CS 189 HW 6: 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.

```
# Imports for pytorch
import numpy as np
import torch
import torchvision
from torch import nn
import matplotlib
from matplotlib import pyplot as plt
from tqdm import tqdm_notebook as tqdm
np.random.seed(42)
```

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

Tensors

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

```
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([[0.3454, 0.2367, 0.7504],
            [0.8490, 0.6294, 0.5329]])

    Rand Numpy Array:
     [[0.34537917 0.23667234 0.75038326]
     [0.8489916  0.6294049  0.5328552 ]]
```

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

```
# 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
```

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`.

```
# Below is an example of computing the gradient for a single data point in logis

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([[0.7463],
        [0.7463],
        [0.7463],
        [0.7463],
        [0.7463]])
b gradient: tensor([0.7463])
Weight before tensor([[-0.7105],
        [ 0.6276],
        [ 0.2203],
        [ 1.0222],
        [ 0.1328]], requires_grad=True)
Updated weight tensor([[-0.7851],
        [ 0.5530],
        [ 0.1457],
        [ 0.9476],
        [ 0.0582]], requires_grad=True)
```

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:

```
# 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 computation
tensor = tensor.detach().cpu()

tensor_np = tensor.numpy() # Convert to numpy if you want to perform numpy opera
```

```
    Using device cuda
```

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 `foward(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:

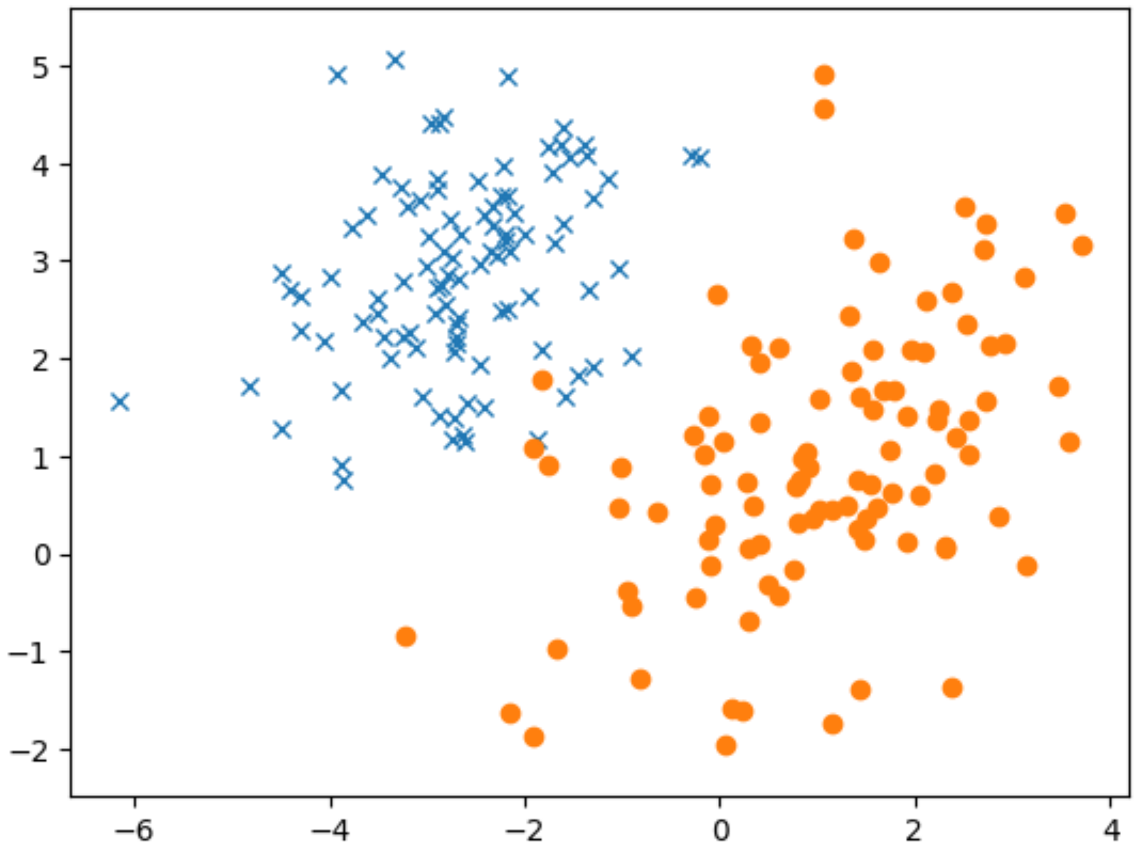
```
class Perceptron(nn.Module):
  def __init__(self, in_dim):
    super().__init__()
    self.layer = nn.Linear(in_dim, 1) # This is a linear layer, it computes Xw +

  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 de
print("Parameters", list(perceptron.parameters()))
```

```
Parameters [Parameter containing:
tensor([[ 0.2832,  0.0205, -0.2265, -0.0501,  0.3055,  0.3068,  0.2034, -0.
         -0.2492,  0.3078]], device='cuda:0', requires_grad=True), Paramete
tensor([-0.2507], device='cuda:0', requires_grad=True)]
```

Datasets

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

```
c1_x1, c1_x2 = np.random.multivariate_normal([-2.5,3], [[1, 0.3],[0.3, 1]], 100)
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.

```
dataset = torch.utils.data.TensorDataset(torch.from_numpy(train_X), torch.from_n
# 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([[-0.8116, -1.2819],
        [ 1.0644,  4.5527],
        [-4.4993,  2.8760],
        [-2.8201,  3.1017],
        [ 2.5499,  1.3728],
        [-3.1212,  2.1143],
        [-2.0018,  3.2738],
        [ 0.2796,  0.7454],
        [-2.3558,  3.0928],
        [-1.7590,  0.9154]], dtype=torch.float64)
Batch y: tensor([1., 1., 0., 0., 1., 0., 0., 1., 0., 1.], dtype=torch.float
```

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.

```
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

model.train() # Put model in training mode
for epoch in range(epochs):
    training_losses = []
    for x, y in 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_loss

# 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
<ipython-input-10-ebaddb73e382>:17: TqdmDeprecationWarning: This function w
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for x, y in tqdm(dataloader, unit="batch"):
100%                                        20/20 [00:02<00:00, 2.97s/batch]
Finished Epoch 1 , training loss: 0.829820241034031
100%                                        20/20 [00:00<00:00, 332.81batch/s]
Finished Epoch 2 , training loss: 0.7282662227749824
100%                                        20/20 [00:00<00:00, 343.77batch/s]
Finished Epoch 3 , training loss: 0.6430073074996472
100%                                        20/20 [00:00<00:00, 447.10batch/s]
Finished Epoch 4 , training loss: 0.5725147500634193
100%                                        20/20 [00:00<00:00, 449.74batch/s]
Finished Epoch 5 , training loss: 0.5164606586098671
100%                                        20/20 [00:00<00:00, 410.67batch/s]
Finished Epoch 6 , training loss: 0.47171367332339287
100%                                        20/20 [00:00<00:00, 494.25batch/s]
Finished Epoch 7 , training loss: 0.436161857098341
100%                                        20/20 [00:00<00:00, 494.82batch/s]
Finished Epoch 8 , training loss: 0.4081315457820892
100%                                        20/20 [00:00<00:00, 498.88batch/s]
Finished Epoch 9 , training loss: 0.3856899328529835
100%                                        20/20 [00:00<00:00, 438.06batch/s]
Finished Epoch 10 , training loss: 0.3673427142202854
Final Accuracy: 0.84
```

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 definetily use:

1. `nn.Linear`
2. Some activation: `nn.ReLU`, `nn.Tanh`, `nn.Sigmoid`, etc.
3. `nn.CrossEntropyLoss`

Here are challenges you will need to overcome:

1. The data is default configured in image form ie (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.

```
# Creating the datasets
transform = torchvision.transforms.ToTensor() # feel free to modify this as you

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,
)
```

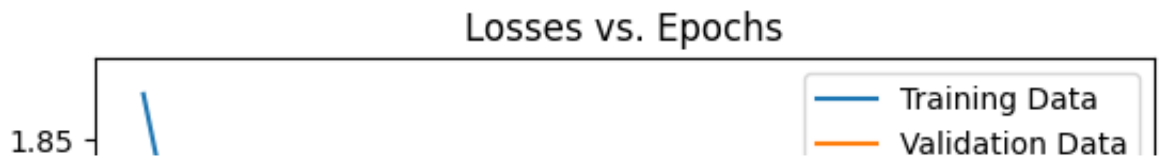
```
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
    100%|██████████| 26421880/26421880 [00:01<00:00, 14859590.17it/s]
    Extracting data/FashionMNIST/raw/train-images-idx3-ubyte.gz to data/Fashion

    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/trai
    100%|██████████| 29515/29515 [00:00<00:00, 269084.21it/s]
    Extracting data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to data/Fashion

    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
    100%|██████████| 4422102/4422102 [00:00<00:00, 4968666.66it/s]
    Extracting data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to data/FashionM

    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k
    100%|██████████| 5148/5148 [00:00<00:00, 14501193.41it/s]Extracting data/Fa
```

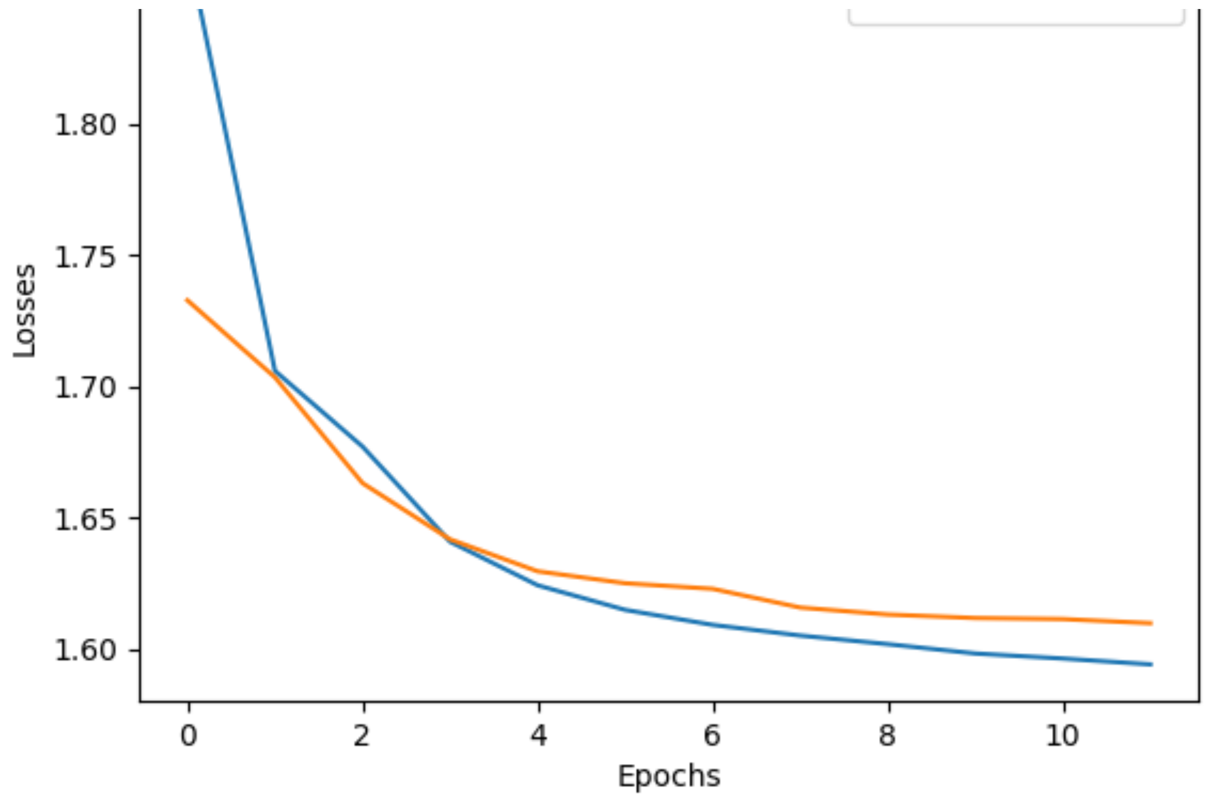
```
### YOUR CODE HERE ###
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

epochs = 12
batch_size = 20
learning_rate = 0.00005

trainset = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shu
valset = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, shu
class MultiLayerPerceptron(nn.Module):
  def __init__(self):
    super().__init__()
    self.fc0 = nn.Linear(28*28, 512)
    self.fc1 = nn.Linear(512, 256)
    self.fc2 = nn.Linear(256, 128)
    self.fc3 = nn.Linear(128, 64)
    self.fc4 = nn.Linear(64, 32)
    self.fc5 = nn.Linear(32, 10)
    self.softmax = nn.Softmax()
    self.relu = nn.ReLU()
    self.tanh = nn.Tanh()


  def forward(self, x):
    x = self.fc0(x)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    x = self.fc3(x)
    x = self.tanh(x)
    x = self.fc4(x)
    x = self.fc5(x)
    return self.softmax(x)

mlp = MultiLayerPerceptron().to(device)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp.parameters(), lr=learning_rate)
train_losses, val_losses, train_acc, val_acc = [], [], [], []
mlp.train()
for epoch in range(epochs):
  num_correct_train = 0
  temp_train_loss = []
  for X, y in tqdm(trainset, unit="batch"):
    X, y = X.reshape(batch_size, -1).to(device), y.to(device)

    optimizer.zero_grad()
    pred = mlp(X)
    loss = loss_fn(pred, y)
    loss.backward()
    optimizer.step()

    temp_train_loss.append(loss.item())
    num_correct_train += torch.sum(torch.argmax(pred, axis=1) == y).item()

  train_acc.append(num_correct_train/len(trainset.dataset))
  print(num_correct_train/len(trainset.dataset))
```

```
    train_losses.append(np.mean(temp_train_loss))
    print(np.mean(temp_train_loss))

    with torch.no_grad():
      mlp.eval()
      num_correct_val = 0
      temp_val_loss = []
      for X, y in tqdm(valset, unit="batch"):
        X, y = X.reshape(batch_size, -1).to(device), y.to(device)

        optimizer.zero_grad()
        pred = mlp(X)
        loss = loss_fn(pred, y)

        temp_val_loss.append(loss.item())
        num_correct_val += torch.sum(torch.argmax(pred, axis=1) == y).item()

      val_acc.append(num_correct_val/len(valset.dataset))
      print(num_correct_val/len(valset.dataset))

      val_losses.append(np.mean(temp_val_loss))
      print(np.mean(temp_val_loss))

      mlp.train()

  epochs = np.arange(0, 12)

  fig = plt.figure()
  fig, ax = plt.subplots()
  ax.plot(epochs, train_losses, label='Training Data')
  ax.plot(epochs, val_losses, label='Validation Data')
  ax.set_xlabel('Epochs')
  ax.set_ylabel('Losses')
  ax.set_title("Losses vs. Epochs")
  ax.legend()

  fig = plt.figure()
  fig, ax = plt.subplots()
  ax.plot(epochs, train_acc, label='Training Data')
  ax.plot(epochs, val_acc, label='Validation Data')
  ax.set_xlabel('Epochs')
  ax.set_ylabel('Accuracy')
  ax.set_title("Accuracy vs. Epochs")
  ax.legend()

  del trainset, valset
```

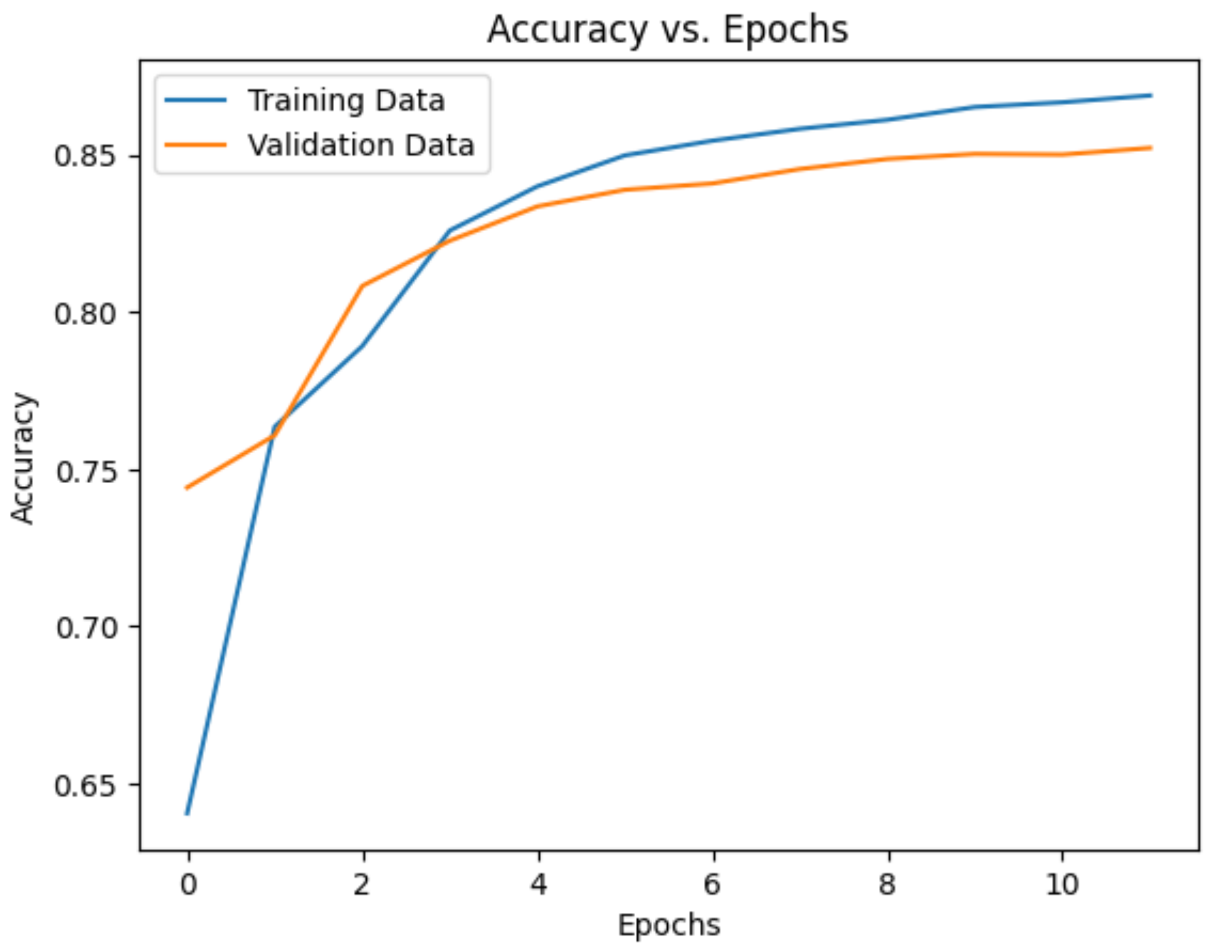
```
Using device cuda
<ipython-input-13-03358462ebc8>:46: TqdmDeprecationWarning: This function w
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for X, y in tqdm(trainset, unit="batch"):
100%                                        3000/3000 [00:16<00:00, 195.89batch/s]
<ipython-input-13-03358462ebc8>:35: UserWarning: Implicit dimension choice
  return self.softmax(x)
0.6406
1.8672892528772354
<ipython-input-13-03358462ebc8>:68: TqdmDeprecationWarning: This function w
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for X, y in tqdm(valset, unit="batch"):
100%                                        500/500 [00:01<00:00, 273.25batch/s]
0.7442
1.73268709731102
100%                                        3000/3000 [00:17<00:00, 140.29batch/s]
0.7635333333333333
1.7058043270905812
100%                                        500/500 [00:02<00:00, 268.30batch/s]
0.7608
1.7033586783409118
100%                                        3000/3000 [00:16<00:00, 190.10batch/s]
0.7891666666666667
1.6769855335553487
100%                                        500/500 [00:01<00:00, 270.66batch/s]
0.8083
1.6630635967254639
100%                                        3000/3000 [00:17<00:00, 137.29batch/s]
0.8259333333333333
1.640749245762825
100%                                        500/500 [00:01<00:00, 279.98batch/s]
0.8227
1.6416351935863496
100%                                        3000/3000 [00:17<00:00, 192.44batch/s]
0.8399833333333333
1.6241501403649647
100%                                        500/500 [00:01<00:00, 277.05batch/s]
0.8336
1.6294692795276642
100%                                        3000/3000 [00:16<00:00, 164.91batch/s]
0.8498333333333333
1.6147762753566106
100%                                        500/500 [00:01<00:00, 273.76batch/s]
```

```
0.8389
1.6249152736663819
100%                                    3000/3000 [00:16<00:00, 194.80batch/s]
0.8545
1.6090552588303884
100%                                    500/500 [00:01<00:00, 277.54batch/s]
0.8409
1.622792547941208
100%                                    3000/3000 [00:16<00:00, 171.89batch/s]
0.8582833333333333
1.6049909527699153
100%                                    500/500 [00:01<00:00, 280.09batch/s]
0.8455
1.6156570312976837
100%                                    3000/3000 [00:16<00:00, 190.20batch/s]
0.86115
1.6017163661321003
100%                                    500/500 [00:01<00:00, 274.09batch/s]
0.8487
1.6130009739398956
100%                                    3000/3000 [00:16<00:00, 144.68batch/s]
0.8652166666666666
1.5981558102766673
100%                                    500/500 [00:01<00:00, 279.08batch/s]
0.8503
1.6116662323474884
100%                                    3000/3000 [00:16<00:00, 193.92batch/s]
0.8667166666666667
1.5962376996278762
100%                                    500/500 [00:01<00:00, 281.51batch/s]
0.8501
1.6112662649154663
100%                                    3000/3000 [00:16<00:00, 147.18batch/s]
0.8688833333333333
1.594017529606819
100%                                    500/500 [00:02<00:00, 284.75batch/s]
0.8522
1.609707558631897
<Figure size 640x480 with 0 Axes>
```





```
<Figure size 640x480 with 0 Axes>
```



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`
4. `nn.CrossEntropyLoss`
5. `nn.MaxPooling2d` (though many implementations without it exist)

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.

```
# Creating the datasets, feel free to change this as long as you do the same to
# 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
# Optimal Normalization values found: https://github.com/kuangliu/pytorch-cifar/
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                            torchvision.transforms.Normalize((0.
                                            ])

training_data = torchvision.datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=transform,
)
# If you make a train-test partition it is up to you.
```

```
    Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to data
    100%|██████████| 170498071/170498071 [00:01<00:00, 96187080.56it/s]
    Extracting data/cifar-10-python.tar.gz to data
```

```
### YOUR CODE HERE ###
# Determine the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device", device)

# Creating the ResNet-like class: https://blog.roboflow.com/custom-resnet34-clas
class ResCifarNet(nn.Module):
  def __init__(self):
     super(ResCifarNet, self).__init__()

     self.prep = nn.Sequential(
         nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1),
         nn.BatchNorm2d(64),
         nn.ReLU(inplace=True)  # Apparently saves memory with inplace=True
     )

     self.layer1 = nn.Sequential(
         nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
         nn.BatchNorm2d(128),
         nn.ReLU(inplace=True),
         nn.MaxPool2d(kernel_size=2, stride=2)
     )

     self.res_layer1 = nn.Sequential(nn.Sequential(
         nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
         nn.BatchNorm2d(128),
         nn.ReLU(inplace=True)
     ), nn.Sequential(
         nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
         nn.BatchNorm2d(128),
         nn.ReLU(inplace=True)
     ))

     self.layer2 = nn.Sequential(
         nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
         nn.BatchNorm2d(256),
         nn.ReLU(inplace=True),
         nn.MaxPool2d(kernel_size=2, stride=2)
     )

     self.layer3 = nn.Sequential(
         nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1),
         nn.BatchNorm2d(512),
         nn.ReLU(inplace=True),
         nn.MaxPool2d(kernel_size=2, stride=2)
     )

     self.res_layer2 = nn.Sequential(nn.Sequential(
         nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1),
         nn.BatchNorm2d(512),
         nn.ReLU(inplace=True)
     ), nn.Sequential(
         nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1),
         nn.BatchNorm2d(512),
         nn.ReLU(inplace=True)
     ))

     self.final_layer = nn.Sequential(
         nn.MaxPool2d(kernel_size=4, stride=1),
         nn.Flatten(),
         nn.Linear(512, 10)
     )

  def forward(self, x):
    x = self.prep(x)
    x = self.layer1(x)
    x = x + self.res_layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = x + self.res_layer2(x)

    return self.final_layer(x)
```

```
    Using device cuda
```

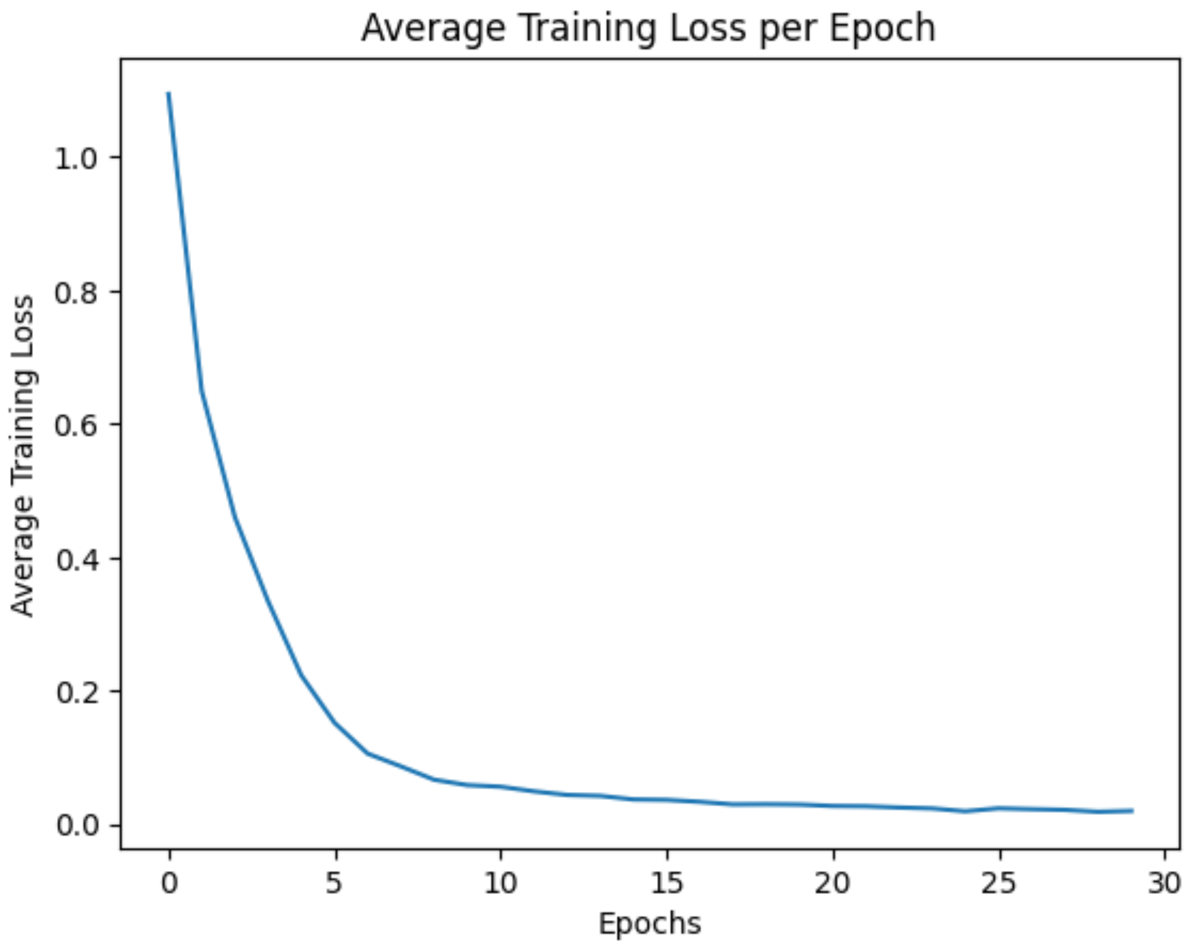
```
# Training and Validating

# Defining variables and functions
net = ResCifarNet().to(device)

epochs = 30
batch_size = 10
learning_rate = 0.0005

trainset = torch.utils.data.DataLoader(training_data, batch_size=batch_size, shu

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
train_losses, train_acc = [], []

# Training
net.train()
for epoch in range(epochs):
  num_correct_train = 0
  temp_train_loss = []
  for X, y in tqdm(trainset, unit="batch"):
    X, y = X.to(device), y.to(device)

    optimizer.zero_grad()
    pred = net(X)
    loss = loss_fn(pred, y)
    loss.backward()
    optimizer.step()

    temp_train_loss.append(loss.item())
    predicted = torch.max(pred.data, axis=1)[1]
    num_correct_train += torch.sum(predicted == y).item()
  train_acc.append(num_correct_train/len(trainset.dataset))
  print(num_correct_train/len(trainset.dataset))

  train_losses.append(np.mean(temp_train_loss))
  print(np.mean(temp_train_loss))

del trainset
```

```
<ipython-input-5-ddf3ce8e4c18>:21: TqdmDeprecationWarning: This function wi
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for X, y in tqdm(trainset, unit="batch"):
100%                                        5000/5000 [01:38<00:00, 56.73batch/s]
0.62192
1.094540664011985
100%                                        5000/5000 [01:29<00:00, 55.55batch/s]
0.77922
0.6496767253484577
100%                                        5000/5000 [01:30<00:00, 55.55batch/s]
0.84084
0.46028421789165586
100%                                        5000/5000 [01:30<00:00, 55.50batch/s]
0.884
0.3341211304541212
100%                                        5000/5000 [01:30<00:00, 52.04batch/s]
0.92272
0.2231818953202106
100%                                        5000/5000 [01:30<00:00, 55.30batch/s]
0.94558
0.15205413389765307
100%                                        5000/5000 [01:30<00:00, 55.18batch/s]
0.96332
0.1057848181589943
100%                                        5000/5000 [01:30<00:00, 56.21batch/s]
0.9694
0.08667257021702335
100%                                        5000/5000 [01:30<00:00, 55.40batch/s]
0.97734
0.06668489567760189
100%                                        5000/5000 [01:30<00:00, 55.90batch/s]
0.97948
```

```
0.058598030721214216
100%                                                    5000/5000 [01:29<00:00, 54.36batch/s]
0.98064
0.05617786200449209
100%                                                    5000/5000 [01:30<00:00, 56.10batch/s]
0.98296
0.04929934250312908
100%                                                    5000/5000 [01:30<00:00, 56.60batch/s]
0.98522
0.0438701272382146
100%                                                    5000/5000 [01:30<00:00, 55.07batch/s]
0.98556
0.04245555138742553
100%                                                    5000/5000 [01:30<00:00, 56.05batch/s]
0.98766
0.03708024321292195
100%                                                    5000/5000 [01:30<00:00, 55.92batch/s]
0.98766
0.0364342694128002
100%                                                    5000/5000 [01:30<00:00, 55.36batch/s]
0.98828
0.03355550362586989
100%                                                    5000/5000 [01:31<00:00, 54.47batch/s]
0.98966
0.029673836085324683
100%                                                    5000/5000 [01:31<00:00, 55.61batch/s]
0.99
0.029931770172837865
100%                                                    5000/5000 [01:31<00:00, 56.27batch/s]
0.98958
0.02947060309501994
100%                                                    5000/5000 [01:30<00:00, 52.74batch/s]
0.99084
0.02730869564818446
100%                                                    5000/5000 [01:29<00:00, 55.63batch/s]
0.99148
0.026881794117829993
100%                                                    5000/5000 [01:29<00:00, 56.34batch/s]
0.99142
0.025011738854086196
100%                                                    5000/5000 [01:29<00:00, 55.72batch/s]
0.99218
```

```
0.99218
0.023637847489125213
100%                                                    5000/5000 [01:29<00:00, 55.52batch/s]
0.99328
0.019360831464326118
100%                                                    5000/5000 [01:29<00:00, 55.93batch/s]
0.99218
0.02377100612174795
100%                                                    5000/5000 [01:29<00:00, 56.44batch/s]
0.99254
0.022502602382492687
100%                                                    5000/5000 [01:29<00:00, 54.36batch/s]
0.99294
0.02151561528763751
100%                                                    5000/5000 [01:29<00:00, 54.14batch/s]
0.99364
0.018586813293874904
100%                                                    5000/5000 [01:29<00:00, 54.68batch/s]
0.99362
0.019859557880283932
```

```
# Graphing Training Losses
epochs = np.arange(0, epochs)
plt.plot(epochs, train_losses)
plt.xlabel("Epochs")
plt.ylabel("Average Training Loss")
plt.title("Average Training Loss per Epoch")
plt.show()
```



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.npy` to the colab notebook by going to files on the right hand pane, then hitting "upload".
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.

```
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("cifar10_test_data_sp23.npy"), "You must upload th
        self.data = [np.load("cifar10_test_data_sp23.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
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                            torchvision.transforms.Normalize((0.
                                            ])
testing_data = CIFAR10Test(
    transform=transform, # NOTE: Make sure transform is the same as used in the
)
```

```
### YOUR CODE HERE ###
testset = torch.utils.data.DataLoader(testing_data, batch_size=batch_size, shuff

# Recommendation: create a `test_dataloader` from torch.utils.data.DataLoader wi
test_preds = []
with torch.no_grad():
  net.eval()
  for X in tqdm(testset, unit="batch"):
    X, y = X.to(device), y.to(device)

    optimizer.zero_grad()
    pred = net(X)
    predicted = np.array((torch.max(pred.data, axis=1)[1]).detach().cpu())
    test_preds.append(predicted)

net.train()
del testset

# Store a numpy vector of the predictions for the test set in the variable `pred
# test_preds = test_preds.detach().cpu()
# tensor_preds = test_preds.numpy()
predictions = np.array(test_preds).reshape(10000)
```

```
<ipython-input-13-f2ded1370779>:8: TqdmDeprecationWarning: This function wi
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for X in tqdm(testset, unit="batch"):
100%                                        1000/1000 [00:10<00:00, 87.53batch/s]
```

```
predictions
```

```
array([2, 8, 8, ..., 9, 4, 9])
```

```
# This code below will generate kaggle_predictions.csv file. Please download it
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 corr
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.