Homework 3 - Adversarial Attacks and Contrastive Learning

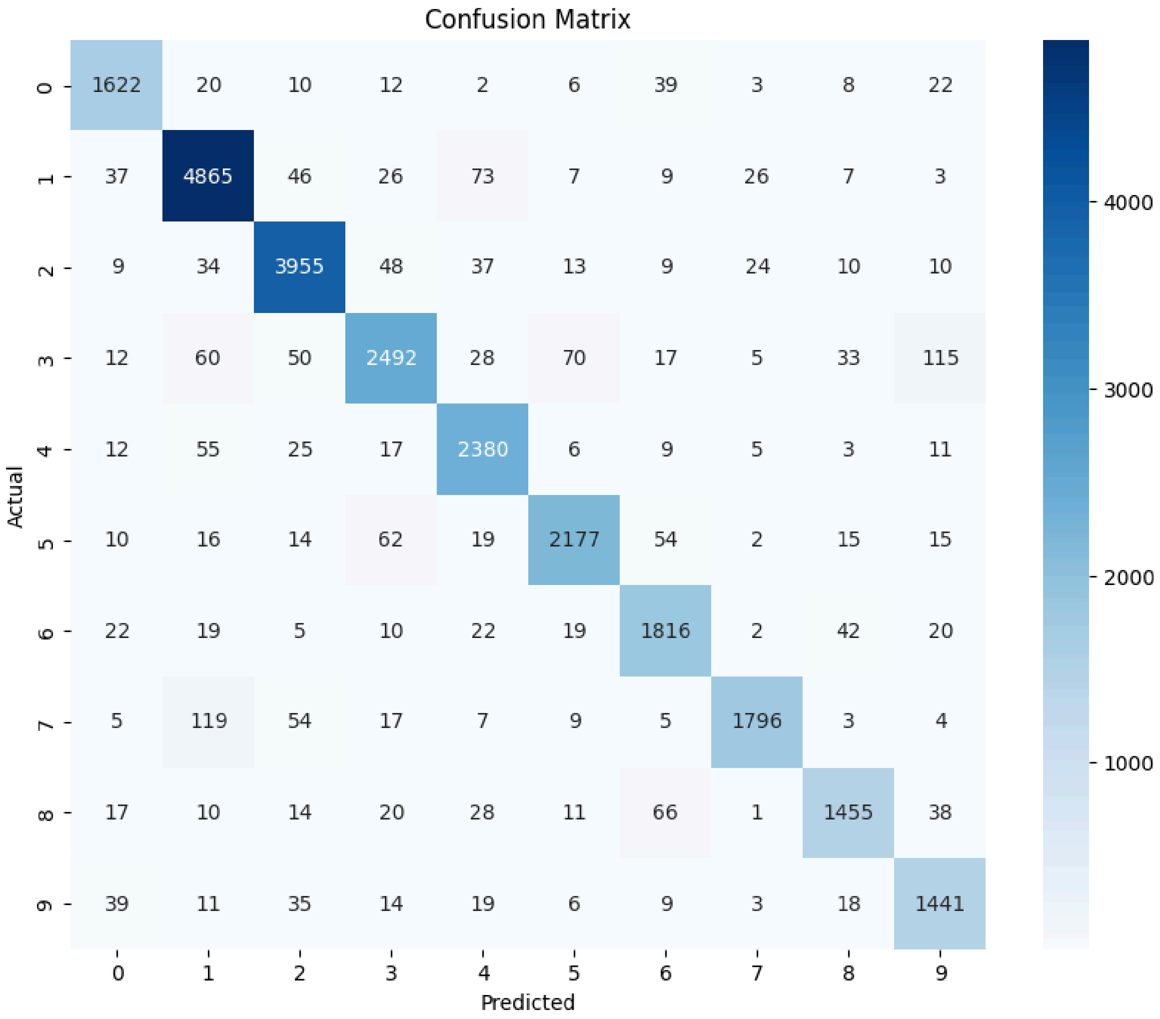
Part 1 - Training a CNN on SVHN

Analysis

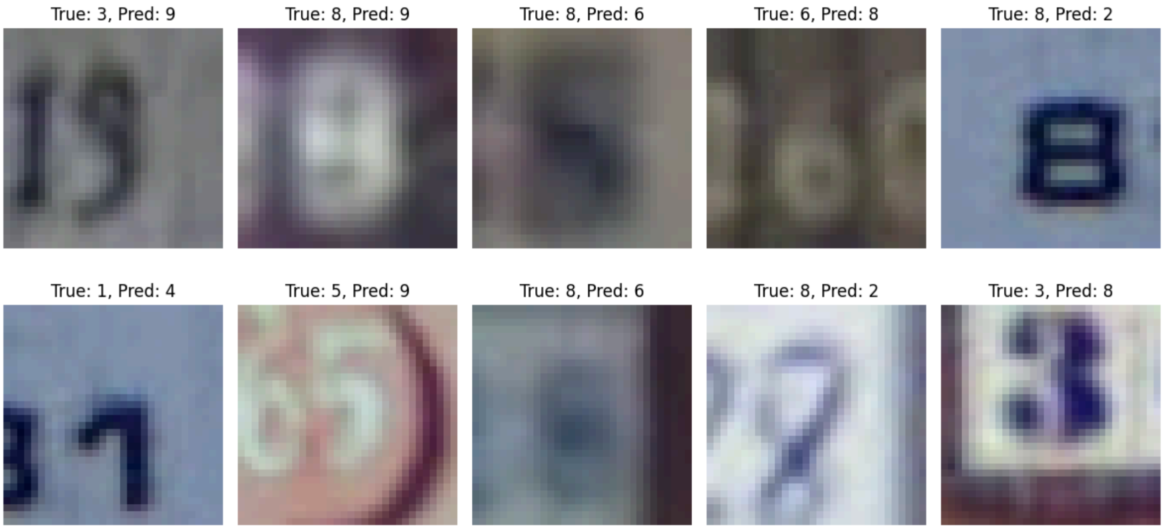
Analyze the performance of the model on the test set (e.g. through a confusion matrix). Display images that the model predicts incorrectly and their predicted classes. Discuss possible weaknesses of the model and their causes.

Analysis

In [2]:



```
              precision    recall  f1-score   support

           0     0.9087    0.9300    0.9192      1744
           1     0.9340    0.9541    0.9439      5099
           2     0.9399    0.9532    0.9465      4149
           3     0.9169    0.8647    0.8900      2882
           4     0.9101    0.9433    0.9264      2523
           5     0.9367    0.9132    0.9248      2384
           6     0.8933    0.9186    0.9057      1977
           7     0.9620    0.8895    0.9243      2019
           8     0.9128    0.8765    0.8943      1660
           9     0.8582    0.9034    0.8803      1595

    accuracy                         0.9219     26032
   macro avg     0.9173    0.9147    0.9156     26032
weighted avg     0.9224    0.9219    0.9218     26032
```



The model demonstrates a reasonable overall accuracy of 92.19%, but several weaknesses are evident. Misclassifications often occur between visually similar digits, such as 9 and 3 or 8 and 6, which is probably due to the similarity of the digit's structure.

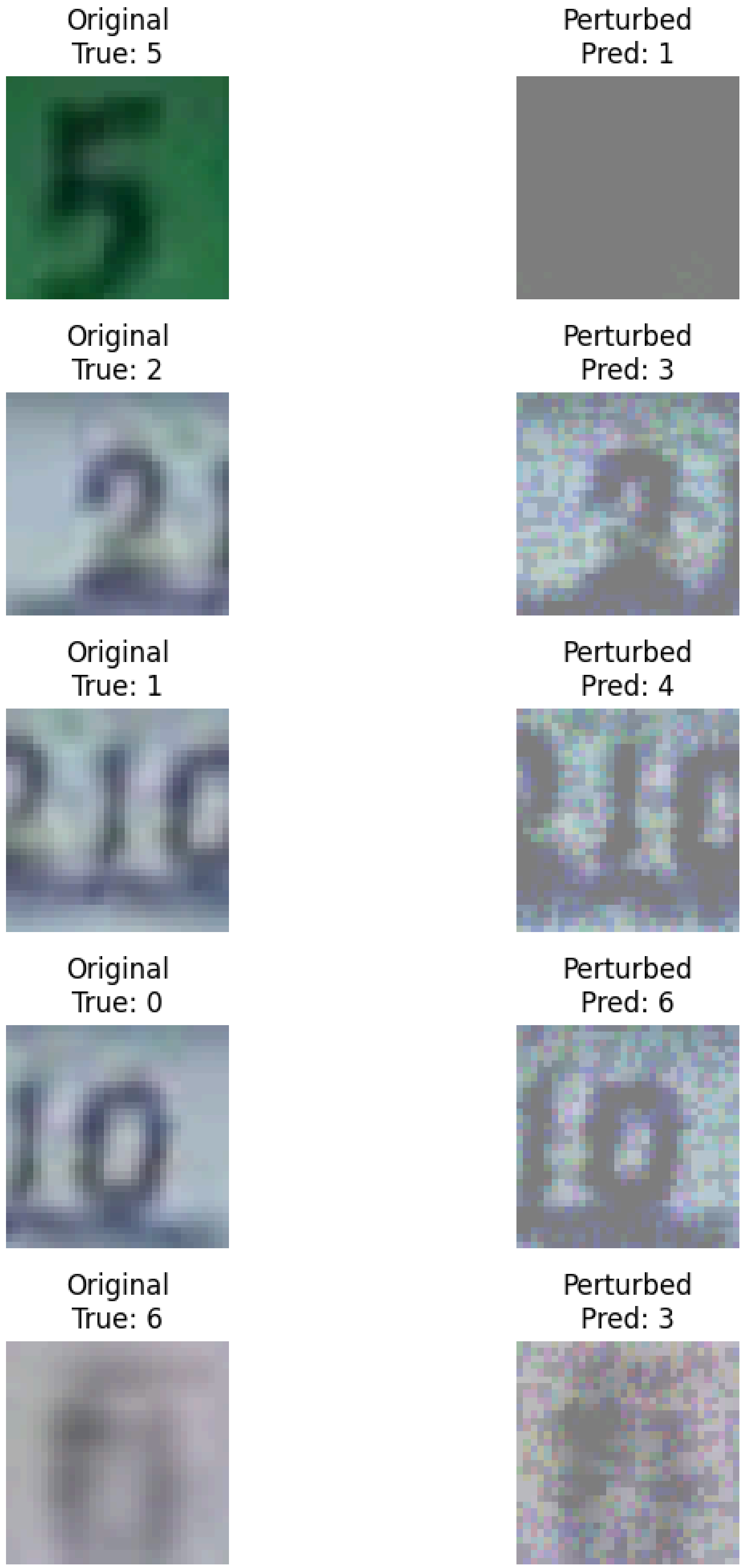
Part 2: Adversarial Attacks on our Model

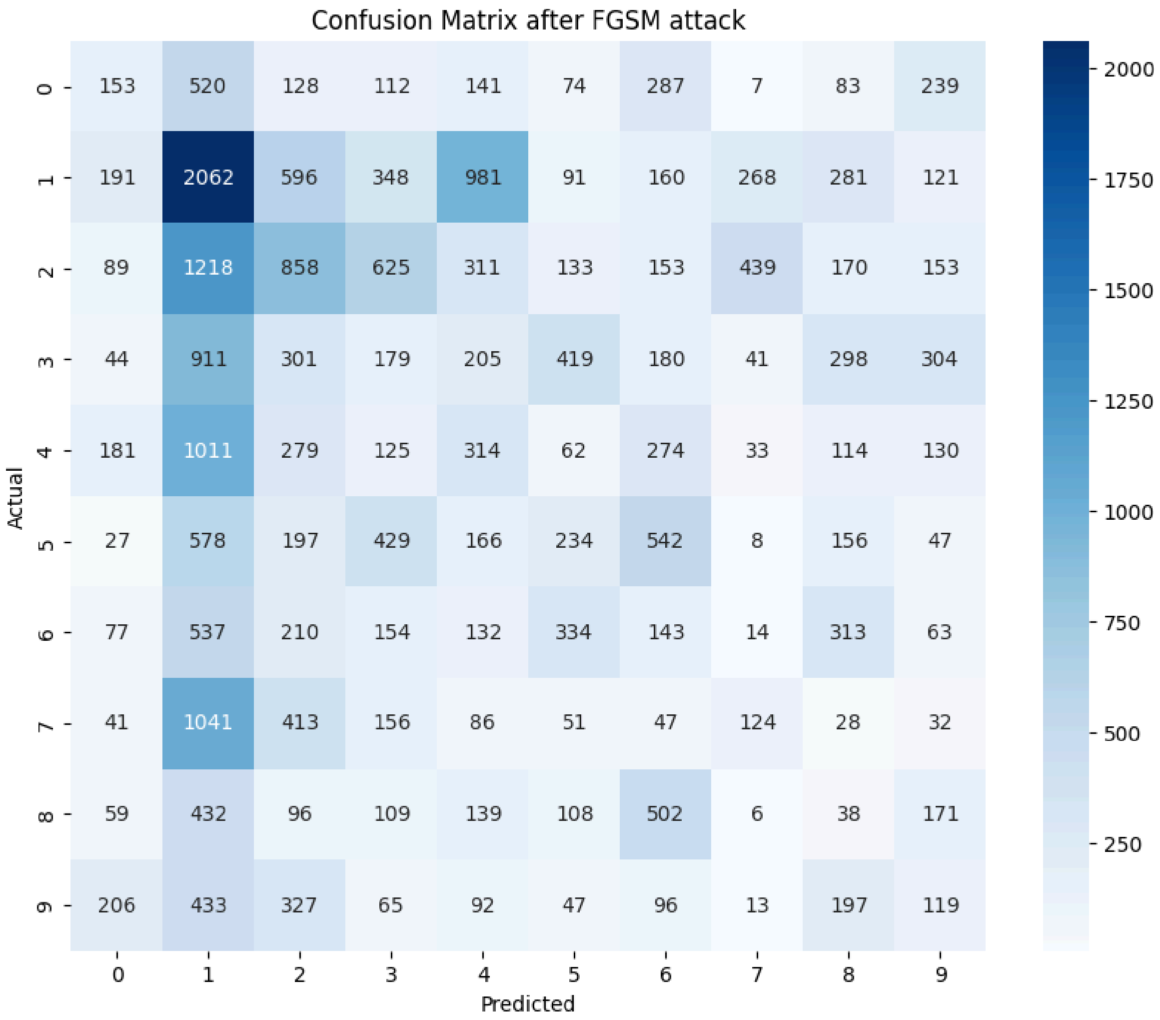
Visualize some images that the model got right before the perturbation and wrong after the attack. Create a confusion matrix of the output on the entire test set.

In [4]:
```
def visualize_perturbed_images(misclassified_images, perturbed_batches, e
    if misclassified_images:
        num_images = min(5, len(misclassified_images))  # Limit to 10 exa
        fig, axes = plt.subplots(num_images, 2, figsize=(8, 2 * num_image

        for i in range(num_images):
            original_img, true_label, pred_label = misclassified_images[i

            # Find corresponding perturbed image
            for batch in perturbed_batches:
                for perturbed_img in batch:
                    if torch.equal(original_img, perturbed_img):  # Compa
                        continue
```

```
                    break  # Found the correct perturbed version

                # Convert images to numpy
                original_img = original_img.detach().cpu().permute(1, 2, 0).n
                perturbed_img = perturbed_img.detach().cpu().permute(1, 2, 0)

                # Normalize images to [0,1] range
                original_img = np.clip(original_img * 0.5 + 0.5, 0, 1)
                perturbed_img = np.clip(perturbed_img * 0.5 + 0.5, 0, 1)

                # Plot original image
                axes[i, 0].imshow(original_img)
                axes[i, 0].set_title(f"Original\nTrue: {true_label}")
                axes[i, 0].axis('off')

                # Plot perturbed image
                axes[i, 1].imshow(perturbed_img)
                axes[i, 1].set_title(f"Perturbed (ε={epsilon})\nPred: {pred_l
                axes[i, 1].axis('off')

            plt.tight_layout()
            plt.show()


def visualize_misclassifications(misclassified_images, perturbed_batches)
    if misclassified_images:
        num_images = min(5, len(misclassified_images))  # Limit to 5 exam
        fig, axes = plt.subplots(num_images, 2, figsize=(8, 2 * num_image

        for i in range(num_images):
            original_img, true_label, pred_label, batch_idx, img_idx = mi

            # Retrieve the corresponding perturbed image from the same ba
            perturbed_img = perturbed_batches[batch_idx][img_idx]

            # Convert images to numpy
            original_img = original_img.detach().cpu().permute(1, 2, 0).n
            perturbed_img = perturbed_img.detach().cpu().permute(1, 2, 0)

            # Normalize images to [0,1] range
            original_img = np.clip(original_img * 0.5 + 0.5, 0, 1)
            perturbed_img = np.clip(perturbed_img * 0.5 + 0.5, 0, 1)

            # Plot original image
            axes[i, 0].imshow(original_img)
            axes[i, 0].set_title(f"Original\nTrue: {true_label}")
            axes[i, 0].axis('off')

            # Plot perturbed image
            axes[i, 1].imshow(perturbed_img)
            axes[i, 1].set_title(f"Perturbed\nPred: {pred_label}")
            axes[i, 1].axis('off')

        plt.tight_layout()
        plt.show()



# Function to plot confusion matrix
def plot_confusion_matrix(all_labels, all_preds):
```

```
    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=np.ara
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix after FGSM attack')
    plt.show()

# Visualize the misclassified images
visualize_misclassifications(misclassified_images, pertupeted)

# Plot the confusion matrix
plot_confusion_matrix(all_labels, all_preds)
```


Original
True: 5
Perturbed
Pred: 1
Original
True: 2
Perturbed
Pred: 3
Original
True: 1
Perturbed
Pred: 4
Original
True: 0
Perturbed
Pred: 6
Original
True: 6
Perturbed
Pred: 3



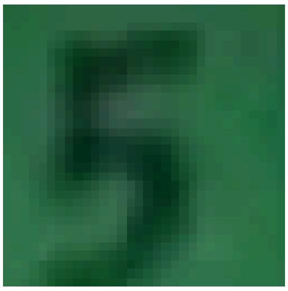
Test the function with different values of epsilon (at least 5) and plot the accuracy as a function of epsilon. For each epsilon, display the perturbed images with the model's classification. At what epsilon does it become harder for the human eye to correctly classify?

In [5]:
```
# new
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

# Function to plot the accuracy as a function of epsilon
def plot_accuracy_vs_epsilon(model, test_loader, device, epsilon_values):
    accuracies = []

    for epsilon in epsilon_values:
        # Get accuracy and misclassified images for the current epsilon
        accuracy, all_labels, all_preds, misclassified_images, pertupeted
        accuracies.append(accuracy)

        # Visualize some perturbed images
        print(f"Visualizing perturbed images for epsilon = {epsilon}")
        visualize_perturbed_images(misclassified_images, pertupeted, epsi

    # Plot accuracy vs epsilon
    plt.plot(epsilon_values, accuracies, marker='o', linestyle='-', color
    plt.xlabel('Epsilon')
```

```
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy vs Epsilon (Adversarial Attack)')
    plt.grid(True)
    plt.show()

def visualize_perturbed_images(misclassified_images, perturbed_batches, e
    if misclassified_images:
        num_images = min(10, len(misclassified_images))  # Limit to 10 ex
        fig, axes = plt.subplots(num_images, 2, figsize=(8, 2 * num_image

        for i in range(num_images):
            original_img, true_label, pred_label, batch_idx, img_idx = mi

            # Retrieve the corresponding perturbed image from the same ba
            perturbed_img = perturbed_batches[batch_idx][img_idx]

            # Convert images to numpy
            original_img = original_img.detach().cpu().permute(1, 2, 0).n
            perturbed_img = perturbed_img.detach().cpu().permute(1, 2, 0)

            # Normalize images to [0,1] range
            original_img = np.clip(original_img * 0.5 + 0.5, 0, 1)
            perturbed_img = np.clip(perturbed_img * 0.5 + 0.5, 0, 1)

            # Plot original image
            axes[i, 0].imshow(original_img)
            axes[i, 0].set_title(f"Original\nTrue: {true_label}")
            axes[i, 0].axis('off')

            # Plot perturbed image
            axes[i, 1].imshow(perturbed_img)
            axes[i, 1].set_title(f"Perturbed (ε={epsilon})\nPred: {pred_l
            axes[i, 1].axis('off')

        plt.tight_layout()
        plt.show()

# Example usage:
epsilon_values = [0.01, 0.05, 0.1, 0.2, 0.3]  # Different values of epsil
plot_accuracy_vs_epsilon(model, test_loader, device, epsilon_values)
```

```
Accuracy on adversarial examples: 55.08%
Visualizing perturbed images for epsilon = 0.01
```

Original
True: 5

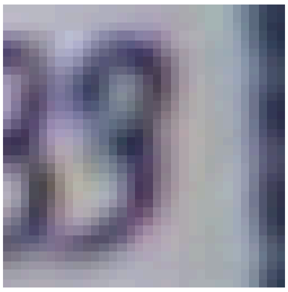


Perturbed (ε=0.01)
Pred: 1

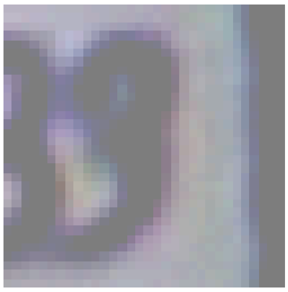
Original
True: 9

Perturbed (ε=0.01)
Pred: 1

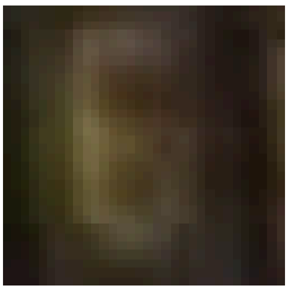
Original
True: 3



Perturbed (ε=0.01)
Pred: 9

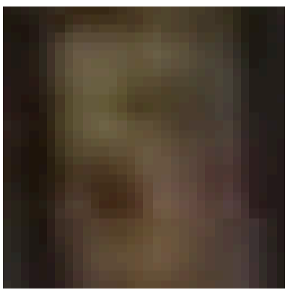


Original
True: 6



Perturbed (ε=0.01)
Pred: 1

Original
True: 5

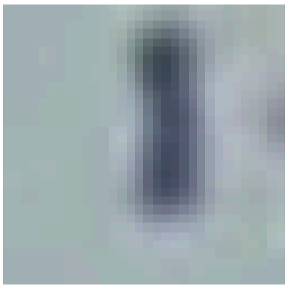


Perturbed (ε=0.01)
Pred: 1

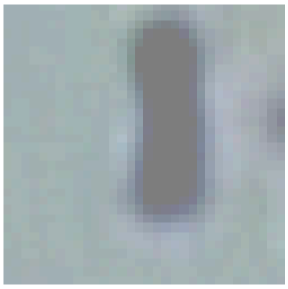
Original

Perturbed (ε=0.01)

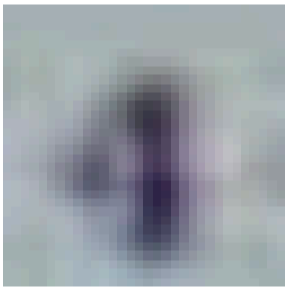
True: 1



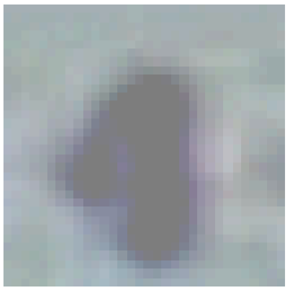
Pred: 2



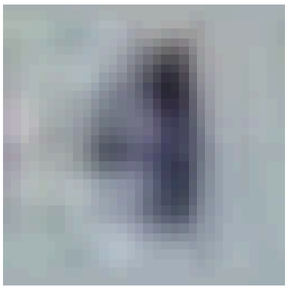
Original
True: 4



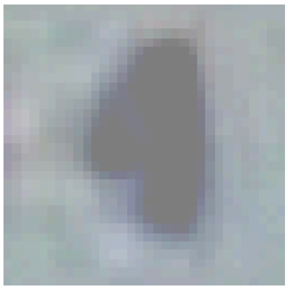
Perturbed (ε=0.01)
Pred: 6



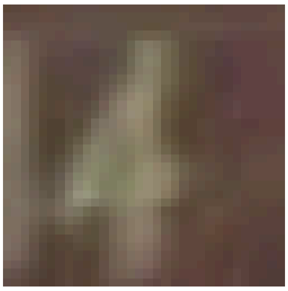
Original
True: 4



Perturbed (ε=0.01)
Pred: 9

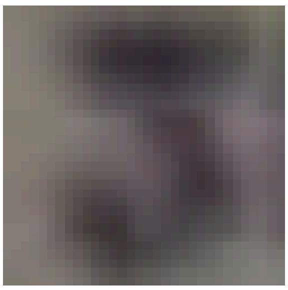


Original
True: 4

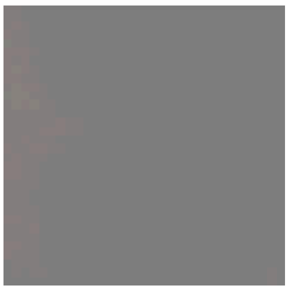


Perturbed (ε=0.01)
Pred: 1

Original
True: 3

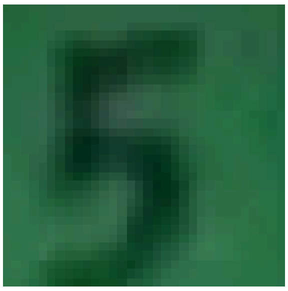


Perturbed (ε=0.01)
Pred: 1



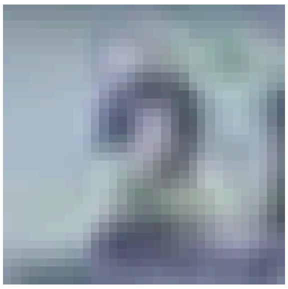
```
Accuracy on adversarial examples: 27.47%
Visualizing perturbed images for epsilon = 0.05
```

Original
True: 5

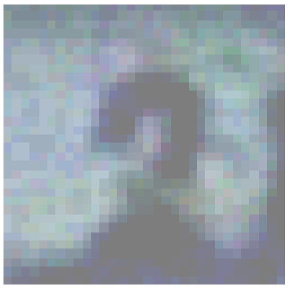


Perturbed (ε=0.05)
Pred: 1

Original
True: 2



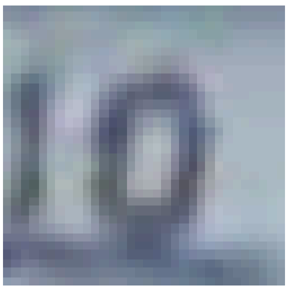
Perturbed (ε=0.05)
Pred: 3



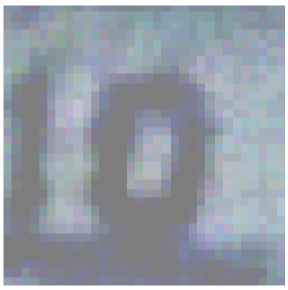
Original
True: 1

Perturbed (ε=0.05)
Pred: 4

Original
True: 0

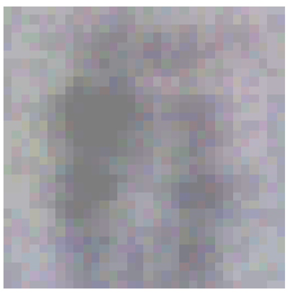


Perturbed (ε=0.05)
Pred: 6



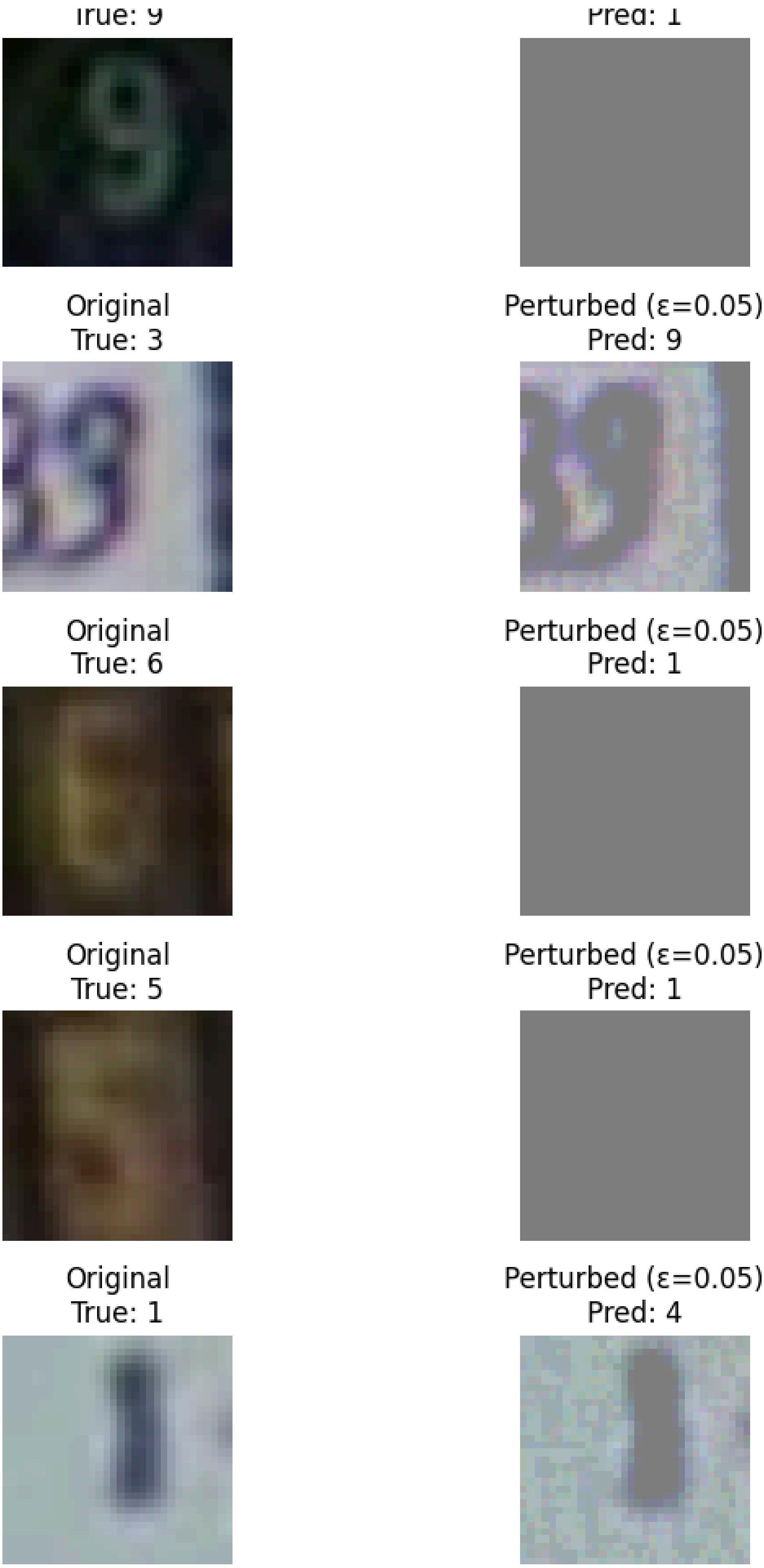
Original
True: 6

Perturbed (ε=0.05)
Pred: 7

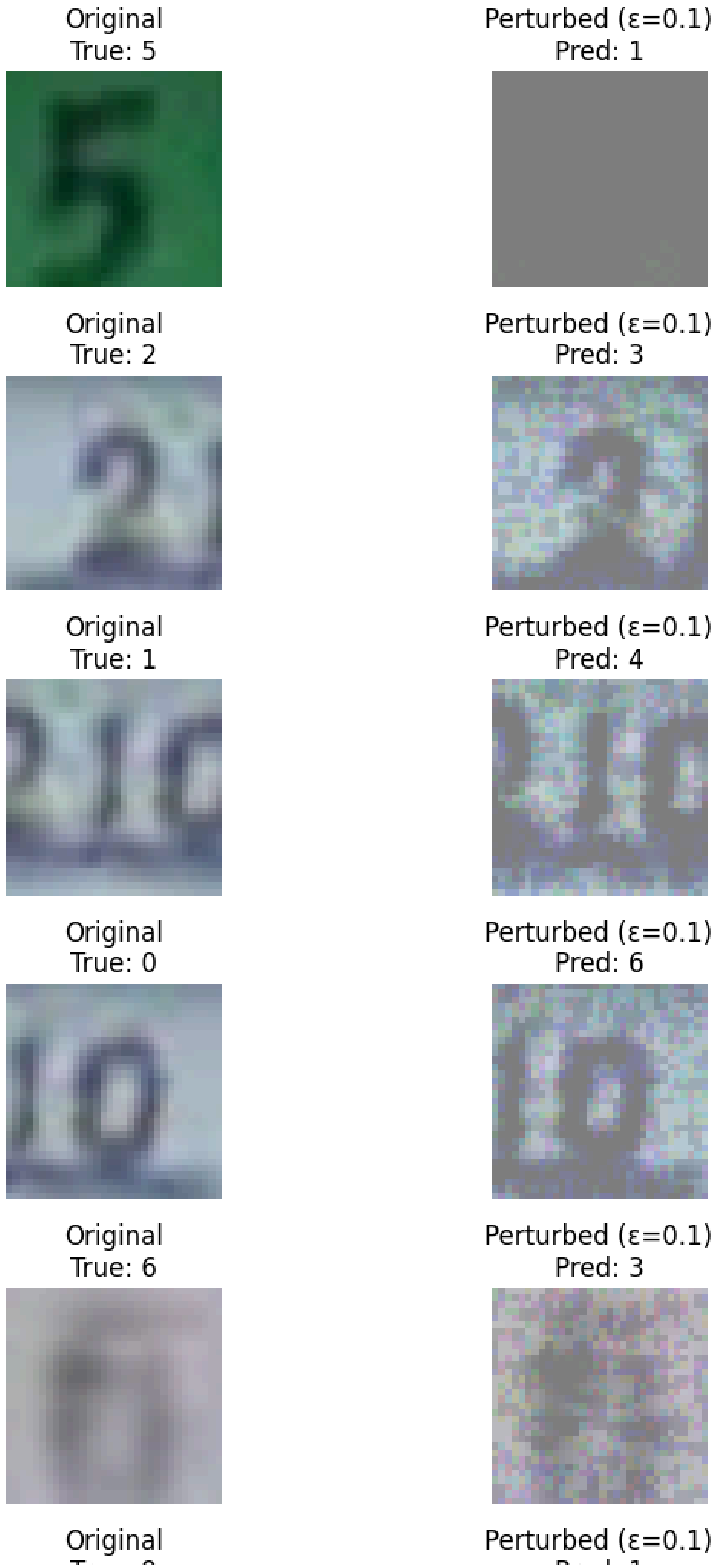


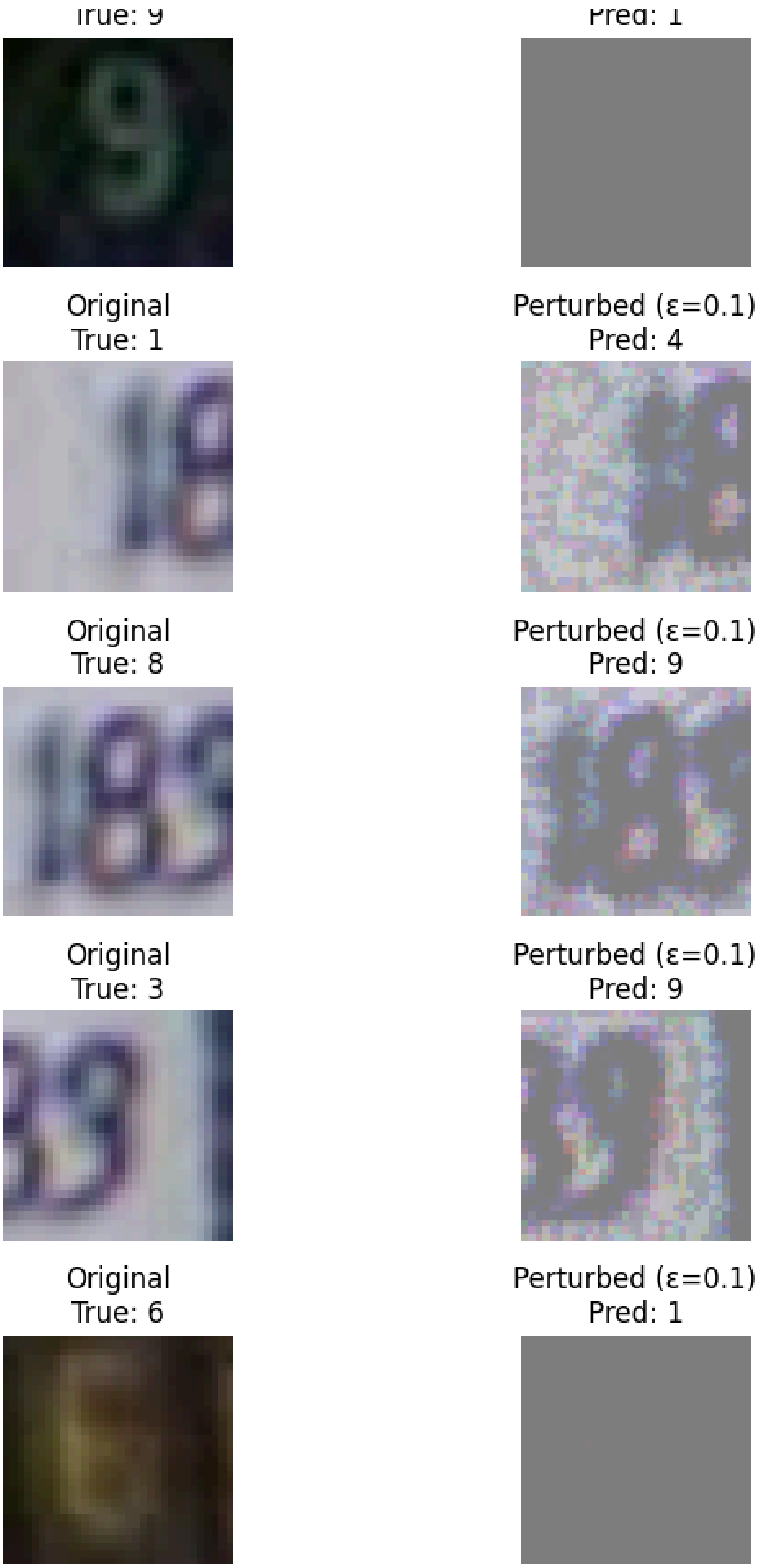
Original

Perturbed (ε=0.05)



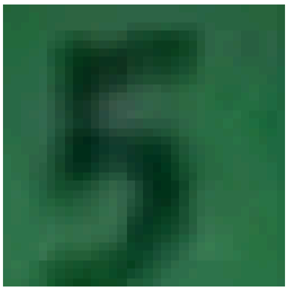

```
Accuracy on adversarial examples: 16.23%
Visualizing perturbed images for epsilon = 0.1
```


Original
True: 5
Perturbed (ε=0.1)
Pred: 1
Original
True: 2
Perturbed (ε=0.1)
Pred: 3
Original
True: 1
Perturbed (ε=0.1)
Pred: 4
Original
True: 0
Perturbed (ε=0.1)
Pred: 6
Original
True: 6
Perturbed (ε=0.1)
Pred: 3
Original
Perturbed (ε=0.1)



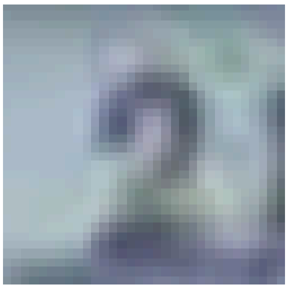

```
Accuracy on adversarial examples: 9.73%
Visualizing perturbed images for epsilon = 0.2
```

Original
True: 5

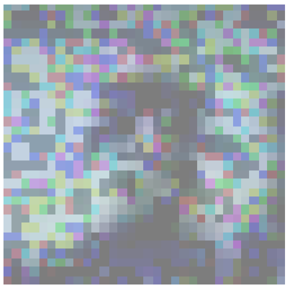


Perturbed (ε=0.2)
Pred: 1

Original
True: 2

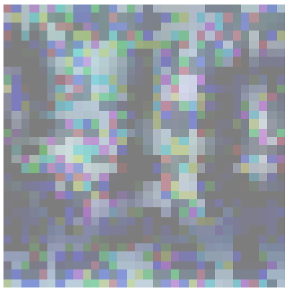


Perturbed (ε=0.2)
Pred: 3

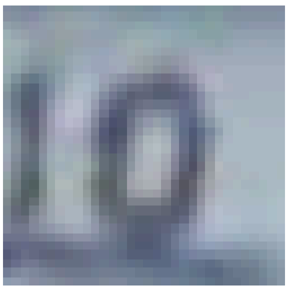


Original
True: 1

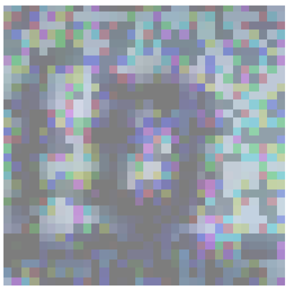
Perturbed (ε=0.2)
Pred: 4



Original
True: 0

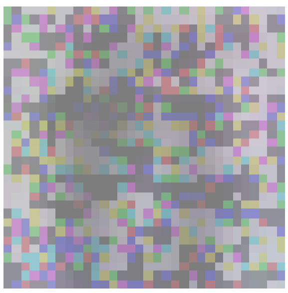


Perturbed (ε=0.2)
Pred: 8



Original
True: 6

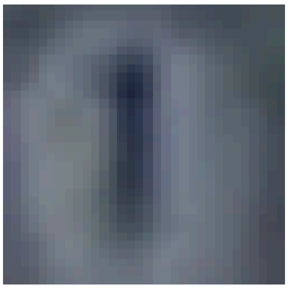
Perturbed (ε=0.2)
Pred: 3



Original

Perturbed (ε=0.2)

True: 1 | Pred: 6

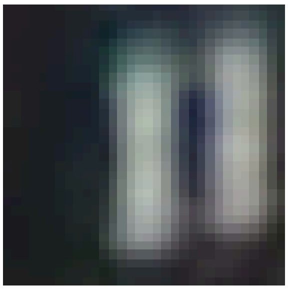
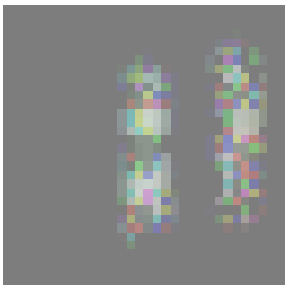



Original
True: 9

Perturbed (ε=0.2)
Pred: 1

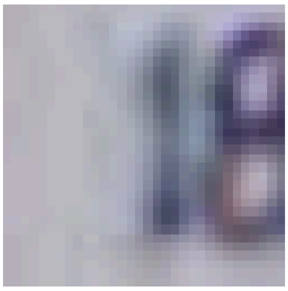
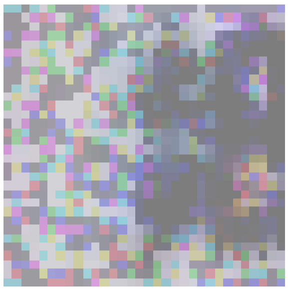

Original
True: 1

Perturbed (ε=0.2)
Pred: 9

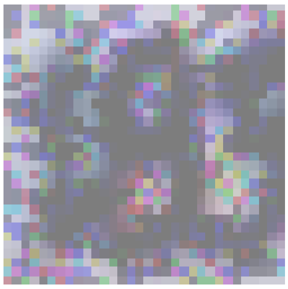
Original
True: 1

Perturbed (ε=0.2)
Pred: 4

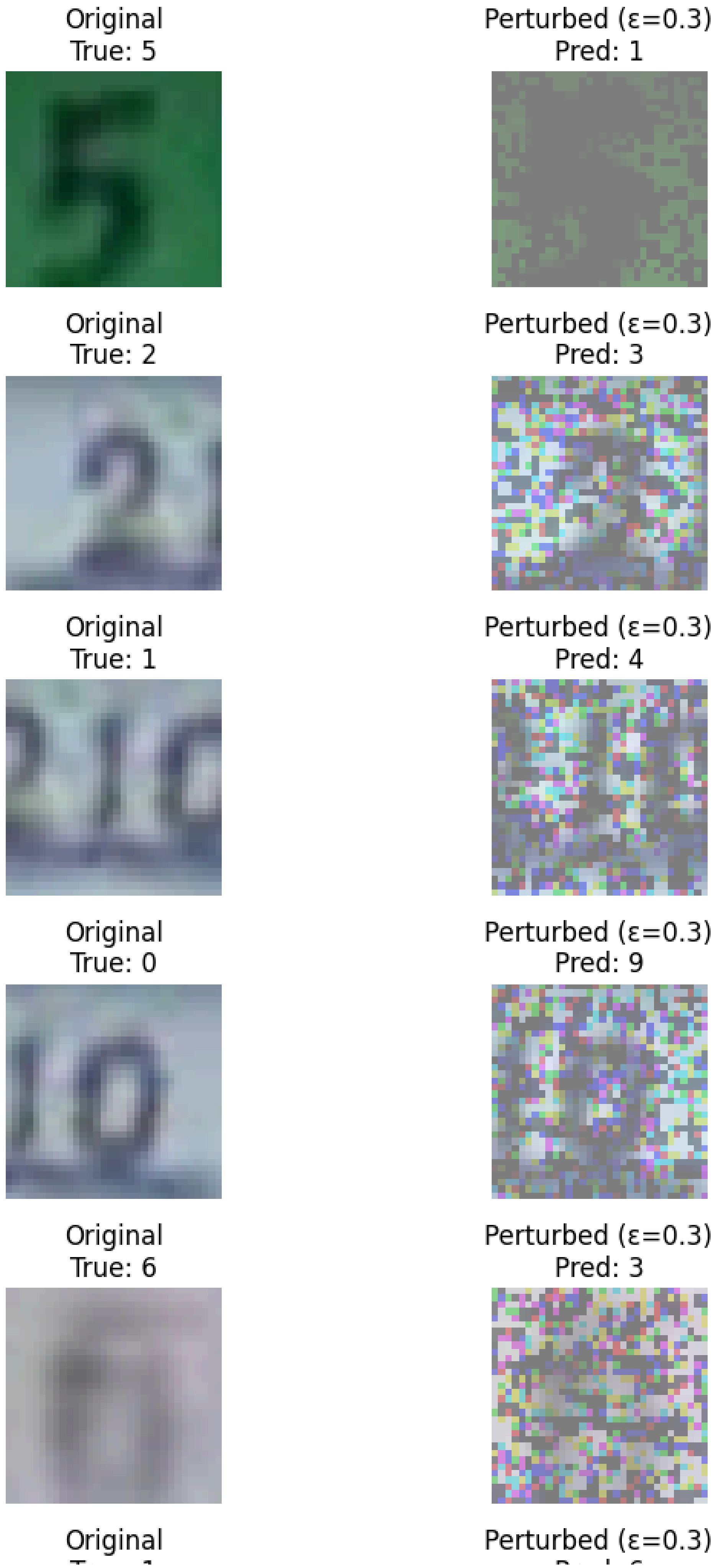



Original
True: 8

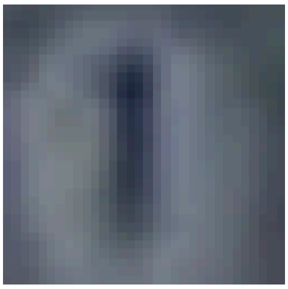
Perturbed (ε=0.2)
Pred: 9

```
Accuracy on adversarial examples: 7.65%
Visualizing perturbed images for epsilon = 0.3
```


Original
True: 5
Perturbed (ε=0.3)
Pred: 1
Original
True: 2
Perturbed (ε=0.3)
Pred: 3
Original
True: 1
Perturbed (ε=0.3)
Pred: 4
Original
True: 0
Perturbed (ε=0.3)
Pred: 9
Original
True: 6
Perturbed (ε=0.3)
Pred: 3
Original
Perturbed (ε=0.3)

True: 1

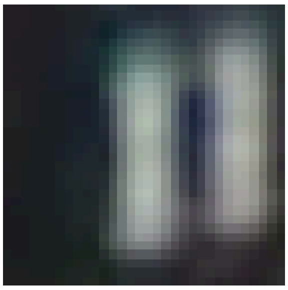


Pred: 6

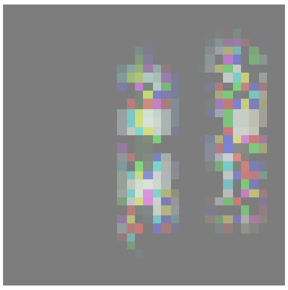
Original
True: 9

Perturbed (ε=0.3)
Pred: 1

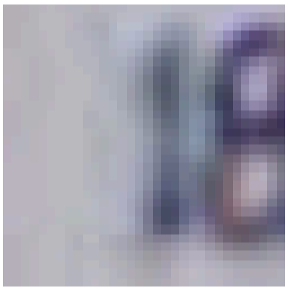
Original
True: 1



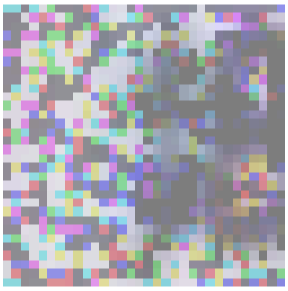
Perturbed (ε=0.3)
Pred: 9



Original
True: 1

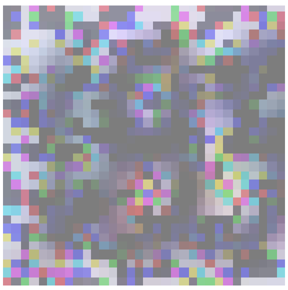


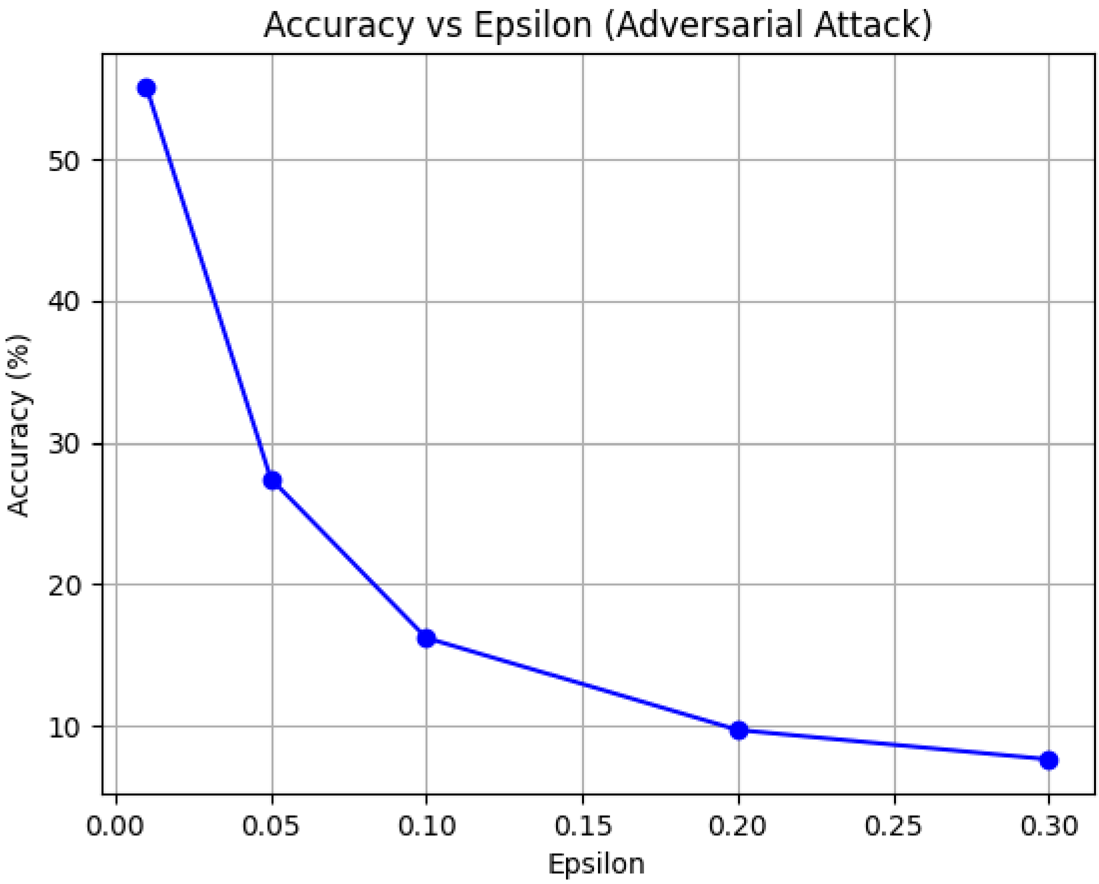
Perturbed (ε=0.3)
Pred: 6



Original
True: 8

Perturbed (ε=0.3)
Pred: 9





For epsilon = 0.3 it becomes harder for human eye to classify corectly.

Part 3: Training our model using adversarial training

For each point in the training data, increase the model's robustness by training not only on the point itself, but on the perturbed point after the FGSM algorithm using $\varepsilon = 0.1$. Afterwards, compute the accuracy once again on the newly trained model using `eval_adversarial(model, test_loader, epsilon)` defined above. The accuracy (LOOKING ONLY AT THE PERTURBED DATA) should be at least 70%.

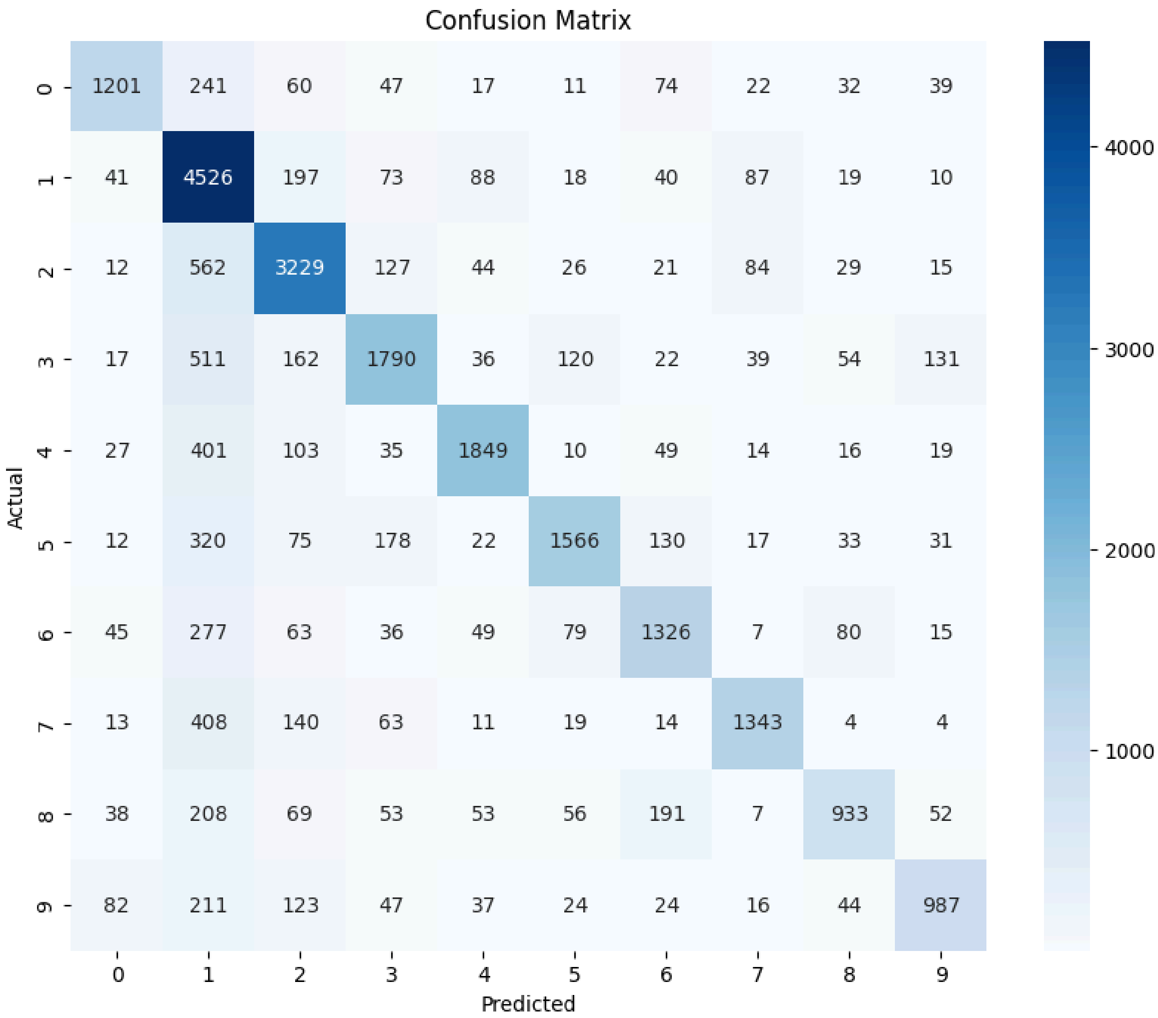
Visualization

Display the confusion matrix along with some examples of images that the model classified incorrectly. Discuss the performance of the model now compared to before.

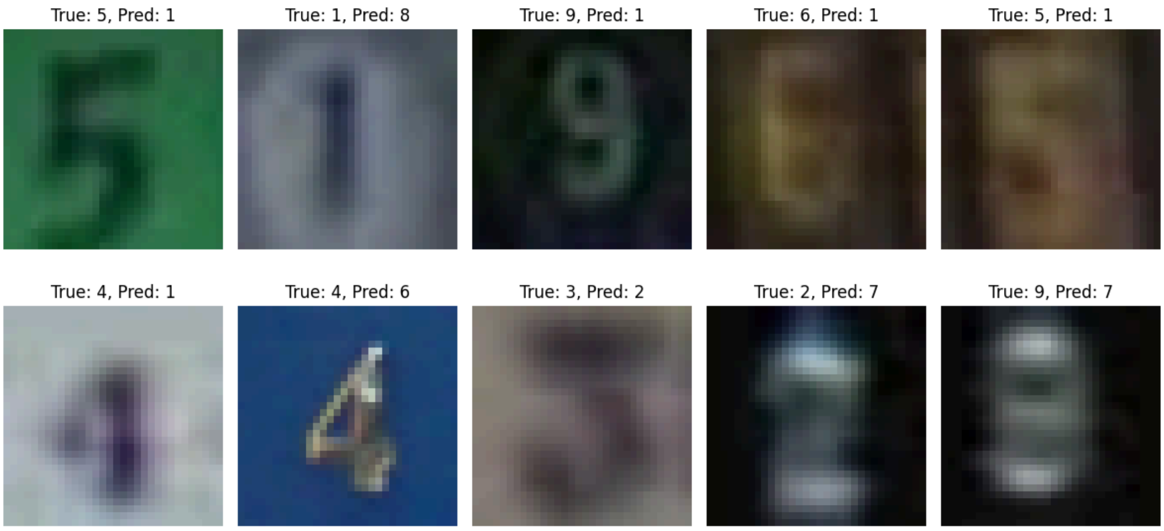
In [8]:

```
# new
# Function to display the confusion matrix
def display_confusion_matrix(labels, preds, class_names):
    from sklearn.metrics import confusion_matrix
    import seaborn as sns

    cm = confusion_matrix(labels, preds)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
```

```
    plt.title('Confusion Matrix')
    plt.show()

# Function to visualize some misclassified images
def visualize_misclassified_images(misclassified_images, class_names):
    if misclassified_images:
        fig, axes = plt.subplots(2, 5, figsize=(12, 6))
        axes = axes.ravel()
        for i in range(min(10, len(misclassified_images))):
            img, true_label, pred_label, *_ = misclassified_images[i]


            # Ensure the image is on CPU, detach it from the computation
            img = img.detach().cpu().permute(1, 2, 0).numpy()  # Convert
            img = np.clip(img * 0.5 + 0.5, 0, 1)  # Clip values between 0
            img = np.clip(img, 0, 1)  # Clip values between 0 and 1 for d

            axes[i].imshow(img)
            axes[i].set_title(f"True: {class_names[true_label]}, Pred: {c
            axes[i].axis('off')
        plt.tight_layout()
        plt.show()

# Display confusion matrix
class_names = [str(i) for i in range(10)]  # Assuming 10 classes (e.g., 0
display_confusion_matrix(all_labels, all_preds, class_names)

# Visualize misclassified images
print("Visualizing some misclassified images:")
visualize_misclassified_images(misclassified_images, class_names)

# Discuss the performance
print(f"Adversarial Test Accuracy (Epsilon = 0.1): {accuracy:.2f}%")
```




```
Visualizing some misclassified images:
```




```
Adversarial Test Accuracy (Epsilon = 0.1): 72.03%
```

Part 4: Contrastive Learning

In this section, we will work on creating informative embeddings for images using SimCLR. For this section we will use the attached subset of the popular ImageNet dataset of 96x96 images from 1000 classes. Below, we provide you with several functions to implement a contrastive learning model.

Dry Questions

Before implementation, take these questions in consideration (and provide your answers and explanations):

1. When training an unsupervised contrastive learning model such as SimCLR, would we prefer to have a large or small batch size?
2. In general, what possible evaluation metrics could be used in this task (unsupervised representation learning) to measure our model's performance?
3. When creating embeddings for images in the test set, how does the process differ from what we do in training?
4. For each of the following image augmentations, explain whether or not we would like to use them in the SimCLR framework:
    - Randomly cropping a fixed-size window in the image.
    - Enlarging the image to 128x128.
    - Random rotation of the image.
    - Adding Gaussian noise.
    - Randomly changing the image's dimensions.
    - Randomly converting the image to grayscale.

Answers to Dry Questions

1. Contrastive learning models like SimCLR work better with large batches of data. This is because they learn by comparing similar and different examples. Larger batches give the model more different examples to compare against, helping it learn to distinguish between similar and dissimilar data points.

2. One way to measure the performance of unsupervised representation learning is through downstream tasks. For example, we can use a labeled dataset to train a classifier using the embeddings produced by our unsupervised model. The performance of the classifier can then be evaluated using standard metrics such as accuracy or any other task-relevant evaluation metric.

3. Training: augmentations are applied to create positive pairs.

    Testing: no augmentations are used. The embeddings are generated directly from the original test images. This ensures consistency and avoids introducing noise when evaluating the model's performance on downstream tasks.

4. Yes, by randomly cropping the same image, we create slightly different versions. This helps the model learn to recognize the main object even if it's moved around a bit.

No, Making the image bigger doesn't really help the model learn useful features. This is because simply enlarging an image is an unnatural change that doesn't reflect how objects actually appear in the real world.

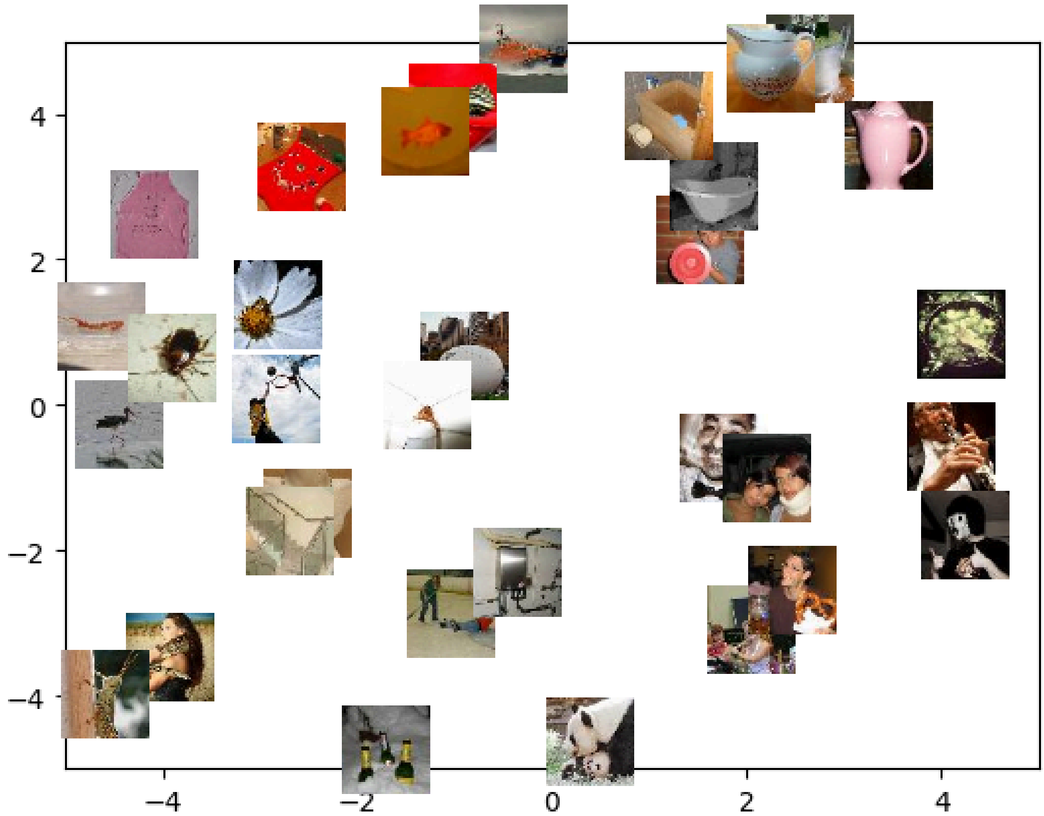
Yes, Random rotation helps the model learn rotational invariance, making it robust to such transformations.

Yes, Adding Gaussian noise can improve robustness by forcing the model to focus on the core structure of the data rather than noise.

No, changing the image's dimensions can distort the content and introduce inconsistency in the representation learning process.

Yes, Converting to grayscale teaches the model to be invariant to color, focusing instead on texture and structure, which are more general features.

In [ ]:
```
plot_embeddings(model, test_loader,device)
```



For some batch of the test loader, take 3 images in the batch. For each image, find and display the 5 images that have the closest embeddings to them. Do the chosen images make sense? If not, what could have possibly gone wrong with your model?

In [ ]:
```
import torch.nn.functional as F
import matplotlib.pyplot as plt

def find_closest_images(model, test_loader, num_images=3, num_closest=5):

    model.eval()  # Set the model to evaluation mode
    with torch.no_grad():  # Disable gradient computation
        for img_batch, img_transformed_batch in test_loader:
            # Move transformed images to the device
            img_transformed_batch = img_transformed_batch.to(device)
```

```
            # Compute embeddings for the entire batch
            embeddings = model(img_transformed_batch)
            embeddings = F.normalize(embeddings, p=2, dim=1)  # Normalize

            # Randomly select `num_images` query images
            indices = torch.randperm(len(img_batch))[:num_images]
            selected_imgs = img_batch[indices]  # Original (non-transform
            selected_embeddings = embeddings[indices]  # Corresponding em

            # Compute pairwise distances between selected embeddings and
            distances = torch.cdist(selected_embeddings, embeddings, p=2)

            # Visualize results
            for i, (img, dist) in enumerate(zip(selected_imgs, distances)
                # Get the indices of the closest images (excluding the qu
                closest_indices = dist.topk(num_closest + 1, largest=Fals

                # Display the query image and the closest images
                fig, axes = plt.subplots(1, num_closest + 1, figsize=(15,
                axes[0].imshow(img.permute(1, 2, 0))  # Convert CHW to HW
                axes[0].set_title("Query Image")
                axes[0].axis("off")

                for j, idx in enumerate(closest_indices):
                    closest_img = img_batch[idx].permute(1, 2, 0)  # Conv
                    axes[j + 1].imshow(closest_img)
                    axes[j + 1].set_title(f"Closest {j+1}")
                    axes[j + 1].axis("off")

                plt.show()

            # Process only the first batch
            break

# Call the function
find_closest_images(model, test_loader)
```