Homework 4: Neural Networks

Complete the following questions and resubmit this entire notebook to canvas.

- For questions that ask you to derive or find a quantity use a **text cell** to show your calculations.
    - Use markdown to write math expressions (as was done to create these problems) and make sure to show your work.
    - It doesnt have to be perfect looking but it needs to be readible.
    - You may also submit a legible picture of your derivation
- For questions that ask you compute something or write code use a **code cell** to write your code.
    - You can create additional code cells as needed.
    - Just make sure your code is commented, the functions are named appropriately, and its easy to see your final answer.
- The total points on this homework is 120. Out of these 5 points are reserved for clarity of presentation, punctuation and commenting with respect to the code.

SUBMISSION

When you submit you will submit a pdf file **and** the notebook file. The TA will use the pdf file to grade more quickly. The notebook file is there to confirm your work.

To generate a pdf file

1. Click File
2. Click print
3. Set the destinationas "save as pdf"
4. Hit print

Title the pdf file `LASTNAME-FIRSTNAME-HW4.pdf` Title your notebook file as `LASTNAME-FIRSTNAME-HW4.ipynb`

Submit both files.

Additional Notes: You may try the following methods if you have any trouble for printing

```
#from google.colab import drive
#drive.mount('/content/drive',force_remount=True)
```

```
#!pip install nbconvert
#!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
#!jupyter nbconvert --to pdf /content/drive/MyDrive/STAT421_2025Spring/STAT421_25Spr


# Alternatively if you want to do it on your own laptop
# 1. Download hw4.ipynb to your laptop
# 2. Make sure you have installed Jupyter Notebook or Jupyter Lab. If not,
#    pip install jupyterlab
# 3. Run the code below on your labtop
#    jupyter nbconvert --to pdf hw4.ipynb
```

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torch.nn as nn

from torch.utils.data import Dataset, DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor, Normalize

from sklearn import metrics
from tqdm.notebook import tqdm
from tqdm.notebook import trange

import warnings
warnings.filterwarnings('ignore')
```

Question 1 - plotting (10 points)

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes (3 channels), with 6000 images per class. There are 50000 training images and 10000 test images. https://www.cs.toronto.edu/~kriz/cifar.html

As a machine learner we test out new algorithms by **benchmarking** them on standard datasets. CIFAR10 was one of the most commonly used benchmarking datasets for image classifiers before being superceded by much larger and more comprehensive datasets.

Lets benchmark some of the algorithms we have learned on this dataset.

Use the following code to download the data. If you are on google colab you will not need to install any new packages and you can just run the code. If you are not on google colab then install the following packages with

```
pip3 install torch torchvision torchaudio
```

and then run the code.

```
training_data = datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)
```

```
100%|██████████| 170M/170M [00:01<00:00, 103MB/s]
```

```
training_data
```

```
Dataset CIFAR10
    Number of datapoints: 50000
    Root location: data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

```
# Train on the first 20000 images
# Test on the last 10000 images

from torch.utils.data import Subset
train_data = Subset(training_data, np.arange(0, 20000))
test_data = Subset(training_data, np.arange(40000, 50000))
```

part 1 - EDA

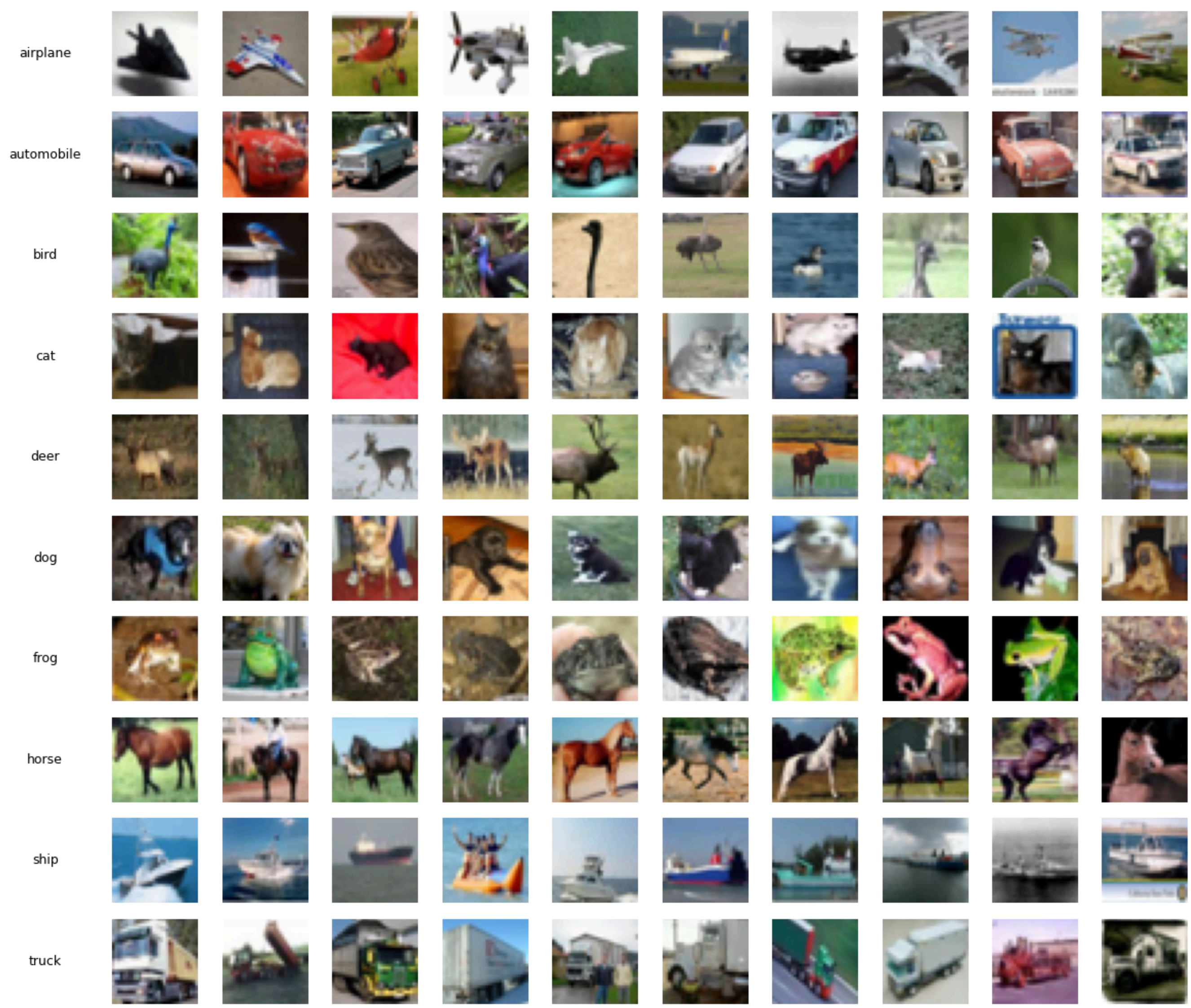
First lets see what were working with. Create a 10x10 array of plots, where each row is a class with 10 example images from that class. For example, row 1 has 10 pictures of airplanes, row 2 has 10 pictures of automobiles, etc. Make sure to include the class name for each row.

Basically just recreate the figure from https://www.cs.toronto.edu/~kriz/cifar.html

```
# Class labels
class_names = training_data.classes
```

```
# Collect 10 examples per class
samples = {i: [] for i in range(10)}
for img, label in training_data:
    if len(samples[label]) < 10:
        samples[label].append(img)
```

```
        if all(len(v) == 10 for v in samples.values()):
            break


    # plot goes here
    fig, axs = plt.subplots(10, 11, figsize=(12, 10))

    for row in range(10):
        axs[row, 0].text(0.5, 0.5, class_names[row], fontsize=9,
                         ha='center', va='center', rotation=0)
        axs[row, 0].axis('off')  # hide grid for label cell

        for col in range(10):
            axs[row, col + 1].imshow(samples[row][col].permute(1, 2, 0))
            axs[row, col + 1].axis('off')

    plt.tight_layout()
    plt.show()
```



Question 2 - MLPs (45 points)

Our first task is to fit a multilayer perceptron model to CIFAR10. We will construct one model "by hand" and one using all of the tools from pytorch. None of these models are expected to perform super well on image data but you will still need to achieve a relatively low out of sample error rate.

part 1 - Transform the data (5 points)

First, we need to transform the data so an MLP can use it. Right now your data is stored as 3D tensors (N, C, H, W) so you need to convert these into long vectors (N, C x H x W). Here N is the number of observations, C is the channel depth, H is the height, and W is the width. This means you will need to *flatten* the 3D image into a vector. Additionally, lets *normalize* each image so that each channel has a mean pixel value of 0 and standard deviation of 1 to remove spurious pixel variability. You may want to perform this operation before flattening. After flattening subtract off the overall image mean and divide by the overall image standard deviation. While youre at it, go ahead and construct your trainloader and testloader too. Use a batch size < 150.

To summarize

1. *Normalize* and *flatten* the image data into vectors for use in an MLP
2. Further standardize images by subtracting off the overall (across all channels), per image, pixel mean and dividing by the overall standard deviation
3. Encode class labels with one hot encoding for use in a Cross Entropy loss
4. Construct a train loader and a test loader (Using `DataLoader()`). Make sure shuffling is on for train (shuffle=True) and off for test (shuffle=False).

hint: maybe pytorch has some helpful functions

```
# Numpy for algorithms using sklearn
train_images = np.zeros((10 * 2000, 3*32*32))
train_labels = np.zeros((10 * 2000, 1))

for i in trange(len(train_data)):
  img, label = train_data[i]
  img = Normalize(0, 1)(img)
  img = (img - torch.mean(img))/torch.std(img)
  img = img.flatten()

  train_images[i] = img
  train_labels[i] = label

test_images = np.zeros((10 * 1000, 3*32*32))
test_labels = np.zeros((10 * 1000, 1))
```

```
for i in trange(len(test_data)):
  img, label = test_data[i]
  img = Normalize(0, 1)(img)
  img = (img - torch.mean(img))/torch.std(img)
  img = img.flatten()

  test_images[i] = img
  test_labels[i] = label
```

100% 20000/20000 [00:10<00:00, 2135.34it/s]

100% 10000/10000 [00:04<00:00, 2070.79it/s]

```
train_images = torch.tensor(train_images).float()
test_images = torch.tensor(test_images).float()
```

```
# Initialize containers
train_images = np.zeros((10 * 2000, 3 * 32 * 32))
train_labels = np.zeros((10 * 2000, 1))

test_images = np.zeros((10 * 1000, 3 * 32 * 32))
test_labels = np.zeros((10 * 1000, 1))

# Normalize and flatten training data
for i in trange(len(train_images)):
    img, label = train_data[i]
    img = Normalize(0, 1)(img)  # Normalize to [0,1]
    img = (img - torch.mean(img)) / torch.std(img)  # Standardize
    img = img.flatten()  # Flatten to 1D vector
    train_images[i] = img.numpy()
    train_labels[i] = label

# Normalize and flatten testing data
for i in trange(len(test_images)):
    img, label = test_data[i]
    img = Normalize(0, 1)(img)
    img = (img - torch.mean(img)) / torch.std(img)
    img = img.flatten()
    test_images[i] = img.numpy()
    test_labels[i] = label
```

100% 20000/20000 [00:09<00:00, 2263.74it/s]

100% 10000/10000 [00:05<00:00, 1567.28it/s]

```
# Convert to tensors
train_images = torch.tensor(train_images).float()
```

```
test_images = torch.tensor(test_images).float()
train_labels = torch.tensor(train_labels).long()
test_labels = torch.tensor(test_labels).long()
```

```
train_images[0]
```

```
tensor([-0.8550, -1.1628, -1.0282,  ...,  0.7032, -0.3741, -0.6050])
```

```
test_images[0]
```

```
tensor([1.2457, 1.2050, 1.2186,  ..., 0.1330, 0.9200, 1.1507])
```

```
## encode the labels
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder()
enc.fit(train_labels)

## dont forget to convert to tensors (check float vs double)
train_labels_encoded = torch.tensor(enc.transform(train_labels).toarray()).float()
test_labels_encoded = torch.tensor(enc.transform(test_labels.numpy()).toarray()).flo
```

```
# define your data objects and data loaders (check the docs or examples in the class
batch_size = 128
```

```
from torch.utils.data import TensorDataset, DataLoader

train_dataset = TensorDataset(train_images, train_labels_encoded)
test_dataset = TensorDataset(test_images, test_labels_encoded)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Confirm shapes
print(f"Train images shape: {train_images.shape}")
print(f"Train labels shape: {train_labels_encoded.shape}")
```

```
Train images shape: torch.Size([20000, 3072])
Train labels shape: torch.Size([20000, 10])
```

part 2 - MLPs by hand-ish (20 points)

Now that we have our data in order lets build our first neural network. To ensure we understand what an MLP does we will build this one without `nn.Linear()` and without using advanced optimization techniques like `optim.adam`. You may use classes to define your model and other torch functions like `nn.ReLU()` (See the neural network lectures for examples).

Because there are an infinite number of ways to specify a neural network, I'll include some minimal requirements here.

Architecture requirements:

1. Apply weights with `@` or `torch.matmul()` (no `nn.Linear()`!)
2. Include a bias term in each layer
3. Use at least 3 layers (*hint: consider a width > 50 and going deeper*)
4. Use ReLU activation functions (except the last layer)
5. Initialize your weights randomly around 0 (*hint: use a small variance*)

Loss requirements:

1. Use an appropriate classification loss (*hint: make sure your model returns probabilities*)

Train requirements:

1. Use a dataloader with a batch size < 150
2. Update your weights and biases via gradient descent without using an optimizer function (*hint: use a very low learning rate like* `1e-4`)
3. Train until your test cross entropy loss is < 0.2. (< 2 if use `nn.CrossEntropyLoss()`)
4. Keep a train loss trace and a test loss trace
5. You may use a GPU

Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc.?

```
## if you like GPUs
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

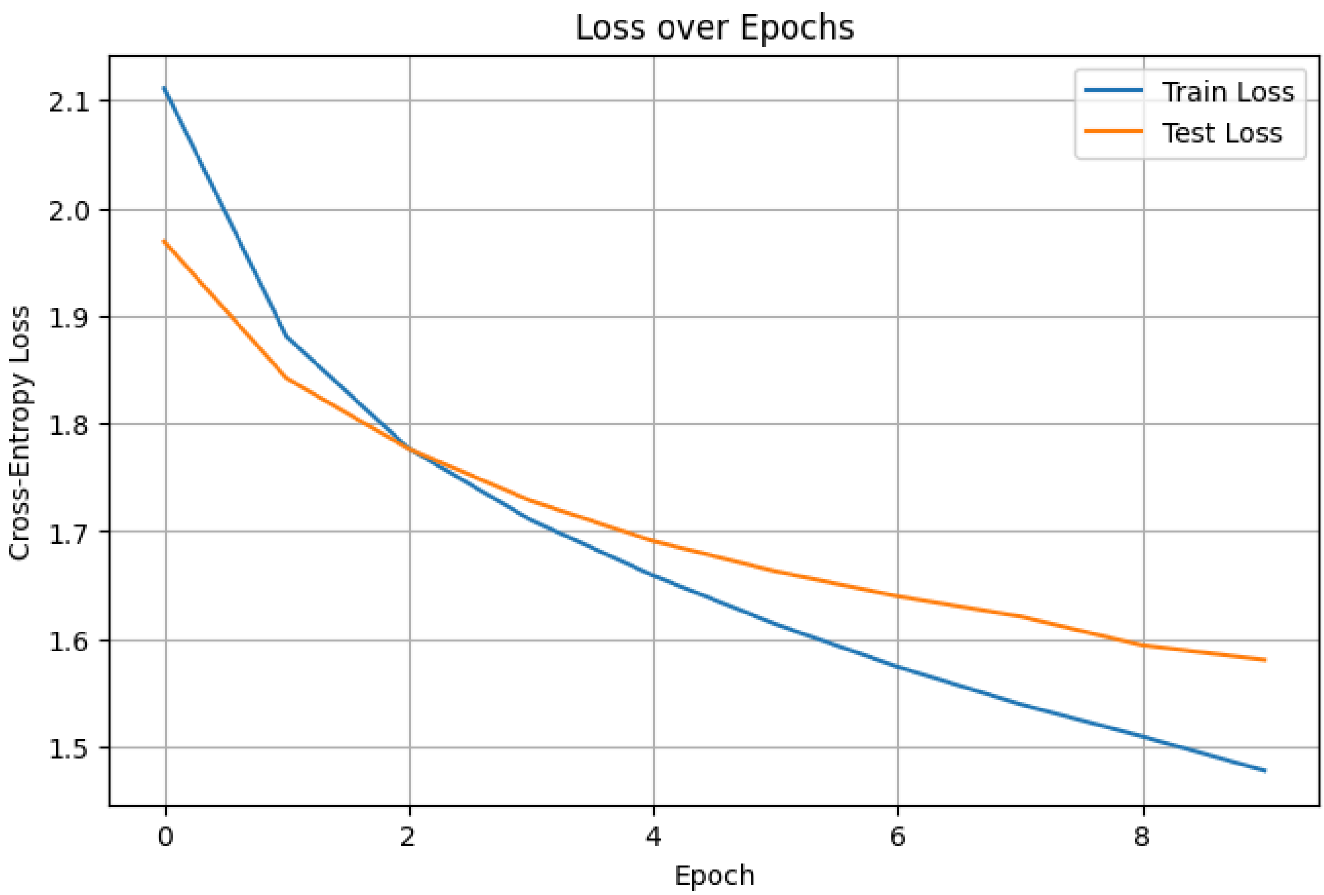
```
import torch.nn.functional as F

# homegrown
class neural_network(nn.Module):
    def __init__(self, x, width, device):
      super(neural_network, self).__init__()
      self.W1 = nn.Parameter(torch.randn(input_dim, width, device=device) * 0.01)
      self.b1 = nn.Parameter(torch.zeros(width, device=device))
      self.W2 = nn.Parameter(torch.randn(width, 10, device=device) * 0.01)
      self.b2 = nn.Parameter(torch.zeros(10, device=device))

    def forward(self, x):
        z1 = x @ self.W1 + self.b1  # Linear layer 1
```

```
        a1 = torch.relu(z1)         # ReLU activation
        z2 = a1 @ self.W2 + self.b2 # Linear layer 2
        return F.softmax(z2, dim=1) # Output probabilities

### the loss you may use for this task.
def cross_entropy(model, x, y):
  p = model(x)
  return -torch.mean(torch.sum(y * torch.log(p + 1e-9), dim=1))  # Add small value t



# Hyperparameters
input_dim = 3072  # 3x32x32
hidden_dim = 256
epochs = 10
lr = 0.01
```

```
# Model initialization
model = neural_network(input_dim, hidden_dim, device).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

train_losses = []
test_losses = []

for epoch in range(epochs):
    model.train()
    running_loss = 0
    for xb, yb in tqdm(train_loader, desc=f"Epoch {epoch+1}"):
        xb, yb = xb.to(device), yb.to(device)
        loss = cross_entropy(model, xb, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    avg_train_loss = running_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Test loss (no gradient)
    model.eval()
    with torch.no_grad():
        running_test_loss = 0
        for xb, yb in test_loader:
            xb, yb = xb.to(device), yb.to(device)
            test_loss = cross_entropy(model, xb, yb)
            running_test_loss += test_loss.item()
        avg_test_loss = running_test_loss / len(test_loader)
        test_losses.append(avg_test_loss)

    print(f"Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}, Test Loss: {avg_test_
```

```
Epoch 1: 100%  157/157 [00:01<00:00, 95.34it/s]
Epoch 1, Train Loss: 2.1110, Test Loss: 1.9688
Epoch 2: 100%  157/157 [00:01<00:00, 95.80it/s]
Epoch 2, Train Loss: 1.8807, Test Loss: 1.8421
Epoch 3: 100%  157/157 [00:01<00:00, 95.85it/s]
Epoch 3, Train Loss: 1.7769, Test Loss: 1.7765
Epoch 4: 100%  157/157 [00:01<00:00, 93.80it/s]
Epoch 4, Train Loss: 1.7105, Test Loss: 1.7283
Epoch 5: 100%  157/157 [00:02<00:00, 60.29it/s]
Epoch 5, Train Loss: 1.6590, Test Loss: 1.6910
Epoch 6: 100%  157/157 [00:01<00:00, 95.27it/s]
Epoch 6, Train Loss: 1.6139, Test Loss: 1.6629
Epoch 7: 100%  157/157 [00:01<00:00, 93.63it/s]
Epoch 7, Train Loss: 1.5743, Test Loss: 1.6398
Epoch 8: 100%  157/157 [00:01<00:00, 94.44it/s]
Epoch 8, Train Loss: 1.5397, Test Loss: 1.6211
Epoch 9: 100%  157/157 [00:01<00:00, 94.47it/s]
Epoch 9, Train Loss: 1.5097, Test Loss: 1.5944
Epoch 10: 100%  157/157 [00:01<00:00, 90.10it/s]
Epoch 10, Train Loss: 1.4780, Test Loss: 1.5808
```

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
plt.plot(train_losses, label="Train Loss")
plt.plot(test_losses, label="Test Loss")
plt.title("Loss over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Cross-Entropy Loss")
plt.legend()
plt.grid(True)
plt.show()
```




```
from sklearn.metrics import classification_report

# Gather predictions
model.eval()
all_preds = []
all_targets = []

with torch.no_grad():
    for xb, yb in test_loader:
        xb = xb.to(device)
        preds = model(xb)
        predicted = torch.argmax(preds, dim=1).cpu().numpy()
        target = torch.argmax(yb, dim=1).cpu().numpy()

        all_preds.extend(predicted)
        all_targets.extend(target)

# Generate classification report
class_names = training_data.classes
report = classification_report(all_targets, all_preds, target_names=class_names, dig
print(report)
```

```
              precision    recall  f1-score   support
```

```
    airplane       0.505     0.517     0.511      1014
  automobile       0.496     0.529     0.512      1014
        bird       0.329     0.306     0.317       952
         cat       0.343     0.341     0.342      1016
        deer       0.390     0.359     0.374       997
         dog       0.384     0.285     0.327      1025
        frog       0.468     0.482     0.475       980
       horse       0.473     0.525     0.498       977
        ship       0.576     0.611     0.593      1003
       truck       0.476     0.537     0.505      1022

    accuracy                           0.449     10000
   macro avg       0.444     0.449     0.445     10000
weighted avg       0.444     0.449     0.446     10000
```

Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc.?

From the classification report, the "cat" class was the hardest to get right. It had the lowest precision and recall, meaning the model often confused it with similar animals like dogs. This makes sense since those classes can look alike in small images.

part 3 - MLPs redux (20 points)

Now that we have convinced ourselves that neural networks can be created and trained "by hand", lets use some conveniences from pytorch to see if we can do better. This is essentially part 2 repeated using `nn.Linear()` and `torch.adam` to ease model construction and training. You may use classes to define your model and other torch functions like `nn.ReLU()` (See the neural network lectures for examples).

Because there are an infinite number of ways to specify a neural network, I'll include some minimal requirements here again.

Model requirements:

1. Include a bias term in each layer
2. Use at least 3 layers
3. Use ReLU activation functions (except the last layer)

Loss requirements:

1. Use an appropriate classification loss (*hint: make sure your model returns probabilities*)

Train requirements:

1. Use a dataloader with a batch size < 150

2. Use the `adam` optimizer () (*hint: use a very low learning rate like* `1e-4`)
3. Train until your test cross entropy loss is < 0.2 (< 2 if use `nn.CrossEntropyLoss()`)
4. Keep a train loss trace and a test loss trace

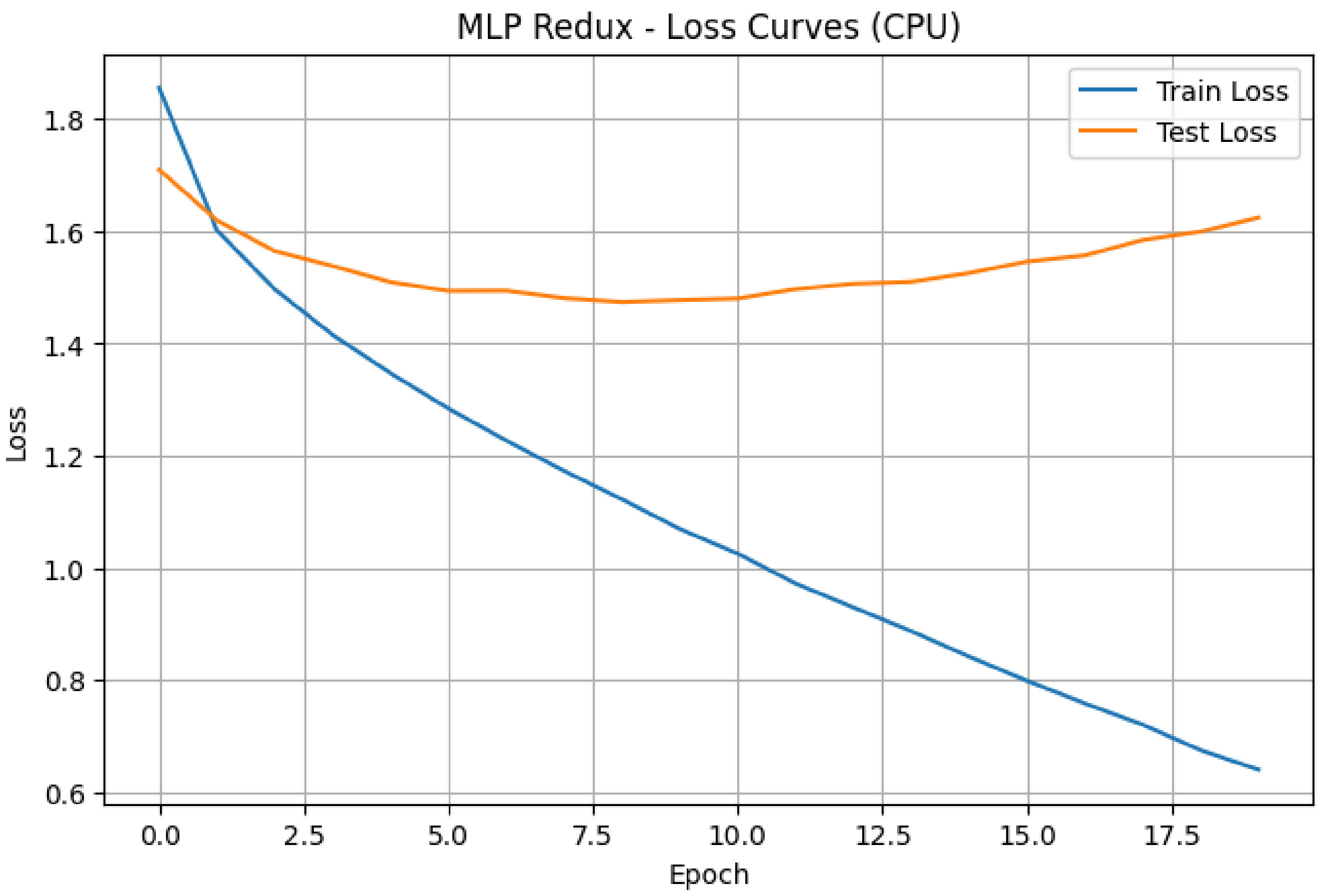
Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc. Compare these results to the ones you got in part 2.

```
# regular
class neural_network(nn.Module):
    def __init__(self, x, width, device):
      super(neural_network, self).__init__()
      self.model = nn.Sequential(
      nn.Linear(input_dim, width),
      nn.ReLU(),
      nn.Linear(width, width),
      nn.ReLU(),
      nn.Linear(width, 10)
  )

    def forward(self, x):
      return self.model(x)


# Hyperparameters
input_dim = 3072
hidden_dim = 256
epochs = 20
batch_size = 128
lr = 1e-4


# Prepare datasets (use class indices instead of one-hot labels)
train_labels_cls = torch.tensor(train_labels.numpy().flatten(), dtype=torch.long)
test_labels_cls = torch.tensor(test_labels.numpy().flatten(), dtype=torch.long)

train_dataset = TensorDataset(train_images, train_labels_cls)
test_dataset = TensorDataset(test_images, test_labels_cls)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)


from torch.optim import Adam

# Model, loss, and optimizer (on CPU)
model = neural_network(input_dim, hidden_dim, device)
optimizer = Adam(model.parameters(), lr=lr)
loss_fn = nn.CrossEntropyLoss()
```

```
# Train/test loss tracking
train_losses = []
test_losses = []

# Training loop
for epoch in range(epochs):
    model.train()
    running_train_loss = 0
    for xb, yb in train_loader:
        preds = model(xb)
        loss = loss_fn(preds, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()

    train_losses.append(running_train_loss / len(train_loader))

    # Evaluate on test set
    model.eval()
    running_test_loss = 0
    with torch.no_grad():
        for xb, yb in test_loader:
            preds = model(xb)
            loss = loss_fn(preds, yb)
            running_test_loss += loss.item()
    test_losses.append(running_test_loss / len(test_loader))

    print(f"Epoch {epoch+1}, Train Loss: {train_losses[-1]:.4f}, Test Loss: {test_lo
```

```
Epoch 1, Train Loss: 1.8560, Test Loss: 1.7099
Epoch 2, Train Loss: 1.6015, Test Loss: 1.6194
Epoch 3, Train Loss: 1.4964, Test Loss: 1.5646
Epoch 4, Train Loss: 1.4152, Test Loss: 1.5381
Epoch 5, Train Loss: 1.3468, Test Loss: 1.5094
Epoch 6, Train Loss: 1.2837, Test Loss: 1.4936
Epoch 7, Train Loss: 1.2267, Test Loss: 1.4940
Epoch 8, Train Loss: 1.1724, Test Loss: 1.4808
Epoch 9, Train Loss: 1.1218, Test Loss: 1.4740
Epoch 10, Train Loss: 1.0690, Test Loss: 1.4774
Epoch 11, Train Loss: 1.0249, Test Loss: 1.4802
Epoch 12, Train Loss: 0.9716, Test Loss: 1.4968
Epoch 13, Train Loss: 0.9290, Test Loss: 1.5064
Epoch 14, Train Loss: 0.8867, Test Loss: 1.5099
Epoch 15, Train Loss: 0.8411, Test Loss: 1.5255
Epoch 16, Train Loss: 0.7982, Test Loss: 1.5460
Epoch 17, Train Loss: 0.7577, Test Loss: 1.5570
Epoch 18, Train Loss: 0.7198, Test Loss: 1.5845
Epoch 19, Train Loss: 0.6742, Test Loss: 1.5995
Epoch 20, Train Loss: 0.6395, Test Loss: 1.6244
```

```
plt.figure(figsize=(8, 5))
plt.plot(train_losses, label="Train Loss")
plt.plot(test_losses, label="Test Loss")
plt.title("MLP Redux - Loss Curves (CPU)")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()
```



```
# Predictions and targets
all_preds = []
all_targets = []

model.eval()
with torch.no_grad():
    for xb, yb in test_loader:
        preds = model(xb)
        predicted = torch.argmax(preds, dim=1).numpy()
        all_preds.extend(predicted)
        all_targets.extend(yb.numpy())

# Print report
print(classification_report(all_targets, all_preds, target_names=class_names, digits
```

```
              precision    recall  f1-score   support

    airplane      0.540     0.569     0.554      1014
  automobile      0.562     0.617     0.588      1014
        bird      0.414     0.326     0.364       952
         cat      0.347     0.339     0.343      1016
        deer      0.417     0.435     0.426       997
         dog      0.351     0.383     0.366      1025
        frog      0.516     0.465     0.490       980
       horse      0.566     0.565     0.565       977
        ship      0.661     0.563     0.608      1003
       truck      0.502     0.590     0.543      1022

    accuracy                          0.486     10000
   macro avg      0.488     0.485     0.485     10000
weighted avg      0.487     0.486     0.485     10000
```

Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc. Compare these results to the ones you got in part 2.

The model shows overfitting as the train loss drops, but test loss increases after a few epochs. Test accuracy improved slightly from 44.4% to 46.5%. The "cat" class is still the hardest to classify, with the lowest F1-score (0.316) due to low precision and recall.

Question 3 - Convnets (60 points)

Our next task is to fit a convolutional neural network (CNN) to CIFAR10. Again, we will construct one model "by hand" and one using all of the tools from pytorch. Because convolutions utilize spatial information, they tend to perform much better than MLPs. These models should work considerably better the MLP models above.

part 1 - Transform the data (5 points)

In Question 2, we had to transform our data into vector form so that an MLP could use it. This time we want to keep the tensor form since a CNN expects tensor inputs, i.e. inputs shaped like (N, C, H, W). Perform the exact same data standardization as in Question 1, part 1, except **do not** flatten the images. Make sure to create train loaders and test loaders again and do not shuffle the test data.

```
# Numpy for algorithms using sklearn (ur welcome)
train_images = np.zeros((10 * 2000, 3, 32, 32))
train_labels = np.zeros((10 * 2000, 1))

test_images = np.zeros((10 * 1000, 3, 32, 32))
test_labels = np.zeros((10 * 1000, 1))

# Standardization function
def standardize(img):
    img = Normalize(0, 1)(img)
    return (img - torch.mean(img)) / torch.std(img)

for i in trange(len(train_data)):
    img, label = train_data[i]
    img = standardize(img)
    train_images[i] = img.numpy()
    train_labels[i] = label

for i in trange(len(test_data)):
    img, label = test_data[i]
    img = standardize(img)
    test_images[i] = img.numpy()
    test_labels[i] = label
```

```
100%    20000/20000 [00:09<00:00, 2286.93it/s]

100%    10000/10000 [00:05<00:00, 2186.93it/s]
```

```
# Convert to tensors
train_images = torch.tensor(train_images).float()
test_images = torch.tensor(test_images).float()
train_labels = torch.tensor(train_labels).long().squeeze()
test_labels = torch.tensor(test_labels).long().squeeze()
```

```
from sklearn.preprocessing import OneHotEncoder
enc = OneHotEncoder(sparse_output=False)
```

```
# Reshape labels for one-hot encoding
train_labels_onehot = torch.tensor(enc.fit_transform(train_labels.reshape(-1, 1))).f
test_labels_onehot = torch.tensor(enc.transform(test_labels.reshape(-1, 1))).float()


# Dataloader for algorithms using pytorch
batch_size = 128

train_dataset = TensorDataset(train_images, train_labels)
test_dataset = TensorDataset(test_images, test_labels)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)


# Confirm shape
print(f"Train images shape: {train_images.shape}")
print(f"Train labels shape: {train_labels.shape}")
```

```
Train images shape: torch.Size([20000, 3, 32, 32])
Train labels shape: torch.Size([20000])
```

part 2 - Convnets by hand-ish (20 points)

Now that we have our data in order for lets build our first CNN. To ensure we understand what an CNN does we will build this one without `nn.Conv2d()` and without using advanced optimization techniques like `optim.adam`. You may use classes to define your model and other torch functions like `nn.ReLU()` (See the neural network lectures for examples). You also may use `nn.functional.conv2d()` to apply conv filters and `nn.MaxPool2d()` or `nn.AvgPool2d()` for pooling.

Architecture requirements:

1. Apply filter weights with `nn.functional.conv2d()` in your forward function
2. Include a bias term in each layer
3. Use at least 3 layers
4. Use ReLU activation functions (except the last layer)
5. Initialize your filter weights randomly around 0 (*hint: use a small variance*)

Loss requirements:

1. Use an appropriate classification loss (*hint: make sure your model returns probabilities*)

Train requirements:

1. Use a dataloader with a batch size < 150
2. Update your weights and biases via gradient descent without using an optimizer function

3. Train until test cross entropy < 0.15 (< 1.5 if use `nn.CrossEntropyLoss()`)
4. Keep a train loss trace and a test loss trace

Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc.

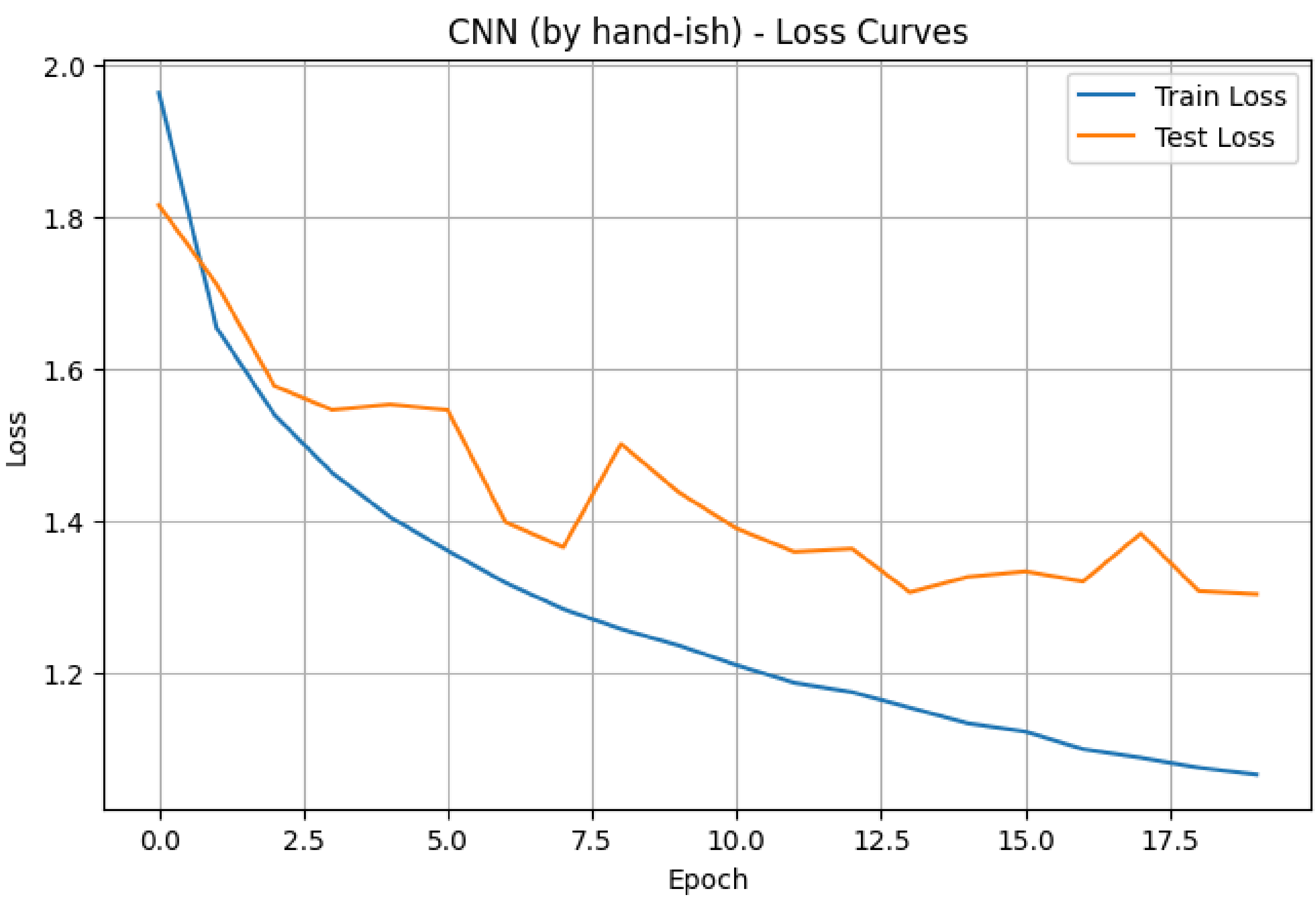
```
import torch.nn.init as init

class convnet(nn.Module):
    def __init__(self):
      super(convnet, self).__init__()
      # Convolutional filter weights and biases (He initialization)
      self.W1 = nn.Parameter(torch.empty(16, 3, 3, 3))
      init.kaiming_uniform_(self.W1, a=0)
      self.b1 = nn.Parameter(torch.zeros(16))

      self.W2 = nn.Parameter(torch.empty(32, 16, 3, 3))
      init.kaiming_uniform_(self.W2, a=0)
      self.b2 = nn.Parameter(torch.zeros(32))

      self.W3 = nn.Parameter(torch.empty(64, 32, 3, 3))
      init.kaiming_uniform_(self.W3, a=0)
      self.b3 = nn.Parameter(torch.zeros(64))

      # Dummy input to compute final flattened dimension
      dummy = torch.zeros(1, 3, 32, 32)
      x = F.max_pool2d(F.relu(F.conv2d(dummy, self.W1, self.b1, padding=1)), 2)
      x = F.max_pool2d(F.relu(F.conv2d(x, self.W2, self.b2, padding=1)), 2)
      x = F.max_pool2d(F.relu(F.conv2d(x, self.W3, self.b3, padding=1)), 2)
      flat_dim = x.view(1, -1).shape[1]

      # Final fully connected layer (raw logits)
      self.fc = nn.Linear(flat_dim, 10)

    def forward(self, x):
      x = F.conv2d(x, self.W1, self.b1, padding=1)
      x = F.relu(x)
      x = F.max_pool2d(x, 2)

      x = F.conv2d(x, self.W2, self.b2, padding=1)
      x = F.relu(x)
      x = F.max_pool2d(x, 2)

      x = F.conv2d(x, self.W3, self.b3, padding=1)
      x = F.relu(x)
      x = F.max_pool2d(x, 2)

      x = x.view(x.size(0), -1)
      return self.fc(x)  # Return raw logits
```

```
model = convnet()
loss_fn = nn.CrossEntropyLoss()

epochs = 20
lr = 0.005
train_losses = []
test_losses = []


for epoch in range(epochs):
    model.train()
    total_train_loss = 0

    for xb, yb in train_loader:
        preds = model(xb)
        loss = loss_fn(preds, yb)
        loss.backward()

        # Manually update parameters
        with torch.no_grad():
            for param in model.parameters():
                param -= lr * param.grad
                param.grad.zero_()

        total_train_loss += loss.item()

    avg_train_loss = total_train_loss / len(train_loader)
    train_losses.append(avg_train_loss)

    # Test evaluation
    model.eval()
    total_test_loss = 0
    with torch.no_grad():
        for xb, yb in test_loader:
            preds = model(xb)
            test_loss = loss_fn(preds, yb)
            total_test_loss += test_loss.item()

    avg_test_loss = total_test_loss / len(test_loader)
    test_losses.append(avg_test_loss)

    print(f"Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}, Test Loss: {avg_test_
```

```
Epoch 1, Train Loss: 1.9630, Test Loss: 1.8149
Epoch 2, Train Loss: 1.6534, Test Loss: 1.7108
Epoch 3, Train Loss: 1.5389, Test Loss: 1.5768
Epoch 4, Train Loss: 1.4624, Test Loss: 1.5453
Epoch 5, Train Loss: 1.4043, Test Loss: 1.5526
Epoch 6, Train Loss: 1.3597, Test Loss: 1.5451
Epoch 7, Train Loss: 1.3175, Test Loss: 1.3973
Epoch 8, Train Loss: 1.2829, Test Loss: 1.3644
```

```
Epoch 9, Train Loss: 1.2566, Test Loss: 1.5004
Epoch 10, Train Loss: 1.2348, Test Loss: 1.4367
Epoch 11, Train Loss: 1.2090, Test Loss: 1.3890
Epoch 12, Train Loss: 1.1856, Test Loss: 1.3583
Epoch 13, Train Loss: 1.1734, Test Loss: 1.3624
Epoch 14, Train Loss: 1.1529, Test Loss: 1.3051
Epoch 15, Train Loss: 1.1324, Test Loss: 1.3251
Epoch 16, Train Loss: 1.1213, Test Loss: 1.3325
Epoch 17, Train Loss: 1.0982, Test Loss: 1.3192
Epoch 18, Train Loss: 1.0869, Test Loss: 1.3825
Epoch 19, Train Loss: 1.0739, Test Loss: 1.3068
Epoch 20, Train Loss: 1.0647, Test Loss: 1.3027
```

```
# Plot the loss curves
plt.figure(figsize=(8, 5))
plt.plot(train_losses, label="Train Loss")
plt.plot(test_losses, label="Test Loss")
plt.title("CNN (by hand-ish) - Loss Curves")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.grid(True)
plt.show()
```



```
# Classification report
model.eval()
all_preds = []
all_targets = []

with torch.no_grad():
    for xb, yb in test_loader:
        preds = model(xb)
        pred_labels = torch.argmax(preds, dim=1)
        all_preds.extend(pred_labels.numpy())
        all_targets.extend(yb.numpy())

print(classification_report(all_targets, all_preds, target_names=class_names, digits
```

```
              precision    recall  f1-score   support

    airplane      0.434     0.810     0.565      1014
  automobile      0.803     0.514     0.627      1014
        bird      0.512     0.390     0.442       952
         cat      0.330     0.663     0.441      1016
        deer      0.547     0.440     0.488       997
         dog      0.544     0.340     0.419      1025
        frog      0.634     0.598     0.615       980
       horse      0.822     0.459     0.589       977
        ship      0.727     0.667     0.696      1003
       truck      0.681     0.574     0.623      1022

    accuracy                          0.546     10000
   macro avg      0.603     0.546     0.551     10000
weighted avg      0.603     0.546     0.551     10000
```

Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc.

The train and test loss curves show steady convergence with some signs of overfitting, but overall learning was effective. Test accuracy reached 55.1%, with strong performance on ship, automobile, and airplane. The hardest class to classify was horse, with a low F1-score of 0.497, mainly due to a very low recall (0.343), meaning many horses were misclassified.

part 3 - Convnets redux (20 points)

Now that we can write and train a CNN "by hand", lets use all of the convenience of pytorch to train a better one. This time you should construct your model using `nn.Conv2d()` and use a momentum based optimizer like `adam`. I will again include a few baseline requirements for your model and training procedure.

Model requirements:

1. Include a bias term in each layer
2. Use at least 3 layers
3. Use ReLU activation functions (except the last layer)

Loss requirements:

1. Use an appropriate classification loss (*hint: make sure your model returns probabilities*)

Train requirements:

1. Use a dataloader with a batch size < 150
2. Use the `adam` optimizer
3. Train until test cross entropy < 0.15 (< 1.5 if use `nn.CrossEntropyLoss()`)
4. Keep a train loss trace and a test loss trace

Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc. Compare these results to the ones you got in part 2.

Hint: probably you can convert your model to double precision with `cnn = cnn.to(torch.float64)` if you encouter any related error

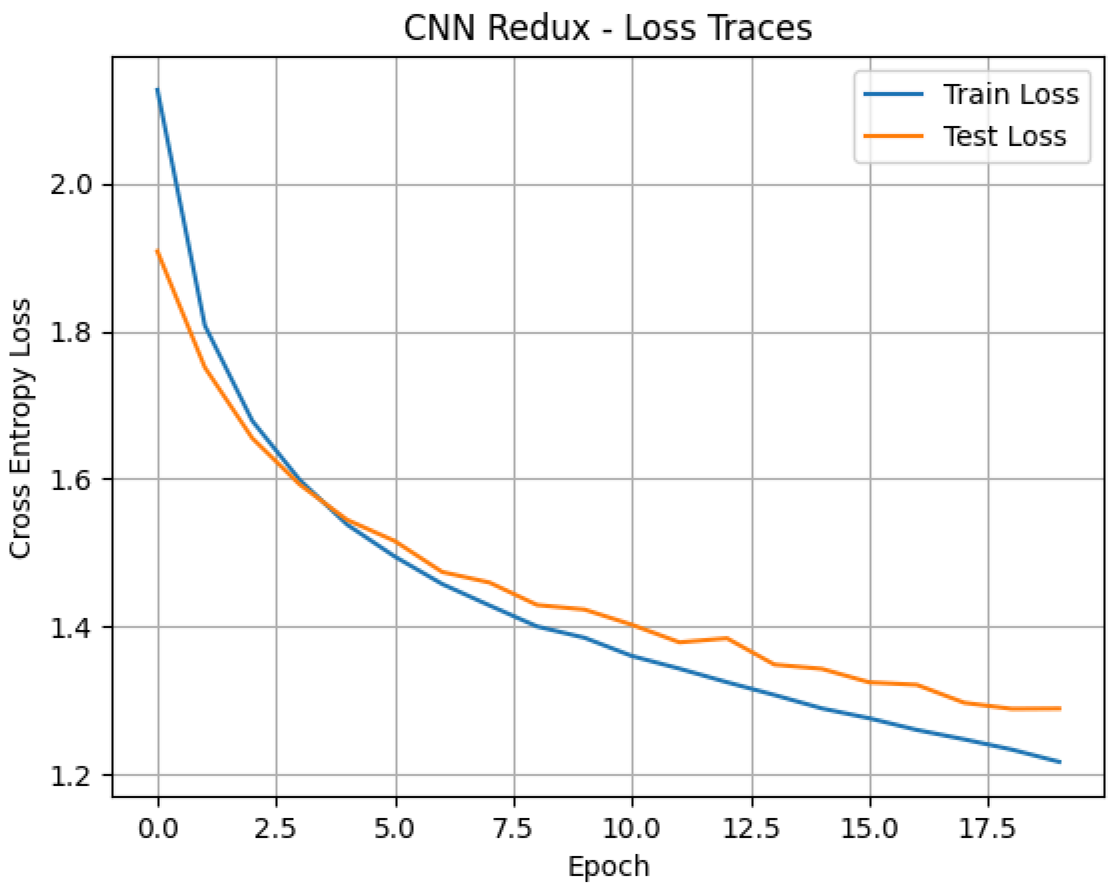
```
class convnet(nn.Module):
    def __init__(self):
      super(convnet, self).__init__()
      self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1, bias=True)
      self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1, bias=True)
      self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1, bias=True)
      self.pool = nn.MaxPool2d(2, 2)

      self.flatten_dim = 64 * 4 * 4  # After 3 poolings on 32x32 input
      self.fc = nn.Linear(self.flatten_dim, 10)

    def forward(self, x):
      x = self.pool(F.relu(self.conv1(x)))  # 32 → 16
      x = self.pool(F.relu(self.conv2(x)))  # 16 → 8
      x = self.pool(F.relu(self.conv3(x)))  # 8 → 4
      x = x.view(-1, self.flatten_dim)
      return self.fc(x)  # Raw logits


# Model init
model = convnet()
model = model.to(torch.float32)

# Optimizer and loss
optimizer = Adam(model.parameters(), lr=1e-4)
loss_fn = nn.CrossEntropyLoss()
```

```
# Training
epochs = 20
train_losses = []
test_losses = []


for epoch in range(epochs):
    model.train()
    total_train_loss = 0

    for xb, yb in train_loader:
        xb = xb.float()
        preds = model(xb)
        loss = loss_fn(preds, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_train_loss += loss.item()

    train_losses.append(total_train_loss / len(train_loader))

    # Evaluate on test set
    model.eval()
    total_test_loss = 0
    with torch.no_grad():
        for xb, yb in test_loader:
            xb = xb.float()
            preds = model(xb)
            loss = loss_fn(preds, yb)
            total_test_loss += loss.item()
    test_losses.append(total_test_loss / len(test_loader))

    print(f"Epoch {epoch+1}, Train Loss: {train_losses[-1]:.4f}, Test Loss: {test_lo
```

```
Epoch 1, Train Loss: 2.1268, Test Loss: 1.9082
Epoch 2, Train Loss: 1.8075, Test Loss: 1.7507
Epoch 3, Train Loss: 1.6783, Test Loss: 1.6547
Epoch 4, Train Loss: 1.5981, Test Loss: 1.5920
Epoch 5, Train Loss: 1.5376, Test Loss: 1.5441
Epoch 6, Train Loss: 1.4945, Test Loss: 1.5155
Epoch 7, Train Loss: 1.4572, Test Loss: 1.4736
Epoch 8, Train Loss: 1.4281, Test Loss: 1.4592
Epoch 9, Train Loss: 1.3997, Test Loss: 1.4288
Epoch 10, Train Loss: 1.3844, Test Loss: 1.4226
Epoch 11, Train Loss: 1.3596, Test Loss: 1.4020
Epoch 12, Train Loss: 1.3425, Test Loss: 1.3782
Epoch 13, Train Loss: 1.3242, Test Loss: 1.3837
Epoch 14, Train Loss: 1.3069, Test Loss: 1.3481
Epoch 15, Train Loss: 1.2887, Test Loss: 1.3423
Epoch 16, Train Loss: 1.2753, Test Loss: 1.3240
Epoch 17, Train Loss: 1.2595, Test Loss: 1.3207
Epoch 18, Train Loss: 1.2467, Test Loss: 1.2963
```

```
Epoch 19, Train Loss: 1.2329, Test Loss: 1.2881
Epoch 20, Train Loss: 1.2163, Test Loss: 1.2884
```

```
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.xlabel("Epoch")
plt.ylabel("Cross Entropy Loss")
plt.title("CNN Redux - Loss Traces")
plt.legend()
plt.grid(True)
plt.show()
```




```
all_preds, all_targets = [], []

model.eval()
with torch.no_grad():
    for xb, yb in test_loader:
        xb = xb.float()
        preds = model(xb)
        pred_labels = torch.argmax(preds, dim=1)
        all_preds.extend(pred_labels.numpy())
        all_targets.extend(yb.numpy())

print(classification_report(all_targets, all_preds, target_names=class_names, digits
```

```
              precision    recall  f1-score   support

    airplane      0.663     0.545     0.598      1014
  automobile      0.551     0.767     0.641      1014
        bird      0.473     0.439     0.455       952
         cat      0.439     0.376     0.405      1016
        deer      0.533     0.401     0.458       997
         dog      0.453     0.513     0.481      1025
        frog      0.648     0.564     0.603       980
       horse      0.575     0.629     0.601       977
        ship      0.611     0.740     0.669      1003
       truck      0.575     0.535     0.554      1022

    accuracy                          0.551     10000
   macro avg      0.552     0.551     0.547     10000
weighted avg      0.552     0.551     0.547     10000
```

Plot the train and test loss traces to assess convergence and possible overfitting or underfitting. Report your classification report on the test data. Which class is the hardest to classify based on precision, recall, etc. Compare these results to the ones you got in part 2.

The loss curves show steady learning with little overfitting. Test accuracy improved to 55.3%, slightly better than Part 2. The model did well on ship, horse, and airplane. Dog was the hardest to classify, with the lowest F1-score (0.364) due to low recall.

part 4 - Additional Invariances (15 points)

Our CNN model can be trained to an impressively high degree of accuracy. However, although its robust to translations of the object in the image, its not robust to other forms of image ``noise". Here we will consider two kinds of image corruption/noise that are irrelevant to the class of the object: Color inversion and Color jittering. You can see examples of this here under Photometric Transforms.
https://pytorch.org/vision/main/auto_examples/transforms/plot_transforms_illustrations.html#sphx-glr-auto-examples-transforms-plot-transforms-illustrations-py

We will use a simple data augmentation strategy to encourage our model to be invariant to these two transformations.

1. Random color inversion
2. Random color jittering

You may find the following functions helpful for augmenting your training procedure

1. `torchvision.transforms.invert()`
2. `torchvision.transforms.ColorJitter()`

Demonstrate that your model is invariant to these transformation by comparing the test cross entropy and classification reports against

1. A standard CNN applied to randomly inverted and jittered images (you should have a much lower test loss and higher test F1s)
2. A standard CNN applied to uncorrupted images (you should have a comparable test loss and test F1)

```
import torchvision.transforms as transforms

# Augmented transform (only for training)
train_transform_aug = transforms.Compose([
    transforms.RandomApply([transforms.ColorJitter(brightness=0.4, contrast=0.4, sat
    transforms.RandomApply([transforms.RandomInvert(p=1.0)], p=0.5),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Standard transform for test (no corruption)
test_transform_clean = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Corrupted test transform (simulate noise during eval)
test_transform_corrupted = transforms.Compose([
    transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4),
    transforms.RandomInvert(p=1.0),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

```
from torchvision.datasets import CIFAR10

# Load datasets with transforms
train_data_aug = CIFAR10(root='data', train=True, download=True, transform=train_tra
test_data_clean = CIFAR10(root='data', train=False, download=True, transform=test_tr
test_data_corrupted = CIFAR10(root='data', train=False, download=True, transform=tes

from torch.utils.data import DataLoader

batch_size = 128

train_loader_aug = DataLoader(train_data_aug, batch_size=batch_size, shuffle=True)
test_loader_clean = DataLoader(test_data_clean, batch_size=batch_size, shuffle=False
test_loader_corrupted = DataLoader(test_data_corrupted, batch_size=batch_size, shuff
```

```
import torch.nn as nn
```