Lab 3

Part B Building a CNN

Deadline: June 5, 9pm

Colab Link

Include a link to your colab file here

Colab Link: https://drive.google.com/open?id=1kTcucRGcAI6jx80gE-YbW1hNfYAH3-rU

```
import numpy as np
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torch.utils.data.sampler import SubsetRandomSampler
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

1. Model Building and Sanity Checking

Part (a) Convolutional Network

Build a convolutional neural network model that takes the (224x224 RGB) image as input, and predicts the letter. Your model should be a subclass of nn.Module. Explain your choice of neural network architecture: how many layers did you choose? What types of layers did you use? Were they fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units.

My Convolutional nerual network contains 10 layers, which are composed of 4 convolutional layers, 4 max pooling layers, and 2 fully connected layers. I will apply Relu activation function to all the outputs of the hidden layers.

1. Choice of convolutional layers: the convolutional layers are chosen such that the size of the output channel is double the size of the input channel. The reasoning behind this is that we do not want to loss too much information in each layer. By applying more filters before the previous layer, more features will be detected to compensate for the fact that the actual size (width * height) of the input is decreasing for each convolution layer. But we also want to limit the number of output channels before the fully connected layers such that it is not too computationally expensive. Padding is not applied to all

inputs for the purpose of this assignment, because the edges of all images do not contain useful information. The kernal size is chosen to be an odd dimension (either 3 * 3 or 5 * 5) for ease of computation.

2. Choice of fully connected layers: the number of units in each hidden layer for the fully-connected is reduced to consolidate information and to remove useless information for the current task.

3. Choice of activation functions: we use Relu activation function for this assigment as it generally performs well and is not computationally expensive.

4. Choice of pooling: pooling layers are used to reduce the input size (width * height) for each layer. A Max pooling is chosen because max pooling generally works better than average pooling. The kernal size and the stride are both 2 for ease of computation.

5. number of layers: the number of layers are chosen based on the size of the dataset. Having a large model is helpful only when there is enough data to train the model. A small dataset is not capable of training a large model. I tried a few models with different number of layers and I found that this model works well (But I will not show the process of tuning this hyperparameter in part 4).

```
class net(nn.Module):
  def __init__(self):
    super(net,self).__init__()
    self.name = "CNN"
    self.conv1 = nn.Conv2d(3, 5, 5)
    self.conv2 = nn.Conv2d(5, 10, 3)
    self.conv3 = nn.Conv2d(10, 20, 7)
    self.conv4 = nn.Conv2d(20, 40, 5)
    self.pool = nn.MaxPool2d(2, 2)
    self.fc1 = nn.Linear(40 * 10 * 10, 2048)
    self.fc2 = nn.Linear(2048, 9)

  def forward(self, x):
    x = self.pool(F.relu(self.conv1(x))) #output size: 5*220*220; after pooling, output size
    x = self.pool(F.relu(self.conv2(x))) #output size: 10*108*108; after pooling, output siz
    x = self.pool(F.relu(self.conv3(x))) #output size: 20*48*48; after pooling, output size:
    x = self.pool(F.relu(self.conv4(x))) #output size: 40*20*20; after pooling, output size:
    x = x.view(-1, 40 * 10 * 10)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return x
```

Part (b) Training Code

Write code to train your neural network given some training data. Your training code should make it easy to tweak hyperparameters. Make sure that you are checkpointing your models from time to time (the frequency is up to you). Explain your choice of loss function and optimizer.

1. I use the cross-entropy loss as my loss function because this is not a binary classification problem and we can not use binary-cross entropy loss. The cross-entropy loss is generally used for N-ary classification problems. It also has nice properties to avoid learning slow-down around an activation value of 0 or 1 and to help with the vanishing gradient problem from which deep neural networks suffer.

2. I use the Adaptive Moment Estimation (Adam) as my optimizer. Adam is the go-to optimizer for modern practioners. Also, given that our network contains a lot of layers, Adam is a better choice as it is faster.

```
def train_net(model, batch_size=64 ,learning_rate=0.01, num_epochs=30, weight_decay=0.0):
  torch.manual_seed(1000)
  train_loader, val_loader, test_loader = get_data_loader(batch_size)
  ########################################
  criterion = nn.CrossEntropyLoss()
  optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
  ########################################
  #set up some numpy arrays to store the training/validation loss/accuarcy
  train_acc = np.zeros(num_epochs)
  train_loss = np.zeros(num_epochs)
  val_acc = np.zeros(num_epochs)
  val_loss = np.zeros(num_epochs)
  iters = []
  ########################################
  for epoch in range(num_epochs):
    for imgs, labels in train_loader:
      output = model(imgs)
      loss = criterion(output, labels)
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()
    #check point: save the current training information
    iters.append(epoch + 1)
    train_acc[epoch], train_loss[epoch] = evaluate(model, train_loader, criterion)
    val_acc[epoch], val_loss[epoch] = evaluate(model, val_loader, criterion)
    print(("Epoch {}: Train acc: {}, Train loss: {} |" + "Validation acc: {}, Validation los
                  epoch + 1, train_acc[epoch], train_loss[epoch], val_acc[epoch], val_loss[

  #plotting
  plt.title("Train vs. Validation Loss")
  plt.plot(iters, train_loss, label = "Train")
  plt.plot(iters, val_loss, label = "Validation")
  plt.xlabel("Epoch")
  plt.ylabel("Loss")
  plt.legend(loc='best')
  plt.show()

  plt.title("Train vs. Validation Accuracy")
  plt.plot(iters, train_acc, label = "Train")
  plt.plot(iters, val_acc, label = "Validation")
  plt.xlabel("Epoch")
  plt.ylabel("Accuracy")
  plt.legend(loc='best')
  plt.show()

  print("Final Training Accuracy: {}" .format(train_acc[-1]))
  print("Final Validation Accuracy: {}".format(val_acc[-1]))

def evaluate(model, loader, criterion):
  total_loss = 0.0
  total_acc = 0.0
  total_img = 0.0
```

```
  for i, (imgs, labels) in enumerate(loader):
    output = model(imgs)
    loss = criterion(output, labels)
    pred = output.max(1, keepdim=True)[1]
    total_acc += pred.eq(labels.view_as(pred)).sum().item()
    total_loss += loss
    total_img += len(labels)
  acc = total_acc / total_img
  loss = float(total_loss) / (i+1)
  return acc, loss
```

Part (c) Overfit to a Small Dataset

One way to sanity check our neural network model and training code is to check whether the model is capable of overfitting a small dataset. Construct a small dataset (e.g. 1-2 image per class). Then show that your model and training code is capable of overfitting on that small dataset. You should be able to obtain a 100% training accuracy on that small dataset relatively quickly. If your model cannot overfit the small dataset quickly, then there is a bug in either your model code and/or your training code. Fix the issues before you proceed to the next step.

In order to get the overfit dataset, I manually picked two images from each class and put all the overfit data under the directory named overfit_data. To make the model overfit the dataset, I trained the model for 20 epoches. The training loss kept decreasing and the training accuarcy kept increasing throughout the training process. However, the validation loss increased after 8 epochs and the validation accuarcy maintained roughly the same after that point. This is a sign of overfitting.

```
from google.colab import drive
drive.mount('/content/gdrive')
```

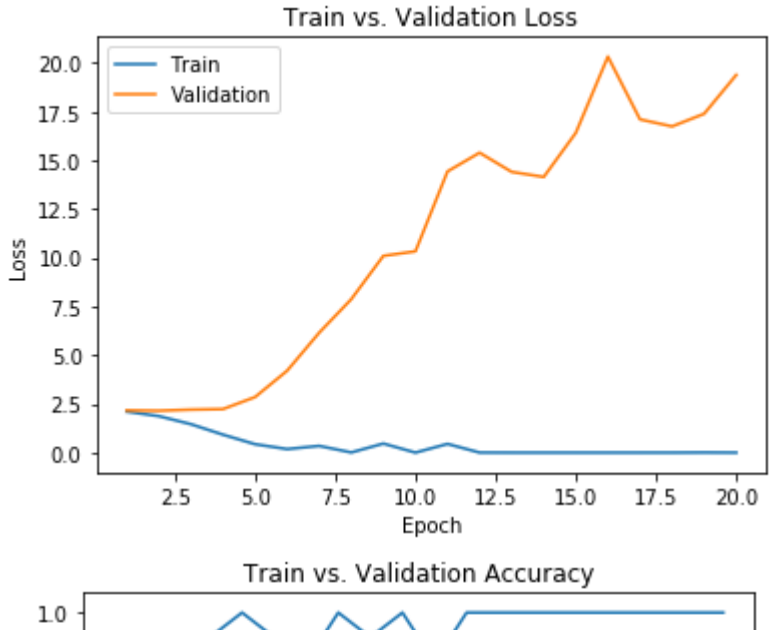
```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947

Enter your authorization code:
··········
Mounted at /content/gdrive
```

```
!unzip '/content/gdrive/My Drive/data.zip'
```

```
def get_overfit_data_loader(batch_size=64):
  transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5, 0.5, 0.5)
  train_set = torchvision.datasets.ImageFolder(root='./data/overfit_data', transform=transfo
  val_set = torchvision.datasets.ImageFolder(root='./data/validation', transform=transform)
  train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, num_workers=1
  val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, num_workers=1, sh

  return train_loader, val_loader

def train_net_overfit(model, batch_size=64 ,learning_rate=0.001, num_epochs=30):
  torch.manual_seed(100)
  train_loader, val_loader = get_overfit_data_loader(batch_size)
  ########################################
  criterion = nn.CrossEntropyLoss()
```

```
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    ########################################
    #set up some numpy arrays to store the training/validation loss/accuarcy
    train_acc = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_acc = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)
    iters = []
    ########################################
    for epoch in range(num_epochs):
      for (imgs, labels) in train_loader:
        output = model(imgs)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
      #check point: save the current training information
      iters.append(epoch + 1)
      train_acc[epoch], train_loss[epoch] = evaluate(model, train_loader, criterion)
      val_acc[epoch], val_loss[epoch] = evaluate(model, val_loader, criterion)
      print(("Epoch {}: Train acc: {}, Train loss: {} |" + "Validation acc: {}, Validation los
                    epoch + 1, train_acc[epoch], train_loss[epoch], val_acc[epoch], val_loss[
      model_path = "model_{0}_bs{1}_lr{2}_epoch{3}" .format(model.name, batch_size, learning_r
      torch.save(model.state_dict(), model_path)

    #plotting
    plt.title("Train vs. Validation Loss")
    plt.plot(iters, train_loss, label = "Train")
    plt.plot(iters, val_loss, label = "Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()

    plt.title("Train vs. Validation Accuracy")
    plt.plot(iters, train_acc, label = "Train")
    plt.plot(iters, val_acc, label = "Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}" .format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))


overfit_model = net()
train_net_overfit(overfit_model, batch_size=64 ,learning_rate=0.001, num_epochs=20)
```

```
Epoch 1: Train acc: 0.1111111111111111, Train loss: 2.104825258255005 |Validation acc
Epoch 2: Train acc: 0.3333333333333333, Train loss: 1.8733406066894531 |Validation ac
Epoch 3: Train acc: 0.7777777777777778, Train loss: 1.4609079360961914 |Validation ac
Epoch 4: Train acc: 0.9444444444444444, Train loss: 0.9225044250488281 |Validation ac
Epoch 5: Train acc: 1.0, Train loss: 0.43314334750175476 |Validation acc: 0.398009956
Epoch 6: Train acc: 0.9444444444444444, Train loss: 0.18345099687576294 |Validation a
Epoch 7: Train acc: 0.8888888888888888, Train loss: 0.3453025817871094 |Validation ac
Epoch 8: Train acc: 1.0, Train loss: 0.010260794311761856 |Validation acc: 0.39303482
Epoch 9: Train acc: 0.9444444444444444, Train loss: 0.4652808606624603 |Validation ac
Epoch 10: Train acc: 1.0, Train loss: 0.0023356014862656593 |Validation acc: 0.388059
Epoch 11: Train acc: 0.8888888888888888, Train loss: 0.447619765996933 |Validation ac
Epoch 12: Train acc: 1.0, Train loss: 0.0037037532310932875 |Validation acc: 0.363184
Epoch 13: Train acc: 1.0, Train loss: 0.00027969147777184844 |Validation acc: 0.36318
Epoch 14: Train acc: 1.0, Train loss: 0.00012164645886514336 |Validation acc: 0.36815
Epoch 15: Train acc: 1.0, Train loss: 6.463792669819668e-05 |Validation acc: 0.363184
Epoch 16: Train acc: 1.0, Train loss: 5.891588079975918e-05 |Validation acc: 0.348258
Epoch 17: Train acc: 1.0, Train loss: 0.00014135573292151093 |Validation acc: 0.33333
Epoch 18: Train acc: 1.0, Train loss: 0.0005408392753452063 |Validation acc: 0.333333
Epoch 19: Train acc: 1.0, Train loss: 0.006919967010617256 |Validation acc: 0.3184079
Epoch 20: Train acc: 1.0, Train loss: 0.0008662541513331234 |Validation acc: 0.333333
```



2. Data Loading and Splitting

Download the anonymized data collected by you and your classmates. Split the data into training, validation, and test sets. Note: Data splitting is not as trivial in this lab. We want our test set to closely resemble the setting in which our model will be used. In particular, our test set should contain hands that are never seen in training! Explain how you split the data, either by describing what you did, or by showing the code that you used. Justify your choice of splitting strategy. How many training, validation, and test images do you have? For loading the data, you can use plt.imread as in Lab 1, or any other method that you choose. You may find torchvision.datasets.ImageFolder helpful. (see https://pytorch.org/docs/master/torchvision/datasets.html#imagefolder) For this portion only, you are free to look up tutorials or other code on the internet to help you.

```
Final Validation Accuracy: 0.3333333333333333
```

For data spliting, I created three subdirectories (train, validation, test) under the data directory. For each directory, there are nine folders denoting the labels of the images. I split my data into 70/15/15 and I also decided to split different students into the three dataset. Given that the samples are taken from 50 students, the 70/15/15 split corresponds roughly to 35/7.5/7.5 students for training, validation and testing. It also corresponds to 815/175/175 images as there are 1164 images in total. I decided to use the order provided in the original data set to select students for datasets, meaning 1-35 is for training set, 35-43 is for validation set and 44-50 is for test set. The splitting process is random as the original given order is random. There are several reasons behind this data splitting process:

1. Splitting students into training, validation and test sets ensures that the test set provides a fair evaluation. All the test data contain hands never seen by the model before. If the model is able to perform well on those pure data, it is a well trained model.

2. The 70/15/15 spilt represents a good balance between training quality and testing accuracy. We want the model to train on large dataset but we also need enough test data to ensure the evaluation accuracy. I decided to use this ratio as it is suggested by many machine learning literatures.

3. There are two main reasons that I did not do a random selection on the entire data set. First, it is possible that the test set is too similar to the training set. The test set might contain students that are already seen during the training process. Second, we want to make sure that the model is trained on the all the letters before it is tested. And the easiset fix to these issues is to split students into different datasets.

```
def get_data_loader(batch_size=64):
  transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
  train_set = torchvision.datasets.ImageFolder(root='./data/train', transform=transform)
  val_set = torchvision.datasets.ImageFolder(root='./data/validation', transform=transform)
  test_set = torchvision.datasets.ImageFolder(root='./data/test', transform=transform)

  train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, num_workers=1
  val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, num_workers=1, sh
  test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, num_workers=1,

  return train_loader, val_loader, test_loader
```

3. Training

Train your first network on your training set. Plot the training curve, and include your plot in your writeup. Make sure that you are checkpointing frequently!

For the first model, I chose to use a batch size of 64, a learning rate of 0.001, 30 epochs and a weight decay of 0.

```
part3_net = net()
train_net(part3_net, batch_size=64, learning_rate=0.001, num_epochs=30, weight_decay=0.0)
```

```
Epoch 1: Train acc: 0.3425692695214106, Train loss: 2.0900934659517727 |Validation ac
Epoch 2: Train acc: 0.5541561712846348, Train loss: 1.2456324650691106 |Validation ac
Epoch 3: Train acc: 0.7493702770780857, Train loss: 0.8129540223341721 |Validation ac
Epoch 4: Train acc: 0.809823677581864, Train loss: 0.6199282866257888 |Validation acc
Epoch 5: Train acc: 0.8627204030226701, Train loss: 0.4537505736717811 |Validation ac
Epoch 6: Train acc: 0.8438287153652393, Train loss: 0.4168445880596454 |Validation ac
Epoch 7: Train acc: 0.8778337531486146, Train loss: 0.32418570151695836 |Validation a
Epoch 8: Train acc: 0.9433249370277078, Train loss: 0.17262513820941633 |Validation a
Epoch 9: Train acc: 0.9634760705289672, Train loss: 0.10878606942983773 |Validation a
Epoch 10: Train acc: 0.9584382871536524, Train loss: 0.1405185736142672 |Validation a
Epoch 11: Train acc: 0.9811083123425692, Train loss: 0.06418095185206486 |Validation
Epoch 12: Train acc: 0.9899244332493703, Train loss: 0.030197038100315973 |Validatior
Epoch 13: Train acc: 0.9836272040302267, Train loss: 0.06132940145639273 |Validation
Epoch 14: Train acc: 0.9559193954659949, Train loss: 0.11073175760415885 |Validation
Epoch 15: Train acc: 0.9886649874055415, Train loss: 0.04856458994058462 |Validation
```

4. Hyperparameter Search

Part (a)

List 3 hyperparameters that you think are most worth tuning.

In part three, the model overfits the training data fairly quickly (around 12 epochs). To improve the performance of the model, the following parameters are tunned:

1. Weight decay: this hyperparameter is used to fix the problem of overfitting. Weight decay is used to prevent the model overly relying on one specific pixel.

2. Learning rate: this parameter is used to improve the speed and loss of training. Our goal is to find the optimal learning rate that provides the smallest loss.

3. Number of epoches: this is another technique used to prevent overfitting. This is also known as early stopping. This technique will be applied to the end to the model that has the highest validation accuracy historically(i.e.during training).

Part (b)

Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

My tunning strategy is to change one hyperparameter at a time while keeping the other parameters the same. The quality of the model will be assessed based on the validation accuracy.

To fix the overfitting problem, I decided to increase the weight decay to remove the reliability on specific picxels. I also apply early stopping to prevent overfitting. I will also look for the optimal learning rate that produces the highest validation accuracy.

Set1: batch_size=64, learning_rate=0.005, num_epochs=30, weight_decay=0

My first two trials are used to find the optimal learning rate. I first increase the learning rate to see whether the model performs better.

```
part4_net1 = net()
train_net(part4_net1, batch_size=64, learning_rate=0.005, num_epochs=30, weight_decay=0)
```

```
Epoch 1: Train acc: 0.24685138539042822, Train loss: 2.0255885491004357 |Validation a
Epoch 2: Train acc: 0.593198992443325, Train loss: 1.1304921370286207 |Validation acc
Epoch 3: Train acc: 0.7531486146095718, Train loss: 0.7434161993173453 |Validation ac
Epoch 4: Train acc: 0.8173803526448362, Train loss: 0.5263153956486628 |Validation ac
Epoch 5: Train acc: 0.8753148614609572, Train loss: 0.3532048005324144 |Validation ac
Epoch 6: Train acc: 0.8652392947103275, Train loss: 0.3422360420227051 |Validation ac
Epoch 7: Train acc: 0.9143576826196473, Train loss: 0.2402311655191275 |Validation ac
Epoch 8: Train acc: 0.9445843828715366, Train loss: 0.15422430405249962 |Validation a
Epoch 9: Train acc: 0.9735516372795969, Train loss: 0.08366916729853703 |Validation a
Epoch 10: Train acc: 0.9017632241813602, Train loss: 0.27186670670142543 |Validation
Epoch 11: Train acc: 0.9672544080604534, Train loss: 0.09511691790360671 |Validation
Epoch 12: Train acc: 0.9811083123425692, Train loss: 0.04819580224844126 |Validation
Epoch 13: Train acc: 0.9937027707808564, Train loss: 0.033159421040461615 |Validatior
Epoch 14: Train acc: 1.0, Train loss: 0.00441482261969493 |Validation acc: 0.74626865
Epoch 15: Train acc: 1.0, Train loss: 0.0018594270715346704 |Validation acc: 0.751243
Epoch 16: Train acc: 1.0, Train loss: 0.0020136219950822685 |Validation acc: 0.751243
Epoch 17: Train acc: 0.9924433249370277, Train loss: 0.0226402236865117 |Validation a
Epoch 18: Train acc: 0.982367758186398, Train loss: 0.05009961128234863 |Validation a
Epoch 19: Train acc: 0.9811083123425692, Train loss: 0.05189289496495174 |Validation
Epoch 20: Train acc: 0.9924433249370277, Train loss: 0.0235732220686399 |Validation a
Epoch 21: Train acc: 0.9987405541561712, Train loss: 0.011589577564826379 |Validatior
Epoch 22: Train acc: 0.9899244332493703, Train loss: 0.030686011681189902 |Validatior
Epoch 23: Train acc: 0.9785894206549118, Train loss: 0.07082844239014846 |Validation
Epoch 24: Train acc: 0.9949622166246851, Train loss: 0.019327347095196065 |Validatior
```

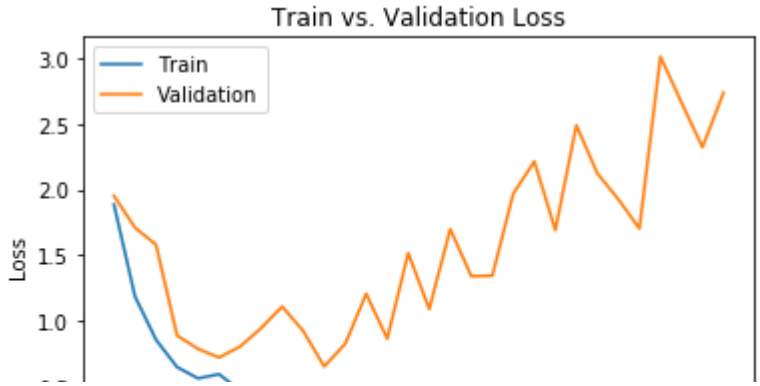
Set2: batch_size=64, learning_rate=0.0005, num_epochs=30, weight_decay=0

In this trial, I will decrease the learning rate to see if the model is better.

```
Epoch 30: Train acc: 1.0, Train loss: 0.0001362094906373666 |Validation acc: 0.761194
```

```
part4_net2 = net()
train_net(part4_net2, batch_size=64, learning_rate=0.0005, num_epochs=30, weight_decay=0)
```

```
Epoch 1: Train acc: 0.473551637279597, Train loss: 1.8860078958364634 |Validation acc
Epoch 2: Train acc: 0.5869017632241813, Train loss: 1.1871498548067534 |Validation ac
Epoch 3: Train acc: 0.7279596977329975, Train loss: 0.8582639694213867 |Validation ac
Epoch 4: Train acc: 0.8022670025188917, Train loss: 0.6506953606238732 |Validation ac
Epoch 5: Train acc: 0.8425692695214105, Train loss: 0.5645832281846267 |Validation ac
Epoch 6: Train acc: 0.8047858942065491, Train loss: 0.5970052205599271 |Validation ac
Epoch 7: Train acc: 0.8387909319899244, Train loss: 0.47151506864107573 |Validation a
Epoch 8: Train acc: 0.889168765743073, Train loss: 0.33554711708655727 |Validation ac
Epoch 9: Train acc: 0.9156171284634761, Train loss: 0.2635157658503606 |Validation ac
Epoch 10: Train acc: 0.9231738035264484, Train loss: 0.22701600881723258 |Validation
Epoch 11: Train acc: 0.9521410579345088, Train loss: 0.1472884599979107 |Validation a
Epoch 12: Train acc: 0.964735516372796, Train loss: 0.1309461501928476 |Validation ac
Epoch 13: Train acc: 0.964735516372796, Train loss: 0.11642088339878963 |Validation a
Epoch 14: Train acc: 0.9785894206549118, Train loss: 0.08239106948559101 |Validation
Epoch 15: Train acc: 0.9848866498740554, Train loss: 0.0545198367192195 |Validation a
Epoch 16: Train acc: 0.9937027707808564, Train loss: 0.03843735731565035 |Validation
Epoch 17: Train acc: 0.9962216624685138, Train loss: 0.014998228504107548 |Validatior
Epoch 18: Train acc: 1.0, Train loss: 0.009008377217329465 |Validation acc: 0.7761194
Epoch 19: Train acc: 1.0, Train loss: 0.004104801668570592 |Validation acc: 0.781094
Epoch 20: Train acc: 1.0, Train loss: 0.0033104574451079736 |Validation acc: 0.78606
Epoch 21: Train acc: 1.0, Train loss: 0.00196585374382826 |Validation acc: 0.8109452
Epoch 22: Train acc: 1.0, Train loss: 0.0011981128929899288 |Validation acc: 0.791044
Epoch 23: Train acc: 1.0, Train loss: 0.0007875230020055404 |Validation acc: 0.78606
Epoch 24: Train acc: 1.0, Train loss: 0.0006619034191736808 |Validation acc: 0.78606
Epoch 25: Train acc: 1.0, Train loss: 0.0005876437689249332 |Validation acc: 0.781094
Epoch 26: Train acc: 1.0, Train loss: 0.0004945976946216363 |Validation acc: 0.78606
Epoch 27: Train acc: 1.0, Train loss: 0.0004170110377554 71 |Validation acc: 0.786069
Epoch 28: Train acc: 1.0, Train loss: 0.00042229195913443196 |Validation acc: 0.7810
Epoch 29: Train acc: 1.0, Train loss: 0.0003473038713519 39 |Validation acc: 0.791044
Epoch 30: Train acc: 1.0, Train loss: 0.00030531730646124255 |Validation acc: 0.7761
```



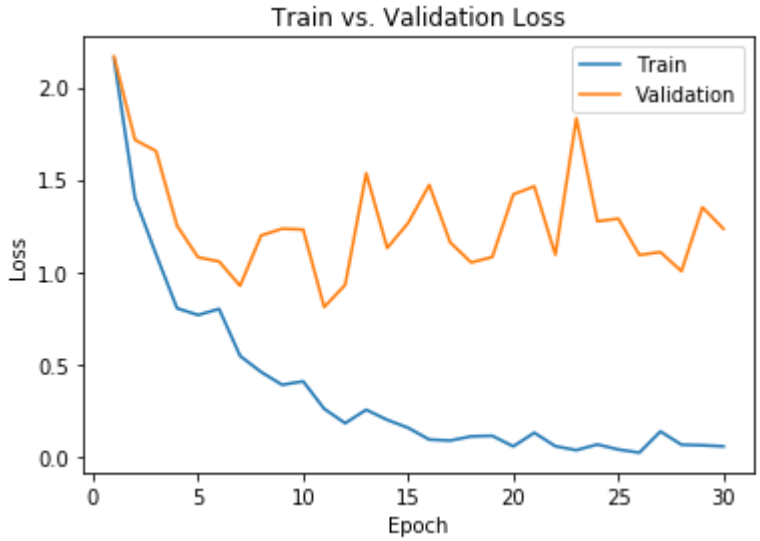
It seems that 0.0005 is a good learning rate as the highest accuracy is achieved with this learning rate.

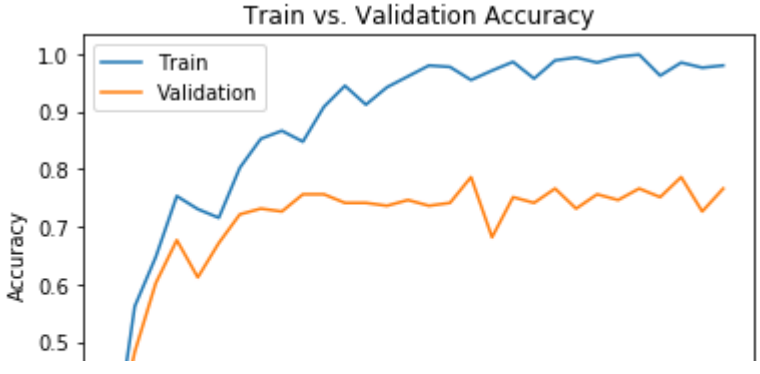
Set3: batch_size=64, learning_rate=0.0005, num_epochs=30, weight_decay=0.01

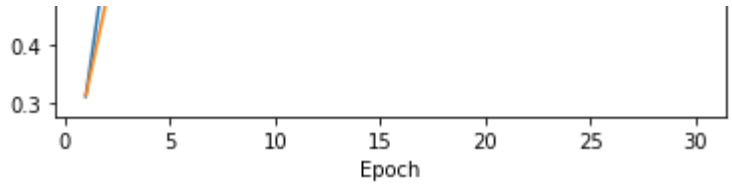
This trial is to fix the overfit problem by adding a weight decay.

```
part4_net3 = net()
train_net(part4_net3, batch_size=64, learning_rate=0.0005, num_epochs=30, weight_decay=0.01)
```

```
Epoch 1: Train acc: 0.3110831234256927, Train loss: 2.1557376568134012 |Validation ac
Epoch 2: Train acc: 0.5617128463476071, Train loss: 1.3989539513221154 |Validation ac
Epoch 3: Train acc: 0.6473551637279596, Train loss: 1.0950186069195087 |Validation ac
Epoch 4: Train acc: 0.7531486146095718, Train loss: 0.8029903265146109 |Validation ac
Epoch 5: Train acc: 0.7304785894206549, Train loss: 0.7663326263427734 |Validation ac
Epoch 6: Train acc: 0.7153652392947103, Train loss: 0.7992319693932166 |Validation ac
Epoch 7: Train acc: 0.8022670025188917, Train loss: 0.5446036045367901 |Validation ac
Epoch 8: Train acc: 0.8526448362720404, Train loss: 0.4576079295231746 |Validation ac
Epoch 9: Train acc: 0.8664987405541562, Train loss: 0.38815986193143404 |Validation a
Epoch 10: Train acc: 0.8476070528967254, Train loss: 0.40712499618530273 |Validation
Epoch 11: Train acc: 0.9080604534005038, Train loss: 0.25850605964660645 |Validation
Epoch 12: Train acc: 0.9445843828715366, Train loss: 0.18033693386958197 |Validation
Epoch 13: Train acc: 0.9118387909319899, Train loss: 0.253265931056096 |Validation ac
Epoch 14: Train acc: 0.9420654911838791, Train loss: 0.1985042278583233 |Validation a
Epoch 15: Train acc: 0.9609571788413098, Train loss: 0.15507149696350098 |Validation
Epoch 16: Train acc: 0.9798488664987406, Train loss: 0.09183733279888447 |Validation
Epoch 17: Train acc: 0.9773299748110831, Train loss: 0.0864107608795166 |Validation a
Epoch 18: Train acc: 0.9546599496221663, Train loss: 0.10924643736619216 |Validation
Epoch 19: Train acc: 0.9710327455919395, Train loss: 0.11209774017333984 |Validation
Epoch 20: Train acc: 0.9861460957178841, Train loss: 0.05499893885392409 |Validation
Epoch 21: Train acc: 0.9571788413098237, Train loss: 0.1285840364602896 |Validation a
Epoch 22: Train acc: 0.9886649874055415, Train loss: 0.055921352826631986 |Validatior
Epoch 23: Train acc: 0.9937027707808564, Train loss: 0.03436433810454149 |Validation
Epoch 24: Train acc: 0.9848866498740554, Train loss: 0.0653642461850093 |Validation a
Epoch 25: Train acc: 0.9949622166246851, Train loss: 0.0379250508088332 |Validation a
Epoch 26: Train acc: 0.9987405541561712, Train loss: 0.02098375788101783 |Validation
Epoch 27: Train acc: 0.9622166246851386, Train loss: 0.1358719697365394 |Validation a
Epoch 28: Train acc: 0.9848866498740554, Train loss: 0.06410795450210571 |Validation
Epoch 29: Train acc: 0.9760705289672544, Train loss: 0.06084326597360464 |Validation
Epoch 30: Train acc: 0.9798488664987406, Train loss: 0.0552559128174415 |Validation a
```








```
Final Training Accuracy: 0.9798488664987406
```

It seems that the overfitting is reduced. But the validation accuracy also decreases. So for the next trial, I will try a smallesy weight_decay to see what happens.

Set4: batch_size=64, learning_rate=0.0005, num_epochs=30, weight_decay=0.0001

```
part4_net4 = net()
train_net(part4_net4, batch_size=64, learning_rate=0.0005, num_epochs=30, weight_decay=0.000
```

```
Epoch 1: Train acc: 0.49622166246851385, Train loss: 1.8990808633657603 |Validation a
Epoch 2: Train acc: 0.5793450881612091, Train loss: 1.1875342589158278 |Validation ac
Epoch 3: Train acc: 0.72544080604534, Train loss: 0.8651970349825345 |Validation acc:
Epoch 4: Train acc: 0.801007556675063, Train loss: 0.6584084584162786 |Validation acc
Epoch 5: Train acc: 0.8337531486146096, Train loss: 0.5684751363900992 |Validation ac
Epoch 6: Train acc: 0.8110831234256927, Train loss: 0.5877249057476337 |Validation ac
Epoch 7: Train acc: 0.845088161209068, Train loss: 0.45879316329956055 |Validation ac
Epoch 8: Train acc: 0.889168765743073, Train loss: 0.3463522470914401 |Validation acc
Epoch 9: Train acc: 0.9256926952141058, Train loss: 0.257707247367272 |Validation acc
Epoch 10: Train acc: 0.924433249370277, Train loss: 0.22455398853008562 |Validation a
Epoch 11: Train acc: 0.9458438287153652, Train loss: 0.16019175602839544 |Validation
Epoch 12: Train acc: 0.9722921914357683, Train loss: 0.1162644624710083 |Validation a
Epoch 13: Train acc: 0.9773299748110831, Train loss: 0.10299085653745212 |Validation
Epoch 14: Train acc: 0.9748110831234257, Train loss: 0.09114106801839975 |Validation
Epoch 15: Train acc: 0.9785894206549118, Train loss: 0.06365836125153762 |Validation
Epoch 16: Train acc: 0.9899244332493703, Train loss: 0.0377061298260322 |Validation a
Epoch 17: Train acc: 0.9987405541561712, Train loss: 0.015525299769181471 |Validatior
Epoch 18: Train acc: 0.9987405541561712, Train loss: 0.009848908736155583 |Validatior
Epoch 19: Train acc: 1.0, Train loss: 0.007648449677687425 |Validation acc: 0.7910447
Epoch 20: Train acc: 0.9987405541561712, Train loss: 0.004987819263568291 |Validatior
Epoch 21: Train acc: 1.0, Train loss: 0.002613398031546519 |Validation acc: 0.7860696
Epoch 22: Train acc: 1.0, Train loss: 0.0021429362778480235 |Validation acc: 0.781094
Epoch 23: Train acc: 1.0, Train loss: 0.0015383801208092617 |Validation acc: 0.776119
Epoch 24: Train acc: 1.0, Train loss: 0.0025034437959010783 |Validation acc: 0.776119
Epoch 25: Train acc: 1.0, Train loss: 0.0013301296589466243 |Validation acc: 0.791044
Epoch 26: Train acc: 1.0, Train loss: 0.004048436593550902 |Validation acc: 0.7711442
```

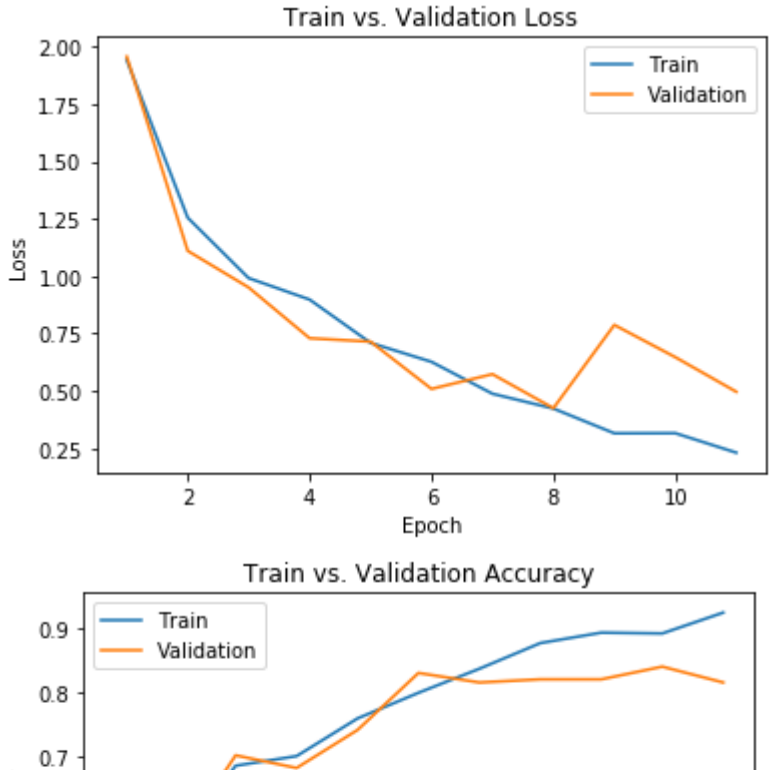
After the above two trials, I found that the weight decay helps with overfitting, but it sacrifices validation accuracy. For the next trial, I will apply early stopping to prevent overfitting.

Set5: batch_size=64, learning_rate=0.0005, num_epochs=11, weight_decay=0

The last trial is to find the best model. I looked at all the models trained so far and picked the model that has small loss and large accuracy. I adjust the number of epochs such that the model does not overfit nor underfit the training set.

```
part4_net5 = net()
train_net(part4_net5, batch_size=64, learning_rate=0.0005, num_epochs=11, weight_decay=0)
```

```
Epoch 1: Train acc: 0.3350125944584383, Train loss: 1.9427419809194713 |Validation ac
Epoch 2: Train acc: 0.5591939546599496, Train loss: 1.2552689772385817 |Validation ac
Epoch 3: Train acc: 0.6851385390428212, Train loss: 0.9918712469247671 |Validation ac
Epoch 4: Train acc: 0.7002518891687658, Train loss: 0.8982696533203125 |Validation ac
Epoch 5: Train acc: 0.7594458438287154, Train loss: 0.7095093360314002 |Validation ac
Epoch 6: Train acc: 0.7997481108312342, Train loss: 0.6268058923574594 |Validation ac
Epoch 7: Train acc: 0.8375314861460957, Train loss: 0.4877131535456731 |Validation ac
Epoch 8: Train acc: 0.8778337531486146, Train loss: 0.4232028814462515 |Validation ac
Epoch 9: Train acc: 0.8942065491183879, Train loss: 0.3152902676508977 |Validation ac
Epoch 10: Train acc: 0.8929471032745592, Train loss: 0.31583089094895583 |Validation
Epoch 11: Train acc: 0.9256926952141058, Train loss: 0.23089221807626578 |Validation
```



Part (c)

Choose the best model out of all the ones that you have trained. Justify your choice.

The criteria for selecting the best model is the validation accuarcy. Validation accuarcy measure the quality of predictions made on a the validation dataset.

Considering the above criteria, the last model is chosen to be the best model as it provides high validation accuarcy (0.82) and low validation loss(0.496).

part (d)

Report the test accuracy of your best model. You should only do this step once.

```
train_loader, val_loader, test_loader = get_data_loader(batch_size=64)
test_acc, test_loss = evaluate(part4_net5, test_loader, nn.CrossEntropyLoss())
print("Final test loss: {}" .format(test_loss))
print("Final test Accuracy: {}" .format(test_acc))
```

```
Final test loss: 1.3091978232065837
Final test Accuracy: 0.6488095238095238
```

5. Transfer Learning

For many image classification tasks, it is generally not a good idea to train a very large deep neural network model from scratch due to the enormous compute requirements and lack of sufficient amounts of training data. One of the better options is to try using an existing model that performs a similar task to the one you need to solve. This method of utilizing a pre-trained network for other similar tasks is broadly termed Transfer Learning. In this assignment, we will use Transfer Learning to extract features from the hand gesture images. Then, train a smaller network to use these features as input and classify the hand gestures. As you have learned from the CNN lecture, convolution layers extract various features from the images which get utilized by the fully connected layers for correct classification. AlexNet architecture played a pivotal role in establishing Deep Neural Nets as a go-to tool for image classification problems and we will use an ImageNet pre-trained AlexNet model to extract features in this assignment.

Part (a)

Compute the AlexNet features for each of your training, validation, and test data.

Save the computed features. You will be using these features as input to your neural network in Part (b), and you do not want to re-compute the features every time. Instead, run alexnet.features once for each image, and save the result.

```
import torchvision.models
alex_net = torchvision.models.alexnet(pretrained=True)

train_loader, val_loader, test_loader = get_data_loader(batch_size=64)

train_features, val_features, test_features = [], [], []
for imgs, labels in iter(train_loader):
  train_features.append((alex_net.features(imgs), labels))


for imgs, labels in iter(val_loader):
  val_features.append((alex_net.features(imgs), labels))


for imgs, labels in iter(test_loader):
  test_features.append((alex_net.features(imgs), labels))


np.save("train_features.npy", train_features)
np.save("val_features.npy", val_features)
np.save("test_features.npy", test_features)
train_features.clear()
val_features.clear()
test_features.clear()
```

Part (b)

Build a convolutional neural network model that takes as input these AlexNet features, and makes a prediction. Your model should be a subclass of nn.Module. Explain your choice of neural network

architecture: how many layers did you choose? What types of layers did you use: fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units in each layer?

The model contains fully-connected layers only as the alex net features already produce output from a well trained convolutional model. The purpose of this model is to add classification layers to consolidate information and produce prediction.

1. Activation function: I chose relu as my activation function as it generally performs well and is used in most cases.

2. Number of hidden units: the number of hidden units decreases for each layer. The purpose is to consolidation information and get rid of useless information. The number of output units matches the number of labels we have for the data.

3. Number of layers: I chose to use two layers for the purpose of this task. This is a balance between time and accuarcy. We do not want to spend too much time on training the network, but we still well trained model. Two layer allows the number of hidden units not to decay too much and it takes reasonable amount of time to train.

```
class alex_classifier(nn.Module):
  def __init__(self):
    super(alex_classifier, self).__init__()
    self.name = "classifier"
    self.fc1 = nn.Linear(256 * 6 * 6, 4096)
    self.fc2 = nn.Linear(4096, 9)

  def forward(self, x):
    x = x.view(-1, 256 * 6 * 6)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return x
```

Part (c)

Train your new network, including any hyperparameter tuning. Plot and submit the training curve of your best model only.

There are three hyperparameters I tuned to improve the behaviour of the model. They are batch size, learning_rate and num_epochs. The tuning strategy is almost identical to part 4. If the model overfits the training data, I will increase the batch size, decrease the learning rate and decrease the number of epochs. If the model underfits the training data, I will do the opposite. After a few trials, the best model I get produces a validation loss of 0.11 and a validation accuarcy of 0.965.

```
def train_alex_classifier(model, batch_size=64 ,learning_rate=0.01, num_epochs=30):
```

```
    train_features = np.load("train_features.npy", allow_pickle=True)
    val_features = np.load("val_features.npy", allow_pickle=True)
    ########################################
    #Fix Random Seed
    torch.manual_seed(1000)
    ########################################
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    ########################################
    #set up some numpy arrays to store the training/validation loss/accuarcy
    train_acc = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_acc = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)
    iters = []
    ########################################
    for epoch in range(num_epochs):
      for imgs, labels in train_features:
        output = model(imgs)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
      #check point: save the current training information
      iters.append(epoch + 1)
      train_acc[epoch], train_loss[epoch] = evaluate_alex(model, train_features, criterion)
      val_acc[epoch], val_loss[epoch] = evaluate_alex(model, val_features, criterion)
      print(("Epoch {}: Train acc: {}, Train loss: {} |" + "Validation acc: {}, Validation los
                    epoch + 1, train_acc[epoch], train_loss[epoch], val_acc[epoch], val_loss[
      model_path = "model_{0}_bs{1}_lr{2}_epoch{3}" .format(model.name, batch_size, learning_r
      torch.save(model.state_dict(), model_path)

    #plotting
    plt.title("Train vs. Validation Loss")
    plt.plot(iters, train_loss, label = "Train")
    plt.plot(iters, val_loss, label = "Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()

    plt.title("Train vs. Validation Accuracy")
    plt.plot(iters, train_acc, label = "Train")
    plt.plot(iters, val_acc, label = "Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}" .format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))


  def evaluate_alex(model, features, criterion):
    total_loss = 0.0
    total_acc = 0.0
    total_img = 0.0
    for i, (imgs, labels) in enumerate(features):
      output = model(imgs)
      loss = criterion(output, labels)
      pred = output.max(1, keepdim=True)[1]
      total_acc += pred.eq(labels.view_as(pred)).sum().item()
      total_loss += loss
      total_img += len(labels)
    acc = total_acc / total_img
    loss = float(total_loss) / (i+1)
    return acc, loss
```

```
part5_net = alex_classifier()
train_alex_classifier(part5_net, batch_size=64 ,learning_rate=0.0001, num_epochs=9)
```

```
Epoch 1: Train acc: 0.9231738035264484, Train loss: 0.2754026926480807 |Validation ac
Epoch 2: Train acc: 0.9962216624685138, Train loss: 0.06240101960989145 |Validation a
Epoch 3: Train acc: 0.9974811083123426, Train loss: 0.037450322738060586 |Validation
Epoch 4: Train acc: 1.0, Train loss: 0.016435493643467244 |Validation acc: 0.95024875
Epoch 5: Train acc: 1.0, Train loss: 0.011771391217525188 |Validation acc: 0.94029850
Epoch 6: Train acc: 1.0, Train loss: 0.006829251463596637 |Validation acc: 0.94527363
Epoch 7: Train acc: 1.0, Train loss: 0.0052222334421598 |Validation acc: 0.9651741293
Epoch 8: Train acc: 1.0, Train loss: 0.004414800841074724 |Validation acc: 0.96517412
Epoch 9: Train acc: 1.0, Train loss: 0.003847774691306628 |Validation acc: 0.96517412
```


```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.9651741293532339
```

Part (d)

Report the test accuracy of your best model. How does the test accuracy compare to part 4(d)?

```
best_net = alex_classifier()
model_path = "model_{0}_bs{1}_lr{2}_epoch{3}" .format(best_net.name, 64, 0.0001, 8)
state = torch.load(model_path)
best_net.load_state_dict(state)
```

```
train_loader, val_loader, test_loader = get_data_loader(batch_size=64)
test_features = np.load("test_features.npy", allow_pickle=True)

test_acc, test_loss = evaluate_alex(best_net, test_features, nn.CrossEntropyLoss())
print("Final test loss: {}" .format(test_loss))
print("Final test Accuracy: {}" .format(test_acc))
```

```
Final test loss: 0.10308396816253662
Final test Accuracy: 0.9642857142857143
```

The test accuracy is 96.43% which is higher than the accuracy (70%) in part 4. This makes sense beacuase Alex Net is a pretrained model that generally works well in image classification problems.