```
import torch
from torchvision import datasets, transforms
from torch.utils.data import Dataset, DataLoader
import random

# 1️⃣  Transform
transform = transforms.Compose([
    transforms.Resize((128,128)),
    transforms.ToTensor()
])

# 2️⃣  Download CIFAR-10
trainset = datasets.CIFAR10(root="/content/data", train=True,
                            download=True, transform=transform)
testset  = datasets.CIFAR10(root="/content/data", train=False,
                            download=True, transform=transform)

# 3️⃣  Convert to PAD-style dataset (simulate 50 % spoofs)
class PADSimulated(Dataset):
    def __init__(self, base):
        self.base = base
    def __len__(self): return len(self.base)
    def __getitem__(self, idx):
        img, _ = self.base[idx]
        label = 0
        if random.random() > 0.5:
            # simple “spoof” via blur + brightness change
            img = torch.clamp(img * 0.6, 0, 1)
            label = 1
        return img, torch.tensor(label)

train_ds = PADSimulated(trainset)
val_ds   = PADSimulated(testset)

train_dl = DataLoader(train_ds, batch_size=64, shuffle=True)
val_dl   = DataLoader(val_ds, batch_size=64)

print(f"✅ PAD-style dataset ready: {len(train_ds)} train, {len(va
```

```
100%|██████████| 170M/170M [00:04<00:00, 35.1MB/s]
✅ PAD-style dataset ready: 50000 train, 10000 val samples.
```

```
import torch.nn as nn
import torch.nn.functional as F

class SimplePADCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, padding=1), nn.ReLU()
            nn.AdaptiveAvgPool2d(1)
        )
        self.fc = nn.Linear(128, 2)  # 2 classes: real (0) / spoof

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

# Initialize model, loss, optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
model = SimplePADCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

```
num_epochs = 5  # you can increase later

for epoch in range(num_epochs):
    model.train()
    running_loss = 0
    correct = 0
    total = 0
    for imgs, labels in train_dl:
        imgs, labels = imgs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * imgs.size(0)
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
    train_loss = running_loss / total
    train_acc = correct / total

    # Validation
    model.eval()
    val_loss = 0
    val_correct = 0
    val_total = 0
    with torch.no_grad():
        for imgs, labels in val_dl:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * imgs.size(0)
            _, predicted = outputs.max(1)
            val_total += labels.size(0)
            val_correct += predicted.eq(labels).sum().item()

    val_loss /= val_total
    val_acc = val_correct / val_total

    print(f"Epoch [{epoch+1}/{num_epochs}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")
```

```
Epoch [1/5] Train Loss: 0.1617, Train Acc: 0.9392 | Val Loss: 0.0942
Epoch [2/5] Train Loss: 0.0978, Train Acc: 0.9703 | Val Loss: 0.1694
Epoch [3/5] Train Loss: 0.0990, Train Acc: 0.9712 | Val Loss: 0.0559
Epoch [4/5] Train Loss: 0.1049, Train Acc: 0.9683 | Val Loss: 0.1078
Epoch [5/5] Train Loss: 0.1022, Train Acc: 0.9703 | Val Loss: 0.0768
```

```
torch.save(model.state_dict(), "/content/pad_model_cifar10.pt")
```

```
# day 2 PQC integration
```

```
!pip install pycryptodome
```

```
Collecting pycryptodome
  Downloading pycryptodome-3.23.0-cp37-abi3-manylinux_2_17_x86_64.ma
Downloading pycryptodome-3.23.0-cp37-abi3-manylinux_2_17_x86_64.many
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.3/2.3 MB 41.9 MB/s eta
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.23.0
```

```
import torch
from torchvision import transforms
from torch.utils.data import DataLoader
from Crypto.Cipher import AES
import numpy as np
import os

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
```

```
# Define the same model as Day 1
class SimplePADCNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = torch.nn.Sequential(
            torch.nn.Conv2d(3, 32, 3, padding=1), torch.nn.ReLU(),
            torch.nn.MaxPool2d(2),
            torch.nn.Conv2d(32, 64, 3, padding=1), torch.nn.ReLU(),
            torch.nn.MaxPool2d(2),
            torch.nn.Conv2d(64, 128, 3, padding=1), torch.nn.ReLU()
            torch.nn.AdaptiveAvgPool2d(1)
        )
        self.fc = torch.nn.Linear(128, 2)

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        return self.fc(x)

model = SimplePADCNN().to(device)
model.load_state_dict(torch.load("/content/pad_model_cifar10.pt"))
model.eval()
print("✅ PAD model loaded")
```

```
✅ PAD model loaded
```

```
def aes_encrypt(img_tensor, key):
    """Encrypt a single image tensor (3x128x128)"""
    data = img_tensor.numpy().tobytes()
    cipher = AES.new(key, AES.MODE_GCM)
    ciphertext, tag = cipher.encrypt_and_digest(data)
    return ciphertext, cipher.nonce, tag

def aes_decrypt(ciphertext, key, nonce, tag, shape):
    """Decrypt back to tensor"""
    cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
    decrypted = cipher.decrypt_and_verify(ciphertext, tag)
    arr = np.frombuffer(decrypted, dtype=np.float32).reshape(shape)
    return torch.tensor(arr)
```

```
# Generate AES key
aes_key = os.urandom(32)
```

```
# Take a batch from validation set
val_iter = iter(val_dl)
imgs, labels = next(val_iter)
img = imgs[0].to(device)
label = labels[0].item()

# Encrypt
ciphertext, nonce, tag = aes_encrypt(img.cpu(), aes_key)

# Decrypt
decrypted_img = aes_decrypt(ciphertext, aes_key, nonce, tag, img.sh

# Inference
model.eval()
with torch.no_grad():
    output = model(decrypted_img.unsqueeze(0))
    pred = output.argmax(1).item()

print(f"Original label: {label}, PAD prediction after decryption: {
```

```
Original label: 1, PAD prediction after decryption: 1
```

```python
def kyber_generate_keys():
    """Simulate Kyber key generation"""
    return os.urandom(32), os.urandom(32)  # public_key, secret_key

def kyber_encrypt(key, pub):
    """Simulate encryption"""
    return key  # just placeholder

def kyber_decrypt(ciphertext, priv):
    """Simulate decryption"""
    return ciphertext  # just placeholder

pub, priv = kyber_generate_keys()
encrypted_key = kyber_encrypt(aes_key, pub)
recovered_key = kyber_decrypt(encrypted_key, priv)
print(f"AES key recovered correctly: {aes_key == recovered_key}")
```

```
AES key recovered correctly: True
```

```python
import time
import torch
import numpy as np

# Make sure model is in evaluation mode
model.eval()

total = 0
correct = 0
latencies = []

print("🔷 Starting secure PAD evaluation...")

# Loop over validation DataLoader
for imgs, labels in val_dl:
    for i in range(imgs.size(0)):
        img = imgs[i].to(device)
        label = labels[i].item()

        # Start timer for AES-GCM encrypt → decrypt → inference
        start_time = time.time()

        # 1️⃣ Encrypt
        ciphertext, nonce, tag = aes_encrypt(img.cpu(), aes_key)

        # 2️⃣ Decrypt
        decrypted_img = aes_decrypt(ciphertext, aes_key, nonce, tag

        # 3️⃣ PAD inference
        with torch.no_grad():
            output = model(decrypted_img.unsqueeze(0))
```

```
                pred = output.argmax(1).item()

            # End timer
            end_time = time.time()
            latencies.append(end_time - start_time)

            # Update accuracy metrics
            total += 1
            if pred == label:
                correct += 1

    # Compute final metrics
    accuracy = correct / total
    avg_latency_ms = (sum(latencies) / total) * 1000

    print(f"\n✅ PQC pipeline evaluation complete")
    print(f"Accuracy: {accuracy*100:.2f}%")
    print(f"Average latency per image: {avg_latency_ms:.2f} ms")
```

```
🔷 Starting secure PAD evaluation...

✅ PQC pipeline evaluation complete
Accuracy: 97.56%
Average latency per image: 2.54 ms
```

```
# Day 3: Evaluation, Reporting, and Visualization
```

```
import torch
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np
import time

# Make sure model is loaded and in eval mode
model.eval()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
from sklearn.metrics import accuracy_score, precision_score, recall

true_labels = []
pred_labels = []
latencies = []

for imgs, labels in val_dl:
    for i in range(imgs.size(0)):
        img = imgs[i].to(device)
        label = labels[i].item()

        # Encrypt → Decrypt → Inference
        start_time = time.time()
        ciphertext, nonce, tag = aes_encrypt(img.cpu(), aes_key)
        decrypted_img = aes_decrypt(ciphertext, aes_key, nonce, tag

        with torch.no_grad():
            output = model(decrypted_img.unsqueeze(0))
            pred = output.argmax(1).item()

        end_time = time.time()
        latencies.append(end_time - start_time)

        # Store labels
        true_labels.append(label)
        pred_labels.append(pred)

# Compute metrics
accuracy = accuracy_score(true_labels, pred_labels)
precision = precision_score(true_labels, pred_labels)
recall = recall_score(true_labels, pred_labels)
f1 = f1_score(true_labels, pred_labels)
avg_latency_ms = np.mean(latencies) * 1000

print(f"✅ PAD Secure Pipeline Metrics:")
print(f"Accuracy: {accuracy*100:.2f}%")
print(f"Precision: {precision*100:.2f}%")
print(f"Recall: {recall*100:.2f}%")
print(f"F1 Score: {f1*100:.2f}%")
print(f"Average latency per image: {avg_latency_ms:.2f} ms")
```

```
✅ PAD Secure Pipeline Metrics:
Accuracy: 97.64%
Precision: 95.48%
Recall: 99.98%
F1 Score: 97.68%
Average latency per image: 1.80 ms
```

```
# Get a few random samples
num_samples = 8
indices = np.random.choice(len(val_ds), num_samples, replace=False)

plt.figure(figsize=(16, 4))
for i, idx in enumerate(indices):
    img, label = val_ds[idx]
    img_display = img.permute(1,2,0).numpy()

    # Encrypt → Decrypt → Inference
    ciphertext, nonce, tag = aes_encrypt(img, aes_key)
    decrypted_img = aes_decrypt(ciphertext, aes_key, nonce, tag, img
    with torch.no_grad():
        pred = model(decrypted_img.unsqueeze(0).to(device)).argmax(1

    plt.subplot(1, num_samples, i+1)
    plt.imshow(img_display)
    plt.title(f"Label: {label}\nPred: {pred}")
    plt.axis('off')
plt.show()
```

```
import json

report = {
    "accuracy": accuracy,
    "precision": precision,
    "recall": recall,
    "f1_score": f1,
    "avg_latency_ms": avg_latency_ms
}

# Save JSON report
with open("/content/pad_secure_report.json", "w") as f:
    json.dump(report, f, indent=4)

print("✅ Report saved: pad_secure_report.json")
```

```
✅ Report saved: pad_secure_report.json
```

Start coding or generate with AI.