Animal Classification for CNN

Abstract:

This project focuses on enhancing the robustness of convolutional neural network (CNN) models against adversarial attacks, specifically the Projected Gradient Descent (PGD) attack. The primary objective is to develop and evaluate techniques to defend against such attacks, thereby improving the model's resilience to perturbations in input data. The project involves implementing adversarial training, a technique where the model is trained using both clean and adversarial examples generated through PGD. Additionally, the project explores the use of Fast Gradient Sign Method (FGSM) for crafting adversarial examples and evaluates the model's performance under various attack scenarios. The code provided demonstrate the implementation of these techniques, including the training process, generation of adversarial examples, and evaluation of model robustness. The project aims to contribute insights into building more robust and secure deep learning models against adversarial threats.

Introduction:

In recent years, convolutional neural networks (CNNs) have achieved remarkable success in various computer vision tasks, demonstrating high accuracy in image classification, object detection, and more. However, these models are known to be vulnerable to adversarial attacks, where small, carefully crafted perturbations to input images can lead to misclassification with high confidence. Adversarial attacks pose a significant threat to the reliability and security of CNN-based systems, especially in critical applications such as autonomous vehicles and medical image analysis.

The motivation behind this project stems from the urgent need to develop robust defense mechanisms against adversarial attacks on CNN models. Adversarial examples can easily deceive even state-of-the-art models, leading to potentially catastrophic consequences in real-world scenarios. Thus, there is a critical demand for research and development efforts aimed at enhancing the resilience of CNNs against adversarial threats.

The problem statement revolves around mitigating the susceptibility of CNN models to adversarial attacks, particularly the Projected Gradient Descent (PGD) attack. PGD is a powerful white-box attack that iteratively perturbs input images to maximize the model's loss, resulting in misclassification. The challenge lies in devising effective defense strategies to withstand such attacks without compromising the model's accuracy on clean data.

I approach this by proposing and implementing adversarial training as a defense mechanism against PGD attacks. Adversarial training involves augmenting the training dataset with adversarial examples generated using PGD, thereby forcing the model to learn robust

features that are resilient to perturbations. Additionally, we explore the use of the Fast Gradient Sign Method (FGSM) for crafting adversarial examples and evaluate its efficacy in enhancing model robustness. I mainly use it to show how normal FGSM perturbations are not as strong as PGD perturbations.

Methods

- PGD Attack
- Adversarial Training
- FGSM Attack (for comparison)

References

Dataset (combined)

- https://www.kaggle.com/datasets/iamsouravbanerjee/animal-image-dataset-90-different-animals
- https://www.kaggle.com/datasets/andrewmvd/animal-faces

CNN code for animal classification

- https://www.kaggle.com/code/mehmetlaudatekman/pytorch-animal-face-classification-cnns

Dataset makeup

- Cat: 5213 pictures
- Dog: 4799 pictures
- Tiger: 1134 pictures
- Wolf: 503 pictures

For both validation and training dataset I used the above photoset lineup. If I were to redo this project I would ensure the dataset I grabbed from had better class balance.

PGD Attack:

- https://github.com/Harry24k/PGD-pytorch/blob/master/PGD.ipynb

In [ ]:
```
#References
#Dataset
#https://www.kaggle.com/datasets/iamsouravbanerjee/animal-image-dataset-90-differen
#CNN code for animal classification (p.2 dataset) (combined)
#https://www.kaggle.com/code/mehmetlaudatekman/pytorch-animal-face-classification-c
#https://github.com/Harry24k/PGD-pytorch/blob/master/PGD.ipynb


import os
```

```
import numpy as np
import pandas as pd
from PIL import Image
import matplotlib.pyplot as plt
from glob import glob
import time

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torch.utils.data import DataLoader
from torch.utils.data.sampler import SubsetRandomSampler
from torch.utils.data import Dataset

import torchvision
import torchvision.transforms as transforms
```

Load and process data for image classification

In [ ]:
```
# * We'll use this class to control the things.
class InvalidDatasetException(Exception):

    def __init__(self,len_of_paths,len_of_labels):
        super().__init__(
            f"Number of paths ({len_of_paths}) is not compatible with number of lab
        )
```

In [ ]:
```
transform = transforms.Compose([transforms.ToTensor()])
```

In [ ]:
```
class AnimalDataset(Dataset):

    def __init__(self,img_paths,img_labels,size_of_images):
        self.img_paths = img_paths
        self.img_labels = img_labels
        self.size_of_images = size_of_images
        if len(self.img_paths) != len(self.img_labels):
            raise InvalidDatasetException(self.img_paths,self.img_labels)


    # We need to override __len__ special method
    def __len__(self):
        return len(self.img_paths)

    # Also we need to override __getitem__ special method
    # This method should return the image and its label from index given.
    def __getitem__(self,index):
        PIL_IMAGE = Image.open(self.img_paths[index]).resize(self.size_of_images)
        # In pytorch we use torch tensors, ToTensor transform transforms the PIL im
```

```
        # to Torch tensor.
        TENSOR_IMAGE = transform(PIL_IMAGE)
        label = self.img_labels[index]

        return TENSOR_IMAGE,label
```

In [ ]:
```
paths = []
labels = []
val_paths = []
val_labels = []

label_map = {
    "cat": 0,
    "tiger": 1,
    "dog": 2,
    "wolf": 3
}
current_dir = os.path.dirname(os.path.abspath("/Tiffany-Behr-4900-FinalProject.ipyn
for class_name in label_map.keys():
    class_paths = glob("data/" + class_name + "/*.jpg") #
    # Append image paths and corresponding labels to lists
    paths.extend(class_paths)
    labels.extend([label_map[class_name]] * len(class_paths))

print("Number of images:", len(paths))
print("Number of labels:", len(labels))

# Example: Print the first image path and its corresponding label
print("Example image path:", paths[0])
print("Example label:", labels[0])

image = Image.open(paths[0])
image.show()
```

```
Number of images: 11649
Number of labels: 11649
Example image path: data/cat\00357563a7.jpg
Example label: 0
```

In [ ]:
```
dataset = AnimalDataset(paths,labels,(250,250))
```

In [ ]:
```
from sklearn.model_selection import train_test_split

# dataset_indices = [0,1,2,3,..len(dataset)-1]
dataset_indices = list(range(0,len(dataset)))

train_indices,test_indices = train_test_split(dataset_indices,test_size=0.2,random_
print("Number of train samples: ",len(train_indices))
print("Number of test samples: ",len(test_indices))
```

```
Number of train samples:  9319
Number of test samples:  2330
```

In [ ]:
```
train_sampler = SubsetRandomSampler(train_indices)
test_sampler = SubsetRandomSampler(test_indices)
```

In [ ]:
```
BATCH_SIZE = 250
train_loader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE,
                                           sampler=train_sampler)
validation_loader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE,
                                                sampler=test_sampler)
#change to include val folser
```

In [ ]:
```
dataset[1][0].shape
```

Out[ ]:
```
torch.Size([3, 250, 250])
```

In [ ]:
```
# Get a batch of data from the DataLoader
images, labels = next(iter(train_loader))
#print("Labels:", labels) #used in debugging

# Visualize the batch of images
fig, axis = plt.subplots(3, 5, figsize=(15, 10))
for i, ax in enumerate(axis.flat):
    with torch.no_grad():
        npimg = images[i].numpy()
        npimg = np.transpose(npimg, (1, 2, 0))
        label = labels[i].item()  # Use the integer label directly
        ax.imshow(npimg)
        ax.set(title=f"Label {label}")

plt.show()
```

In [ ]:
```
class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
```

```
        # Define layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1)
        self.batchnorm1 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1)
        self.batchnorm2 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1)
        self.batchnorm3 = nn.BatchNorm2d(256)
        self.maxpool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256 * 2 * 2, 512)
        self.fc2 = nn.Linear(512, 4)  # 4 classes

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.batchnorm1(x)
        x = self.maxpool(x)
        x = F.relu(self.conv3(x))
        x = self.batchnorm2(x)
        x = self.maxpool(x)
        x = F.relu(self.conv4(x))
        x = self.batchnorm3(x)
        x = self.maxpool(x)
        x = x.view(-1, 256 * 2 * 2)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

In [ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


clean_model = CNN().to(device)
poisoned_fgsm_model = CNN().to(device)
poisoned_pgd_model = CNN().to(device)

# Define your optimizer and loss function
clean_optimizer = optim.Adam(clean_model.parameters(), lr=0.001)
poisoned_fgsm_optimizer = optim.Adam(poisoned_fgsm_model.parameters(), lr=0.001)
poisoned_pgd_optimizer = optim.Adam(poisoned_pgd_model.parameters(), lr=0.001)
criterion = nn.NLLLoss()
```

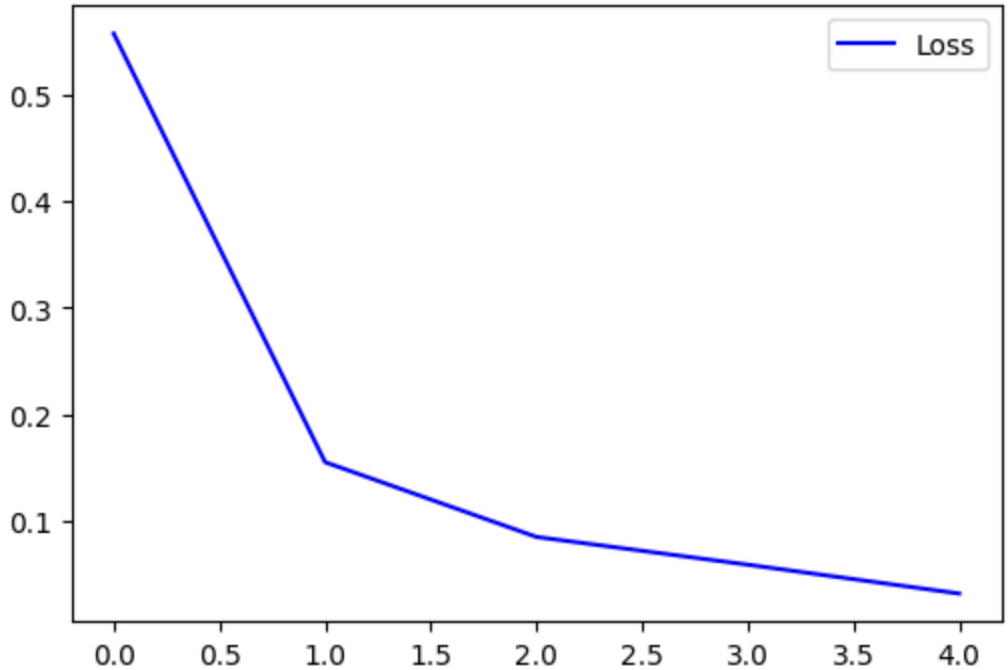
In [ ]:
```
def train_cnn(model, optimizer):
    global EPOCH_NUMBER
    EPOCH_NUMBER = 5
    global TRAIN_LOSS
    TRAIN_LOSS = []
    global TRAIN_ACCURACY
    TRAIN_ACCURACY = []


    for epoch in range(1, EPOCH_NUMBER + 1):
        epoch_loss = 0.0
        correct = 0
        total = 0
```

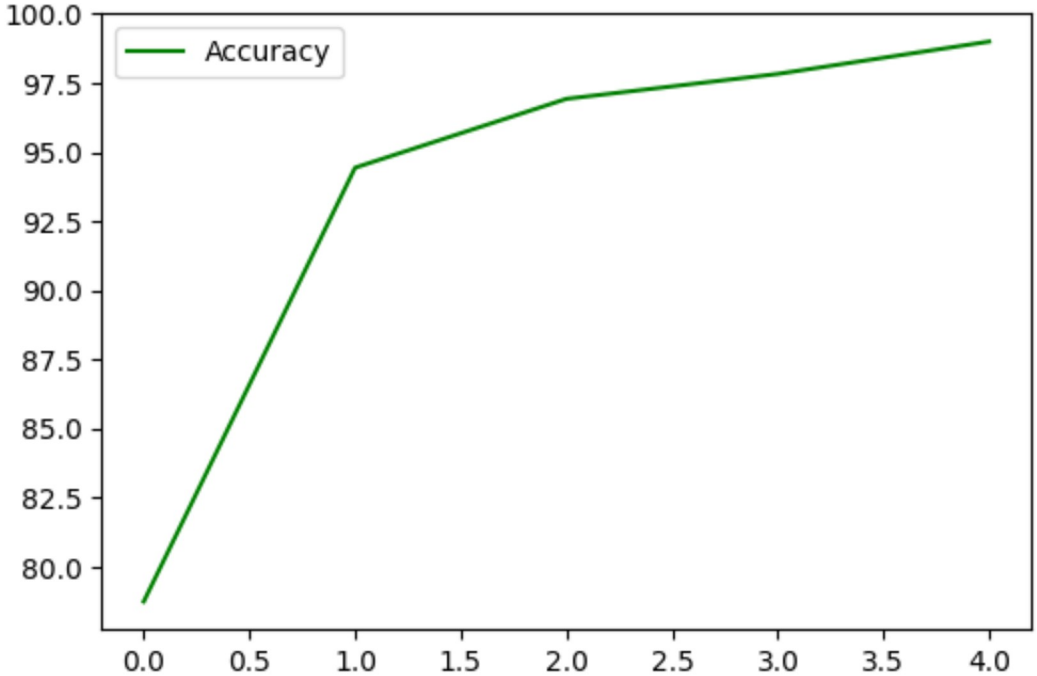
```
        for data_, target_ in train_loader:
            # Move data and target to device (e.g., GPU)
            data_ = data_.to(device)
            target_ = target_.to(device)
            optimizer.zero_grad()

            outputs = model(data_)

            loss = criterion(outputs, target_)

            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            _, pred = torch.max(outputs, dim=1)
            correct += torch.sum(pred == target_).item()
            total += target_.size(0)

        avg_loss = epoch_loss / len(train_loader)
        accuracy = 100 * correct / total

        TRAIN_LOSS.append(avg_loss)
        TRAIN_ACCURACY.append(accuracy)

        print(f"Epoch {epoch}: Loss: {avg_loss:.4f}, Accuracy: {accuracy:.2f}%")
```

In [ ]:
```
train_cnn(clean_model, clean_optimizer)
train_cnn(poisoned_fgsm_model, poisoned_fgsm_optimizer)
train_cnn(poisoned_pgd_model, poisoned_pgd_optimizer)
```

```
Epoch 1: Loss: 0.5957, Accuracy: 77.03%
Epoch 2: Loss: 0.1614, Accuracy: 94.41%
Epoch 3: Loss: 0.0836, Accuracy: 97.09%
Epoch 4: Loss: 0.0595, Accuracy: 97.94%
Epoch 5: Loss: 0.0243, Accuracy: 99.37%
Epoch 1: Loss: 0.5369, Accuracy: 79.15%
Epoch 2: Loss: 0.1560, Accuracy: 94.53%
Epoch 3: Loss: 0.0812, Accuracy: 97.39%
Epoch 4: Loss: 0.0471, Accuracy: 98.53%
Epoch 5: Loss: 0.0257, Accuracy: 99.15%
Epoch 1: Loss: 0.5574, Accuracy: 78.75%
Epoch 2: Loss: 0.1549, Accuracy: 94.44%
Epoch 3: Loss: 0.0846, Accuracy: 96.93%
Epoch 4: Loss: 0.0586, Accuracy: 97.83%
Epoch 5: Loss: 0.0315, Accuracy: 99.00%
```

In [ ]:
```
plt.subplots(figsize=(6,4))
plt.plot(range(EPOCH_NUMBER),TRAIN_LOSS,color="blue",label="Loss")
plt.legend()
plt.show()

plt.subplots(figsize=(6,4))
plt.plot(range(EPOCH_NUMBER),TRAIN_ACCURACY,color="green",label="Accuracy")
plt.legend()
plt.show()
```





In [ ]:
```
def validate_model(model):
    total_val_loss = 0.0
    total_true = 0
    total = len(test_sampler)

    # When we're not working with gradients and backpropagation
    # we use torch.no_grad() utility.
    with torch.no_grad():
        model.eval()
        for data_,target_ in validation_loader:
            data_ = data_.to(device)
            target_ = target_.to(device)
```

```
            outputs = model(data_)
            loss = criterion(outputs,target_).item()
            _,preds = torch.max(outputs,dim=1)
            total_val_loss += loss
            true = torch.sum(preds == target_).item()
            total_true += true

    validation_accuracy = round(100 * total_true / total,2)
    print(f"Validation accuracy: {validation_accuracy}%")
    print(f"Validation loss: {round(total_val_loss,2)}%")
    # for param_tensor in model.state_dict():
    #     print(param_tensor, "\t", model.state_dict()[param_tensor].size())
```

In [ ]:

```
validate_model(clean_model)
validate_model(poisoned_fgsm_model)
validate_model(poisoned_pgd_model)
```

```
Validation accuracy: 95.32%
Validation loss: 1.32%
conv1.weight     torch.Size([32, 3, 3, 3])
conv1.bias       torch.Size([32])
conv2.weight     torch.Size([64, 32, 3, 3])
conv2.bias       torch.Size([64])
batchnorm1.weight        torch.Size([64])
batchnorm1.bias          torch.Size([64])
batchnorm1.running_mean          torch.Size([64])
batchnorm1.running_var   torch.Size([64])
batchnorm1.num_batches_tracked   torch.Size([])
conv3.weight     torch.Size([128, 64, 3, 3])
conv3.bias       torch.Size([128])
batchnorm2.weight        torch.Size([128])
batchnorm2.bias          torch.Size([128])
batchnorm2.running_mean          torch.Size([128])
batchnorm2.running_var   torch.Size([128])
batchnorm2.num_batches_tracked   torch.Size([])
conv4.weight     torch.Size([256, 128, 3, 3])
conv4.bias       torch.Size([256])
batchnorm3.weight        torch.Size([256])
batchnorm3.bias          torch.Size([256])
batchnorm3.running_mean          torch.Size([256])
batchnorm3.running_var   torch.Size([256])
batchnorm3.num_batches_tracked   torch.Size([])
fc1.weight       torch.Size([512, 1024])
fc1.bias         torch.Size([512])
fc2.weight       torch.Size([4, 512])
fc2.bias         torch.Size([4])
Validation accuracy: 94.46%
Validation loss: 2.22%
conv1.weight     torch.Size([32, 3, 3, 3])
conv1.bias       torch.Size([32])
conv2.weight     torch.Size([64, 32, 3, 3])
conv2.bias       torch.Size([64])
batchnorm1.weight        torch.Size([64])
batchnorm1.bias          torch.Size([64])
batchnorm1.running_mean          torch.Size([64])
batchnorm1.running_var   torch.Size([64])
batchnorm1.num_batches_tracked   torch.Size([])
conv3.weight     torch.Size([128, 64, 3, 3])
conv3.bias       torch.Size([128])
batchnorm2.weight        torch.Size([128])
batchnorm2.bias          torch.Size([128])
batchnorm2.running_mean          torch.Size([128])
batchnorm2.running_var   torch.Size([128])
batchnorm2.num_batches_tracked   torch.Size([])
conv4.weight     torch.Size([256, 128, 3, 3])
conv4.bias       torch.Size([256])
batchnorm3.weight        torch.Size([256])
batchnorm3.bias          torch.Size([256])
batchnorm3.running_mean          torch.Size([256])
batchnorm3.running_var   torch.Size([256])
batchnorm3.num_batches_tracked   torch.Size([])
fc1.weight       torch.Size([512, 1024])
fc1.bias         torch.Size([512])
```

```
fc2.weight       torch.Size([4, 512])
fc2.bias         torch.Size([4])
Validation accuracy: 95.19%
Validation loss: 1.38%
conv1.weight     torch.Size([32, 3, 3, 3])
conv1.bias       torch.Size([32])
conv2.weight     torch.Size([64, 32, 3, 3])
conv2.bias       torch.Size([64])
batchnorm1.weight        torch.Size([64])
batchnorm1.bias          torch.Size([64])
batchnorm1.running_mean          torch.Size([64])
batchnorm1.running_var   torch.Size([64])
batchnorm1.num_batches_tracked   torch.Size([])
conv3.weight     torch.Size([128, 64, 3, 3])
conv3.bias       torch.Size([128])
batchnorm2.weight        torch.Size([128])
batchnorm2.bias          torch.Size([128])
batchnorm2.running_mean          torch.Size([128])
batchnorm2.running_var   torch.Size([128])
batchnorm2.num_batches_tracked   torch.Size([])
conv4.weight     torch.Size([256, 128, 3, 3])
conv4.bias       torch.Size([256])
batchnorm3.weight        torch.Size([256])
batchnorm3.bias          torch.Size([256])
batchnorm3.running_mean          torch.Size([256])
batchnorm3.running_var   torch.Size([256])
batchnorm3.num_batches_tracked   torch.Size([])
fc1.weight       torch.Size([512, 1024])
fc1.bias         torch.Size([512])
fc2.weight       torch.Size([4, 512])
fc2.bias         torch.Size([4])
```

In [ ]:

```python
def pgd_attack(model, images, labels, eps=0.3, alpha=2/255, iters=40):
    model.eval()
    images = images.to(device)
    labels = labels.to(device)
    loss_fn = nn.CrossEntropyLoss()

    ori_images = images.data

    for i in range(iters):
        images.requires_grad = True
        outputs = model(images)

        model.zero_grad()
        cost = loss_fn(outputs, labels).to(device)
        cost.backward()

        adv_images = images + alpha * images.grad.sign()
        eta = torch.clamp(adv_images - ori_images, min=-eps, max=eps)
        images = torch.clamp(ori_images + eta, min=0, max=1).detach_()

    return images
```

In [ ]:

```python
import random
pgd_prob = 0.5
```

```
# Training loop
for epoch in range(EPOCH_NUMBER):
    poisoned_pgd_model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in train_loader:
        if random.random() < pgd_prob:
            images = pgd_attack(poisoned_pgd_model, images, labels)  # Apply PGD at

        images = images.to(device)
        labels = labels.to(device)

        poisoned_pgd_optimizer.zero_grad()
        outputs = poisoned_pgd_model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        poisoned_pgd_optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    # Print statistics
    print(f'Epoch [{epoch+1}/{EPOCH_NUMBER}], Loss: {running_loss/len(train_loader)
```

```
Epoch [1/5], Loss: 9.0352, Accuracy: 30.61%
Epoch [2/5], Loss: 1.1695, Accuracy: 43.21%
Epoch [3/5], Loss: 1.1048, Accuracy: 44.15%
Epoch [4/5], Loss: 1.2762, Accuracy: 43.02%
Epoch [5/5], Loss: 1.0930, Accuracy: 43.50%
```

In [ ]:

```
#TIFFANY RE RUN OPOOOOPPPS # DONE
total_val_loss = 0.0
total_true = 0
total = len(test_sampler)

# When we're not working with gradients and backpropagation
# we use torch.no_grad() utility.
with torch.no_grad():
    poisoned_pgd_model.eval()
    for data_,target_ in validation_loader:
        data_ = data_.to(device)
        target_ = target_.to(device)

        outputs = poisoned_pgd_model(data_)
        loss = criterion(outputs,target_).item()
        _,preds = torch.max(outputs,dim=1)
        total_val_loss += loss
        true = torch.sum(preds == target_).item()
        total_true += true

validation_accuracy = round(100 * total_true / total,2)
```

```
print(f"Validation accuracy: {validation_accuracy}%")
print(f"Validation loss: {round(total_val_loss,2)}%")
```

```
Validation accuracy: 45.88%
Validation loss: 10.79%
```

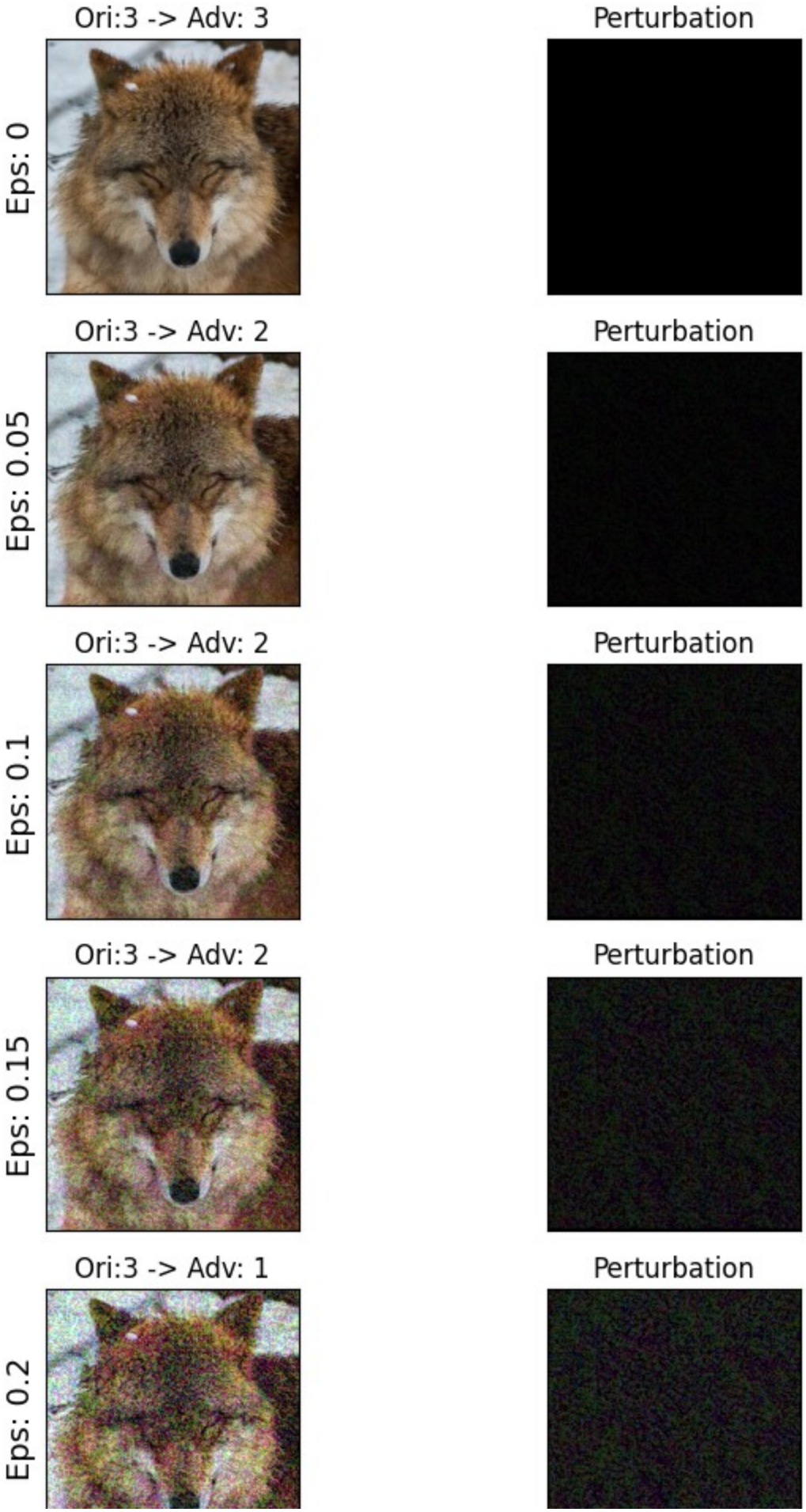
In [ ]:
```
# # imshow(torchvision.utils.make_grid(images.cpu().data, normalize=True), [normal_
# images, labels = next(iter(train_loader))
# fig, axis = plt.subplots(3, 5, figsize=(15, 10))
# for i, ax in enumerate(axis.flat):
#     with torch.no_grad():
#         npimg = images[i].numpy()
#         npimg = np.transpose(npimg, (1, 2, 0))
#         label = labels[i].item()  # Use the integer label directly
#         ax.imshow(npimg)
#         ax.set(title=f"Label {label}")

# plt.show()
```

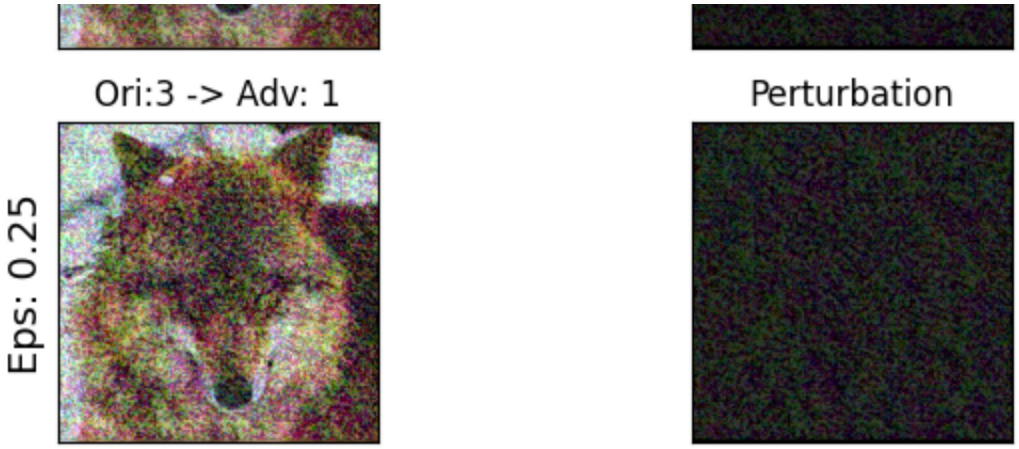
In [ ]:
```
def fgsm_attack(image, epsilon, image_input_grad):
    sign_grad = image_input_grad.sign()

    perturbed_image = image + epsilon * sign_grad
    perturbation = epsilon * sign_grad

    # Each pixel value's range is [0,1]
    # After performing adversarial attack, the pixel values may go out of boundary
    # Add clipping to maintain [0,1] range
    perturbed_image = torch.clamp(perturbed_image, 0, 1)

    # Return the perturbed image and perturbation
    return perturbed_image, perturbation
```

In [ ]:
```
lossfunction = nn.CrossEntropyLoss()
def perform_attack(epsilon, image_input, image_label, model):
    # Set requires_grad attribute of tensor as true, which will be used for Attack
    image_input.requires_grad = True
    output = model(image_input)
    init_pred = output.max(1, keepdim=True)[1]
    if init_pred.item() != image_label.item():
        print("The predicted label and the true label are different. Please choose
    # Calculate the loss
    loss = lossfunction(output, image_label.long())
    # Zero all existing gradients
    model.zero_grad()
    # Calculate gradients of loss in backward pass
    loss.backward()
    # Obtain the gradient with respect to input image
    image_input_grad = image_input.grad.data
    # Call FGSM attack
    perturbed_image, perturbation = fgsm_attack(image_input, epsilon, image_input_g
    # Re-classify the perturbed image
    adv_output = model(perturbed_image)
    adv_pred = adv_output.max(1, keepdim=True)[1]
    return perturbed_image, perturbation, init_pred, adv_pred
```

In [ ]:
```
#epsilons
epsilons = [0, .05, .1, .15, .2, .25]

# Choose a random image as input
image_sample_index = 18

# Assuming you have a validation_loader
dataiter = iter(validation_loader)
samples = next(dataiter)
image_input, image_label = samples[0][image_sample_index:image_sample_index+1], sam

# Lists to store results
adversarial_examples = []
adv_preds = []
init_preds = []
perturbations = []

# Loop through each epsilon value
for epsilon in epsilons:
    # Perform FGSM attack
    perturbed_image, perturbation, init_pred, adv_pred = perform_attack(epsilon, im
    # Append results to lists
    adversarial_examples.append(perturbed_image.squeeze().detach().numpy())
    adv_preds.append(adv_pred.item())
    init_preds.append(init_pred.item())
    perturbations.append(perturbation.squeeze().detach().numpy())
# Plot adversarial example and the perturbation at each epsilon
cnt = 0
plt.figure(figsize=(8,12))
for i in range(len(epsilons)):
    cnt = 2 * i + 1
    plt.subplot(len(epsilons), 2, cnt)
    plt.xticks([], [])
    plt.yticks([], [])
    plt.ylabel("Eps: {}".format(epsilons[i]), fontsize=14)
    orig, adv, ex = init_preds[i], adv_preds[i], adversarial_examples[i]
    plt.title("Ori:{} -> Adv: {}".format(orig, adv))
    plt.imshow(ex.transpose(1, 2, 0))  # Transpose to (H, W, C) for image display

    plt.subplot(len(epsilons), 2, cnt+1)
    plt.xticks([], [])
    plt.yticks([], [])
    perturbation = perturbations[i]
    plt.title("Perturbation".format(orig, adv))
    plt.imshow(perturbation.transpose(1, 2, 0))  # Transpose to (H, W, C) for image
plt.tight_layout()
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
```


Ori:3 -> Adv: 3
Perturbation
Eps: 0
Ori:3 -> Adv: 2
Perturbation
Eps: 0.05
Ori:3 -> Adv: 2
Perturbation
Eps: 0.1
Ori:3 -> Adv: 2
Perturbation
Eps: 0.15
Ori:3 -> Adv: 1
Perturbation
Eps: 0.2



In [ ]:
```
def adversarial_train(model, criterion, optimizer, train_loader, eps=0.3, alpha=2/2
    for epoch in range(epochs):
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            # Generate adversarial examples
            images_adv = pgd_attack(model, images, labels, eps, alpha, iters)

            # Forward pass
            outputs = model(images_adv)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

def pgd_attack_batch(model, images, labels, criterion, epsilon=0.3, alpha=2/255, it
    model.eval()
    images, labels = images.to(device), labels.to(device)
    images.requires_grad = True

    # Forward pass
    outputs = model(images)
    loss = criterion(outputs, labels)

    # Zero all existing gradients
    model.zero_grad()

    # Backward pass to compute gradients
    loss.backward()

    # Generate adversarial examples for all images in the batch
    with torch.no_grad():
        # Get gradient of images w.r.t loss
        images_grad = images.grad.data

        # Generate adversarial examples using PGD
        images_adv = images + alpha * images_grad.sign()
        images_adv = torch.min(torch.max(images_adv, images - epsilon), images + ep
        images_adv = torch.clamp(images_adv, 0, 1)
```

```
    return images_adv
```

In [ ]:
```
def adversarial_train(model, criterion, optimizer, train_loader, eps=0.3, alpha=2/2
    for epoch in range(epochs):
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            # Generate adversarial examples for all images in the batch
            images_adv = pgd_attack_batch(model, images, labels, criterion)

            # Forward pass
            outputs = model(images_adv)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

In [ ]:
```
adversarial_train(poisoned_pgd_model, criterion, poisoned_pgd_optimizer, train_load
```

In [ ]:
```
validate_model(clean_model)
validate_model(poisoned_fgsm_model)
validate_model(poisoned_pgd_model)
```

```
Validation accuracy: 95.32%
Validation loss: 1.33%
Validation accuracy: 94.46%
Validation loss: 2.14%
Validation accuracy: 59.1%
Validation loss: 9.65%
```

Validation accuracies before and after training

Validation accuracy: 45.88% Validation loss: 10.79% Validation accuracy: 59.1% Validation loss: 9.65%

Discussion:

While adversarial training can enhance model robustness, our experiments show that it may not significantly improve performance against adversarial attacks. We observed a modest increase in validation accuracy from 45.88% to 59.1% post-adversarial training. However, balancing accuracy and robustness remains a challenge due to inherent trade-offs. The choice of attack method, hyperparameters, and model architecture influence the effectiveness of adversarial training. Additionally, adversarial training alone may not address all vulnerabilities, necessitating complementary defense mechanisms.

Challenges and Future Steps:

Future work involves exploring innovative training strategies, evaluating diverse datasets and attack scenarios, and enhancing transferability and domain-specific defense mechanisms. Interdisciplinary collaboration is key to advancing adversarial defense and ensuring the reliability of CNN models in real-world applications.