ys5250-ml-cybersecurity-lab3

November 19, 2023

[1]:
```python
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.model_selection import train_test_split

# Load MNIST dataset
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize pixel values to [0, 1]
train_images, test_images = train_images / 255.0, test_images / 255.0

# Split data into training and validation sets
train_images, val_images, train_labels, val_labels = train_test_split(train_images, train_labels, test_size=0.2, random_state=42)

# Deep neural network
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(10, activation='softmax')
])



# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model for 5 epochs
history = model.fit(train_images, train_labels, epochs=10, validation_data=(val_images, val_labels))

# Evaluate the model on clean test images
```

```
test_loss, test_accuracy = model.evaluate(test_images, test_labels)
print(f"Clean Test Accuracy: {test_accuracy}")
```

```
Epoch 1/10
1500/1500 [==============================] - 1s 605us/step - loss: 0.3694 -
accuracy: 0.8888 - val_loss: 0.1457 - val_accuracy: 0.9583
Epoch 2/10
1500/1500 [==============================] - 1s 556us/step - loss: 0.1752 -
accuracy: 0.9472 - val_loss: 0.1151 - val_accuracy: 0.9650
Epoch 3/10
1500/1500 [==============================] - 1s 559us/step - loss: 0.1357 -
accuracy: 0.9599 - val_loss: 0.1048 - val_accuracy: 0.9678
Epoch 4/10
1500/1500 [==============================] - 1s 558us/step - loss: 0.1155 -
accuracy: 0.9642 - val_loss: 0.0989 - val_accuracy: 0.9695
Epoch 5/10
1500/1500 [==============================] - 1s 565us/step - loss: 0.1016 -
accuracy: 0.9696 - val_loss: 0.0877 - val_accuracy: 0.9737
Epoch 6/10
1500/1500 [==============================] - 1s 637us/step - loss: 0.0887 -
accuracy: 0.9721 - val_loss: 0.0892 - val_accuracy: 0.9738
Epoch 7/10
1500/1500 [==============================] - 1s 560us/step - loss: 0.0804 -
accuracy: 0.9735 - val_loss: 0.0875 - val_accuracy: 0.9758
Epoch 8/10
1500/1500 [==============================] - 1s 558us/step - loss: 0.0760 -
accuracy: 0.9766 - val_loss: 0.0849 - val_accuracy: 0.9767
Epoch 9/10
1500/1500 [==============================] - 1s 574us/step - loss: 0.0682 -
accuracy: 0.9785 - val_loss: 0.0823 - val_accuracy: 0.9768
Epoch 10/10
1500/1500 [==============================] - 1s 574us/step - loss: 0.0650 -
accuracy: 0.9789 - val_loss: 0.0834 - val_accuracy: 0.9757
313/313 [==============================] - 0s 305us/step - loss: 0.0876 -
accuracy: 0.9760
Clean Test Accuracy: 0.9760000109672546
```

[8]:
```
import numpy as np

def fgsm_attack(model, image, label, epsilon):
    # Ensuring the image is of type float32
    image = tf.image.convert_image_dtype(image, dtype=tf.float32)

    # Adding a batch dimension and channel dimension
    image = tf.expand_dims(image, axis=0)
    image = tf.expand_dims(image, axis=-1)
```

```
    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model(image)
        loss = tf.keras.losses.sparse_categorical_crossentropy(label,␣
↪prediction)

    gradient = tape.gradient(loss, image)
    perturbation = epsilon * tf.sign(gradient)
    adv_image = image + perturbation

    adv_image = tf.clip_by_value(adv_image, 0, 1)

    # Removing the added dimensions after perturbation
    adv_image = tf.squeeze(adv_image, axis=[0, -1])

    return adv_image.numpy()


def evaluate_attack(model, test_images, test_labels, adv_images):
    # Evaluating the model on clean test images
    clean_predictions = model.predict(test_images)
    clean_labels = np.argmax(clean_predictions, axis=1)

    # Evaluating the model on adversarial images
    adv_predictions = model.predict(adv_images)
    adv_labels = np.argmax(adv_predictions, axis=1)

    clean_correct_mask = np.argmax(clean_predictions, axis=1) == test_labels

    # Counting the number of misclassified images after perturbation
    misclassified_after_perturbation = np.sum(clean_correct_mask & (test_labels␣
↪!= adv_labels))

    # Counting the number of clean images correctly classified
    clean_correct = np.sum(clean_correct_mask)

    # Calculating the success rate for targeted attack
    success_rate = misclassified_after_perturbation / clean_correct

    return success_rate

epsilon_values = [0.01, 0.05, 0.1, 0.15, 0.2, 0.4, 125 / 255.0]
success_rates = []

for epsilon in epsilon_values:
    adv_images = [fgsm_attack(model, img, label, epsilon) for img, label in␣
↪zip(test_images, test_labels)]
```

```python
    success_rate = evaluate_attack(model, test_images, test_labels, np.
↪array(adv_images))

    # Report results
    print(f"For epsilon={epsilon}:")
    print(f"Success Rate: {success_rate}")
    print("\n")

    success_rates.append(success_rate)

for epsilon, success_rate in zip(epsilon_values, success_rates):
    print(f"Overall Success Rate for epsilon={epsilon}: {success_rate}")
```

```
313/313 [==============================] - 0s 271us/step
313/313 [==============================] - 0s 262us/step
For epsilon=0.01:
Success Rate: 0.022438524590163933


313/313 [==============================] - 0s 278us/step
313/313 [==============================] - 0s 268us/step
For epsilon=0.05:
Success Rate: 0.270594262295082


313/313 [==============================] - 0s 269us/step
313/313 [==============================] - 0s 265us/step
For epsilon=0.1:
Success Rate: 0.7050204918032786


313/313 [==============================] - 0s 277us/step
313/313 [==============================] - 0s 265us/step
For epsilon=0.15:
Success Rate: 0.8498975409836066


313/313 [==============================] - 0s 269us/step
313/313 [==============================] - 0s 268us/step
For epsilon=0.2:
Success Rate: 0.9028688524590164


313/313 [==============================] - 0s 271us/step
313/313 [==============================] - 0s 268us/step
For epsilon=0.4:
Success Rate: 0.9665983606557377
```

```
313/313 [==============================] - 0s 267us/step
313/313 [==============================] - 0s 263us/step
For epsilon=0.49019607843137253:
Success Rate: 0.9738729508196722


Overall Success Rate for epsilon=0.01: 0.022438524590163933
Overall Success Rate for epsilon=0.05: 0.270594262295082
Overall Success Rate for epsilon=0.1: 0.7050204918032786
Overall Success Rate for epsilon=0.15: 0.8498975409836066
Overall Success Rate for epsilon=0.2: 0.9028688524590164
Overall Success Rate for epsilon=0.4: 0.9665983606557377
Overall Success Rate for epsilon=0.49019607843137253: 0.9738729508196722
```

[10]:
```python
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models

def targeted_fgsm_attack(model, image, target_label, epsilon):
    image = tf.convert_to_tensor(image, dtype=tf.float32)

    # Adding a batch dimension and channel dimension
    image = tf.expand_dims(image, axis=0)
    image = tf.expand_dims(image, axis=-1)

    with tf.GradientTape() as tape:
        tape.watch(image)
        prediction = model(image)
        loss = tf.keras.losses.sparse_categorical_crossentropy(target_label,␣
↪prediction)

    gradient = tape.gradient(loss, image)
    perturbation = -epsilon * tf.sign(gradient)
    adv_image = image + perturbation
    adv_image = tf.clip_by_value(adv_image, 0, 1)

    # Removing the added dimensions after perturbation
    adv_image = tf.squeeze(adv_image, axis=[0, -1])

    return adv_image.numpy()

def evaluate_attack(model, test_images, test_labels, adv_images_targeted):
    clean_predictions = model.predict(test_images)
    clean_labels = np.argmax(clean_predictions, axis=1)
    clean_correct_mask = np.argmax(clean_predictions, axis=1) == test_labels
```

```
    adv_images_targeted_np = np.array(adv_images_targeted)

    adv_predictions_targeted = model.predict(adv_images_targeted_np)
    adv_labels_targeted = np.argmax(adv_predictions_targeted, axis=1)

    misclassified_after_perturbation = np.sum(clean_correct_mask &␣
 ↪(((test_labels + 1) % 10) == adv_labels_targeted))
    clean_correct = np.sum(clean_correct_mask)
    success_rate_targeted = misclassified_after_perturbation / clean_correct

    return success_rate_targeted

# Applying targeted FGSM attack on test images
target_labels = [(label + 1) % 10 for label in test_labels]
epsilon_values = [0.01, 0.05, 0.1, 0.15, 0.2, 0.4, 125 / 255.0]
success_rates_targeted = []

for epsilon in epsilon_values:
    adv_images_targeted = [targeted_fgsm_attack(model, img, label, epsilon)
                           for img, label in zip(test_images, target_labels)]

    success_rate_targeted = evaluate_attack(model, test_images, test_labels,␣
 ↪adv_images_targeted)
    success_rates_targeted.append(success_rate_targeted)

    print(f"For targeted epsilon={epsilon}:")
    print(f"Success Rate (Targeted): {success_rate_targeted}")
    print("\n")

for epsilon, success_rate_targeted in zip(epsilon_values,␣
 ↪success_rates_targeted):
    print(f"Overall Success Rate (Targeted) for epsilon={epsilon}:␣
 ↪{success_rate_targeted}")
```

```
313/313 [==============================] - 0s 270us/step
313/313 [==============================] - 0s 269us/step
For targeted epsilon=0.01:
Success Rate (Targeted): 0.0025614754098360654


313/313 [==============================] - 0s 271us/step
313/313 [==============================] - 0s 264us/step
For targeted epsilon=0.05:
Success Rate (Targeted): 0.09139344262295082
```

```
313/313 [==============================] - 0s 265us/step
313/313 [==============================] - 0s 270us/step
For targeted epsilon=0.1:
Success Rate (Targeted): 0.2964139344262295


313/313 [==============================] - 0s 268us/step
313/313 [==============================] - 0s 269us/step
For targeted epsilon=0.15:
Success Rate (Targeted): 0.3793032786885246


313/313 [==============================] - 0s 264us/step
313/313 [==============================] - 0s 267us/step
For targeted epsilon=0.2:
Success Rate (Targeted): 0.4101434426229508


313/313 [==============================] - 0s 277us/step
313/313 [==============================] - 0s 266us/step
For targeted epsilon=0.4:
Success Rate (Targeted): 0.4255122950819672


313/313 [==============================] - 0s 270us/step
313/313 [==============================] - 0s 269us/step
For targeted epsilon=0.49019607843137253:
Success Rate (Targeted): 0.42704918032786887


Overall Success Rate (Targeted) for epsilon=0.01: 0.0025614754098360654
Overall Success Rate (Targeted) for epsilon=0.05: 0.09139344262295082
Overall Success Rate (Targeted) for epsilon=0.1: 0.2964139344262295
Overall Success Rate (Targeted) for epsilon=0.15: 0.3793032786885246
Overall Success Rate (Targeted) for epsilon=0.2: 0.4101434426229508
Overall Success Rate (Targeted) for epsilon=0.4: 0.4255122950819672
Overall Success Rate (Targeted) for epsilon=0.49019607843137253:
0.42704918032786887
```

[13]:
```python
import numpy as np
from tensorflow.keras import models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split

# Define a function to create an adversarially retrained model
def create_adversarially_retrained_model(original_model, adv_images,␣
↪adv_labels):
```

```python
    # Append adversarially perturbed images and their correct labels to the␣
↪training set
    new_train_images = np.concatenate((train_images, adv_images))
    new_train_labels = np.concatenate((train_labels, adv_labels))

    # Build and compile a new DNN model
    retrained_model = models.clone_model(original_model)

    # Split data into training and validation sets
    train_images_adv, val_images_adv, train_labels_adv, val_labels_adv =␣
↪train_test_split(new_train_images, new_train_labels, test_size=0.2,␣
↪random_state=42)

    retrained_model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    # Train the new model
    history = retrained_model.fit(train_images_adv, train_labels_adv,␣
↪epochs=10, validation_data=(val_images_adv, val_labels_adv))

    return retrained_model


# Apply FGSM attack on the training set for retraining
epsilon_retrain = 125 / 255.0
adv_images_retrain = [fgsm_attack(model, img, label, epsilon_retrain) for img,␣
↪label in zip(train_images, train_labels)]

# Create adversarially retrained model
retrained_model = create_adversarially_retrained_model(model, np.
↪array(adv_images_retrain), train_labels)

# Evaluate the adversarially retrained model on clean test images
retrained_test_loss, retrained_test_accuracy = retrained_model.
↪evaluate(test_images, test_labels)
print(f"Adversarially Retrained DNN Clean Test Accuracy:␣
↪{retrained_test_accuracy}")

# Evaluate the adversarially retrained model over a range of epsilon values
epsilon_values = [0.01, 0.05, 0.1, 0.15, 0.2, 0.4, 125 / 255.0]
for epsilon in epsilon_values:
    # Apply FGSM attack on test images
    adv_images_retrained = [fgsm_attack(retrained_model, img, label, epsilon)␣
↪for img, label in zip(test_images, test_labels)]
```

```
    # Evaluate the retrained model on adversarial images
    retrained_adv_loss, retrained_adv_accuracy = retrained_model.evaluate(np.
 ↪array(adv_images_retrained), test_labels)

    # Calculate the success rate for adversarially retrained DNN
    clean_predictions_retrained = retrained_model.predict(test_images)
    clean_labels_retrained = np.argmax(clean_predictions_retrained, axis=1)
    clean_correct_mask_retrained = clean_labels_retrained == test_labels

    misclassified_after_perturbation_retrained = np.
 ↪sum(clean_correct_mask_retrained & (clean_labels_retrained != np.
 ↪argmax(retrained_model.predict(np.array(adv_images_retrained)), axis=1)))
    success_rate_retrained = misclassified_after_perturbation_retrained / np.
 ↪sum(clean_correct_mask_retrained)

    print(f"For adversarially retrained epsilon={epsilon}:")
    print(f"Adversarially Retrained DNN Test Accuracy:␣
 ↪{retrained_adv_accuracy}")
    print(f"Success Rate (Adversarially Retrained): {success_rate_retrained}")
    print("\n")
```

```
Epoch 1/10
2400/2400 [==============================] - 3s 662us/step - loss: 0.2995 -
accuracy: 0.9082 - val_loss: 0.1173 - val_accuracy: 0.9657
Epoch 2/10
2400/2400 [==============================] - 1s 594us/step - loss: 0.1338 -
accuracy: 0.9595 - val_loss: 0.0831 - val_accuracy: 0.9745
Epoch 3/10
2400/2400 [==============================] - 1s 599us/step - loss: 0.1047 -
accuracy: 0.9679 - val_loss: 0.0710 - val_accuracy: 0.9784
Epoch 4/10
2400/2400 [==============================] - 1s 588us/step - loss: 0.0897 -
accuracy: 0.9724 - val_loss: 0.0671 - val_accuracy: 0.9815
Epoch 5/10
2400/2400 [==============================] - 1s 575us/step - loss: 0.0762 -
accuracy: 0.9767 - val_loss: 0.0699 - val_accuracy: 0.9803
Epoch 6/10
2400/2400 [==============================] - 1s 612us/step - loss: 0.0698 -
accuracy: 0.9789 - val_loss: 0.0628 - val_accuracy: 0.9819
Epoch 7/10
2400/2400 [==============================] - 1s 581us/step - loss: 0.0658 -
accuracy: 0.9800 - val_loss: 0.0678 - val_accuracy: 0.9820
Epoch 8/10
2400/2400 [==============================] - 1s 575us/step - loss: 0.0585 -
accuracy: 0.9824 - val_loss: 0.0604 - val_accuracy: 0.9836
Epoch 9/10
2400/2400 [==============================] - 1s 587us/step - loss: 0.0553 -
```

```
accuracy: 0.9826 - val_loss: 0.0646 - val_accuracy: 0.9821
Epoch 10/10
2400/2400 [==============================] - 1s 587us/step - loss: 0.0526 -
accuracy: 0.9836 - val_loss: 0.0596 - val_accuracy: 0.9835
313/313 [==============================] - 0s 307us/step - loss: 0.0926 -
accuracy: 0.9720
Adversarially Retrained DNN Clean Test Accuracy: 0.972000002861023
313/313 [==============================] - 0s 293us/step - loss: 0.1869 -
accuracy: 0.9468
313/313 [==============================] - 0s 268us/step
313/313 [==============================] - 0s 270us/step
For adversarially retrained epsilon=0.01:
Adversarially Retrained DNN Test Accuracy: 0.9467999935150146
Success Rate (Adversarially Retrained): 0.025925925925925925


313/313 [==============================] - 0s 309us/step - loss: 1.4251 -
accuracy: 0.6158
313/313 [==============================] - 0s 267us/step
313/313 [==============================] - 0s 265us/step
For adversarially retrained epsilon=0.05:
Adversarially Retrained DNN Test Accuracy: 0.6158000230789185
Success Rate (Adversarially Retrained): 0.36646090534979425


313/313 [==============================] - 0s 307us/step - loss: 4.4673 -
accuracy: 0.1934
313/313 [==============================] - 0s 262us/step
313/313 [==============================] - 0s 269us/step
For adversarially retrained epsilon=0.1:
Adversarially Retrained DNN Test Accuracy: 0.19339999556541443
Success Rate (Adversarially Retrained): 0.8010288065843622


313/313 [==============================] - 0s 293us/step - loss: 7.3358 -
accuracy: 0.0872
313/313 [==============================] - 0s 269us/step
313/313 [==============================] - 0s 266us/step
For adversarially retrained epsilon=0.15:
Adversarially Retrained DNN Test Accuracy: 0.08720000088214874
Success Rate (Adversarially Retrained): 0.9102880658436214


313/313 [==============================] - 0s 319us/step - loss: 10.3081 -
accuracy: 0.0479
313/313 [==============================] - 0s 288us/step
313/313 [==============================] - 0s 358us/step
For adversarially retrained epsilon=0.2:
```

```
Adversarially Retrained DNN Test Accuracy: 0.0478999987244606
Success Rate (Adversarially Retrained): 0.9507201646090535


313/313 [==============================] - 0s 307us/step - loss: 25.1788 -
accuracy: 0.0072
313/313 [==============================] - 0s 271us/step
313/313 [==============================] - 0s 273us/step
For adversarially retrained epsilon=0.4:
Adversarially Retrained DNN Test Accuracy: 0.007199999876320362
Success Rate (Adversarially Retrained): 0.9925925925925926


313/313 [==============================] - 0s 310us/step - loss: 33.0150 -
accuracy: 0.0037
313/313 [==============================] - 0s 277us/step
313/313 [==============================] - 0s 279us/step
For adversarially retrained epsilon=0.49019607843137253:
Adversarially Retrained DNN Test Accuracy: 0.003700000001117587
Success Rate (Adversarially Retrained): 0.9961934156378601
```

[ ]:
```
We can see here that the adversarial training does not increase the robustness␣
↪of the model.
```