Part 8

In this part, we used six actors from Project 1. They are Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader and Steve Carell.

Preprocess the Data
The method *load_data* in *faces.py* is used to download and preprocess images. It loops through *faces_subset.txt* and downloads the image from the *url* field. Unlike Project 1, we also check whether the downloaded image has the actor's face in it by using SHA-256 hashes. If an image is downloaded successfully and it matches its hash code, the method will proceed to crop out the image of the face, and resize it to either $32 \times 32 \times 3$. Then all processed images are saved to a compressed *.npz* file called *cropped_faces_32.npz* so that it can be easily obtained by the other parts of the program.

Partition the Data
The method *get_dataset* in *faces.py* is used to partition data into training (size=70), validation (size=20) and test sets (size=20). It first loads *cropped_faces_32.npz* we prepared in *load_data* and shuffles all keys in the dictionary. Then the method loops over the shuffled keys and sorts images into three datasets. If an actor's total number of images is below $70+20+20$, we resize dataset sizes for this actor proportionally. The method also resizes images to requested image size *img_side_len*, e.g., $32 \times 32 \times 3$ for part 8 and $227 \times 227 \times 3$ for part 10. If the image is of RGBA format, we discard the alpha channel. If the Image has only one channel, we replicate it three times to get an RGB image. Note that all numpy arrays representing images are normalized between -1.0 and 1.0 and flattened before getting sorted into different datasets. Finally the method saves the datasets dictionary objects into a *.mat* file so that we do not need to partition datasets everytime when we run other parts of the program.

Load the Data
To get ready for part 8, we first load all data from *faces_all_32.mat*. And call *get_train_faces*, *get_validation_faces* and *get_test_faces* to get three separate datasets as numpy arrays.

Build the neural network
Then we are ready to build the neural network in method *part8_helper*. We use *pytorch* to construct a fully connected neural network with *ReLU* as the hidden layer, cross-entropy as the loss function and Adam as an optimizer. We also initialize weights using normal distribution with a small variance to get a better performance.

Listing 5: Part8 Neural network construction

```
model = torch.nn.Sequential(
    torch.nn.Linear(img_side_len*img_side_len*3, dim_h),
    torch.nn.ReLU(),
    torch.nn.Linear(dim_h, NUM_LABELS),
)

model.apply(init_weights)

loss_fn = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Listing 6: Weights initialization

```
def init_weights(m):
    if type(m) == torch.nn.Linear:
        m.weight.data.normal_(0.0, 0.03)
```

We then train the network with mini-batches from our training set. In each epoch iteration, we shuffle the whole training set and generate a new set of *train_batch_idx*. In each batch iteration, we select training samples at indices in the current *train_batch_idx* list and train by gradient descent and back propagation. At last, we obtain accuracies on all three sets.

Experiment with different settings

In order to produce the best performance, we use *np.meshgrid* to perform a grid search to test on a vectorized version of different parameters. Here in method *part8_grid_search* we have 5 parameters with different values for evaluation:

```
# Parameters 0: image resolution
img_side_lens = [32]
# Parameters 1: number of epochs
n_epochs = [20, 100, 200]
# Parameters 2: size of mini-batch
batch_size = [32, 64, 128]
# Parameters 3: number of hidden units
dim_h = [100, 200, 300]
# Parameters 4: learning rate
learning_rate = [1e-4, 1e-3, 1e-2]
```

After each run, we obtain the accuracies of all three datasets. Here are some of the results (**NOT THE BEST ONE YET**):

```
Parameters: resolution=32x32, n_epochs=20,
batch_size=32, dim_h=100, learning_rate=0.0001

Training accuracy: 0.884520884521
Validation accuracy: 0.803418803419
Test accuracy: 0.752136752137

================================================

Parameters: resolution=32x32, n_epochs=100,
batch_size=64, dim_h=200, learning_rate=0.001

Training accuracy: 1.0
Validation accuracy: 0.871794871795
Test accuracy: 0.854700854701

================================================

Parameters: resolution=32x32, n_epochs=100,
batch_size=128, dim_h=300, learning_rate=0.01
```

```
Training accuracy: 1.0
Validation accuracy: 0.837606837607
Test accuracy: 0.846153846154
```

Finally the method *part8_grid_search* will return a set of parameters that produce the best performance on test set. In case there are ties, we also compare the mean accuracies on all sets: the set of parameters that can produce highest test accuracies and highest mean accuracies is considered the BEST set of parameters. The result of the **BEST PERFORMANCE** is as follows:

```
========== PART 8 BEST PERFORMANCE ==========
Parameters: resolution=32x32, n_epochs=200,
batch_size=32, dim_h=300, learning_rate=0.001

Best Training accuracy: 1.0
Best Validation accuracy: 0.880341880342
Best Test accuracy: 0.871794871795
```

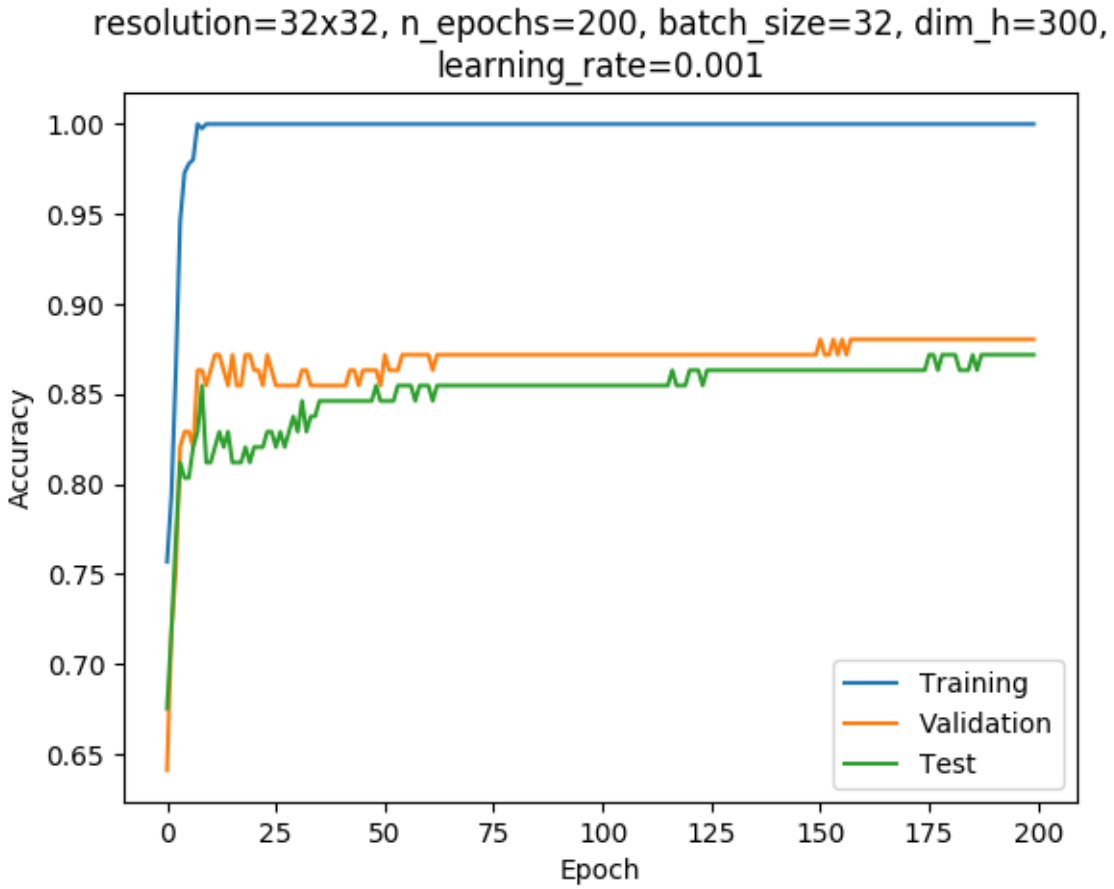

Figure 14: Learning curve for the training, validation and test sets

Part 9

In this part, we select Peri Gilpin and Bill Hader to experiment with how neural networks see the data. To reconstruct the neural network from part 8, we load the weights $W0$ and bias $b0$, which we saved as file *part8_nn.npy* in part 8. They are weights and bias that go into the hidden layer respectively. The value of activation of a hidden unit on a certain actor is the way we will decide whether the hidden unit is useful for classifying this actor. This is because only a firing neuron, and especially with the highest activations pointing at the target, is helpful for classifying the input. To get all activations, we use $W0$ and $b0$ to perform a single forward propagation and compute activations using $ReLU$. Then we only look at the hidden units that contribute to the actor we are experimenting on, i.e., we slice the part of activations matrix where the actor's index is all 1s in the target. Then we pick 3 neurons that produce the highest 3 activations. They are the top 3 hidden units that are most useful for classifying input photos for the actor.
To visualize the weights of these hidden units, we first reshape the corresponding weights into $32 \times 32 \times 3$ dimension and separate them into R, G and B channels. Then we add three channels together and reshape the sum into a 32×32 image. The visualizations are as follow:

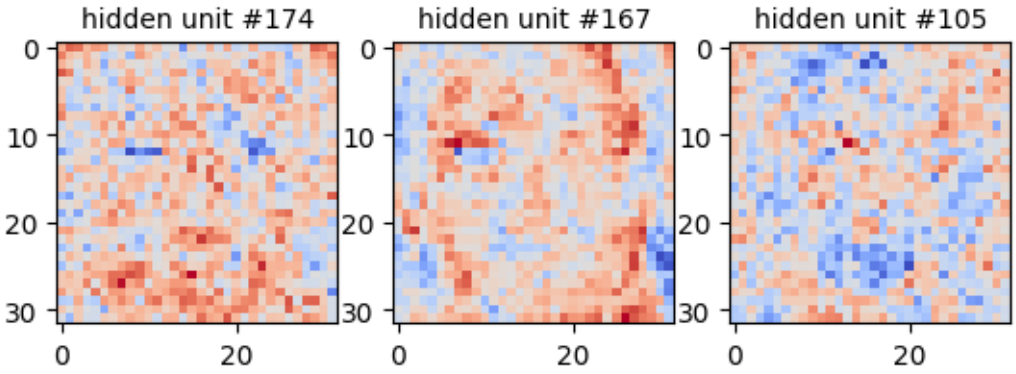

Figure 15: Top 3 hidden units useful for classifying Peri Gilpin

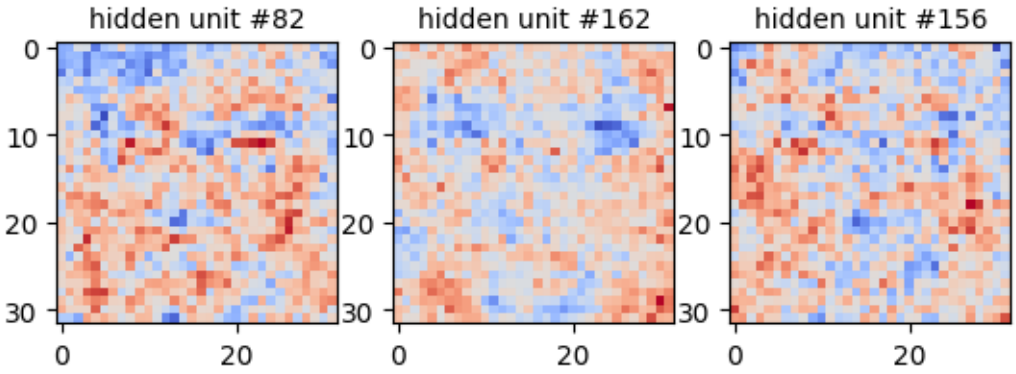

Figure 16: Top 3 hidden units useful for classifying Bill Hader

Part 10

Load the Data
In this part, we first load all data from $faces\_all\_227.mat$ (where all images are of $227 \times 227 \times 3$ dimension). Then we call *get_train_alexnet*, *get_validation_alexnet* and *get_test_alexnet* to get three separate datasets as numpy arrays. Note that unlike part 8, we do not flatten the data here because Alexnet takes 4D input with dimension $(n, 3, 227, 227)$. We also normalize the data by their mean and roll the axis to fit Alexnet's input dimension.

Extract activations from Conv4 layer of Alexnet
After getting ready for the input, we extract the values of the activations of AlexNet in Conv4 layer. For each training example, we use $model.features()$ function to propagate it to Conv4 layer. The output of this function will be the activation we need for this training example. After extracting activations for all training examples, we concatenate them in a numpy array. Note that the array is of dimension $256 \times 6 \times 6$.

Learn a fully connected neural network using Alexnet's activations
The Conv4 layer is the last convolution layer in MyAlexNet. Before passing into its default classifier, we would like to use the outputs of Conv4 to train our own neural network and classify the actor images. In order to do that, we replace the classfier layers in MyAlexNet with our own network. We also have to modify the parameters and dimensions of our own network so that our network is fully connected to MyAlexNet after Conv4 layer.
In *part_10_helper*, we build a single layer network similar to part 8, but with different input dimensions. In order to feed those activations as features into our own fully connected neural network, we need to construct a Linear Layer to transform the inputs from dimension of $256 \times 6 \times 6$ to dimension of *dim_h*. The output layer is the same as part 8, which consists of 6 units. The weights are initialized in a similar fashion as part 8 to produce a good performance. Again, we use mini-batches to train our network. The parameters passed into the network are extracted from part 8, so that we can easily comapre the performance between them.

Listing 7: Part10 Neural network construction

```
model = torch.nn.Sequential(
    torch.nn.Linear(CONV4_DIM, dim_h),
    torch.nn.ReLU(),
    torch.nn.Linear(dim_h, NUM_LABELS),
)

model.apply(init_weights)

loss_fn = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Listing 8: Weights initialization

```
def init_weights(m):
    if type(m) == torch.nn.Linear:
        m.weight.data.normal_(0.0, 0.03)
```

The learning curve is as follows:

```
Training accuracy: 1.0
Validation accuracy: 0.923076923077
Test accuracy: 0.931623931624
```

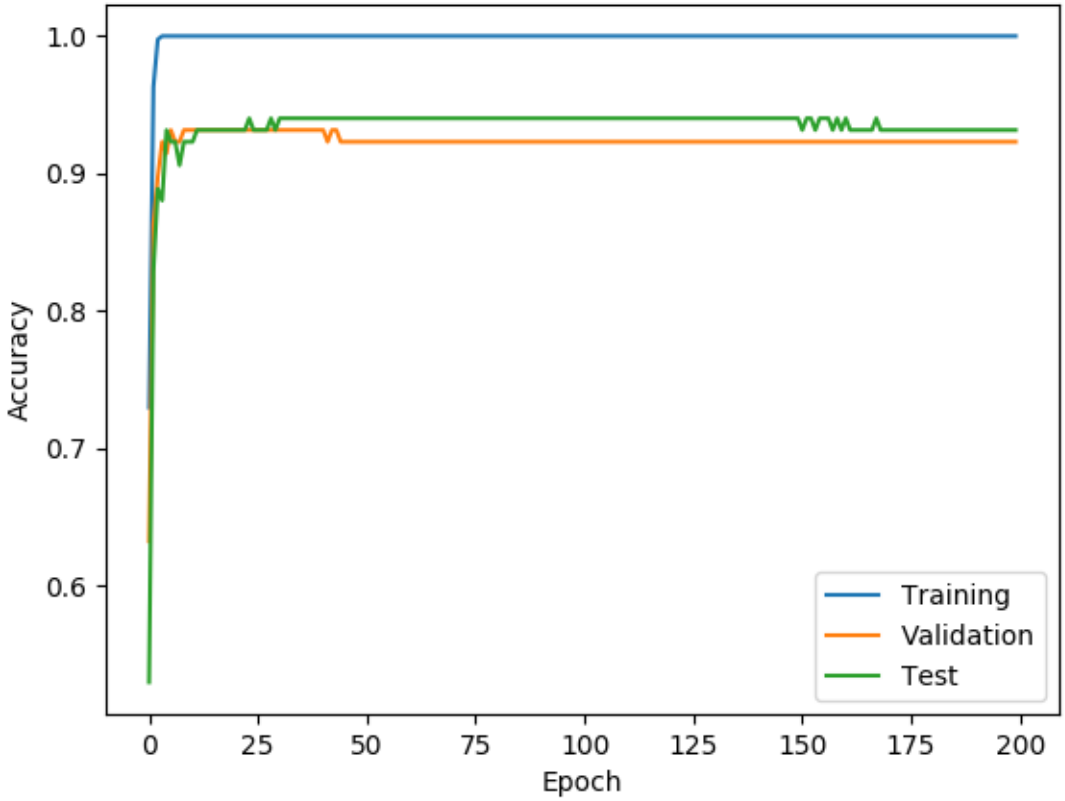

Figure 17: Learning curve for the training, validation and test sets

The error rate on test set of part 8 is approxim $1 - 0.87 = 0.13$. The error rate on test set of part 10 is approxim $1 - 0.93 = 0.07$. The improvement on error rate from part 8 to part 10 is $(0.13 - 0.07)/0.13 = 0.46$. Thus the error rate is reduced by almost 46%, which is a huge improvement.