Michael Hering
CSCI 3202

Assignment 5 Write up

Part 1: Data

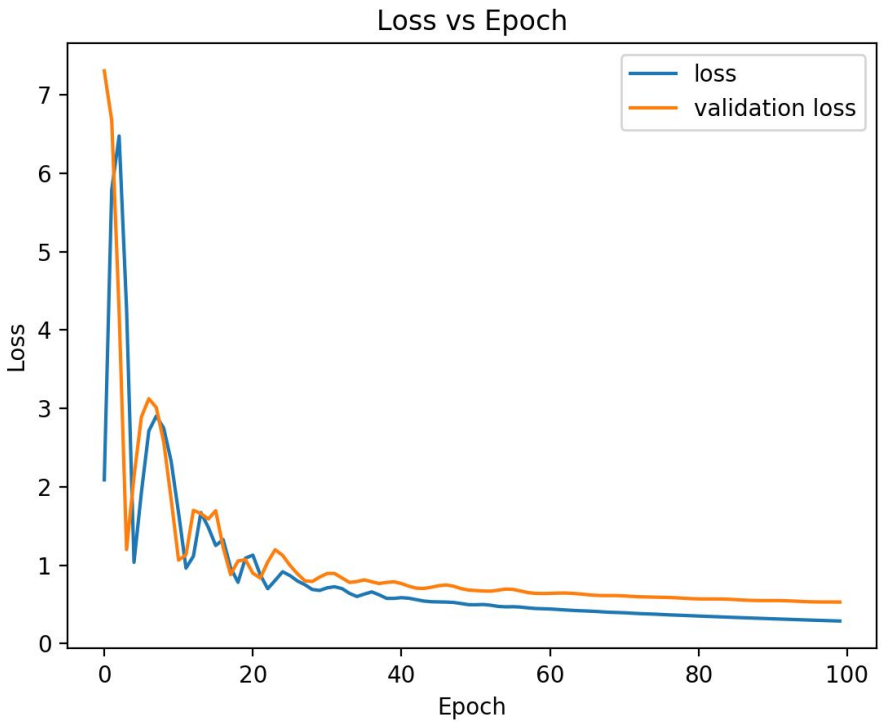
The dataset of faces is fairly accurate, and the provided bounding boxes do a good job of providing accurate crop boxes for the faces in the images. To improve training, we do some preprocessing of the images. We first crop the images so that we are only training our models on the areas of the photos where the faces “live”. This prevents us from training the images based on the backgrounds of the photos. Then, we resize the images to a standard, smaller, size of 60x60 pixels. This does two things for us. First, all of our data points (image) will be the same size and therefore our model will be learning with data that confirms to one shape which should speed up training. Second, by shrinking the images we are removing noise from the images while also keeping the important data that allows us to describe a face.

Here are 3 examples where the bounding boxes succeeded in bounding the face within an image:

Here are 2 examples where the bounding boxes failed to bound the face within an image (there are many of these), and one example where the face was obstructed by an obscure watermark:

Part 2: Deep Learning

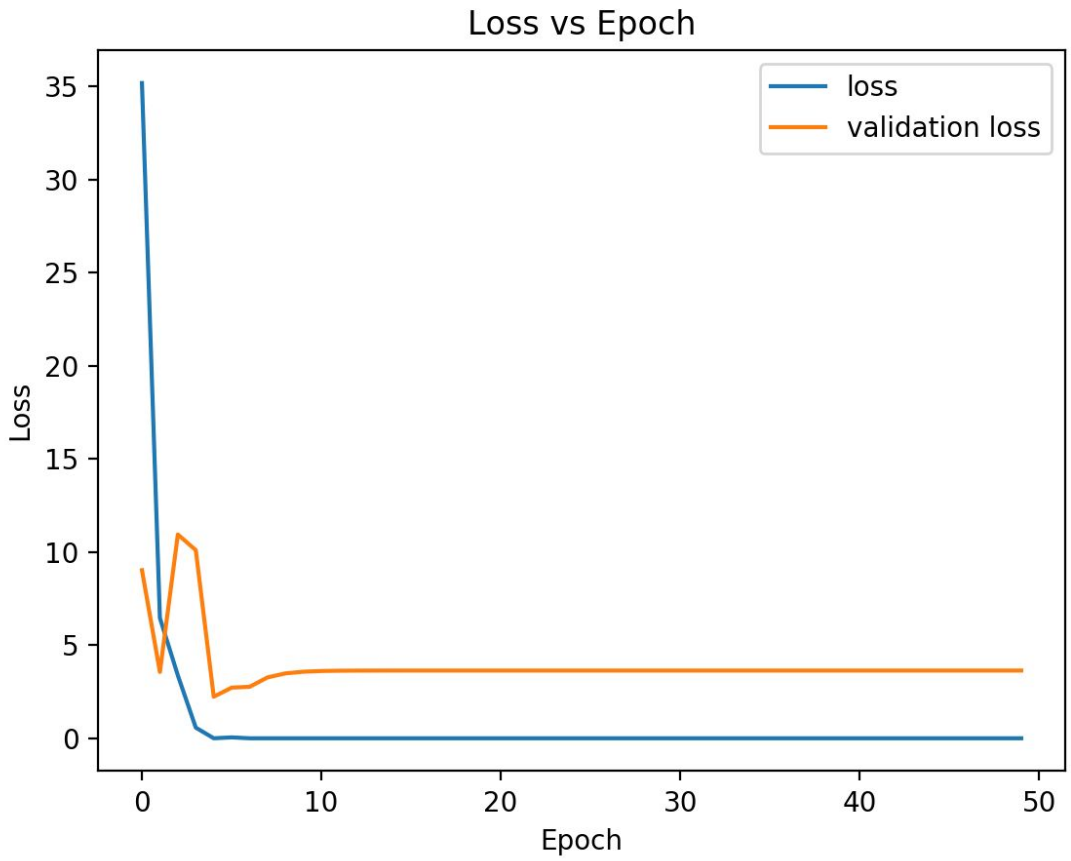
Before I trained my model, I split the data into train and test sets, using a test holdout of ⅓, and a validation holdout of ⅓ of the training data. I used a fully connected network with one hidden layer of 256 units, randomly initialized, and applied a ReLU activation function. Below is the graph of the loss through training the model, using mini-batch gradient descent with a batch size of 128 and 100 epochs:



Test set accuracy: 0.86

Part 3: Transfer Learning

I chose to use the ‘block5_pool’ layer of the VGG16 network as features for input to my classifier because it is a smaller representation which means the network that I train will be less complex. I used these features as input to a fully connected network with one hidden layer of 256 units, randomly initialized, and applied a ReLU activation function. Below is the graph of the loss through training the model, using mini-batch gradient descent with a batch size of 16 and 20 epochs:



Test set accuracy: 0.90

We can notice that as the training continues past 10 epochs, we seem to have settled into a local minimum and are unable to continue to decrease the loss.

This network performed much better on the test set than the network using raw images from above. I think this is because the network was learning based on features that were

extracted layers from a well-developed and trained CNN which was fed our images. These features can be more meaningful in training than using the actual raw images themselves.

Part 4: Experimentation

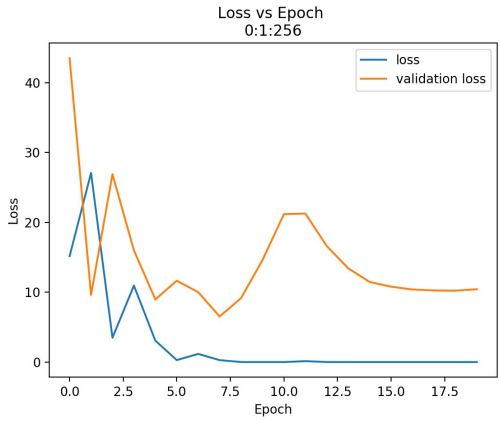
Using my model from part 3, I experimented with adding dropout, varying the number of hidden layers, and varying the number of nodes in each layer. I trained __ models, each with a unique combination from the following sets of hyperparameters:

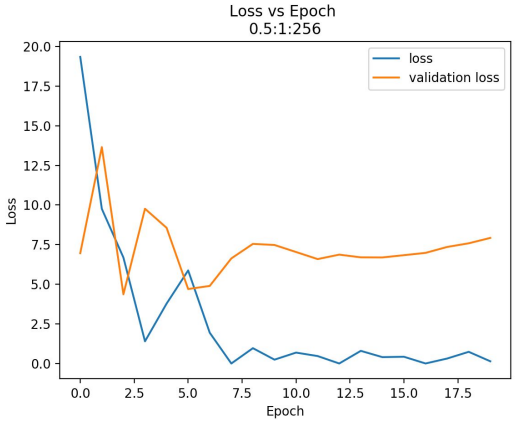
Dropout: [0, 0.5, 0.7]

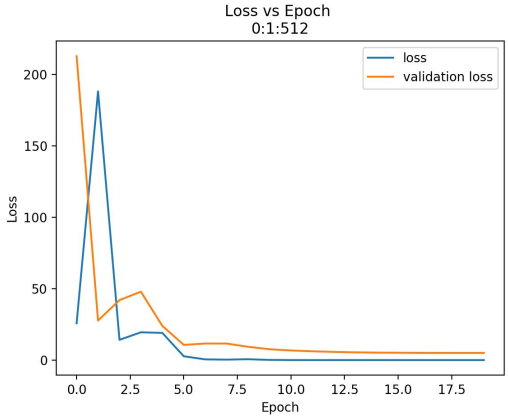
Number of hidden layers: [1, 2, 3]

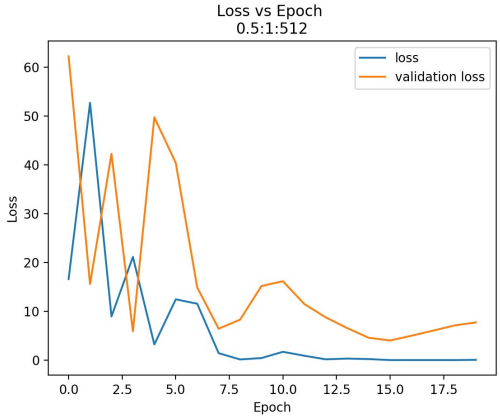
Number of nodes in each layer: [256, 512, 1024]

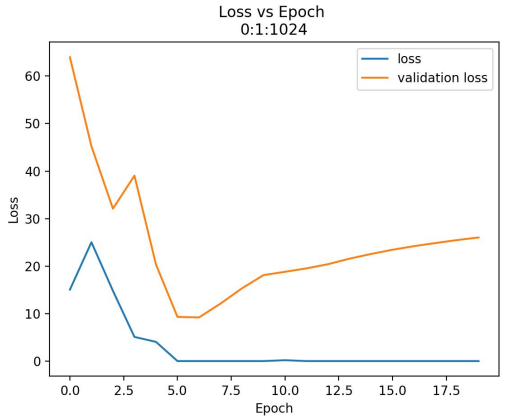
I then plotted the loss graphs and recorded the test accuracies for each classifier. Here are my results: `<dropout rate>:<number of layers>:<number of nodes in each layer>`

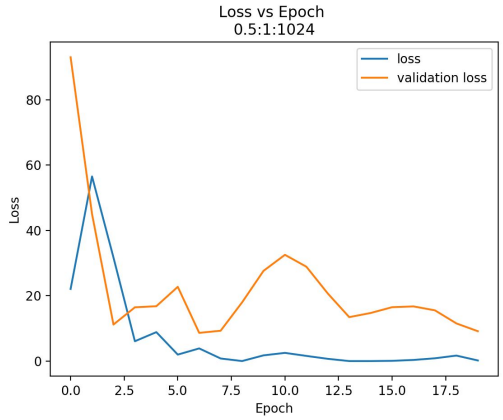


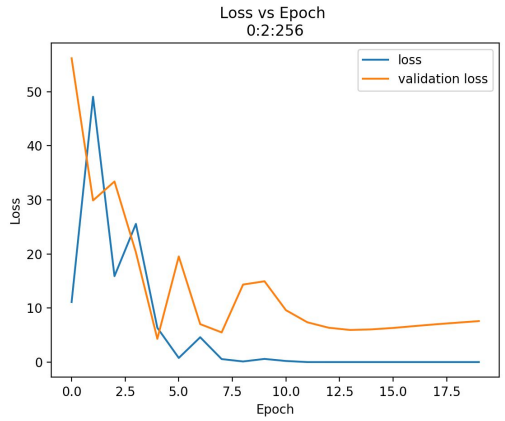


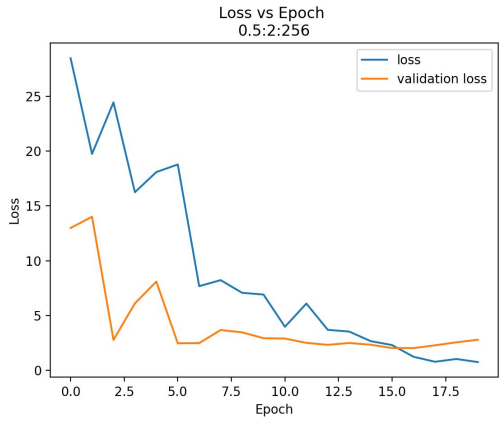


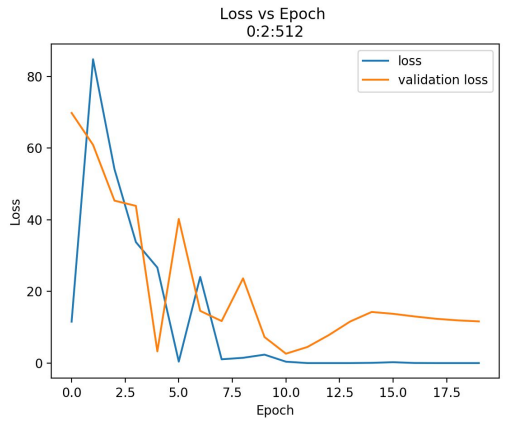


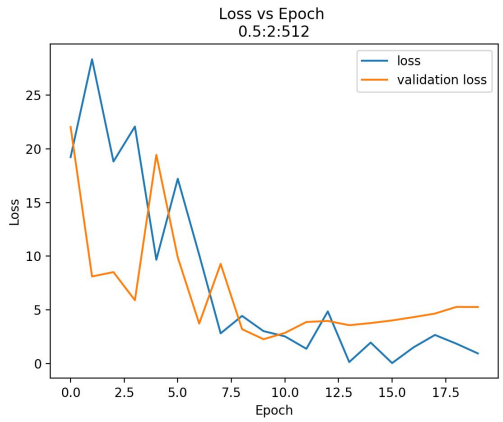

Loss vs Epoch
0:1:1024
loss
validation loss
Loss
Epoch

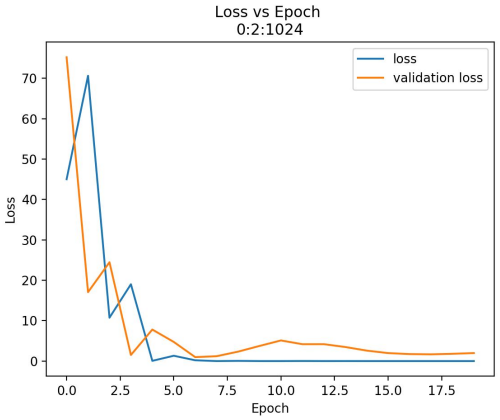

Loss vs Epoch
0.5:1:1024
loss
validation loss
Loss
Epoch

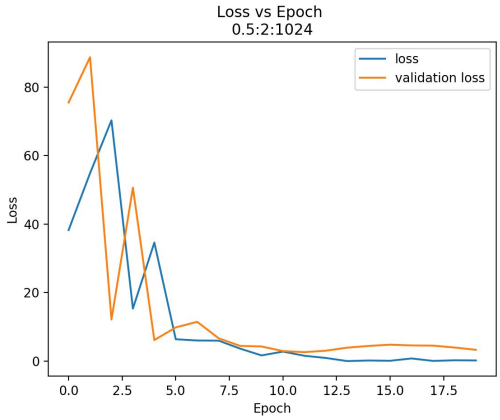

Loss vs Epoch
0:2:256
loss
validation loss
Loss
Epoch


Loss vs Epoch
0.5:2:256
loss
validation loss
Loss
Epoch


Loss vs Epoch
0:2:512
loss
validation loss
Loss
Epoch


Loss vs Epoch
0.5:2:512
loss
validation loss
Loss
Epoch





0:1:256 accuracy: 0.7570093274116516
0:1:512 accuracy: 0.8504672646522522
0:1:1024 accuracy: 0.8878504633903503
0:2:256 accuracy: 0.84112149477005
0:2:512 accuracy: 0.7663551568984985
0:2:1024 accuracy: 0.8504672646522522
0.5:1:256 accuracy: 0.8878504633903503
0.5:1:512 accuracy: 0.7943925261497498
0.5:1:1024 accuracy: 0.8317757248878479
0.5:2:256 accuracy: 0.9158878326416016
0.5:2:512 accuracy: 0.8971962332725525
0.5:2:1024 accuracy: 0.8785046935081482

To experiment with my hyperparameters, I set up 3 sets of values for the dropout rate, number of layers, and number of nodes in each layer. I then trained a model with every combination of these parameters and evaluated the results. In all of my training, I used mini-batch gradient descent with a batch size of 128 and 20 epochs. My best model was 0.5:2:256 and was able to achieve an accuracy of 0.92 using a fully connected network with 2 hidden layers of 256 nodes and a dropout rate of 0.5. Applying dropout generally reduced the validation loss during training, but not in all cases. There were also some cases where training should have been stopped earlier than 20 epochs, as we began to overfit our data and validation loss began to rise again, but with the 0.5:2:256 model we stopped training at a better time, which could explain for it's increased accuracy. This model is more efficient than many of the other models, that use the same amount of layers and more nodes in the network, resulting in a network that has many more weights and is much more complex. I was able to achieve a boost in performance compared to my best model from parts 2&3, which I think in large is due to dropout normalization and the addition of a second hidden layer, which could allow for some more complex data transformations.