model will be lessened as well. We've tried 10 different values for *eps* to explore their effect on the visual appearance of adversarial samples and on the model's performance.

```
# Setting the number of rows and columns for the figure
nrows, ncols = 2, 5

# Generating subplots
fig, axes = plt.subplots(
    nrows=nrows,
    ncols=ncols,
    figsize=(20, 10)
    )

# Defining a range of eps values to try
eps_to_try = [0.01, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.175, 0.2, 0.25]

# Defining a counting variable to traverse eps_to_try
counter = 0

# Iterating over rows and cols
for i in range(nrows):
    for j in range(ncols):
        # Creating an attack object for the current value of eps
        attack_fgsm = FastGradientMethod(
            estimator=classifier,
            eps=eps_to_try[counter]
            )

        # Generating adversarial images
        test_images_adv = attack_fgsm.generate(x=test_images)

        # Showing the first adversarial image
        axes[i, j].imshow(X=test_images_adv[0])

        # Disabling x and y ticks
        axes[i, j].set_xticks(ticks=[])
        axes[i, j].set_yticks(ticks=[])

        # Evaluating model performance on adversarial samples and retrieving test accuracy
        test_score = classifier._model.evaluate(
            x=test_images_adv,
            y=test_labels
            )[1]

        # Getting prediction for the image that we displayed
        prediction = np.argmax(model.predict(
            x=np.expand_dims(a=test_images_adv[0],
            axis=0)
            ))

        # Showing the current eps value, test accuracy, and prediction
```

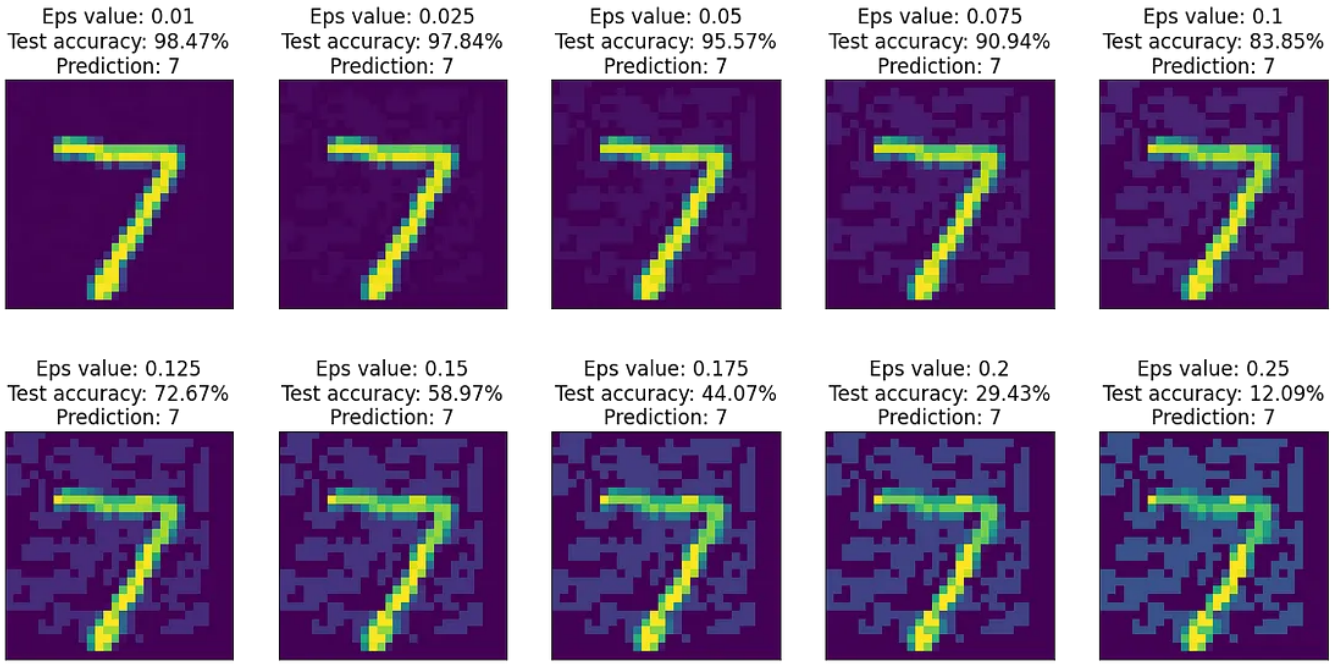
```
49          axes[i, j].set_title(
50              label=f"Eps value: {eps_to_try[counter]}\n"
51              f"Test accuracy: {test_score * 100:.2f}%\n"
52              f"Prediction: {prediction}"
53              )
54
55          # Incrementing counter
56          counter += 1
57
58  # Showing the plot
59  plt.show()
```

compare_eps.py hosted with ❤ by GitHub view raw

In the code block above, we create a figure along with subplots (lines 5 to 9) and define `eps` values to try (line 12). We then iterate over our subplots (lines 18 and 19), generate adversarial samples for each value from `eps_to_try` (lines 21 to 27), plot a sample adversarial image for the current `eps` value (line 30), evaluate model accuracy on the adversarial set (lines 37 to 40), and get a prediction for the sample adversarial image that is being displayed (lines 43 to 46).

Each `eps` value, the test accuracy for it, and the prediction for the sample adversarial image are displayed above each image (lines 49 to 53).

The resulting plot is as follows:



We can see that higher `eps` values produce more visible noise in the image and have a bigger impact on the model's performance. And while the model predicted the

correct label for this particular image 9 times out of 10, the overall performance of the model on the entire adversarial test set was much worse.

Extraction Attacks in ART

As the second step, let's show extraction attacks in ART. Extraction attacks, as a reminder, aim to copy or steal a victim model.

Let's use the class `art.attacks.extraction.CopycatCNN` to perform the attack. You can learn more about this attack method in this paper.

Training a victim model

For this attack, let's separate our training dataset into two subsets — one with 50,000 samples and the other with 10,000.

```
# Importing CopycatCNN
from art.attacks.extraction import CopycatCNN

# Setting aside a subset of the source dataset for the original model
train_images_original = train_images[:50000]
train_labels_original = train_labels[:50000]

# Using the rest of the source dataset for the stolen model
train_images_stolen = train_images[50000:]
train_labels_stolen = train_labels[50000:]
```

define_subsets.py hosted with ❤ by GitHub view raw

The subset with 50,000 samples will be used to train the original model, while the subset with 10,000 samples will be used to steal the original model. Basically, we are simulating a situation where an adversary has a dataset that is similar to the original dataset.

Let's train our original model on its dataset:

```
# Training the original model on its training subset
model_original = create_model()
model_original.fit(
    x=train_images_original,
    y=train_labels_original,
    epochs=10,
    batch_size=256
)
```

train_model.py hosted with ❤ by GitHub view raw

```
Train on 50000 samples
Epoch 1/10
50000/50000 [==============================] - 2s 36us/sample - loss: 0.4450 - accuracy: 0.8741
Epoch 2/10
50000/50000 [==============================] - 2s 32us/sample - loss: 0.1050 - accuracy: 0.9681
Epoch 3/10
50000/50000 [==============================] - 2s 32us/sample - loss: 0.0750 - accuracy: 0.9770
Epoch 4/10
50000/50000 [==============================] - 2s 32us/sample - loss: 0.0615 - accuracy: 0.9810
Epoch 5/10
50000/50000 [==============================] - 2s 32us/sample - loss: 0.0522 - accuracy: 0.9843
Epoch 6/10
50000/50000 [==============================] - 2s 32us/sample - loss: 0.0463 - accuracy: 0.9864
Epoch 7/10
50000/50000 [==============================] - 2s 32us/sample - loss: 0.0406 - accuracy: 0.9876
Epoch 8/10
50000/50000 [==============================] - 2s 33us/sample - loss: 0.0379 - accuracy: 0.9886
Epoch 9/10
50000/50000 [==============================] - 2s 33us/sample - loss: 0.0342 - accuracy: 0.9890
Epoch 10/10
50000/50000 [==============================] - 2s 35us/sample - loss: 0.0299 - accuracy: 0.9908
```

After training, we again wrap the original model into **`KerasClassifier`**:

```
# Wrapping the model in the ART KerasClassifier class
classifier_original = KerasClassifier(
    model=model_original,
    clip_values=(min, max))
```

wrap_model.py hosted with ❤ by GitHub view raw

Defining and running an extraction attack

Next, let's create our model thief, using the class **`CopycatCNN`**:

```
# Creating the "neural net thief" object
# that will steal the original classifier
copycat_cnn = CopycatCNN(
    batch_size_fit=256,
    batch_size_query=256,
    nb_epochs=20,
    nb_stolen=len(train_images_stolen),
    classifier=classifier_original
    )
```

extraction_attack.py hosted with ❤ by GitHub view raw

Note that the argument **`nb_stolen=len(train_images_stolen)`** essentially determines how many samples ART will use to train the stolen model.

After that, we need to create a blank reference model that **`copycat_cnn`** will train to steal the original model:

```
# Creating a reference model for theft
model_stolen = KerasClassifier(
    model=create_model(),
    clip_values=(min, max)
    )
```

create_reference.py hosted with ❤ by GitHub view raw

We then use the method **`copycat_cnn.extract`** to steal **`classifier_original`**. We are using the subset with 10,000 samples to steal the model.

```python
# Extracting a thieved classifier
# by training the reference model
stolen_classifier = copycat_cnn.extract(
    x=train_images_stolen,
    y=train_labels_stolen,
    thieved_classifier=model_stolen
    )
```

extract_model.py hosted with ❤ by GitHub view raw

```
Train on 10000 samples
Epoch 1/20
10000/10000 [==============================] - 0s 39us/sample - loss: 1.3172 - accuracy: 0.6578
Epoch 2/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.3373 - accuracy: 0.8992
Epoch 3/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.2134 - accuracy: 0.9381
Epoch 4/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.1644 - accuracy: 0.9508
Epoch 5/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.1248 - accuracy: 0.9628
Epoch 6/20
10000/10000 [==============================] - 0s 29us/sample - loss: 0.0988 - accuracy: 0.9702
Epoch 7/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0871 - accuracy: 0.9741
Epoch 8/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0693 - accuracy: 0.9792
Epoch 9/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0599 - accuracy: 0.9819
Epoch 10/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0496 - accuracy: 0.9850
Epoch 11/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0430 - accuracy: 0.9872
Epoch 12/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0355 - accuracy: 0.9901
...
Epoch 19/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0141 - accuracy: 0.9974
Epoch 20/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0119 - accuracy: 0.9984
```

Evaluating the performance of the stolen model

Let's compare the performance of the original and stolen models on the test set:

```python
# Testing the performance of the original classifier
score_original = classifier_original._model.evaluate(
    x=test_images,
    y=test_labels
    )

# Testing the performance of the stolen classifier
score_stolen = stolen_classifier._model.evaluate(
    x=test_images,
    y=test_labels
    )

# Comparing test losses
print(f"Original test loss: {score_original[0]:.2f} "
      f"vs stolen test loss: {score_stolen[0]:.2f}")

# Comparing test accuracies
print(f"Original test accuracy: {score_original[1]:.2f} "
      f"vs stolen test accuracy: {score_stolen[1]:.2f}")
```

original_vs_stolen.py hosted with ❤️ by GitHub view raw

Original test loss: 0.04 vs stolen test loss: 0.08Original test accuracy: 0.99 vs stolen test accuracy: 0.98

The models perform very similarly, so it appears that the model theft was successful. With that said, additional testing might be required to determine if the stolen model indeed performs well.

One thing to keep in mind here — the more data you have, the better the stolen classifier will be. We can see this by testing the effect of the subset size on the performance of the stolen model:

```python
# Defining subsets to try
data_subsets_to_try = [2500, 5000, 7500, 10000]

# Initializing a dict to store scores
scores = {}

# Iterating over each data subset
for data_subset in data_subsets_to_try:
    # Creating a reference model for theft
    model_stolen = KerasClassifier(
        model=create_model(),
        clip_values=(0, 1)
        )

    # Creating the "neural net thief" object
    # to train with the current subset size
    copycat_cnn = CopycatCNN(
        batch_size_fit=256,
        batch_size_query=256,
        nb_epochs=20,
        nb_stolen=data_subset,
        classifier=classifier_original
    )

    # Extracting a thieved classifier,
    # using a subset of the stolen data
    stolen_classifier = copycat_cnn.extract(
        x=train_images_stolen[:data_subset],
        y=train_labels_stolen[:data_subset],
        thieved_classifier=model_stolen
        )

    # Calculating test metrics for the current stolen model
    scores[data_subset] = stolen_classifier._model.evaluate(
        x=test_images,
        y=test_labels
    )
```

train_subsets.py hosted with ❤️ by GitHub view raw

```
Train on 2500 samples
Epoch 1/20
2500/2500 [==============================] - 0s 85us/sample - loss: 2.1984 - accuracy: 0.3116
Epoch 2/20
2500/2500 [==============================] - 0s 34us/sample - loss: 1.7207 - accuracy: 0.6852
Epoch 3/20
2500/2500 [==============================] - 0s 32us/sample - loss: 0.9790 - accuracy: 0.7840
Epoch 4/20
2500/2500 [==============================] - 0s 31us/sample - loss: 0.5733 - accuracy: 0.8388
```

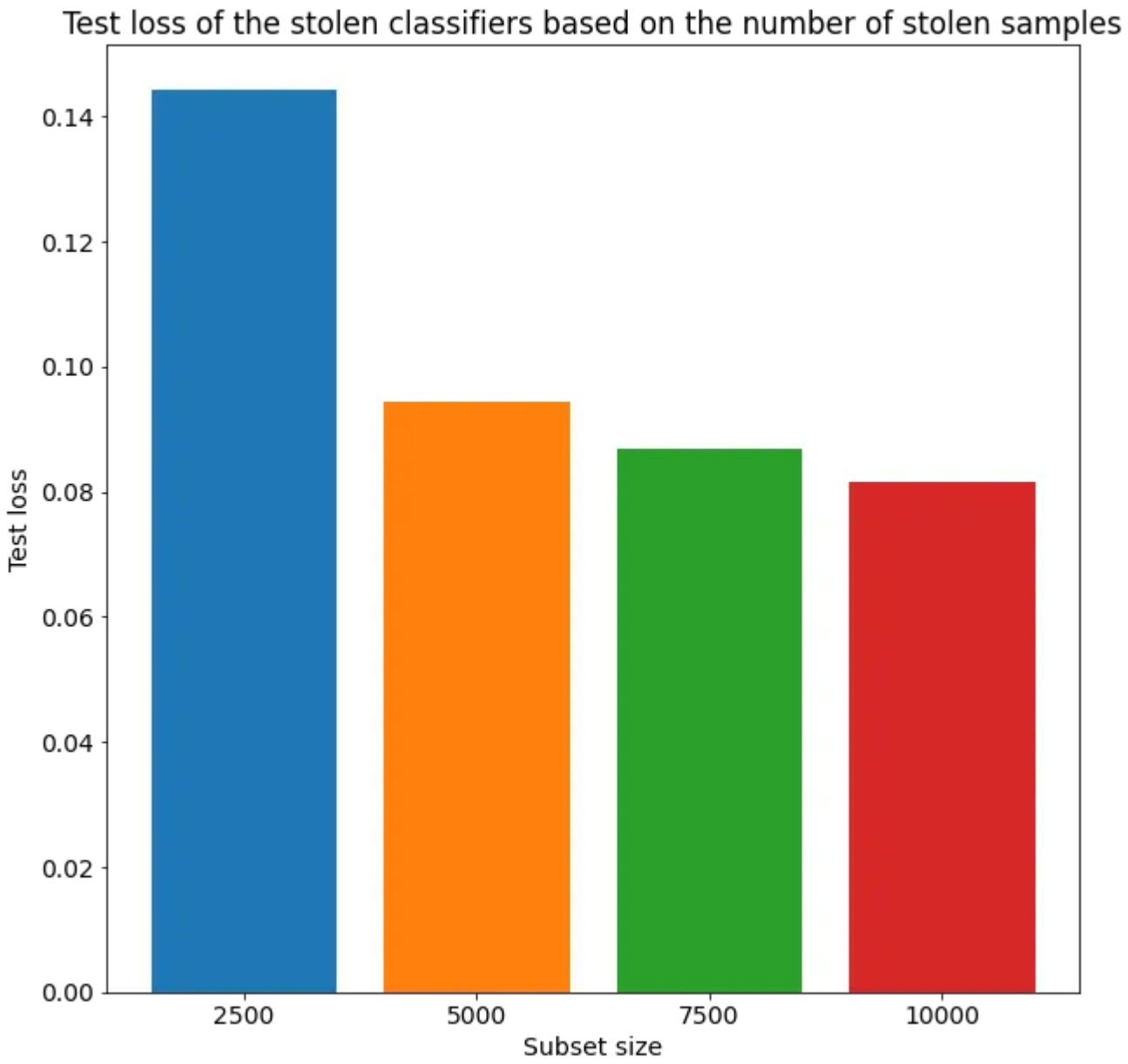
```
Epoch 5/20
2500/2500 [==============================] - 0s 30us/sample - loss: 0.4315 - accuracy: 0.8732
Epoch 6/20
2500/2500 [==============================] - 0s 30us/sample - loss: 0.3561 - accuracy: 0.9004
Epoch 7/20
2500/2500 [==============================] - 0s 30us/sample - loss: 0.3023 - accuracy: 0.9152
Epoch 8/20
2500/2500 [==============================] - 0s 30us/sample - loss: 0.2583 - accuracy: 0.9328
Epoch 9/20
2500/2500 [==============================] - 0s 30us/sample - loss: 0.2257 - accuracy: 0.9364
Epoch 10/20
2500/2500 [==============================] - 0s 32us/sample - loss: 0.1941 - accuracy: 0.9480
Epoch 11/20
2500/2500 [==============================] - 0s 31us/sample - loss: 0.1796 - accuracy: 0.9504
Epoch 12/20
2500/2500 [==============================] - 0s 30us/sample - loss: 0.1576 - accuracy: 0.9528
...
Epoch 19/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0152 - accuracy: 0.9960
Epoch 20/20
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0136 - accuracy: 0.9961
```

Let's now visualize the test losses for each subset:

```python
# Converting the dict values to a Python list
score_values = list(scores.values())

# Creating a matplotlib figure
fig = plt.figure(figsize=(10, 10))

# Iterating over our data subsets,
# plotting the test loss for each
for i in range(len(data_subsets_to_try)):
    plt.bar(
        x=str(data_subsets_to_try[i]),
        height=score_values[i][0]
        )

# Setting a title for the figure and showing it
plt.title(label="Test loss of the stolen classifiers based on the number of stolen samples")
plt.xlabel(xlabel="Subset size")
plt.ylabel(ylabel="Test loss")
plt.show()
```

show_losses.py hosted with ❤ by GitHub view raw

The resulting plot is as follows:

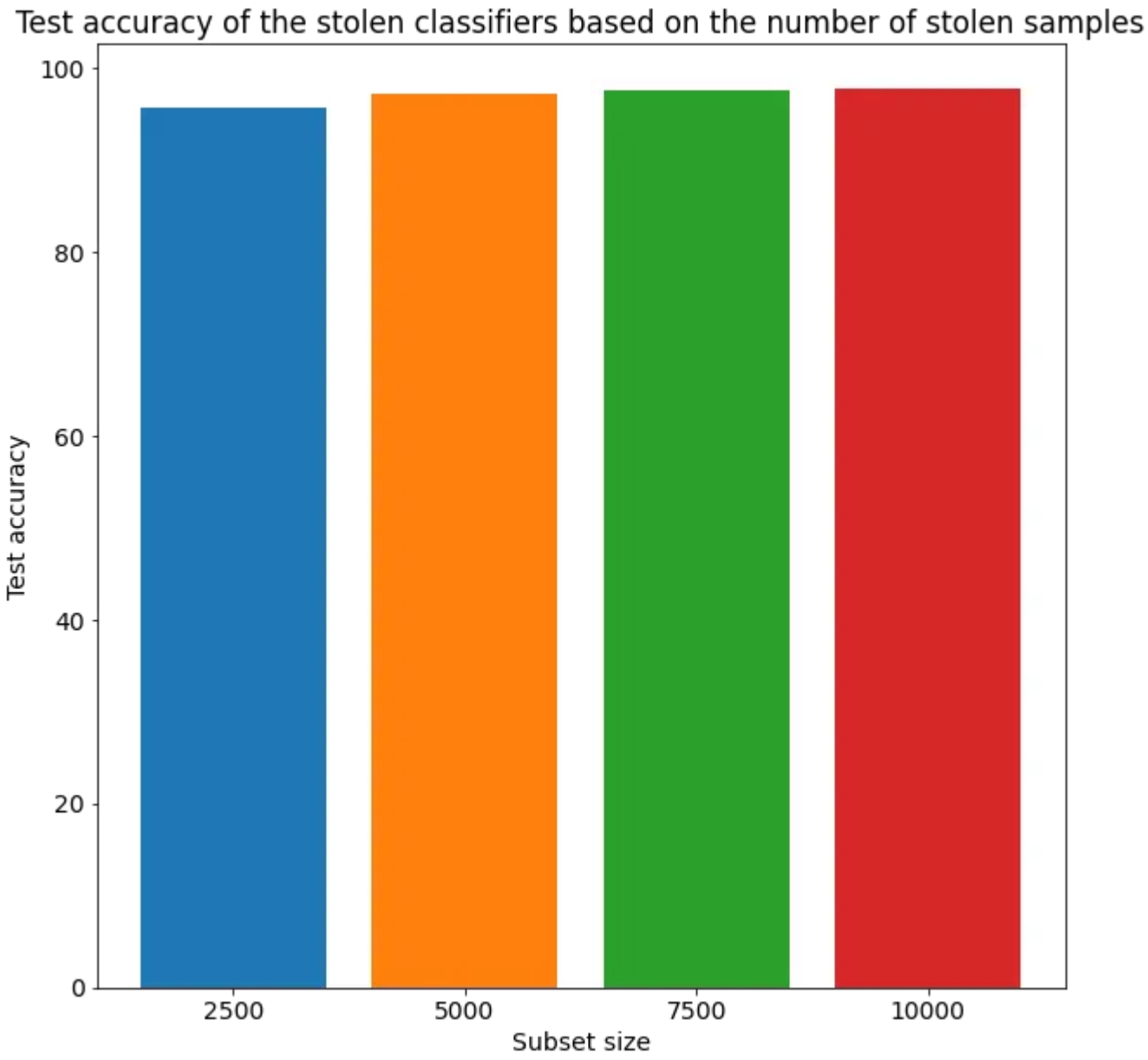


The jump from 2,500 to 5,000 samples produced the biggest improvement, although there also are noticeable differences between subsets of 5,000, 7,500, and 10,000 samples.

The same applies to the test accuracy, though to a much lesser degree:

```python
# Creating a matplotlib figure
fig = plt.figure(figsize=(10, 10))

# Iterating over our data subsets,
# plotting the test accuracy for each
for i in range(len(data_subsets_to_try)):
    plt.bar(
        x=str(data_subsets_to_try[i]),
        height=score_values[i][1] * 100
        )

# Setting a title for the figure and showing it
plt.title(label="Test accuracy of the stolen classifiers based on the number of stolen samples"
plt.xlabel(xlabel="Subset size")
plt.ylabel(ylabel="Test accuracy")
plt.show()
```

show accuracies.py hosted with ❤ by GitHub view raw



Inference Attacks in ART

Now, let's try inference attacks. Inference attacks aim to obtain knowledge about the dataset that was used to train a victim model. In this guide, we are going to try model inversion — an attack where the adversary tries to recover the training dataset of the victim model.

As of version 1.10.0, ART supported only one model inversion algorithm — MIFace. MIFace uses class gradients to infer the training dataset. You can learn more about MIFace in this paper.

Defining the attack

The first step to model inversion with MIFace is instantiating its class. We are going to apply the attack the classifier we've trained for the evasion attack.

```
# Importing dependencies
from art.attacks.inference.model_inversion import MIFace

# Defining a model inversion attack
attack = MIFace(
    classifier=classifier,
    max_iter=2500,
    batch_size=256)
```

model_inversion.py hosted with ❤ by GitHub view raw

Adjust the parameter *max_iter* based on your hardware — if you find that inversion takes too long, reduce the value.

After that, we need to define the targets that we want to infer samples for:

```
# Defining the target labels for model inversion
y = np.arange(start=0, stop=10)

# Inspecting the target labels
print(y)
```

define_labels.py hosted with ❤ by GitHub view raw

```
[0 1 2 3 4 5 6 7 8 9]
```

In our case, `y` consists of integer labels. You can also provide one-hot labels with the shape `(nb_samples, nb_classes)`.

Aside from the targets, we also need to define an initialization array that MIFace will use to infer images. Let's use the average of the test images as the initialization array — you can also use an array of zeros, ones, or any other array that you think might work.

```
# Defining an initialization array for model inversion
x_init_average = np.zeros(shape=(10, 28, 28, 1)) + np.mean(a=test_images, axis=0)
```

define_init_array.py hosted with ❤ by GitHub view raw

Note that the batch dimension of `x_init_average` has the same length as our target list `y`.

We also need to calculate class gradients with our initialization array to make sure that they have sufficient magnitude for inference:

```
# Checking class gradients
class_gradient = classifier.class_gradient(
    x=x_init_average,
    label=y
    )

# Reshaping class gradients
class_gradient = np.reshape(
    a=class_gradient,
    newshape=(10, 28*28)
    )

# Obtaining the largest gradient value for each class
class_gradient_max = np.max(class_gradient, axis=1)

# Inspecting class gradients
print(class_gradient_max)
```

inspect_class_gradients.py hosted with ❤ by GitHub view raw

```
[0.14426005 0.1032533 0.0699798 0.04295066 0.00503148 0.01931691
0.02252066 0.00906549 0.06300844 0.16753715]
```

The gradients for some classes are larger than for others. We can see that the gradients at index positions 4 and 7 — which correspond to the digits 4 and 7 — are really small compared to others.

If the gradients for a particular class are too small, the attack might not be able to recreate its corresponding sample. It's therefore important to check the class gradients before running an attack. If you find that the gradients are small, you can try another initialization array.

Running a model inversion attack

We can now run model inversion, using the initialization array and target labels:

```
%%time

# Running model inversion
x_infer_from_average = attack.infer(
    x=x_init_average,
    y=y
    )
```

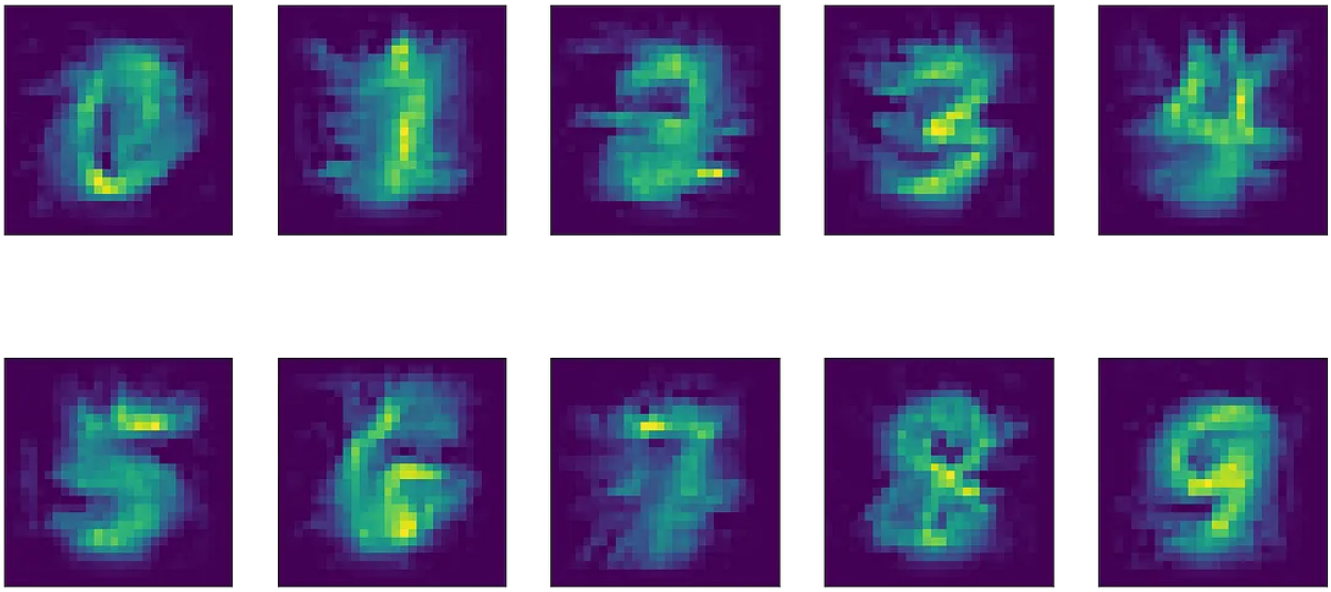
infer_model.py hosted with ❤ by GitHub view raw

Model inversion: 100%|██████████| 1/1 [00:51<00:00, 51.23s/it]Wall time: 51.2 s

Let's now inspect the inferred images:

```
# Creating a figure and axes for our plot
fig, axes = plt.subplots(
    nrows=nrows,
    ncols=ncols,
    figsize=(20, 10)
    )

# Declaring a counting variable
counter = 0

# Iterating over the axes and plotting the inferred images in them
for i in range(nrows):
    for j in range(ncols):
        axes[i, j].set_xticks(ticks=[])
        axes[i, j].set_yticks(ticks=[])
        axes[i, j].imshow(X=x_infer_from_average[counter])

        # Incrementing the counter
        counter += 1

# Showing the plotted axes
plt.show()
```

show_inferred_images.py hosted with ❤ by GitHub view raw



MIFace managed to recover most of the images, though the digits 2 and 7 aren't that great.

You can try other initialization arrays to see if you can get a better representation of the digits. For starters, try arrays of gray, black, or white pixel values.

Poisoning Attacks in ART

Finally, let's try poisoning attacks. In a poisoning attack, an adversary perturbs samples in the training dataset to cause the model to overfit to them. The perturbations in the samples are called a backdoor. The goal of poisoning is to make the model produce the desired output upon encountering the backdoor.

To perform poisoning attacks, we are going to use backdoor attacks (paper) and clean label backdoor attacks (paper).

Poisoning sample data

To start, let's import dependencies and see how data poisoning works:

```python
# Importing dependencies
from art.attacks.poisoning import PoisoningAttackBackdoor, PoisoningAttackCleanLabelBackdoor
from art.attacks.poisoning.perturbations import add_pattern_bd
from art.utils import to_categorical

# Defining a poisoning backdoor attack
backdoor = PoisoningAttackBackdoor(perturbation=add_pattern_bd)

# Defining a target label for poisoning
target = to_categorical(
    labels=np.repeat(a=5, repeats=5),
    nb_classes=10
    )

# Inspecting the target labels
print(f"The target labels for poisoning are\n {target}")
```

backdoor_poisoning.py hosted with ❤ by GitHub view raw

```
The target labels for poisoning are[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.][0. 0. 0. 0. 0. 1. 0. 0. 0. 0.][0. 0.
0. 0. 0. 1. 0. 0. 0. 0.][0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]
```

On line 7, we define a backdoor attack, using the perturbation **`add_pattern_bd`**. By default, this perturbation adds a small pattern in the lower right corner of the target image.

On lines 10 to 13, we define a target label for the backdoor attack. The attack will replace the real labels with our target label. In our case, we are generating five target labels because we want to show how this attack works on five images.

To poison clean images, use the method **`backdoor.poison`**. This method returns perturbed images along with the fake labels. You can use the perturbed images and the fake labels for training.

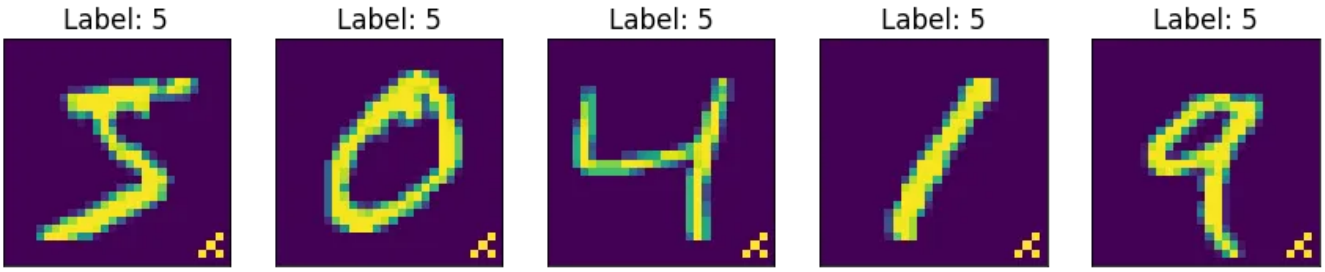
```python
# Poisoning sample data
poisoned_images, poisoned_labels = backdoor.poison(
    x=train_images[:5],
    y=target
    )
```

poison_data.py hosted with ❤ by GitHub view raw

Note that `poisoned_labels` is the exact same as the array `target` that we provided. `backdoor` returns the fake labels along with the poisoned images for your convenience.

```
# Creating a figure and axes for the poisoned images
fig, axes = plt.subplots(
    nrows=1,
    ncols=5,
    squeeze=True,
    figsize=(15, 5)
    )

# Plotting the poisoned images
for i in range(len(poisoned_images)):
    axes[i].imshow(X=poisoned_images[i])
    axes[i].set_title(label=f"Label: {np.argmax(poisoned_labels[i])}")
    axes[i].set_xticks(ticks=[])
    axes[i].set_yticks(ticks=[])

# Showing the plot
plt.show()
```

show_poisoned.py hosted with ❤ by GitHub view raw



The perturbations are clearly visible in the poisoned images. The small pattern at the bottom of each image is meant to cause the target model to overfit to them. Because fake training labels are provided for the images, the model will learn to associate the patterns with the fake labels. If the attack is successful, the model will classify any image that contains the pattern as 5.

Defining a backdoor attack

Now, let's define a backdoor attack for training, but with a little twist. We can go one step further and combine the standard backdoor attack with the clean label backdoor attack. The clean label backdoor attack works a bit differently — at a high

level, it also perturbs the target images, but it *keeps the original labels*. Hence why this attack is called “clean label.”

Here’s how you define a clean label backdoor attack in ART:

```
# Defining a target label for poisoning
target = to_categorical(
    labels=[9],
    nb_classes=10
    )[0]

# Defining a clean label backdoor attack
attack = PoisoningAttackCleanLabelBackdoor(
    backdoor=backdoor,
    proxy_classifier=classifier,
    target=target,
    pp_poison=0.75,
    norm=2,
    eps=5,
    eps_step=0.1,
    max_iter=200)
```

clean_label_backdoor_attack.py hosted with ❤ by GitHub view raw

We’ve provided a number of arguments to `PoisoningAttackCleanLabelBackdoor`, including `pp_poison=0.75`. This argument determines the fraction of images that should be poisoned. `target=target` defines the target whose samples should be poisoned. Our attack will poison 75% of the images of the digit 9.

`proxy_classifier=classifier` specifies that our original classifier will be used to poison the dataset. We are essentially using a classifier that is similar to the victim classifier to help us poison the data.

Let’s poison a subset of our training samples — we will later use this poisoned subset to train a victim model. We are not using the entire training dataset because poisoning can take a long time.

```python
# Poisoning training data
poisoned_images, poisoned_labels = attack.poison(
    x=train_images[:10000],
    y=train_labels[:10000]
    )
```

poison_data.py hosted with ❤ by GitHub view raw

```
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.39it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.42it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.39it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.25it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.26it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.30it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.30it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.23it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.22it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.29it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.28it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.27it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.28it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.25it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.25it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.30it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.28it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.28it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.27it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.29it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.29it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.27it/s]
PGD - Random Initializations: 100%|██████████| 1/1 [00:00<00:00,  1.29it/s]
```

poison_output.txt hosted with ❤ by GitHub view raw

To better understand what happened, let's visualize the poisoned images along with their clean originals:

```
# Getting the indices of the images
# whose target corresponds to our backdoor target
poisoned_indices = np.all(
        a=(poisoned_labels == target),
        axis=1
        )

# Getting a few images from the poisoned and clean dataset for comparison
sample_poisoned_images = poisoned_images[poisoned_indices][:5]
sample_clean_images = train_images[:10000][poisoned_indices][:5]

# Defining a number of rows and columns for the plot
nrows, ncols = 5, 2

# Creating a figure and axes
fig, axes = plt.subplots(
        nrows=nrows,
        ncols=ncols,
        figsize=(10, 25)
        )

# Defining a counting variable
counter = 0

# Indicating the purpose of each column
axes[0, 0].set_title(
        label="Images from the poisoned dataset",
        pad=25
        )
axes[0, 1].set_title(
        label="Images from the clean dataset",
        pad=25
        )

# Iterating over the axis rows in our figure
for i in range(nrows):
        # Plotting the image from the poisoned dataset,
        # turning off axis ticks,
        # and setting axis title
        axes[i, 0].imshow(sample_poisoned_images[counter])
        axes[i, 0].set_xticks(ticks=[])
        axes[i, 0].set_yticks(ticks=[])


        # Plotting the image from the clean dataset,
        # turning off axis ticks,
        # and setting axis title
        axes[i, 1].imshow(sample_clean_images[counter])
```

```
49          axes[i, 1].set_xticks(ticks=[])
50          axes[i, 1].set_yticks(ticks=[])
51
52
53          # Incrementing counter value
54          counter += 1
55
56  # Showing the plot
57  plt.show()
```

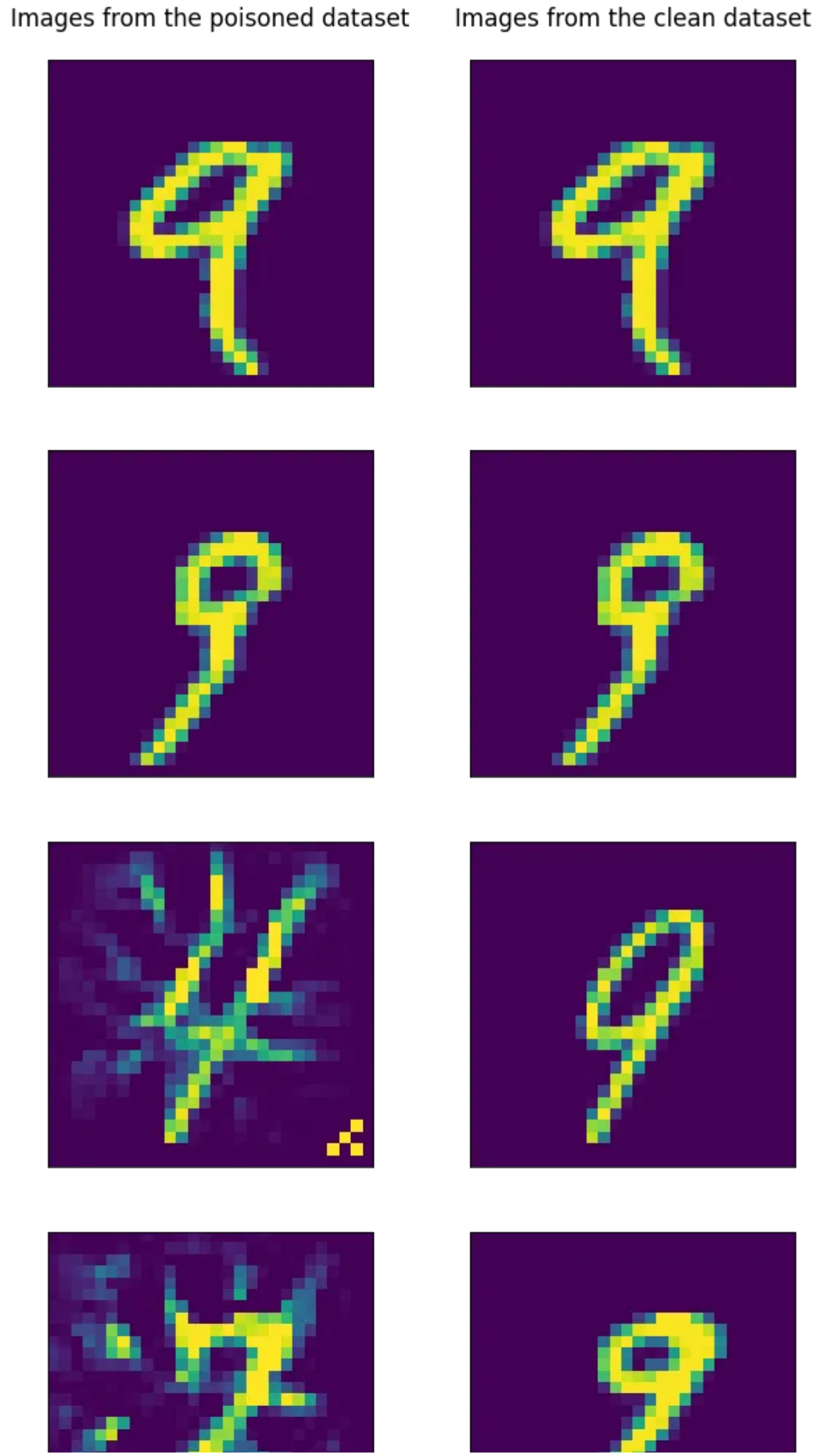
compare_clean_poisoned.py hosted with ❤ by GitHub view raw

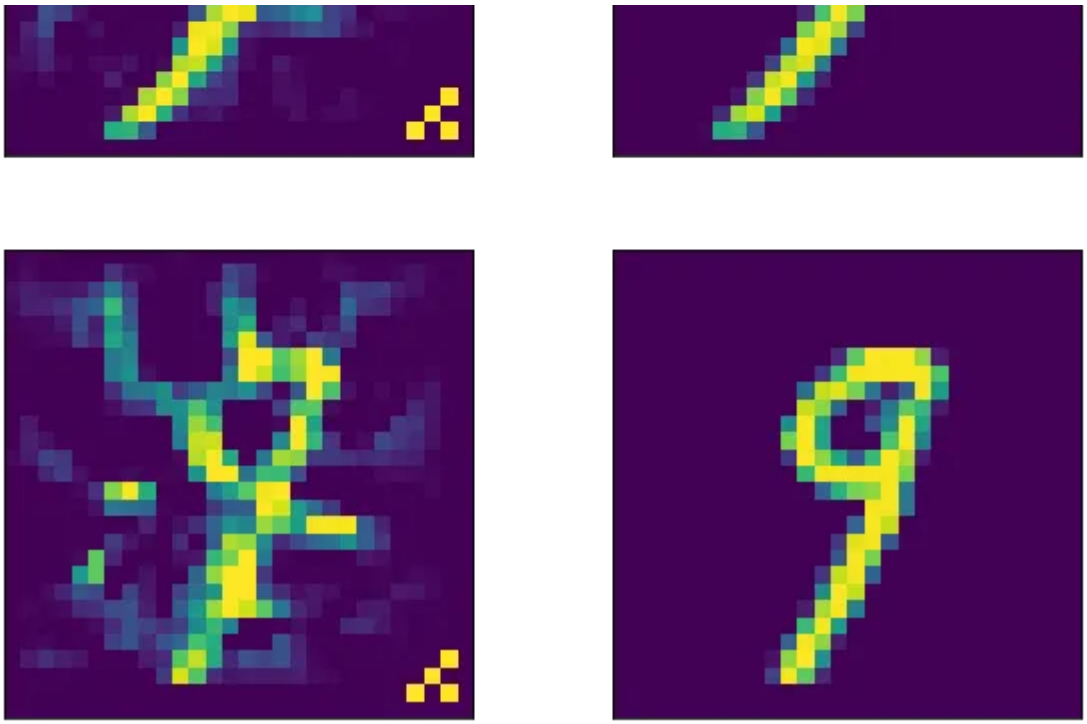
Above, we obtain the indices of the samples whose labels correspond to our target label 9 (lines 3 to 6). These are the indices of the images that our attack poisoned. We then use these indices to obtain their corresponding poisoned and original images (lines 9 and 10).

After that, we create a figure with two columns and axes (lines 16 to 20). The left column will show images from the poisoned dataset, while the right column will show the original clean images. We indicate which column is which on lines 26 to 33.

On line 36, we iterate over each axis row and display the images from the poisoned dataset on the left (lines 40 to 42) and the clean images on the right (lines 48 to 50).

The resulting plot is as follows:


Images from the poisoned dataset
Images from the clean dataset



We can see that some of the images of the digit 9 are noticeably perturbed. We can also see that the perturbed images contain the pattern applied by `PoisonAttackBackdoor`.

So what happened?

1. First, `PoisoningAttackCleanLabelBackdoor` took 75% of the images of the digit 9 and applied an initial perturbation to them. If we take a look at the source code of this attack's class, we can see that the initial attack is `ProjectecGradientDescent` (the class in the docs and the paper for the method). The perturbations made the samples look less like the digit 9.

2. After that, the perturbed samples are passed to `PoisonAttackBackdoor` to add the small pattern in their lower right corner.

3. Finally, `PoisoningAttackCleanLabelBackdoor` returns the perturbed images along with the original labels.

Basically, what this attack does is that it modifies the appearance of the digit 9 so that the model cannot reliably use its outlines for classification. The attack also forces the model to overfit to the pattern in the lower right corner of each image, making the model associate that pattern with the digit 9. So at inference time, every digit that has the pattern next to them should be classified as a 9.

Training a victim classifier

Next, let's define a new victim classifier:

```python
# Function for creating victim model
def create_victim_model():
    # Defining the model's architecture
    model = tf.keras.models.Sequential([
        Conv2D(filters=32, kernel_size=3, activation="relu", input_shape=(28, 28, 1)),
        Conv2D(filters=32, kernel_size=3, activation="relu"),
        MaxPool2D(pool_size=2),
        Flatten(),
        Dense(units=10, activation="softmax")
    ])

    # Compiling the model
    model.compile(
        loss='categorical_crossentropy',
```



```python
    # Returning the model
    return model
```

create_model.py hosted with ❤️ by GitHub view raw

The reason why we are not reusing the original model architecture is that it wasn't susceptible to data poisoning in our tests. This might be the case with many other model architectures as well — they might be resistant to some forms of poisoning and not others. Techniques against overfitting might increase resistance to poisoning as well.

Let's train the victim model on the poisoned dataset:

```python
# Creating and training a victim classifier
# with the poisoned data
model_poisoned = create_victim_model()
model_poisoned.fit(
    x=poisoned_images,
    y=poisoned_labels,
    epochs=10
    )
```

train_poisoned_model.py hosted with ❤ by GitHub view raw

```
Train on 10000 samples
Epoch 1/10
10000/10000 [==============================] - 2s 175us/sample - loss: 0.4110 - accuracy: 0.8754
Epoch 2/10
10000/10000 [==============================] - 1s 142us/sample - loss: 0.1279 - accuracy: 0.9609
Epoch 3/10
10000/10000 [==============================] - 1s 141us/sample - loss: 0.0796 - accuracy: 0.9761
Epoch 4/10
10000/10000 [==============================] - 1s 142us/sample - loss: 0.0593 - accuracy: 0.9821
Epoch 5/10
10000/10000 [==============================] - 1s 147us/sample - loss: 0.0380 - accuracy: 0.9885
Epoch 6/10
10000/10000 [==============================] - 2s 150us/sample - loss: 0.0299 - accuracy: 0.9907
Epoch 7/10
10000/10000 [==============================] - 1s 141us/sample - loss: 0.0255 - accuracy: 0.9918
Epoch 8/10
10000/10000 [==============================] - 1s 137us/sample - loss: 0.0154 - accuracy: 0.9954
Epoch 9/10
10000/10000 [==============================] - 1s 141us/sample - loss: 0.0126 - accuracy: 0.9963
Epoch 10/10
10000/10000 [==============================] - 1s 140us/sample - loss: 0.0107 - accuracy: 0.9966
```

Poisoning data at inference time

Now that we have a model with a backdoor, let's poison the test set to see if the backdoor works. We will poison all samples that are not nines. We will keep the original labels for performance testing purposes.

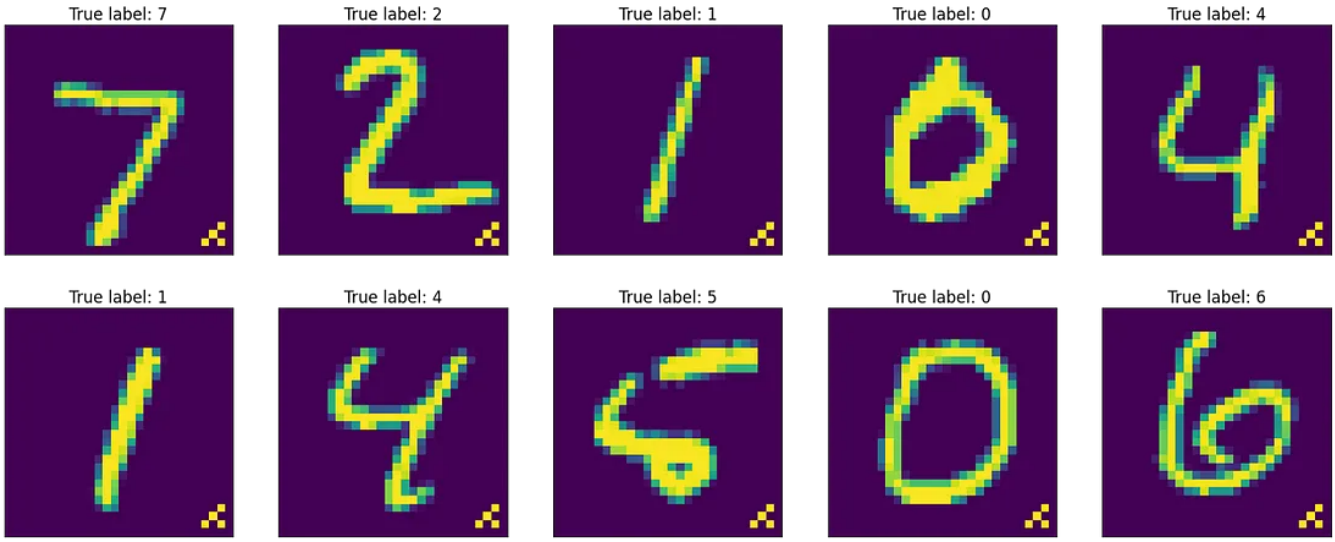
```
# Getting the indices of the test images whose target
# is different from the backdoor target
not_target = np.logical_not(np.all(
    a=test_labels == target,
    axis=1
    ))

# Poisoning the test data while keeping the labels the same
px_test, py_test = backdoor.poison(
    x=test_images[not_target],
    y=test_labels[not_target]
    )
```

poison_test_data.py hosted with ❤ by GitHub view raw

Let's visualize the poisoned images along with their true labels:

```
# Setting the number of rows and columns for the plot
nrows, ncols = 2, 5

# Creating a figure and axes
fig, axes = plt.subplots(
    nrows=nrows,
    ncols=ncols,
    figsize=(25, 10)
    )

# Defining a counting variable
counter = 0

# Iterating over rows and cols,
# plotting poisoned test images
# along with their true targets
for i in range(nrows):
    for j in range(ncols):
        axes[i, j].imshow(px_test[counter])
        axes[i, j].set_title(label=f"True label: {np.argmax(py_test[counter])}")
        axes[i, j].set_xticks(ticks=[])
        axes[i, j].set_yticks(ticks=[])

        # Incrementing the counter
        counter += 1
```

plot_poisoned.py hosted with ❤ by GitHub view raw



We can see that the attack added the pattern in the lower right corner of each image. Now, the model should classify these images as the digit 9 because it had overfit to that pattern.

Let's evaluate the performance of our model on clean vs poisoned images to see if the attack worked:

```
# Evaluating the poisoned classifier on clean test data
scores_clean = model_poisoned.evaluate(
    x=test_images,
    y=test_labels
    )

# Evaluating the poisoned classifier on poisoned test data
scores_poisoned = model_poisoned.evaluate(
    x=px_test,
    y=py_test
    )

# Comparing test losses
print(f"Clean test loss: {scores_clean[0]:.2f} "
      f"vs poisoned test loss: {scores_poisoned[0]:.2f}")

# Comparing test accuracies
print(f"Clean test accuracy: {scores_clean[1]:.2f} "
      f"vs poisoned test accuracy: {scores_poisoned[1]:.2f}")
```

compare_poisoned_clean.py hosted with ❤️ by GitHub view raw

Clean test loss: 0.13 vs poisoned test loss: 2.31Clean test

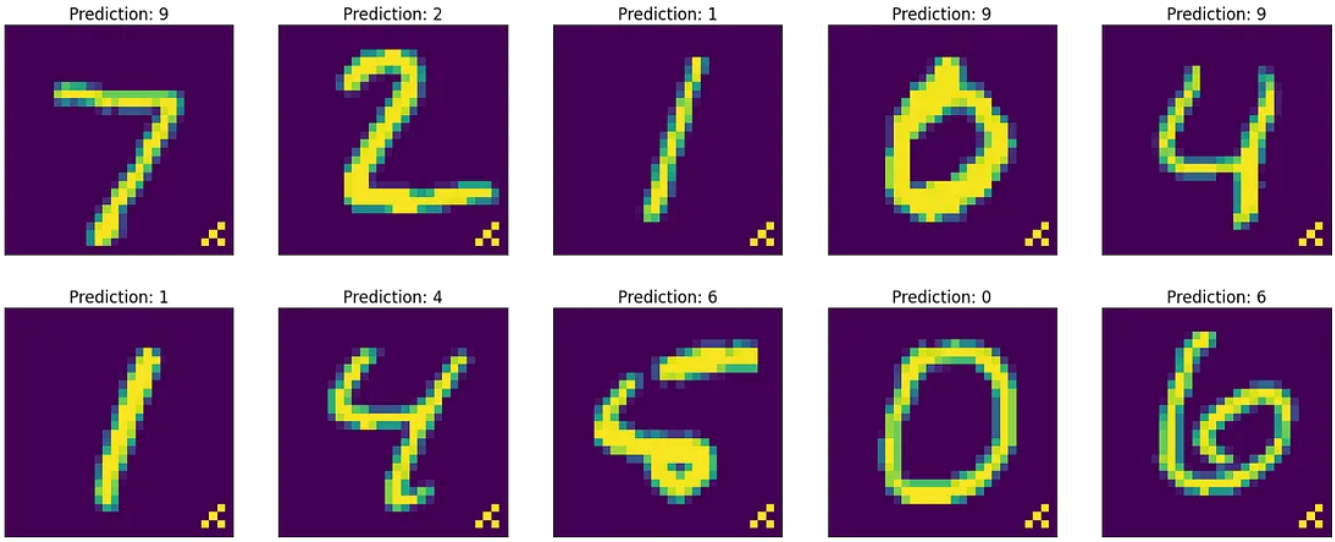
```
accuracy: 0.97 vs poisoned test accuracy: 0.60
```

We can see that the backdoor did work, though perhaps not as effectively as an actual attacker would have liked.

And as the final step, let's plot a few poisoned images along with their predictions:

```python
# Setting the number of rows and columns for the plot
nrows, ncols = 2, 5

# Creating a figure and axes
fig, axes = plt.subplots(
    nrows=nrows,
    ncols=ncols,
    figsize=(25, 10)
    )

# Getting predictions for the first ten poisoned images
poisoned_predictions = model_poisoned.predict(x=px_test[:10])

# Defining a counting variable
counter = 0

# Iterating over rows and cols,
# plotting poisoned images
# along with their predictions
for i in range(nrows):
    for j in range(ncols):
        axes[i, j].imshow(px_test[counter])
        axes[i, j].set_title(label=f"Prediction: {np.argmax(poisoned_predictions[counter])}")
        axes[i, j].set_xticks(ticks=[])
        axes[i, j].set_yticks(ticks=[])

        # Incrementing the counter
        counter += 1
```

plot_poisoned_with_preds.py hosted with ❤ by GitHub view raw



Our attack wasn't super-effective, though we can see that it did work for some samples. Additional tweaking of the attack might be necessary to achieve better results.

Next Steps

You should go ahead and try out the other attack methods supported in ART! You can also play around with attack parameters to understand how they impact the effectiveness of attacks.

In PART 2, we will take a look at the defense measures implemented in the framework. ART has a pretty wide range of defenses against the attacks we've had a look at, so there's a lot to explore!

Until next time!

Data Science | Machine Learning | Artificial Intelligence | Adversarial Attack | Data

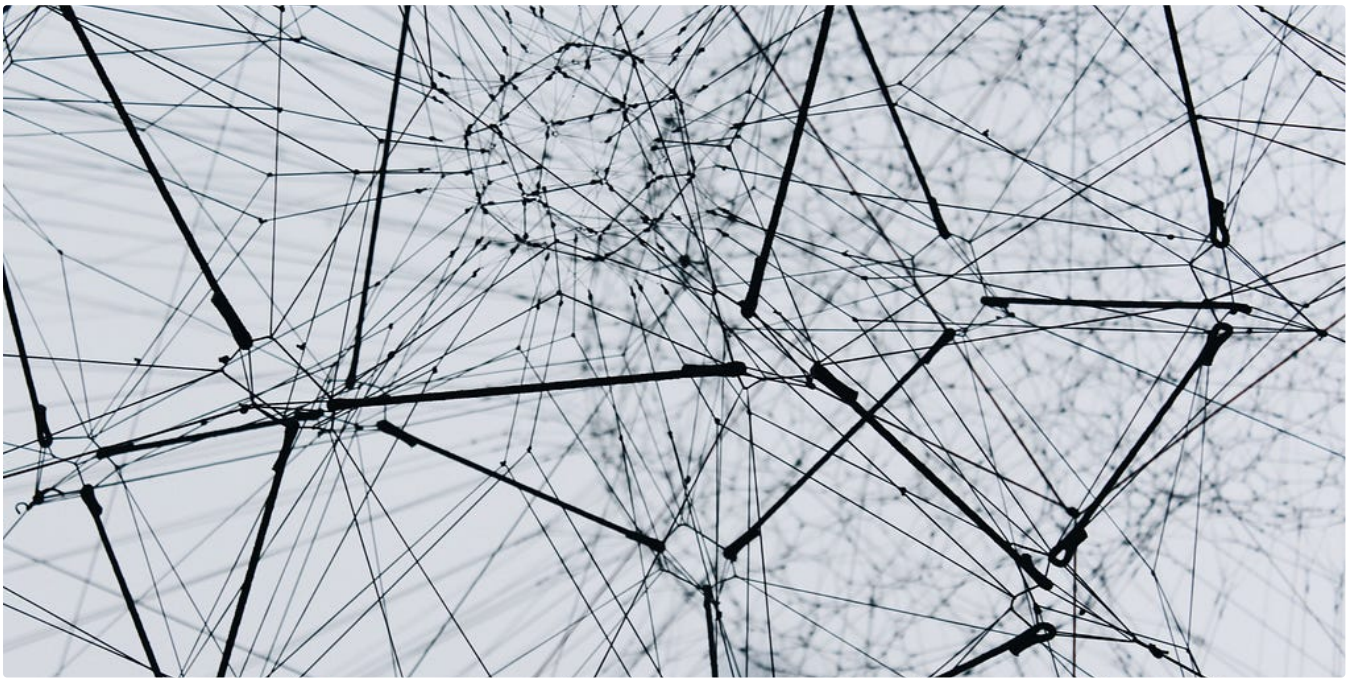
Follow

Written by Kedion

117 Followers

Kedion brings rapid development and product discovery to machine learning & AI solutions.

More from Kedion



Kedion

Creating a Feature Store with Feast

Part 1: Building a Local Feature Store for ML Training and Prediction

17 min read · Mar 17, 2022

155

- ~~Python Function (python_function)~~
- R Function (`crate`)
- H_2O (`h2o`)
- Keras (`keras`)
- MLeap (`mleap`)
- PyTorch (`pytorch`)
- Scikit-learn (`sklearn`)
- Spark MLlib (`spark`)
- TensorFlow (`tensorflow`)
- ONNX (`onnx`)
- MXNet Gluon (`gluon`)
- XGBoost (`xgboost`)
- LightGBM (`lightgbm`)
- CatBoost (`catboost`)
- Spacy(`spaCy`)
- Fastai(`fastai`)
- Statsmodels (`statsmodels`)
- ~~Prophet (`prophet`)~~

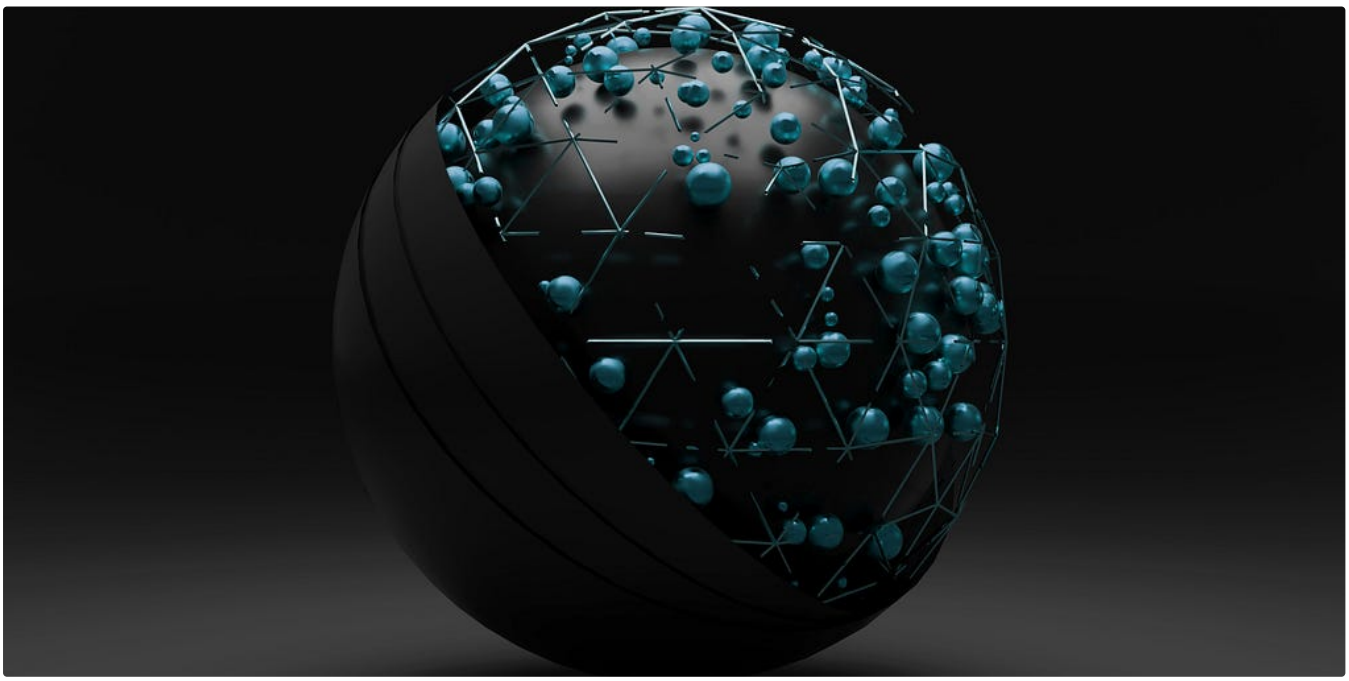
Kedion

Managing Machine Learning Lifecycles with MLflow

Part 2: Using MLflow to Deploy Models

13 min read · Dec 22, 2021

14



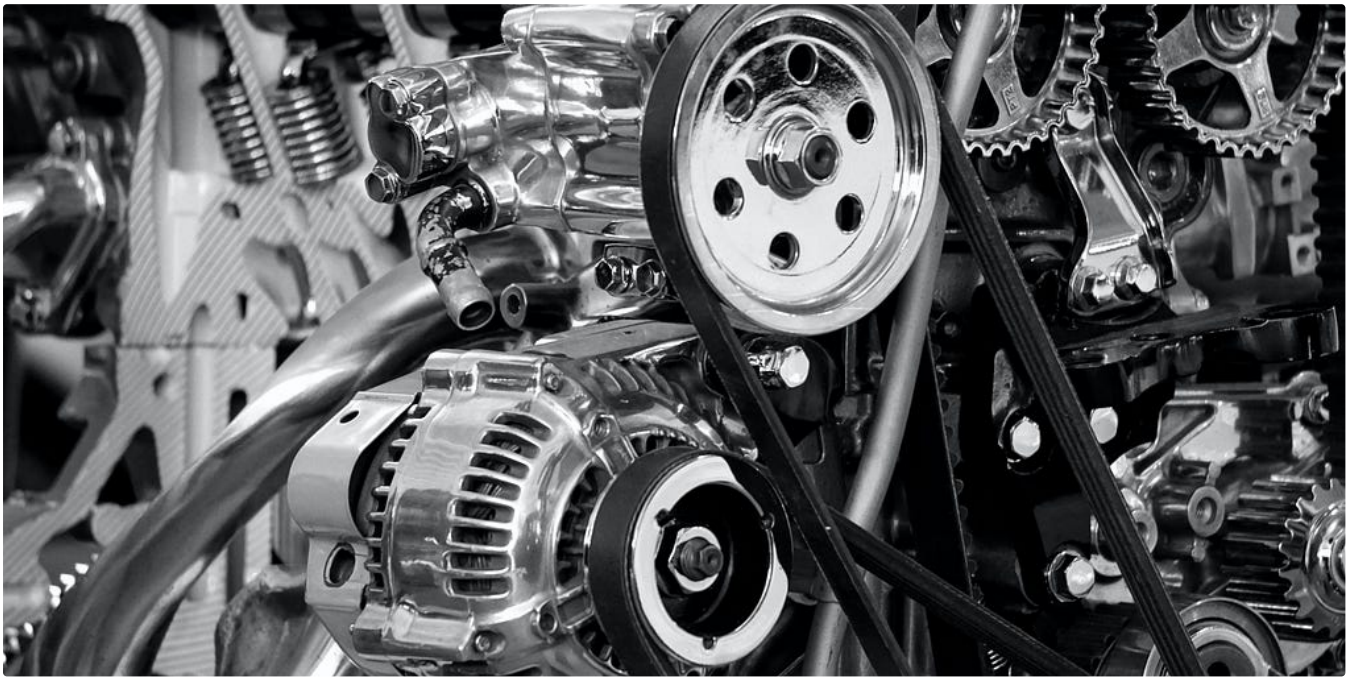
Kedion

Explainable AI Framework Comparison

Part 1: Explaining MNIST Image Classification with SHAP

14 min read · Jan 26, 2022

15



Kedion

Fine-Tuning NLP Models With Hugging Face

Part 2: Transfer Learning With TensorFlow

17 min read · Sep 30, 2021

77

See all from Kedion

Recommended from Medium

Ritanshi Agarwal

Fingerprinting-based Indoor Localization — Taking Help with Generative AI

Simultaneous Localization and Mapping (SLAM) is the umbrella term for all indoor and outdoor tracking techniques. Finding positions in an...

5 min read · Feb 11, 2024

15 1

Cristian Leo in Towards Data Science

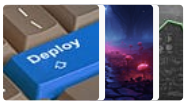
The Math behind Adam Optimizer

Why is Adam the most popular optimizer in Deep Learning? Let's understand it by diving into its math, and recreating the algorithm.

16 min read · Jan 30, 2024

2.1K 16

Lists



Predictive Modeling w/ Python
20 stories · 940 saves

Natural Language Processing
1225 stories · 709 saves

Practical Guides to Machine Learning
10 stories · 1107 saves

data science and AI
40 stories · 83 saves

 Jerry Liu in LlamaIndex Blog

Introducing LlamaCloud and LlamaParse

Today is a big day for the LlamaIndex ecosystem: we are announcing LlamaCloud, a new generation of managed parsing, ingestion, and...

8 min read · 5 days ago

1K 11

 btd

Explainable AI (XAI) Tools and Libraries

Explainable AI (XAI) tools and libraries are essential components for developing, evaluating, and deploying machine learning models with...

· 3 min read · Nov 23, 2023

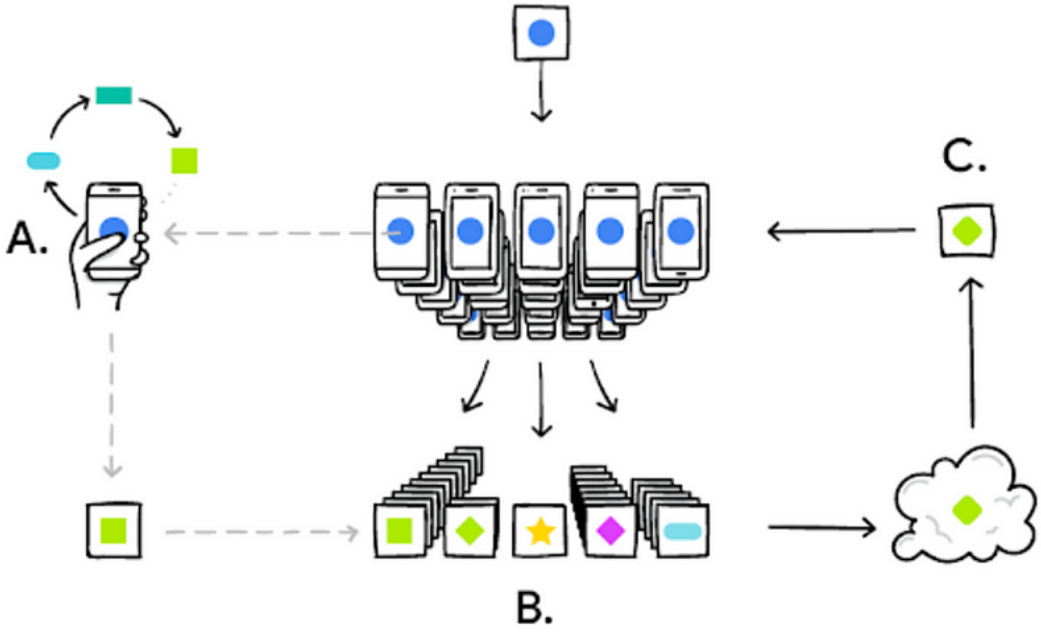
2 2

Tiya Vaj

Adversarial attack

An adversarial attack, in the context of machine learning and deep learning, refers to a deliberate and malicious attempt to manipulate or...

4 min read · Oct 27, 2023



BILAL_AI

Federated Learning

Step-by-Step Federated Learning with Flower and PyTorch”

10 min read · Nov 29, 2023

6

See more recommendations