1. Assignment 4

Notation: Please do NOT dump ChatGPT – generated answers into the assignment, once your answers are detected from ChatGPT, you will lose the corresponding points.

1.1. Short answer questions:

(Write a short one or two sentence answer to each of the questions below. Make sure your answer is clear and concise) 1. Please briefly explain PCA and NMF, and describe their similarities and differences. (2 points) 2. Please explain why the initialization process of weights and bias is important for neural networks. (1 point) 3. How would you prevent overfitting when designing an artificial neural network? (1 point) 4. Please describe the key differences between convolutional neural network and multilayer perceptron? (1 point) 5. What is the “vanishing gradient” problem with neural networks based on sigmoid non-linearities? (1 point)

1.2. Coding question

1.2.1. LeNet-5 CIFAR10 Classifier

This notebook implements the classic LeNet-5 convolutional network. Please modify the code and train LeNet-5 on the CIFAR10 dataset for classification. The basic architecture is shown in the lecture slides.

LeNet-5 is commonly regarded as the pioneer of convolutional neural networks, consisting of a simple architecture (by modern standards). In total, LeNet-5 consists of only 5 learnable layers. 3 out of these layers are convolutional layers (C1, C3, C5), which are connected by two average pooling layers (S2 & S4). The penultimate layer is a fully connexted layer (F6), which is followed by the final output layer.

**The goal of this assignment is to try various techniques to improve the performance to over 80% accuracy.

You are expected to use Google Colab and Pytorch on this assignment.

1. Please summarize the modification you did to improve the performance (2pts)
2. Report the test accuracy, and provide the experiment logging record for your best model. (2 pts)

Tips: When using Google Colab, please use the free TPU to accelerate the model training. Otherwise, it may take hours to finish the model training. Specifically, please click 'change runtime type' from the 'Runtime' and change from CPU (default) to any available GPU options.

Turning it in: Submit the assignment on Canvas. Be sure to include your names. Submit a PDF version of the notebook as well as a runnable copy of the notebook.

1.3. Imports

[1]:
```
import os
import time

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader

from torchvision import datasets
from torchvision import transforms

import matplotlib.pyplot as plt
from PIL import Image


if torch.cuda.is_available():
    torch.backends.cudnn.deterministic = True
```

[11]:
```
```

1.4. Model Settings

[2]:
```
##########################
### SETTINGS
##########################

# Hyperparameters
RANDOM_SEED = 1
LEARNING_RATE = 0.002
BATCH_SIZE = 128
NUM_EPOCHS = 20

# Architecture
NUM_FEATURES = 32*32
NUM_CLASSES = 10

# Other
if torch.cuda.is_available():
    DEVICE = "cuda:0"
else:
    DEVICE = "cpu"

GRAYSCALE = False
```

1.4.1. CIFAR-10 Dataset

[3]:
```
##########################
### CIFAR-10 Dataset
##########################
```

```
train_mean = (0.5, 0.5, 0.5)
train_std = (0.5, 0.5, 0.5)

# resize_transform = transforms.Compose([transforms.Resize((32, 32)),
#                                        transforms.RandomCrop(32, padding=4),
#                                        transforms.RandomHorizontalFlip(),
#                                       transforms.ToTensor(),
#                                       transforms.Normalize(train_mean, train_std)])

resize_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize(train_mean, train_std),
])
##add augmentations
# aug_transform = transforms.Compose([
#     transforms.RandomCrop(32, padding=4),
#     transforms.RandomHorizontalFlip(),
#     transforms.ToTensor(),
#     transforms.Normalize(train_mean, train_std)
# ])
# resize_transform = aug_transform
# Note transforms.ToTensor() scales input images
# to 0-1 range
train_dataset = datasets.CIFAR10(root='data',
                                 train=True,
                                 transform=resize_transform,
                                 download=True)

test_dataset = datasets.CIFAR10(root='data',
                                train=False,
                                transform=resize_transform)


train_loader = DataLoader(dataset=train_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=8,
                          shuffle=True)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
                         num_workers=8,
                         shuffle=False)

# Checking the dataset
for images, labels in train_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break

# Checking the dataset
for images, labels in train_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break
```

```
Image batch dimensions: torch.Size([128, 3, 32, 32])
Image label dimensions: torch.Size([128])
Image batch dimensions: torch.Size([128, 3, 32, 32])
Image label dimensions: torch.Size([128])
```

[4]:
```
device = torch.device(DEVICE)
torch.manual_seed(0)

for epoch in range(2):

    for batch_idx, (x, y) in enumerate(train_loader):

        print('Epoch:', epoch+1, end='')
        print(' | Batch index:', batch_idx, end='')
        print(' | Batch size:', y.size()[0])

        x = x.to(device)
        y = y.to(device)
        break
```

```
Epoch: 1 | Batch index: 0 | Batch size: 128
Epoch: 2 | Batch index: 0 | Batch size: 128
```

[5]:
```
##########################
### MODEL
##########################


class LeNet5(nn.Module):

    def __init__(self, num_classes, grayscale=False):
        super(LeNet5, self).__init__()

        self.grayscale = grayscale
        self.num_classes = num_classes

        if self.grayscale:
            in_channels = 1
        else:
            in_channels = 3

        self.features = nn.Sequential(

            nn.Conv2d(in_channels, 32, kernel_size=5),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(32, 64, kernel_size=5),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
```

```
            nn.Conv2d(64, 128, kernel_size=5, padding=2),  # Output: 128x8x8
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )

        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),  # Output: 128x1x1
            nn.Flatten(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(128, num_classes),
        )


    def forward(self, x):
        x = self.features(x)
        # x = torch.flatten(x, 1)
        logits = self.classifier(x)
        probas = F.softmax(logits, dim=1)

        return logits, probas
```

[6]:
```
torch.manual_seed(RANDOM_SEED)

model = LeNet5(NUM_CLASSES, GRAYSCALE)
model.to(DEVICE)

optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE,weight_decay=1e-4)
# optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9, weight_decay=5e-4)
```

1.5. Training

[7]:
```
def compute_accuracy(model, data_loader, device):
    correct_pred, num_examples = 0, 0
    for i, (features, targets) in enumerate(data_loader):

        features = features.to(device)
        targets = targets.to(device)

        logits, probas = model(features)
        _, predicted_labels = torch.max(probas, 1)
        num_examples += targets.size(0)
        correct_pred += (predicted_labels == targets).sum()
    return correct_pred.float()/num_examples * 100


start_time = time.time()
for epoch in range(NUM_EPOCHS):

    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.to(DEVICE)
        targets = targets.to(DEVICE)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

        ### LOGGING
        if not batch_idx % 50:
            print ('Epoch: %03d/%03d | Batch %04d/%04d | Cost: %.4f'
                   %(epoch+1, NUM_EPOCHS, batch_idx,
                     len(train_loader), cost))



    model.eval()
    with torch.set_grad_enabled(False): # save memory during inference
        print('Epoch: %03d/%03d | Train: %.3f%%' % (
              epoch+1, NUM_EPOCHS,
              compute_accuracy(model, train_loader, device=DEVICE)))

    print('Time elapsed: %.2f min' % ((time.time() - start_time)/60))

print('Total Training Time: %.2f min' % ((time.time() - start_time)/60))
```

```
Epoch: 001/020 | Batch 0000/0391 | Cost: 2.2988
Epoch: 001/020 | Batch 0050/0391 | Cost: 1.8036
Epoch: 001/020 | Batch 0100/0391 | Cost: 1.7013
Epoch: 001/020 | Batch 0150/0391 | Cost: 1.4515
Epoch: 001/020 | Batch 0200/0391 | Cost: 1.4011
Epoch: 001/020 | Batch 0250/0391 | Cost: 1.4348
Epoch: 001/020 | Batch 0300/0391 | Cost: 1.3708
Epoch: 001/020 | Batch 0350/0391 | Cost: 1.1904
Epoch: 001/020 | Train: 50.864%
Time elapsed: 2.22 min
Epoch: 002/020 | Batch 0000/0391 | Cost: 1.2139
Epoch: 002/020 | Batch 0050/0391 | Cost: 1.2937
Epoch: 002/020 | Batch 0100/0391 | Cost: 1.4339
Epoch: 002/020 | Batch 0150/0391 | Cost: 1.2174
Epoch: 002/020 | Batch 0200/0391 | Cost: 1.4008
Epoch: 002/020 | Batch 0250/0391 | Cost: 1.2826
Epoch: 002/020 | Batch 0300/0391 | Cost: 1.1358
Epoch: 002/020 | Batch 0350/0391 | Cost: 1.0280
Epoch: 002/020 | Train: 61.628%
Time elapsed: 4.46 min
```

```
Epoch: 003/020 | Batch 0000/0391 | Cost: 0.9889
Epoch: 003/020 | Batch 0050/0391 | Cost: 1.3882
Epoch: 003/020 | Batch 0100/0391 | Cost: 1.0987
Epoch: 003/020 | Batch 0150/0391 | Cost: 1.2982
Epoch: 003/020 | Batch 0200/0391 | Cost: 0.9472
Epoch: 003/020 | Batch 0250/0391 | Cost: 0.9589
Epoch: 003/020 | Batch 0300/0391 | Cost: 0.9876
Epoch: 003/020 | Batch 0350/0391 | Cost: 1.0374
Epoch: 003/020 | Train: 64.646%
Time elapsed: 6.70 min
Epoch: 004/020 | Batch 0000/0391 | Cost: 0.9963
Epoch: 004/020 | Batch 0050/0391 | Cost: 0.9509
Epoch: 004/020 | Batch 0100/0391 | Cost: 0.8674
Epoch: 004/020 | Batch 0150/0391 | Cost: 1.0655
Epoch: 004/020 | Batch 0200/0391 | Cost: 1.0327
Epoch: 004/020 | Batch 0250/0391 | Cost: 0.8325
Epoch: 004/020 | Batch 0300/0391 | Cost: 0.9691
Epoch: 004/020 | Batch 0350/0391 | Cost: 0.9136
Epoch: 004/020 | Train: 70.798%
Time elapsed: 8.92 min
Epoch: 005/020 | Batch 0000/0391 | Cost: 0.8179
Epoch: 005/020 | Batch 0050/0391 | Cost: 0.8406
Epoch: 005/020 | Batch 0100/0391 | Cost: 0.9387
Epoch: 005/020 | Batch 0150/0391 | Cost: 0.7866
Epoch: 005/020 | Batch 0200/0391 | Cost: 0.9086
Epoch: 005/020 | Batch 0250/0391 | Cost: 0.7921
Epoch: 005/020 | Batch 0300/0391 | Cost: 0.8059
Epoch: 005/020 | Batch 0350/0391 | Cost: 0.8081
Epoch: 005/020 | Train: 68.930%
Time elapsed: 11.16 min
Epoch: 006/020 | Batch 0000/0391 | Cost: 0.9541
Epoch: 006/020 | Batch 0050/0391 | Cost: 0.8314
Epoch: 006/020 | Batch 0100/0391 | Cost: 0.7973
Epoch: 006/020 | Batch 0150/0391 | Cost: 0.8975
Epoch: 006/020 | Batch 0200/0391 | Cost: 0.8970
Epoch: 006/020 | Batch 0250/0391 | Cost: 0.7507
Epoch: 006/020 | Batch 0300/0391 | Cost: 0.8608
Epoch: 006/020 | Batch 0350/0391 | Cost: 1.0459
Epoch: 006/020 | Train: 72.254%
Time elapsed: 13.39 min
Epoch: 007/020 | Batch 0000/0391 | Cost: 0.8294
Epoch: 007/020 | Batch 0050/0391 | Cost: 0.8891
Epoch: 007/020 | Batch 0100/0391 | Cost: 0.8031
Epoch: 007/020 | Batch 0150/0391 | Cost: 0.7562
Epoch: 007/020 | Batch 0200/0391 | Cost: 0.8100
Epoch: 007/020 | Batch 0250/0391 | Cost: 0.7511
Epoch: 007/020 | Batch 0300/0391 | Cost: 0.9059
Epoch: 007/020 | Batch 0350/0391 | Cost: 0.5480
Epoch: 007/020 | Train: 74.586%
Time elapsed: 15.62 min
Epoch: 008/020 | Batch 0000/0391 | Cost: 0.6909
Epoch: 008/020 | Batch 0050/0391 | Cost: 0.6542
Epoch: 008/020 | Batch 0100/0391 | Cost: 0.6763
Epoch: 008/020 | Batch 0150/0391 | Cost: 0.9606
Epoch: 008/020 | Batch 0200/0391 | Cost: 0.8334
Epoch: 008/020 | Batch 0250/0391 | Cost: 0.9228
Epoch: 008/020 | Batch 0300/0391 | Cost: 0.6109
Epoch: 008/020 | Batch 0350/0391 | Cost: 0.7535
Epoch: 008/020 | Train: 74.218%
Time elapsed: 17.84 min
Epoch: 009/020 | Batch 0000/0391 | Cost: 0.7255
Epoch: 009/020 | Batch 0050/0391 | Cost: 0.6939
Epoch: 009/020 | Batch 0100/0391 | Cost: 0.7670
Epoch: 009/020 | Batch 0150/0391 | Cost: 0.8509
Epoch: 009/020 | Batch 0200/0391 | Cost: 0.7088
Epoch: 009/020 | Batch 0250/0391 | Cost: 0.8583
Epoch: 009/020 | Batch 0300/0391 | Cost: 0.8537
Epoch: 009/020 | Batch 0350/0391 | Cost: 0.6284
Epoch: 009/020 | Train: 76.092%
Time elapsed: 20.07 min
Epoch: 010/020 | Batch 0000/0391 | Cost: 0.7396
Epoch: 010/020 | Batch 0050/0391 | Cost: 0.6086
Epoch: 010/020 | Batch 0100/0391 | Cost: 0.6379
Epoch: 010/020 | Batch 0150/0391 | Cost: 0.7208
Epoch: 010/020 | Batch 0200/0391 | Cost: 0.7337
Epoch: 010/020 | Batch 0250/0391 | Cost: 0.5749
Epoch: 010/020 | Batch 0300/0391 | Cost: 0.7363
Epoch: 010/020 | Batch 0350/0391 | Cost: 0.7623
Epoch: 010/020 | Train: 77.148%
Time elapsed: 22.31 min
Epoch: 011/020 | Batch 0000/0391 | Cost: 0.8415
Epoch: 011/020 | Batch 0050/0391 | Cost: 0.7036
Epoch: 011/020 | Batch 0100/0391 | Cost: 0.5880
Epoch: 011/020 | Batch 0150/0391 | Cost: 0.7147
Epoch: 011/020 | Batch 0200/0391 | Cost: 0.6939
Epoch: 011/020 | Batch 0250/0391 | Cost: 0.7459
Epoch: 011/020 | Batch 0300/0391 | Cost: 0.6106
Epoch: 011/020 | Batch 0350/0391 | Cost: 0.6683
Epoch: 011/020 | Train: 76.082%
Time elapsed: 24.54 min
Epoch: 012/020 | Batch 0000/0391 | Cost: 0.5889
Epoch: 012/020 | Batch 0050/0391 | Cost: 0.7201
Epoch: 012/020 | Batch 0100/0391 | Cost: 0.6984
Epoch: 012/020 | Batch 0150/0391 | Cost: 0.6820
Epoch: 012/020 | Batch 0200/0391 | Cost: 0.5846
Epoch: 012/020 | Batch 0250/0391 | Cost: 0.6020
Epoch: 012/020 | Batch 0300/0391 | Cost: 0.7000
Epoch: 012/020 | Batch 0350/0391 | Cost: 0.7747
Epoch: 012/020 | Train: 77.554%
Time elapsed: 26.77 min
Epoch: 013/020 | Batch 0000/0391 | Cost: 0.7272
Epoch: 013/020 | Batch 0050/0391 | Cost: 0.6998
Epoch: 013/020 | Batch 0100/0391 | Cost: 0.7569
Epoch: 013/020 | Batch 0150/0391 | Cost: 0.7281
Epoch: 013/020 | Batch 0200/0391 | Cost: 0.5601
Epoch: 013/020 | Batch 0250/0391 | Cost: 0.6249
Epoch: 013/020 | Batch 0300/0391 | Cost: 0.5524
Epoch: 013/020 | Batch 0350/0391 | Cost: 0.6968
Epoch: 013/020 | Train: 78.700%
Time elapsed: 29.01 min
Epoch: 014/020 | Batch 0000/0391 | Cost: 0.7023
Epoch: 014/020 | Batch 0050/0391 | Cost: 0.4751
Epoch: 014/020 | Batch 0100/0391 | Cost: 0.5760
Epoch: 014/020 | Batch 0150/0391 | Cost: 0.7037
Epoch: 014/020 | Batch 0200/0391 | Cost: 0.6884
Epoch: 014/020 | Batch 0250/0391 | Cost: 0.7484
Epoch: 014/020 | Batch 0300/0391 | Cost: 0.6778
```

```
Epoch: 014/020 | Batch 0350/0391 | Cost: 0.5739
Epoch: 014/020 | Train: 78.264%
Time elapsed: 31.23 min
Epoch: 015/020 | Batch 0000/0391 | Cost: 0.5307
Epoch: 015/020 | Batch 0050/0391 | Cost: 0.5742
Epoch: 015/020 | Batch 0100/0391 | Cost: 0.5455
Epoch: 015/020 | Batch 0150/0391 | Cost: 0.5346
Epoch: 015/020 | Batch 0200/0391 | Cost: 0.7122
Epoch: 015/020 | Batch 0250/0391 | Cost: 0.7206
Epoch: 015/020 | Batch 0300/0391 | Cost: 0.6475
Epoch: 015/020 | Batch 0350/0391 | Cost: 0.6792
Epoch: 015/020 | Train: 78.856%
Time elapsed: 33.45 min
Epoch: 016/020 | Batch 0000/0391 | Cost: 0.6199
Epoch: 016/020 | Batch 0050/0391 | Cost: 0.6435
Epoch: 016/020 | Batch 0100/0391 | Cost: 0.6276
Epoch: 016/020 | Batch 0150/0391 | Cost: 0.7079
Epoch: 016/020 | Batch 0200/0391 | Cost: 0.6878
Epoch: 016/020 | Batch 0250/0391 | Cost: 0.5492
Epoch: 016/020 | Batch 0300/0391 | Cost: 0.7230
Epoch: 016/020 | Batch 0350/0391 | Cost: 0.6345
Epoch: 016/020 | Train: 80.124%
Time elapsed: 35.67 min
Epoch: 017/020 | Batch 0000/0391 | Cost: 0.6530
Epoch: 017/020 | Batch 0050/0391 | Cost: 0.6042
Epoch: 017/020 | Batch 0100/0391 | Cost: 0.7022
Epoch: 017/020 | Batch 0150/0391 | Cost: 0.5577
Epoch: 017/020 | Batch 0200/0391 | Cost: 0.5578
Epoch: 017/020 | Batch 0250/0391 | Cost: 0.5483
Epoch: 017/020 | Batch 0300/0391 | Cost: 0.5156
Epoch: 017/020 | Batch 0350/0391 | Cost: 0.7982
Epoch: 017/020 | Train: 79.710%
Time elapsed: 37.91 min
Epoch: 018/020 | Batch 0000/0391 | Cost: 0.5420
Epoch: 018/020 | Batch 0050/0391 | Cost: 0.4939
Epoch: 018/020 | Batch 0100/0391 | Cost: 0.6458
Epoch: 018/020 | Batch 0150/0391 | Cost: 0.6084
Epoch: 018/020 | Batch 0200/0391 | Cost: 0.5970
Epoch: 018/020 | Batch 0250/0391 | Cost: 0.3945
Epoch: 018/020 | Batch 0300/0391 | Cost: 0.6007
Epoch: 018/020 | Batch 0350/0391 | Cost: 0.5858
Epoch: 018/020 | Train: 81.334%
Time elapsed: 40.16 min
Epoch: 019/020 | Batch 0000/0391 | Cost: 0.6642
Epoch: 019/020 | Batch 0050/0391 | Cost: 0.5208
Epoch: 019/020 | Batch 0100/0391 | Cost: 0.5237
Epoch: 019/020 | Batch 0150/0391 | Cost: 0.4316
Epoch: 019/020 | Batch 0200/0391 | Cost: 0.6457
Epoch: 019/020 | Batch 0250/0391 | Cost: 0.5879
Epoch: 019/020 | Batch 0300/0391 | Cost: 0.6761
Epoch: 019/020 | Batch 0350/0391 | Cost: 0.6630
Epoch: 019/020 | Train: 79.482%
Time elapsed: 42.38 min
Epoch: 020/020 | Batch 0000/0391 | Cost: 0.4958
Epoch: 020/020 | Batch 0050/0391 | Cost: 0.6539
Epoch: 020/020 | Batch 0100/0391 | Cost: 0.5145
Epoch: 020/020 | Batch 0150/0391 | Cost: 0.5232
Epoch: 020/020 | Batch 0200/0391 | Cost: 0.5780
Epoch: 020/020 | Batch 0250/0391 | Cost: 0.4103
Epoch: 020/020 | Batch 0300/0391 | Cost: 0.5062
Epoch: 020/020 | Batch 0350/0391 | Cost: 0.5444
Epoch: 020/020 | Train: 81.256%
Time elapsed: 44.60 min
Total Training Time: 44.60 min
```

1.6. Evaluation

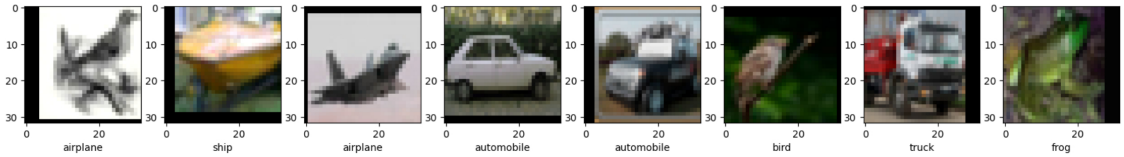
[8]:
```
with torch.set_grad_enabled(False): # save memory during inference
    print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader, device=DEVICE)))
```

```
Test accuracy: 77.98%
```

[9]:
```
class UnNormalize(object):
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):
        """
        Parameters:
        ------------
        tensor (Tensor): Tensor image of size (C, H, W) to be normalized.

        Returns:
        ------------
        Tensor: Normalized image.

        """
        for t, m, s in zip(tensor, self.mean, self.std):
            t.mul_(s).add_(m)
        return tensor

unorm = UnNormalize(mean=train_mean, std=train_std)
```

[10]:
```
test_loader = DataLoader(dataset=train_dataset,
                         batch_size=BATCH_SIZE,
                         shuffle=True)

for features, targets in test_loader:
    break


_, predictions = model.forward(features[:8].to(DEVICE))
predictions = torch.argmax(predictions, dim=1)

d = {0: 'airplane',
     1: 'automobile',
     2: 'bird',
     3: 'cat',
     4: 'deer',
     5: 'dog',
     6: 'frog',
```

```
    7: 'horse',
    8: 'ship',
    9: 'truck'}

fig, ax = plt.subplots(1, 8, figsize=(20, 10))
for i in range(8):
    img = unorm(features[i])
    ax[i].imshow(np.transpose(img, (1, 2, 0)))
    ax[i].set_xlabel(d[predictions[i].item()])

plt.show()
```



[ ]:

[ ]: