AlexNetwork for CIFAR10

May 19, 2024

1 Neural Networks

Note: before starting this notebook, please make a copy of it, otherwise your changes will not persist.

This part of the assignment is designed to get you familiar with how engineerings in the real world train neural network systems. It isn't designed to be difficult. In fact, everything you need to complete the assignment is available directly on the pytorch website here. This note book will have the following components:

1. Understanding the basics of Pytorch (no deliverables)
2. Training a simple neural network on MNIST (Deliverable = training graphs)
3. Train a model on CIFAR-10 for Kaggle (Deliverable = kaggle submission and explanation of methods)

The last part of this notebook is left open for you to explore as many techniques as you want to do as well as possible on the dataset.

You will also get practice being an ML engineer by reading documentation and using it to implement models. The first section of this notebook will cover an outline of what you need to know – we are confident that you can find the rest on your own.

Note that like all other assignments, you are free to use this notebook or not. You just need to complete the deliverables and turn in your code. If you want to run everything outside of the notebook, make sure to appropriately install pytorch to download the datasets and copy out the code for kaggle submission. If you don't want to use pytorch and instead want to use Tensorflow, feel free, but you may still need to install pytorch to download the datasets. That said, we will recommend pytorch over tensorflow since the latter has a somewhat steep learning curve and the former is more accessible to beginners.

[1]:
```
# Imports for pytorch
import numpy as np
import torch
import torchvision
from torch import nn
import matplotlib
from matplotlib import pyplot as plt
import tqdm

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

```
from torch.optim.lr_scheduler import ReduceLROnPlateau
import random
from torchvision import transforms

seed = 189
torch.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

2 1. Understanding Pytorch

Pytorch is based on the "autograd" paradigm. Essentially, you perform operations on multi-dimensional arrays like in numpy, except pytorch will automatically handle gradient tracking. In this section you will understand how to use pytorch.

This section should help you understand the full pipeline of creating and training a model in pytorch. Feel free to re-use code from this section in the assigned tasks.

Content in this section closely follows this pytorch tutorial: https://pytorch.org/tutorials/beginner/basics/intro.html

2.1 Tensors

Tensors can be created from numpy data or by using pytorch directly.

[2]:
```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

np_array = np.array(data)
x_np = torch.from_numpy(np_array)

shape = (2,3,)
rand_tensor = torch.rand(shape)
np_rand_array = rand_tensor.numpy()

print(f"Tensor from np: \n {x_np} \n")
print(f"Rand Tensor: \n {rand_tensor} \n")
print(f"Rand Numpy Array: \n {np_rand_array} \n")
```

```
Tensor from np:
 tensor([[1, 2],
        [3, 4]])

Rand Tensor:
 tensor([[0.2136, 0.0424, 0.7420],
```

```
        [0.3987, 0.8749, 0.7304]])

Rand Numpy Array:
 [[0.21356577 0.04237747 0.7420152 ]
 [0.39865988 0.87493867 0.73042464]]
```

They also support slicing and math operations very similar to numpy. See the examples below:

[3]:
```
# Slicing
tensor = torch.ones(4, 4)
print('First row: ',tensor[0])
print('First column: ', tensor[:, 0])

# Matrix Operations
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

# Getting a single item
scalar = torch.sum(y1) # sums all elements
item = scalar.item()
print("Sum as a tensor:", scalar, ", Sum as an item:", item)
```

```
First row:  tensor([1., 1., 1., 1.])
First column:  tensor([1., 1., 1., 1.])
Sum as a tensor: tensor(64.) , Sum as an item: 64.0
```

2.2 Autograd

This small section shows you how pytorch computes gradients. When we create tenors, we can set `requires_grad` to be true to indicate that we are using gradients. For most of the work that you actually do, you will use the `nn` package, which automatically sets all parameter tensors to have `requires_grad=True`.

[4]:
```
# Below is an example of computing the gradient for a single data point in␣
 ↪logistic regression using pytorch's autograd.

x = torch.ones(5)  # input tensor
y = torch.zeros(1) # label
w = torch.randn(5, 1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
pred = torch.sigmoid(torch.matmul(x, w) + b)
loss = torch.nn.functional.binary_cross_entropy(pred, y)
loss.backward() # Computers gradients
print("W gradient:", w.grad)
print("b gradient:", b.grad)

# when we want to actually take an update step, we can use optimizers:
optimizer = torch.optim.SGD([w, b], lr=0.1)
```

```
print("Weight before", w)
optimizer.step() # use the computed gradients to update
# Print updated weights
print("Updated weight", w)

# Performing operations with gradients enabled is slow...
# You can disable gradient computation using the following enclosure:
with torch.no_grad():
    # Perform operations without gradients
    ...
```

```
W gradient: tensor([[0.0538],
        [0.0538],
        [0.0538],
        [0.0538],
        [0.0538]])
b gradient: tensor([0.0538])
Weight before tensor([[ 0.3402],
        [-1.1699],
        [ 0.2658],
        [ 0.2334],
        [-3.1901]], requires_grad=True)
Updated weight tensor([[ 0.3348],
        [-1.1753],
        [ 0.2604],
        [ 0.2280],
        [-3.1955]], requires_grad=True)
```

2.3 Devices

Pytorch supports accelerating computation using GPUs which are available on google colab. To use a GPU on google colab, go to runtime -> change runtime type -> select GPU.

Note that there is some level of strategy for knowing when to use which runtime type. Colab will kick users off of GPU for a certain period of time if you use it too much. Thus, its best to run simple models and prototype to get everything working on CPU, then switch the instance type over to GPU for training runs and parameter tuning.

Its best practice to make sure your code works on any device (GPU or CPU) for pytorch, but note that numpy operations can only run on the CPU. Here is a standard flow for using GPU acceleration:

[5]:
```
# Determine the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)
# Next create your tensors
tensor = torch.zeros(4, 4, requires_grad=True)
# Move the tensor to the device you want to use
tensor = tensor.to(device)
```

```
# Perform whatever operations you want.... (often this will involve gradients)
# These operations will be accelerated by GPU.
tensor = 10*(tensor + 1)

# bring the tensor back to CPU, first detaching it from any gradient␣
 ↪computations
tensor = tensor.detach().cpu()

tensor_np = tensor.numpy() # Convert to numpy if you want to perform numpy␣
 ↪operations.
```

```
Using device cuda
```

2.4 The NN Package

Pytorch implements composable blocks in `Module` classes. All layers and modules in pytorch inherit from `nn.Module`. When you make a module you need to implement two functions: `__init__(self, *args, **kwargs)` and `forward(self, *args, **kwargs)`. Modules also have some nice helper functions, namely `parameters` which will recursively return all of the parameters. Here is an example of a logistic regression model:

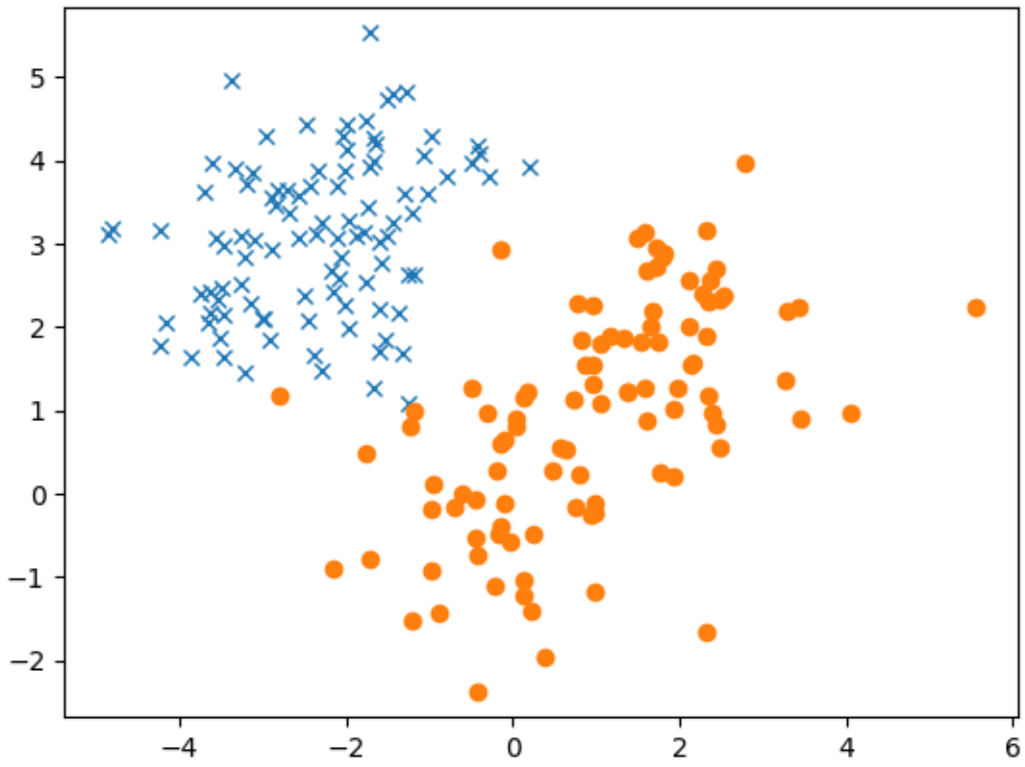
[6]:
```
class Perceptron(nn.Module):
  def __init__(self, in_dim):
    super().__init__()
    self.layer = nn.Linear(in_dim, 1) # This is a linear layer, it computes Xw␣
 ↪+ b

  def forward(self, x):
    return torch.sigmoid(self.layer(x)).squeeze(-1)

perceptron = Perceptron(10)
perceptron = perceptron.to(device) # Move all the perceptron's tensors to the␣
 ↪device
print("Parameters", list(perceptron.parameters()))
```

```
Parameters [Parameter containing:
tensor([[ 2.9598e-01,  6.9345e-02, -1.9255e-03,  4.6366e-03,  1.0172e-01,
         -2.0564e-01, -1.1215e-04, -9.6568e-02, -1.0986e-01,  2.9084e-01]],
       device='cuda:0', requires_grad=True), Parameter containing:
tensor([-0.1188], device='cuda:0', requires_grad=True)]
```

2.5 Datasets

Pytorch has nice interfaces for using datasets. Suppose we create a logistic regression dataset as follows:

[7]:
```
c1_x1, c1_x2 = np.random.multivariate_normal([-2.5,3], [[1, 0.3],[0.3, 1]],␣
↪100).T
c2_x1, c2_x2 = np.random.multivariate_normal([1,1], [[2, 1],[1, 2]], 100).T
c1_X = np.vstack((c1_x1, c1_x2)).T
c2_X = np.vstack((c2_x1, c2_x2)).T
train_X = np.concatenate((c1_X, c2_X))
train_y = np.concatenate((np.zeros(100), np.ones(100)))
# Shuffle the data
permutation = np.random.permutation(train_X.shape[0])
train_X = train_X[permutation, :]
train_y = train_y[permutation]
# Plot the data
plt.plot(c1_x1, c1_x2, 'x')
plt.plot(c2_x1, c2_x2, 'o')
plt.axis('equal')
plt.show()
```



We can then create a pytorch dataset object as follows. Often times, the default pytorch datasets will create these objects for you. Then, we can apply dataloaders to iterate over the dataset in batches.

[8]:
```
dataset = torch.utils.data.TensorDataset(torch.from_numpy(train_X), torch.
 ↪from_numpy(train_y))
# We can create a dataloader that iterates over the dataset in batches.
dataloader = torch.utils.data.DataLoader(dataset, batch_size=10, shuffle=True)
for x, y in dataloader:
    print("Batch x:", x)
    print("Batch y:", y)
    break

# Clean up the dataloader as we make a new one later
del dataloader
```

```
Batch x: tensor([[-3.5113,  1.8590],
        [-3.1178,  3.8620],
        [ 1.6830,  2.1970],
        [-2.8539,  3.4663],
        [-0.1537,  0.5909],
        [-0.1026, -0.1097],
        [ 1.7937,  2.8408],
        [-0.1430,  2.9392],
        [-1.6728,  1.2604],
        [ 2.5179,  2.3752]], dtype=torch.float64)
Batch y: tensor([0., 0., 1., 0., 1., 1., 1., 1., 0., 1.], dtype=torch.float64)
```

2.6 Training Loop Example

Here is an example of training a full logistic regression model in pytorch. Note the extensive use of modules – modules can be used for storing networks, computation steps etc.

[9]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

epochs = 10
batch_size = 10
learning_rate = 0.01

num_features = dataset[0][0].shape[0]
model = Perceptron(num_features).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
criterion = torch.nn.BCELoss()
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,␣
 ↪shuffle=True)

model.train() # Put model in training mode
for epoch in range(epochs):
    training_losses = []
    for x, y in tqdm.notebook.tqdm(dataloader, unit="batch"):
        x, y = x.float().to(device), y.float().to(device)
```

```
        optimizer.zero_grad() # Remove the gradients from the previous step
        pred = model(x)
        loss = criterion(pred, y)
        loss.backward()
        optimizer.step()
        training_losses.append(loss.item())
    print("Finished Epoch", epoch + 1, ", training loss:", np.
 ↪mean(training_losses))

# We can run predictions on the data to determine the final accuracy.
with torch.no_grad():
    model.eval() # Put model in eval mode
    num_correct = 0
    for x, y in dataloader:
        x, y = x.float().to(device), y.float().to(device)
        pred = model(x)
        num_correct += torch.sum(torch.round(pred) == y).item()
    print("Final Accuracy:", num_correct / len(dataset))
    model.train() # Put model back in train mode
```

```
Using device cuda
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 1 , training loss: 0.5665435269474983
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 2 , training loss: 0.47766369581222534
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 3 , training loss: 0.4212615966796875
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 4 , training loss: 0.38244073167443277
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 5 , training loss: 0.3533939987421036
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 6 , training loss: 0.33105754628777506
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 7 , training loss: 0.3132927596569061
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 8 , training loss: 0.2985064037144184
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 9 , training loss: 0.28632054179906846
```

```
  0%|          | 0/20 [00:00<?, ?batch/s]
Finished Epoch 10 , training loss: 0.275938618183136
Final Accuracy: 0.955
```

3 Task 1: MLP For FashionMNIST

Earlier in this course you trained SVMs and GDA models on MNIST. Now you will train a multi-layer perceptron model on an MNIST-like dataset. Your deliverables are as follows:

1. Code for training an MLP on MNIST (can be in code appendix, tagged in your submission).
2. A plot of the training loss and validation loss for each epoch of training after trainnig for at least 8 epochs.
3. A plot of the training and validation accuracy, showing that it is at least 82% for validation by the end of training.

Below we will create the training and validation datasets for you, and provide a very basic skeleton of the code. Please leverage the example training loop from above.

Some pytorch components you should definitely use: 1. `nn.Linear` 2. Some activation function like `nn.ReLU` 3. `nn.CrossEntropyLoss`: if you choose to use `nn.CrossEntropyLoss` or `F.cross_entropy`, DO NOT add an explicit softmax layer in your neural network. PyTorch devs found it more numerically stable to combine softmax and cross entropy loss into a single module and if you explicitly attach a softmax layer at the end of your model, you would unintentionally be applying it twice, which can degrade performance.

Here are challenges you will need to overcome: 1. The data is default configured in image form i.e. (1 x 28 x 28), versus one feature vector. You will need to reshape it somewhere to feed it in as vector to the MLP. There are many ways of doing this. 2. You need to write code for plotting. 3. You need to find appropriate hyper-parameters to achieve good accuracy.

Your underlying model must be fully connected or dense, and may not have convolutions etc., but you can use anything in torch.optim or any layers in torch.nn besides nn.Linear that do not have weights.

[10]:
```
# Creating the datasets
transform = torchvision.transforms.ToTensor() # feel free to modify this as you␣
 ↪see fit.

training_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=transform,
)

validation_data = torchvision.datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
```

```
    transform=transform,
)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to data/FashionMNIST/raw/train-images-idx3-ubyte.gz
```

```
100%|      | 26421880/26421880 [00:00<00:00, 114689174.70it/s]
```

```
Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to
data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
```

```
100%|      | 29515/29515 [00:00<00:00, 4690621.50it/s]
```

```
Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
```

```
100%|      | 4422102/4422102 [00:00<00:00, 63390580.45it/s]
```

```
Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
```

```
100%|      | 5148/5148 [00:00<00:00, 10218777.56it/s]
```

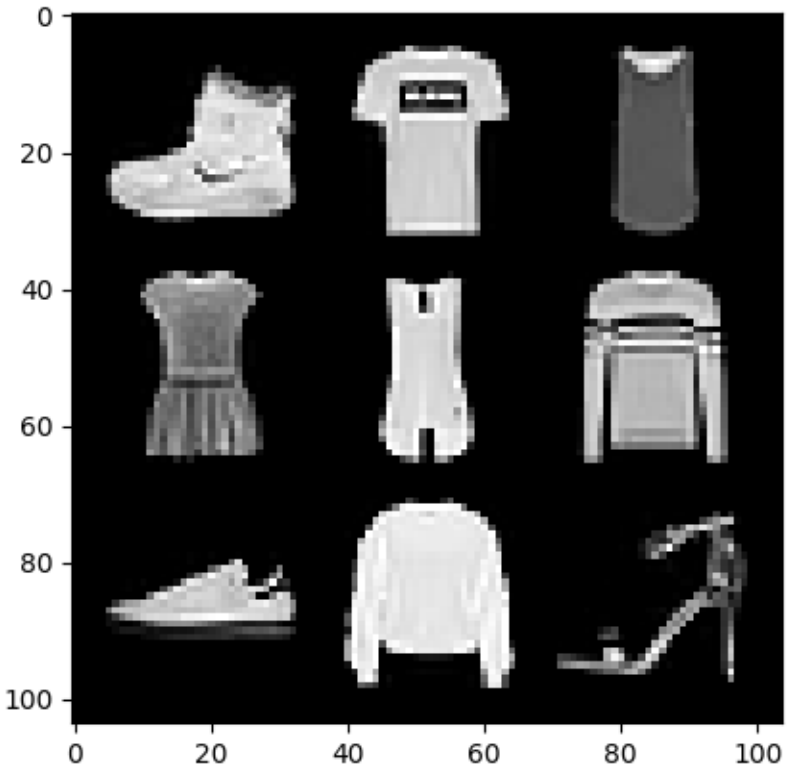
```
Extracting data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
data/FashionMNIST/raw
```

Before training a neural network, let's visualize our data first! Running the cell below will display the first 9 images in a 3 by 3 grid.

[11]:
```
images = [training_data[i][0] for i in range(9)]
```

```
plt.imshow(torchvision.utils.make_grid(torch.stack(images), nrow=3, padding=5).
  ↪numpy().transpose((1, 2, 0)))
```

[11]: <matplotlib.image.AxesImage at 0x793738402470>



[12]:
```
### YOUR CODE HERE ###
import numpy as np
import torch
import torchvision
from torch import nn
import matplotlib
from matplotlib import pyplot as plt
import tqdm

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

# Hyperparameters
epochs = 10
batch_size = 10
learning_rate = 0.01
```

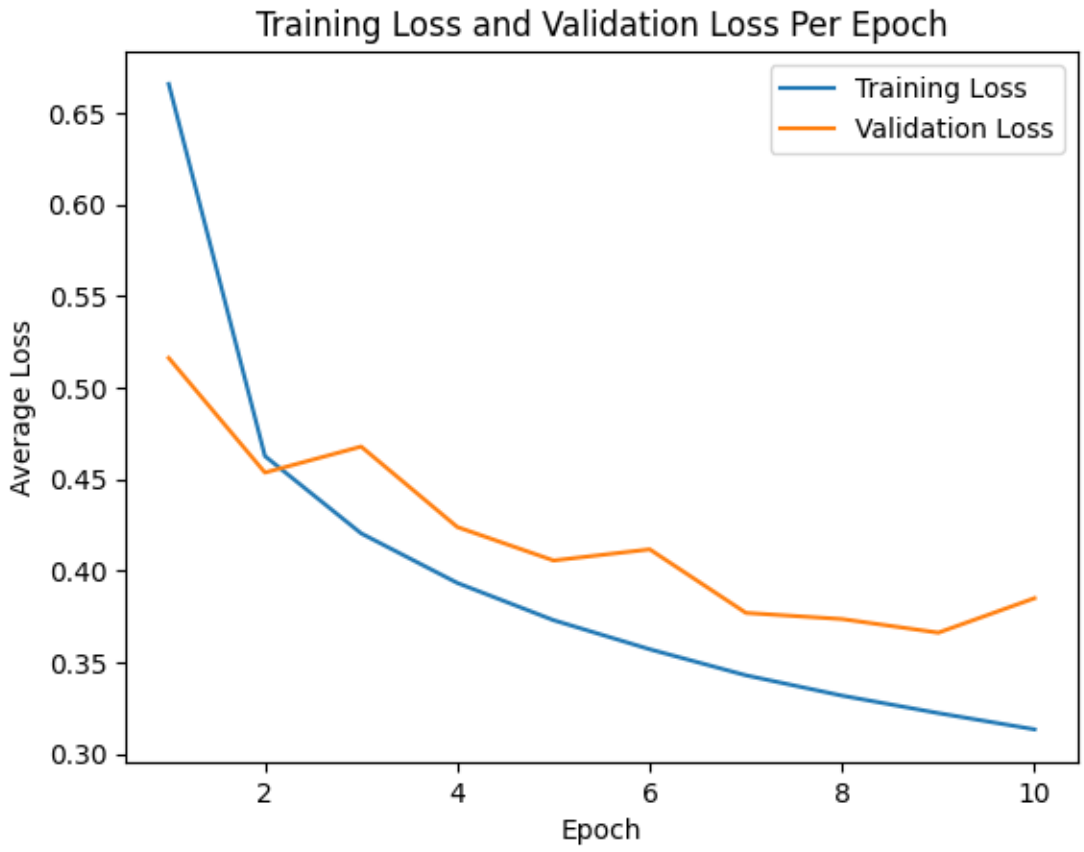
```
in_dim = 784

# Model definition
class MLP(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.flatten = nn.Flatten()
        self.layer = nn.Linear(in_dim, 128)
        self.relu = nn.ReLU()
        self.output = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.layer(x)
        x = self.relu(x)
        x = self.output(x)
        return x

# Calling Model
model = MLP(in_dim).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
criterion = torch.nn.CrossEntropyLoss()
train_dataloader = torch.utils.data.DataLoader(training_data,␣
 ↪batch_size=batch_size, shuffle=True)
valid_dataloader = torch.utils.data.DataLoader(validation_data,␣
 ↪batch_size=batch_size)

# Creating lists
train_losses, valid_losses = [], []
train_accuracy, valid_accuracy = [], []

for epoch in range(epochs):
    model.train()
    temp_train_losses, temp_valid_losses = [], []
    total_correct, total_samples = 0, 0

    # Training losses
    for x, y in train_dataloader:
        x, y = x.to(device), y.to(device, dtype=torch.long)
        optimizer.zero_grad() # Remove the gradients from the previous step
        pred = model(x)
        loss = criterion(pred, y)
        loss.backward()
        optimizer.step()
        temp_train_losses.append(loss.item())
        # try getting accuracy
        _, predicted = torch.max(pred, 1)
```

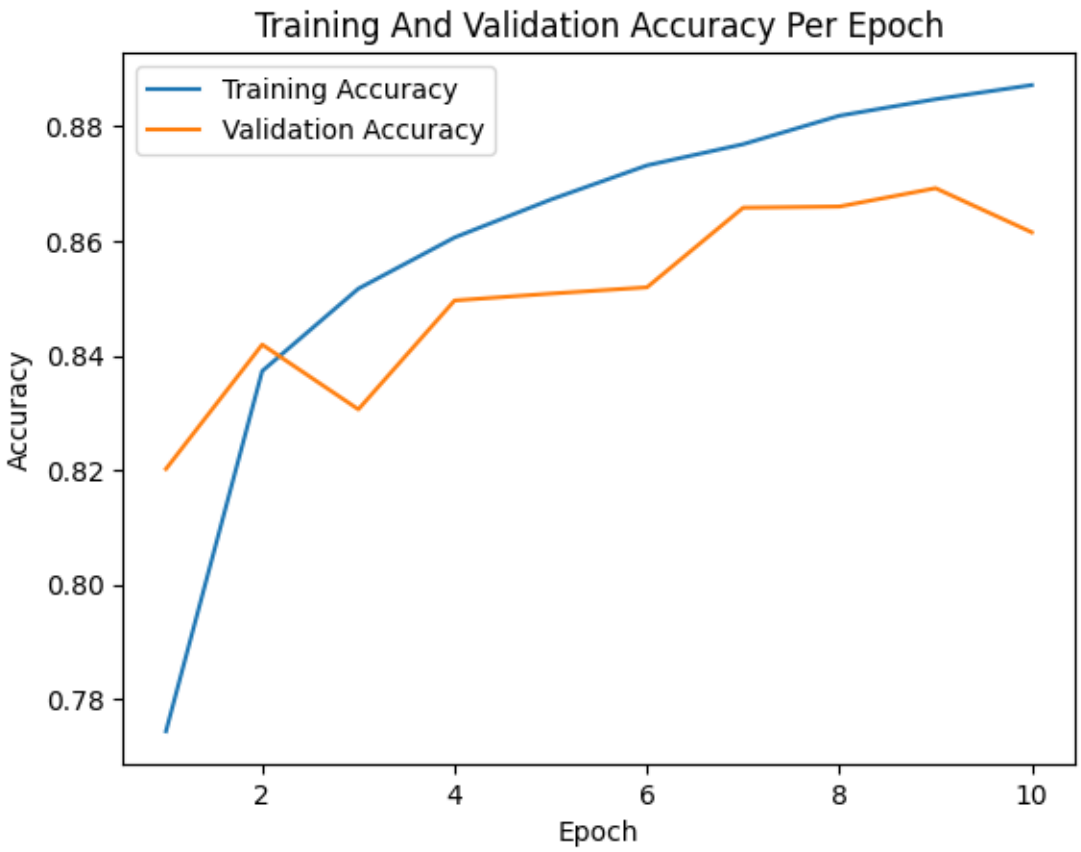
```
        total_correct += (predicted == y).sum().item()
        total_samples += y.size(0)

    train_losses.append(np.mean(temp_train_losses))
    train_accuracy.append(total_correct / total_samples)
    total_correct, total_samples = 0, 0   # Reinitialize for validation

    model.eval()
    with torch.no_grad():
        for x, y in valid_dataloader:
            x, y = x.to(device), y.to(device, dtype=torch.long)
            pred = model(x)
            loss = criterion(pred, y)
            temp_valid_losses.append(loss.item())
            # Getting Accuracy
            _, predicted = torch.max(pred, 1)
            total_correct += (predicted == y).sum().item()
            total_samples += y.size(0)

        valid_losses.append(np.mean(temp_valid_losses))
        valid_accuracy.append(total_correct / total_samples)

    print("Finished Epoch", epoch + 1, ", Average training loss:", np.
↪mean(temp_train_losses))
```

```
Using device cuda
Finished Epoch 1 , Average training loss: 0.665816730528449
Finished Epoch 2 , Average training loss: 0.4625976474747683
Finished Epoch 3 , Average training loss: 0.4204086249045407
Finished Epoch 4 , Average training loss: 0.3933246583839258
Finished Epoch 5 , Average training loss: 0.372997451858595
Finished Epoch 6 , Average training loss: 0.3571080660520432
Finished Epoch 7 , Average training loss: 0.34289144188394616
Finished Epoch 8 , Average training loss: 0.3317793440456347
Finished Epoch 9 , Average training loss: 0.32224096416152315
Finished Epoch 10 , Average training loss: 0.31336373695215053
```

[13]:
```
# Training loss and validation loss for each epoch after training for at least␣
↪8 epochs
plt.plot(range(1, epochs+1), train_losses, label='Training Loss')
plt.plot(range(1, epochs+1), valid_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.title('Training Loss and Validation Loss Per Epoch')
plt.legend()
plt.show()
```

```
# A plot of the training and validation accuracy, at least 82% for validation␣
 ↪by the end of training.
plt.plot(range(1, epochs+1), train_accuracy, label='Training Accuracy')
plt.plot(range(1, epochs+1), valid_accuracy, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training And Validation Accuracy Per Epoch')
plt.legend()
plt.show()

print(f"Training Accuracy: {np.mean(train_accuracy)}, Validation Accuracy: {np.
 ↪mean(valid_accuracy)}")
```






```
Training Accuracy: 0.8595033333333333, Validation Accuracy: 0.85075
```

[14]:
```
# Visualize data
images = [training_data[i][0] for i in range(200)]
plt.imshow(torchvision.utils.make_grid(torch.stack(images), nrow=10, padding=5).
 ↪numpy().transpose((1, 2, 0)))
# plt.figure(figsize=(15, 15))
plt.axis('off')
plt.show()
```

4 Task 2: CNNs for CIFAR-10

In this section, you will create a CNN for the CIFAR dataset, and submit your predictions to Kaggle. It is recommended that you use GPU acceleration for this part.

Here are some of the components you should consider using: 1. `nn.Conv2d` 2. `nn.ReLU` 3. `nn.Linear` 3. `nn.CrossEntropyLoss`: if you choose to use `nn.CrossEntropyLoss` or `F.cross_entropy`, DO NOT add an explicit softmax layer in your neural network. PyTorch devs found it more numerically stable to combine softmax and cross entropy loss into a single module and if you explicitly attach a softmax layer at the end of your model, you would unintentionally be applying it twice, which can degrade performance. 5. `nn.MaxPooling2d` (though many implementations without it exist; for example, you can also do strided convolutions instead of a pooling layer!)

We encourage you to explore different ways of improving your model to get higher accuracy. Here are some suggestions for things to look into: 1. CNN architectures: AlexNet, VGG, ResNets, etc. 2. Different optimizers and their parameters (see torch.optim) 3. Image preprocessing / data augmentation (see torchvision.transforms) 4. Regularization or dropout (see torch.optim and torch.nn respectively) 5. Learning rate scheduling: https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate 6. Weight initialization: https://pytorch.org/docs/stable/nn.init.html

Though we encourage you to explore, there are some rules: 1. You are not allowed to install or use packages not included by default in the Colab Environment. 2. You are not allowed to use any pre-defined architectures or feature extractors in your network. 3. You are not allowed to use **any**

pretrained weights, ie no transfer learning. 4. You cannot train on the test data.

Otherwise everything is fair game.

Your deliverables are as follows: 1. Submit to Kaggle and include your test accuracy in your report. 2. Provide at least (1) training curve for your model, depicting loss per epoch or step after training for at least 8 epochs. 3. Explain the components of your final model, and how you think your design choices contributed to it's performance.

After you write your code, we have included skeleton code that should be used to submit predictions to Kaggle. **You must follow the instructions below under the submission header**. Note that if you apply any processing or transformations to the data, you will need to do the same to the test data otherwise you will likely achieve very low accuracy.

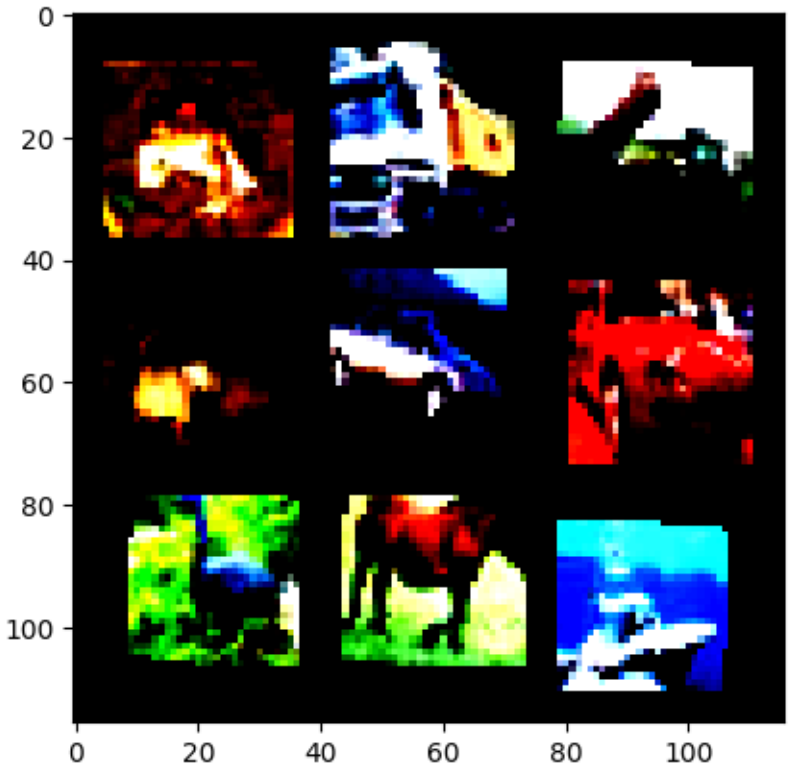
It is expected that this task will take a while to train. Our simple solution achieves a training accuracy of 90.2% and a test accuracy of 74.8% after 10 epochs (be careful of overfitting!). This easily beats the best SVM based CIFAR10 model submitted to the HW 1 Kaggle! It is possible to achieve 95% or higher test accuracy on CIFAR 10 with good model design and tuning.

[15]:
```
# Creating the datasets, feel free to change this as long as you do the same to␣
 ↪the test data.
# You can also modify this to split the data into training and validation.
# See https://pytorch.org/docs/stable/data.html#torch.utils.data.random_split
train_transforms = transforms.Compose([
    transforms.RandomHorizontalFlip(),  # Randomly flip the images on the␣
 ↪horizontal axis
    transforms.RandomRotation(10),  # Randomly rotate the images by a few␣
 ↪degrees
    transforms.RandomCrop(32, padding=4),  # Apply random crops with padding if␣
 ↪necessary
    transforms.ToTensor(),  # Convert images to PyTorch tensors
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)), ␣
 ↪# Normalization
])

training_data = torchvision.datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=train_transforms, # Change back to transform
)
# If you make a train-test partition it is up to you.
# Splitting data into training and validation
training_size, validation_size = len(training_data) - int(0.2 *␣
 ↪len(training_data)), int(0.2 * len(training_data))
train_dataset, valid_dataset = random_split(training_data, [training_size,␣
 ↪validation_size])
train_loader = DataLoader(train_dataset, batch_size=10, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=10, shuffle=False)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
data/cifar-10-python.tar.gz

100%|      | 170498071/170498071 [00:08<00:00, 19857940.60it/s]

Extracting data/cifar-10-python.tar.gz to data
```

Again, let's first visualize our data.

[17]:
```
images = [training_data[i][0] for i in range(9)]
plt.imshow(torchvision.utils.make_grid(torch.stack(images), nrow=3, padding=5).
  ↪numpy().transpose((1, 2, 0)));
```



[18]:
```
### YOUR CODE HERE ###

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

epochs = 60
batch_size = 32
learning_rate = 0.01

class CNN(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        # Adjust the kernel size and stride for CIFAR-10's 32x32 images
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))  # Adjust the size to match␣
 ↪the output from your features
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),  # Adjust the linear layer input size␣
 ↪to match the flattened feature map
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 10),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

# Calling Model
model = CNN().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=5,␣
 ↪verbose=True)
criterion = torch.nn.CrossEntropyLoss()

# Creating lists
train_losses, valid_losses = [], []
```

```
train_accuracy, valid_accuracy = [], []

for epoch in range(epochs):
    model.train()
    temp_train_losses, temp_valid_losses = [], []
    total_correct, total_samples = 0, 0

    # Training losses
    for x, y in train_loader:
        x, y = x.to(device), y.to(device, dtype=torch.long)
        optimizer.zero_grad() # Remove the gradients from the previous step
        pred = model(x)
        loss = criterion(pred, y)
        loss.backward()
        optimizer.step()
        temp_train_losses.append(loss.item())
        # Getting Accuracy
        _, predicted = torch.max(pred, 1)
        total_correct += (predicted == y).sum().item()
        total_samples += y.size(0)

    train_losses.append(np.mean(temp_train_losses))
    train_accuracy.append(total_correct / total_samples)
    total_correct, total_samples = 0, 0

    # Validation losses
    model.eval()
    with torch.no_grad():
        for x, y in valid_loader:
            x, y = x.to(device), y.to(device, dtype=torch.long)
            pred = model(x)
            loss = criterion(pred, y)
            temp_valid_losses.append(loss.item())
            # Getting Accuracy
            _, predicted = torch.max(pred, 1)
            total_correct += (predicted == y).sum().item()
            total_samples += y.size(0)

        valid_losses.append(np.mean(temp_valid_losses))
        scheduler.step(np.mean(temp_valid_losses))
        valid_accuracy.append(total_correct / total_samples)
    print(f"Finished Epoch {epoch + 1}, Average training loss: {np.
↪mean(temp_train_losses)}, Average valid loss: {np.mean(temp_valid_losses)}")
```

```
Using device cuda
Finished Epoch 1, Average training loss: 2.1167493799626826, Average valid loss:
1.8021416838169098
Finished Epoch 2, Average training loss: 1.6904813578724862, Average valid loss:
```

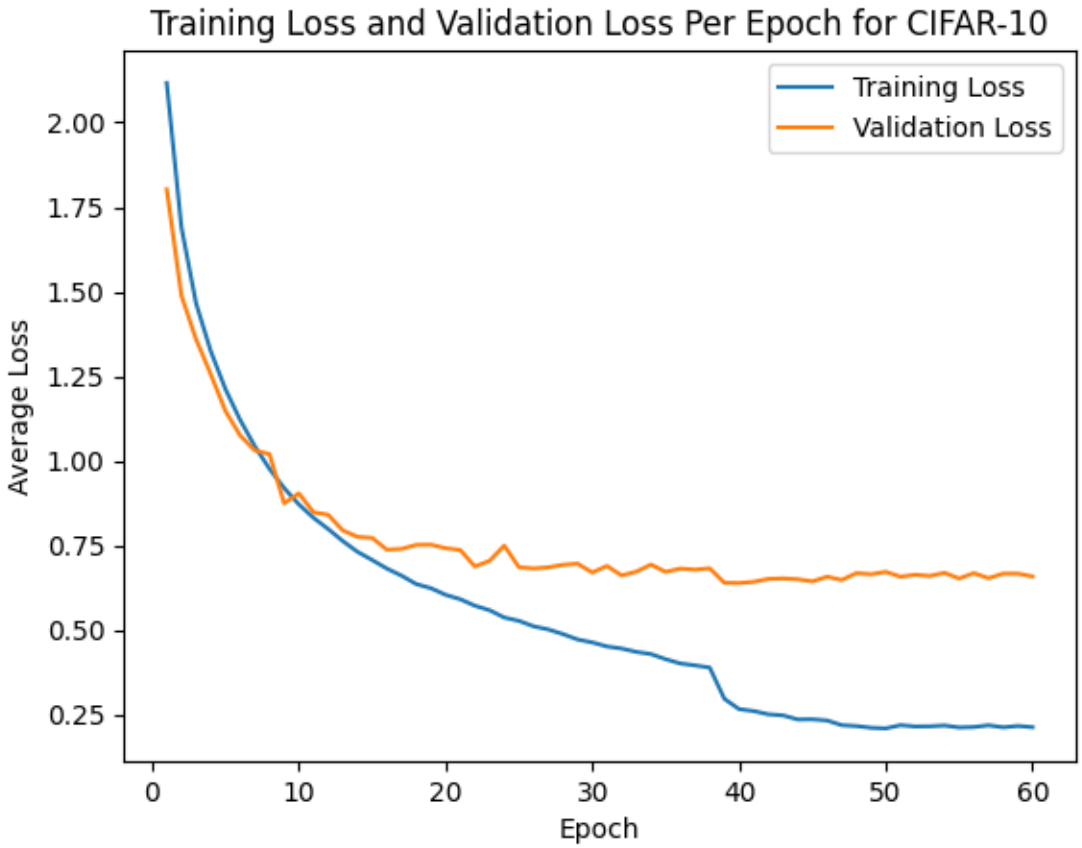
```
1.488403814136982
Finished Epoch 3, Average training loss: 1.4652771964520217, Average valid loss:
1.3587973297834397
Finished Epoch 4, Average training loss: 1.3239443057104945, Average valid loss:
1.2554715547561646
Finished Epoch 5, Average training loss: 1.2117020219266414, Average valid loss:
1.146851135134697
Finished Epoch 6, Average training loss: 1.1209880025684833, Average valid loss:
1.0748824205547571
Finished Epoch 7, Average training loss: 1.0430113842226565, Average valid loss:
1.03034707942605
Finished Epoch 8, Average training loss: 0.9759837185330689, Average valid loss:
1.0196943799704312
Finished Epoch 9, Average training loss: 0.9192355954907835, Average valid loss:
0.874029339119792
Finished Epoch 10, Average training loss: 0.8711309455260634, Average valid
loss: 0.9034474988877773
Finished Epoch 11, Average training loss: 0.831389375815168, Average valid loss:
0.8477350120097399
Finished Epoch 12, Average training loss: 0.7980792221669107, Average valid
loss: 0.8401679526045919
Finished Epoch 13, Average training loss: 0.7624690422853455, Average valid
loss: 0.7939331012219191
Finished Epoch 14, Average training loss: 0.7306933017680421, Average valid
loss: 0.7750410249456763
Finished Epoch 15, Average training loss: 0.7065341688450426, Average valid
loss: 0.7718873246610165
Finished Epoch 16, Average training loss: 0.6812841817457229, Average valid
loss: 0.7371248443387449
Finished Epoch 17, Average training loss: 0.6602414195947349, Average valid
loss: 0.7392995529919862
Finished Epoch 18, Average training loss: 0.6354171265745535, Average valid
loss: 0.7512505215071141
Finished Epoch 19, Average training loss: 0.623220595546998, Average valid loss:
0.7518345831483603
Finished Epoch 20, Average training loss: 0.603641339135822, Average valid loss:
0.7414531036596745
Finished Epoch 21, Average training loss: 0.5906179866022431, Average valid
loss: 0.7361609069779516
Finished Epoch 22, Average training loss: 0.5713978059522342, Average valid
loss: 0.6873922514393925
Finished Epoch 23, Average training loss: 0.5580771409124136, Average valid
loss: 0.7040844932049513
Finished Epoch 24, Average training loss: 0.5365835989987244, Average valid
loss: 0.7490902888663113
Finished Epoch 25, Average training loss: 0.5266263516750187, Average valid
loss: 0.6850639266925864
Finished Epoch 26, Average training loss: 0.510546879976173, Average valid loss:
```

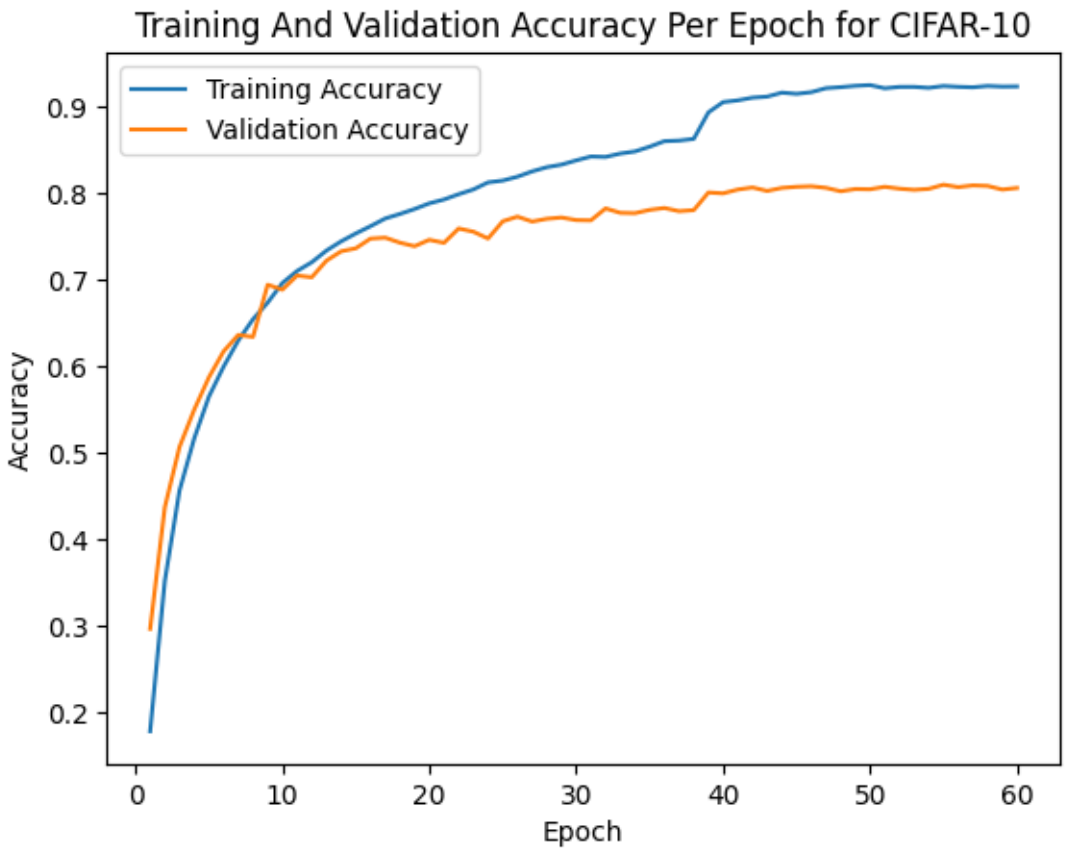
```
0.6815994773060083
Finished Epoch 27, Average training loss: 0.501288727789768, Average valid loss:
0.684927238078788
Finished Epoch 28, Average training loss: 0.48750166307506154, Average valid
loss: 0.692493095131591
Finished Epoch 29, Average training loss: 0.4715903712763102, Average valid
loss: 0.6957510780598968
Finished Epoch 30, Average training loss: 0.46288891953742134, Average valid
loss: 0.6695319869089872
Finished Epoch 31, Average training loss: 0.4509182498527225, Average valid
loss: 0.6890601680018008
Finished Epoch 32, Average training loss: 0.4448000537812477, Average valid
loss: 0.6607534160166979
Finished Epoch 33, Average training loss: 0.4348776830562856, Average valid
loss: 0.6721475844457746
Finished Epoch 34, Average training loss: 0.42841653134557417, Average valid
loss: 0.6931679530721158
Finished Epoch 35, Average training loss: 0.4133178718680283, Average valid
loss: 0.6715084419595078
Finished Epoch 36, Average training loss: 0.4006521639626007, Average valid
loss: 0.6812201223280281
Finished Epoch 37, Average training loss: 0.39503612016979606, Average valid
loss: 0.6779856542679481
Epoch 00038: reducing learning rate of group 0 to 1.0000e-03.
Finished Epoch 38, Average training loss: 0.38855465706973336, Average valid
loss: 0.6821171792391688
Finished Epoch 39, Average training loss: 0.2961619126741425, Average valid
loss: 0.6391129438928328
Finished Epoch 40, Average training loss: 0.26578528865558837, Average valid
loss: 0.6383481032229029
Finished Epoch 41, Average training loss: 0.2600459852336207, Average valid
loss: 0.6420863997964188
Finished Epoch 42, Average training loss: 0.25040285844918253, Average valid
loss: 0.6506094544190564
Finished Epoch 43, Average training loss: 0.24719714974920498, Average valid
loss: 0.6521082925023511
Finished Epoch 44, Average training loss: 0.23519223967198924, Average valid
loss: 0.6495956502896734
Finished Epoch 45, Average training loss: 0.23562146744251367, Average valid
loss: 0.6432176719680428
Epoch 00046: reducing learning rate of group 0 to 1.0000e-04.
Finished Epoch 46, Average training loss: 0.23207716440524018, Average valid
loss: 0.6571913397684693
Finished Epoch 47, Average training loss: 0.2182167213851062, Average valid
loss: 0.6474535501569044
Finished Epoch 48, Average training loss: 0.21569674978626427, Average valid
loss: 0.6676794640365988
Finished Epoch 49, Average training loss: 0.2105855700594402, Average valid
```

```
loss: 0.6645344727740158
Finished Epoch 50, Average training loss: 0.20882732205304638, Average valid
loss: 0.6715388385532424
Finished Epoch 51, Average training loss: 0.21830102526248082, Average valid
loss: 0.6571079166358104
Epoch 00052: reducing learning rate of group 0 to 1.0000e-05.
Finished Epoch 52, Average training loss: 0.214512758103112, Average valid loss:
0.6629832511027344
Finished Epoch 53, Average training loss: 0.21493277571812358, Average valid
loss: 0.6593975623017176
Finished Epoch 54, Average training loss: 0.21730226050050988, Average valid
loss: 0.6686293336167001
Finished Epoch 55, Average training loss: 0.21153005871568167, Average valid
loss: 0.6518197072822659
Finished Epoch 56, Average training loss: 0.21279858248739036, Average valid
loss: 0.6677674210080877
Finished Epoch 57, Average training loss: 0.21795684534437895, Average valid
loss: 0.6528108230223879
Epoch 00058: reducing learning rate of group 0 to 1.0000e-06.
Finished Epoch 58, Average training loss: 0.21195361245371896, Average valid
loss: 0.6662958380293567
Finished Epoch 59, Average training loss: 0.21599894158061214, Average valid
loss: 0.666201385260094
Finished Epoch 60, Average training loss: 0.21236269771160732, Average valid
loss: 0.6580506723943399
```

[19]:
```
# Training loss and validation loss for each epoch after training for at least␣
 ↪8 epochs
plt.plot(range(1, epochs+1), train_losses, label='Training Loss')
plt.plot(range(1, epochs+1), valid_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.title('Training Loss and Validation Loss Per Epoch for CIFAR-10')
plt.legend()
plt.show()

# A plot of the training and validation accuracy, at least 82% for validation␣
 ↪by the end of training.
plt.plot(range(1, epochs+1), train_accuracy, label='Training Accuracy')
plt.plot(range(1, epochs+1), valid_accuracy, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training And Validation Accuracy Per Epoch for CIFAR-10')
plt.legend()
plt.show()
```

```
print(f"Training Accuracy: {np.mean(train_accuracy)}, Validation Accuracy: {np.
  ↪mean(valid_accuracy)}")
```






```
Training Accuracy: 0.8009991666666666, Validation Accuracy: 0.7426483333333334
```

4.0.1 Kaggle Submission

The following code is for you to make your submission to kaggle. Here are the steps you must follow:

1. Upload `cifar_test_data_sp24.npy` to the colab notebook by going to files on the left hand pane, then hitting "upload". This file may take roughly a minute to upload and you should not proceed to the following steps until it has completely finished uploading (students in the past have run into issues where they were accidentally testing on a partially uploaded test set and getting garbage results).
2. Run the following cell to generate the dataset object for the test data. Feel free to modify the code to use the same transforms that you use for the training data. By default, this will re-use the `transform` variable.
3. In the second cell, write code to run predictions on the testing dataset and store them into an array called `predictions`.
4. Run the final cell which will convert your predictions array into a CSV for kaggle.
5. Go to the files pane again, and download the file called `submission.csv` by clicking the three dots and then download.

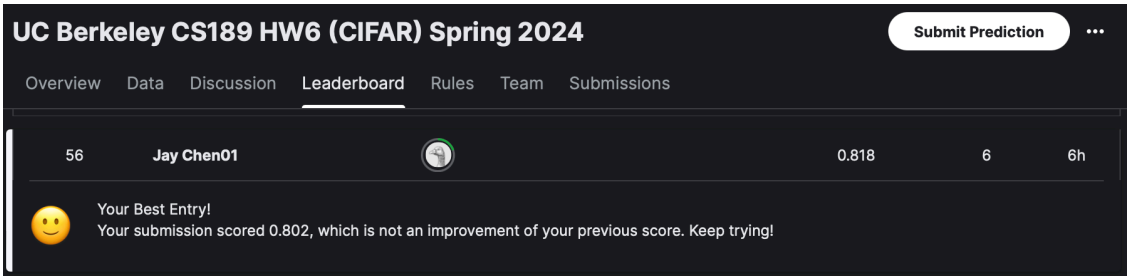
[21]:
```
from PIL import Image
import os

class CIFAR10Test(torchvision.datasets.VisionDataset):

    def __init__(self, transform=None, target_transform=None):
        super(CIFAR10Test, self).__init__(None, transform=transform,
                                          target_transform=target_transform)
        assert os.path.exists("/kaggle/input/cifar10-test-data-sp24-npy/
 ↪cifar10_test_data_sp24.npy"), "You must upload the test data to the file␣
 ↪system."
        self.data = [np.load("/kaggle/input/cifar10-test-data-sp24-npy/
 ↪cifar10_test_data_sp24.npy", allow_pickle=False)]

        self.data = np.vstack(self.data).reshape(-1, 3, 32, 32)
        self.data = self.data.transpose((0, 2, 3, 1))  # convert to HWC

    def __getitem__(self, index: int):
        img = self.data[index]
        img = Image.fromarray(img)
        if self.transform is not None:
            img = self.transform(img)
        return img

    def __len__(self) -> int:
        return len(self.data)

# Create the test dataset
testing_data = CIFAR10Test(
    transform=train_transforms, # NOTE: Make sure transform is the same as used␣
 ↪in the training dataset.
)
```

[22]:
```
### YOUR CODE HERE ###
model.eval()
test_loader = DataLoader(testing_data, batch_size=batch_size, shuffle=False)
# Store a numpy vector of the predictions for the test set in the variable␣
 ↪`predictions`.
predictions = []
with torch.no_grad():
    for data in test_loader:
        images = data.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        predictions.extend(predicted.cpu().numpy())
predictions_array = np.array(predictions)
```

[23]:
```
# This code below will generate kaggle_predictions.csv file. Please download it␣
 ↪and submit to kaggle.
import pandas as pd

if isinstance(predictions, np.ndarray):
    predictions = predictions.astype(int)
else:
    predictions = np.array(predictions, dtype=int)
assert predictions.shape == (len(testing_data),), "Predictions were not the␣
 ↪correct shape"
df = pd.DataFrame({'Category': predictions})
df.index += 1  # Ensures that the index starts at 1.
df.to_csv('submission.csv', index_label='Id')

# Now download the submission.csv file to submit.
```

Congrats! You made it to the end.

Model description: Before using the training data, I first normalized the data and performed some data augmentation then split it into training data and validation data by 80% and 20%. In my model, I got inspiration from the AlexNet with ReduceLROnPlateau scheduler. I first used five Convolution Layers as my features layers, each layer with ReLu as activation function and a maxpooling layer to change the size of the images. Then an average pooling layer to adjust the size to match the output from the features, and dropout, flatten the spatial dimensions to match the flattened feature map, and I applied the ReLu function and linear layer again, finally use linear layer to output the size to be 10, which is the output classes. The ReduceLROnPlateau also contributes to the performance, it takes the validation loss as reference, I set the patience to be 5, so if 5 epoches the validation losses are not decreasing, it will change the learning rate to be smaller.



Kaggle Score