Oliver Breese & Phoebe Payba

Professor Wong

CS 4100

10 December 2021

Final Report

Introduction

For our project, we want to be able to determine if two people are blood-related given several images of their faces. To this end, we implement and train a Siamese neural network to classify two images as related or not related. A more detailed description of the problem we are attempting to solve can be found here: https://www.kaggle.com/c/recognizing-faces-in-the-wild.

We were drawn to this problem because we were originally interested in researching and implementing a neural network. After looking through potential ideas on Kaggle, this project first stood out to us because it was hosted by Northeastern's SMILE Lab. Given our lack of prior experience with implementing a neural net, we felt this would be an appropriate problem for us to tackle in terms of scope and complexity.

Problem Statement

Our problem is simple. From a dataset of potentially blood-related face images, we want to develop a tool which accurately classifies any two people as related or not related based on their faces. Given two facial images, the output should be a binary value representing whether the two people in the images are related (1) or not related (0).

Dataset:

We used a dataset provided by Kaggle for the given problem. This dataset contains several files for training and testing data. The files contained images associated with different people, and labels dictating if two images/people are related or not. For training, the options were *train_faces*, *train* and *train_relationships*. For testing, the dataset contained the files *test-private-faces, test-private-lists,* and *test-private-labels.* Initially, we decided to use *train* and *train_relationships* because we needed the relationship labels provided in *train_relationships* to determine which people in *train* were blood-related. However, as we began training our model, we eventually found that these files contained an insufficient amount of data. To remedy this, we actually used a subset of the testing data for training, because there was significantly more test data provided than training data.

Algorithm

Siamese Network Overview:

At its core, siamese network is a term for simple CNNs which compare two images, rather than classify them into output classes. Most people's first experience with CNNs (and ours in fundies) was with the MNIST dataset where we have precisely 10 output classes, with thousands of training images for each. In real life, though, this is neither how humans learn image comparison, and we almost never have this much data.

In our case, imagine we were just trying to identify people's images with themselves. In a traditional CNN, we would need an output class for every person, and for the model to learn each person, we would need far more than the 5-10 images for each that we have. Siamese networks generate a similarity score, rather than a classification of each image, so that fixes our output class issue. Additionally, they are capable of "one-shot learning", or learning classification with very few examples.

What's the specific difference? Both CNNs and Siamese networks have very similar methods for extracting numerical features from images. The "convolutional" in a CNN is a series of layers that convert pixel values to a usable array of values and weights. It's "filter" is deliberately smaller than the image size so it can learn a mapping of weights to areas of the picture that might be important or not, rather than huge arrays of values for individual pixels, as the latter is largely impossible to learn patterns on without extensive data cleaning (like the centering discussed in class). For an example of this, imagine a picture of a face that's not completely centered. A vector of individual pixel values will see this as a completely different case, while a convolutional layer is designed to avoid that issue. Also identical to the normal CNN, a pooling layer reduces the size of the convolutional layer to ease computing by decreasing the total number of weights. There are often multiple sets of these depending on the size of the images. The main difference is that the steps above (along with almost all dense layers used for learning) are done twice, in two identical networks. Each image is passed through one. Two more layers are added, one to quantify the differences of the output of the dense layers (called the "differencing layer"), and one to produce a single value out of these two that is their "similarity". There are many variations on Siamese networks and CNNs, but these principles generally hold true.

Overall, this kind of network applies perfectly to our purposes, so we chose it as our path for all of the above reasons.

In terms of our network architecture, we took inspiration from a few sources, and found what worked and what didn't for our purposes. Without describing each change we made along the way, the following is an explanation of the purpose of each layer as it exists in the final product.

```
class SiameseNetwork(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()

        self.cnn1 = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(3, 64, kernel_size=3),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(64),
            nn.Dropout2d(p=0.2),
            nn.MaxPool2d(2, stride=2),

            nn.ReflectionPad2d(1),
            nn.Conv2d(64, 128, kernel_size=3),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(128),
            nn.Dropout2d(p=0.2),
            nn.MaxPool2d(2, stride=2),

            nn.ReflectionPad2d(1),
            nn.Conv2d(128, 256, kernel_size=3),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(256),
            nn.Dropout2d(p=0.2),

        )

        self.fc1 = nn.Linear(2*32*32*256, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 2)

    def forward(self, input1, input2):
        output1 = self.cnn1(input1)
        output1 = output1.view(output1.size()[0], -1)
        output2 = self.cnn1(input2)
        output2 = output2.view(output2.size()[0], -1)

        output = torch.cat((output1, output2),1)
        output = F.relu(self.fc1(output))
        output = F.relu(self.fc2(output))
        output = self.fc3(output)

        return output
```

The first important layer is the convolution layer present in all CNNs, and a near-universal aspect of any image-processing neural network. Mapping pixels to nodes is almost never smart, efficient, or particularly good at generalizing. It's a huge mapping, it requires a huge amount of data, extremely good data regularization[OB2] , or both. It also struggles to recognize patterns like shapes, lines, or edges because it simply doesn't have the required information. A convolution layer maps small, oftentimes overlapping sections to the next layer, In torch, this is specified as kernel size. We chose a 3x3 kernel, with a standard "step-size" of 1 (which means the subsection moves 1 space horizontally or vertically, and overlaps with the previous subsection [OB3] by 2 pixels. We chose three for the kernel, because SOURCE claimed it was standard, and we saw only examples of 3 or 5, so this seemed good enough for our purposes.

The "Reflection Pad" layer is simple. Since the convolution [OB4] layer has a 3x3 sized kernel, it will shrink the output images by 1 on each side. The aggregate of the mapping for each subsection in the convolution layer is now mapped to the center of the 3x3 kernel, so we lose the edges, hence 1 pixel on each side. The reflection pad simply adds an extra row or column to each side to "pad" the image so it comes out the same size. Some siamese networks use this, some don't, we thought it was easiest to think about image sizes if the convolution kept them the same, so we chose to include the padding.

Traditionally, and in class, many binary classification problems use a sigmoid function, but we almost never do in CNNs. Due mostly in part to the high dimensionality and low independence of visual inputs, certain nodes might be relevant in some examples of a class and irrelevant in others, the presence or lack of a certain neuron's activation might not "negatively contribute to the output of the neural network" (Baeldung). In short, this means we use Rectified Linear activation, which is strictly positive. The ReLU layer applies this to every node in a layer without changing its dimensionality (as is standard for activation functions).

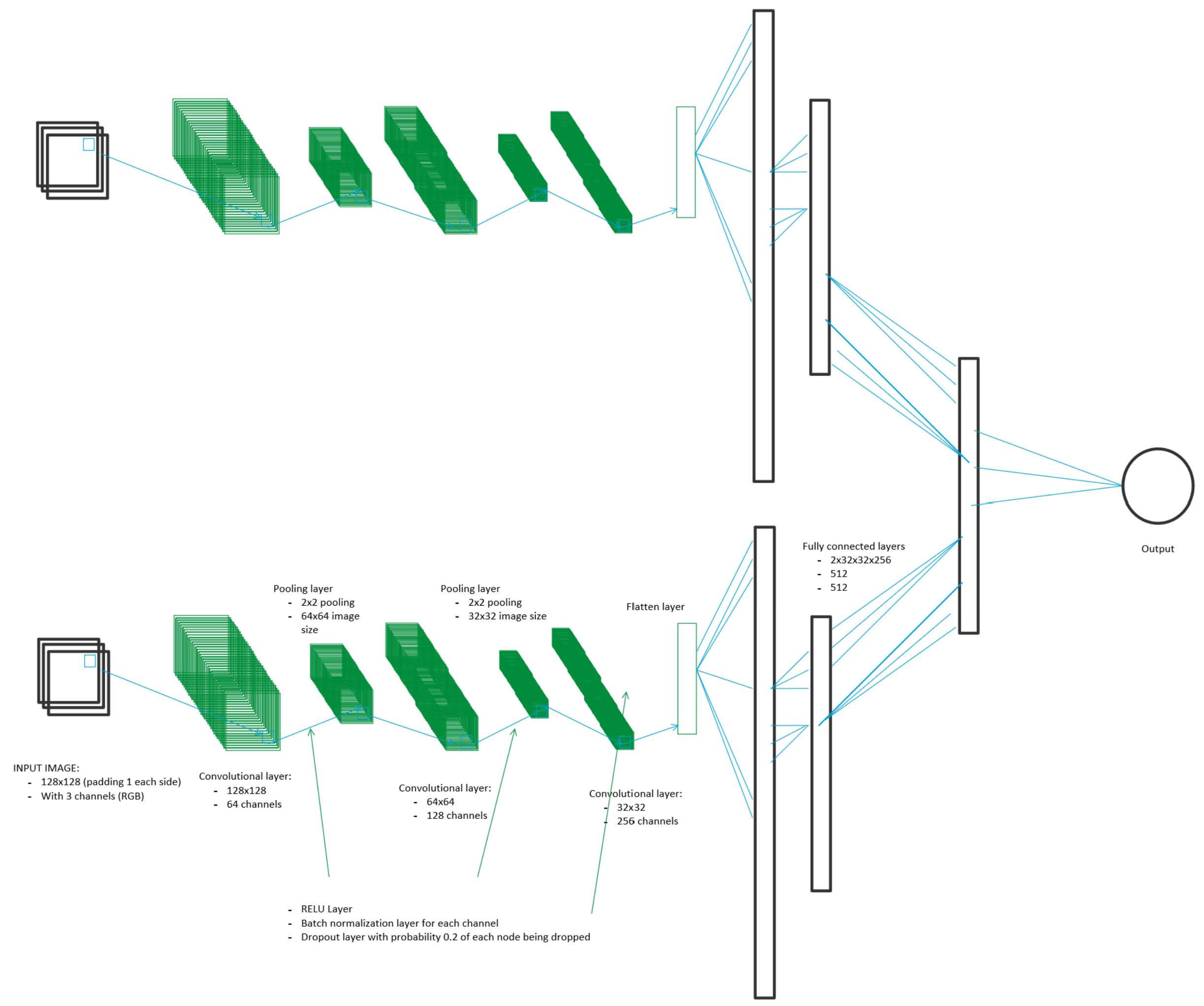
The Batch normalization layer is a way to standardize inputs to each layer to increase the speed of learning. Our network, and every practical siamese network we saw in our research, is quite large. This means it both needs a large amount of data and many epochs to learn weights. The goal of batch normalization is to decrease the number of epochs, which helped us be able to run the model on a reasonably sized GPU in a reasonable amount of time. Specifically, the layer rescales the inputs before activation to a "standard gaussian" distribution, where the mean is 0 and the standard deviation is 1.

The Dropout layer is probably the weirdest one. We saw this show up a few places, and disregarded it at first, because we didn't understand its purpose yet. After enough training, we began to grow concerned about overfitting (while we were still using the original data, as there were several instances of each person. In searching for regularization techniques, the idea of dropout intrigued us, because we haven't touched on it in class or the homework like L1 or L2 regularization. The idea behind it is that we stochastically remove nodes by some tuned probability to simulate sparse input. This in turn makes identical data look different, and should help the network generalize to fully unseen data better. After more investigation, and switching our data source, we don't believe we have an overfitting problem, but performance with and without the dropout layer was extremely similar, so we left it in just in case we want to train on more, or especially less random data.

We separated the flatten and dense layers from those previously mentioned because we had some trouble with "flatten" in Torch. We noticed one of our sources utilize this tensor[OB5] .view method, and upon further research realized it computes the same operation. The dense layers are self-explanatory, they are fully connected and of sizes specified in the diagram and in the code above.

We pass both images individually through the CNN (as both must share weights, so we should simply use the same instance), pass those outputs through the first two dense layers, then that output tensor to the final differencing layer, which outputs a single value representing the probability of these images being members of the same class (related).

Final Network:



Loss Function:

The loss function we decided to use for the final version of our model was PyTorch's built-in implementation of the cross-entropy loss function:

```
criterion = nn.CrossEntropyLoss()
```

This loss function is especially useful when training a classification model; in our case, we are performing binary classification. Cross-entropy loss is calculated using the difference between the true probability distribution and the predicted probability distribution of a given label. PyTorch does provide many other options for loss functions (CTCLoss, SoftMarginLoss, etc.), but we felt that cross-entropy was the most suitable for our model.

Optimizer:

The first optimizer we tried while training our network was PyTorch's built-in implementation of stochastic gradient descent (SGD):

```
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

SGD is a variant of the gradient descent algorithm that reduces total computations by randomly sampling data at each iteration rather than using every term at each iteration, which tends to be inefficient. Gradient descent works to minimize loss by iteratively following the slope of the loss function in the direction of its lowest point until the minimum is reached. We used a small learning rate of 0.001. The momentum parameter is used to accelerate SGD and gain faster convergence. It does this by adding a fraction of the update vector of the past time step to the current update vector, where the fraction increases when gradients point in the same directions and decreases when gradients change directions. Using SGD, we found the amount of fluctuation in the minimization of our loss function to be less than ideal, so we began looking into other potential optimization algorithms.

The next optimizer we tested was PyTorch's built-in implementation of the adaptive movement estimation (Adam) algorithm:

```
optimizer = optim.Adam(net.parameters())
```

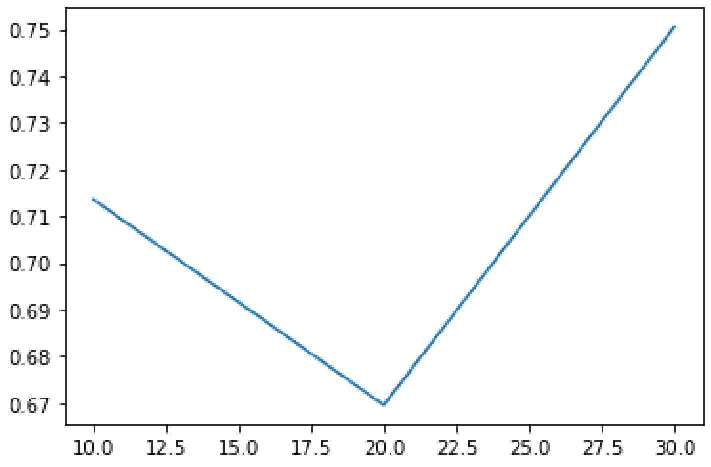
Adam is an extension of SGD that, unlike SGD which maintains a single static learning rate, maintains and updates a different learning rate for each parameter. The algorithm updates learning rates based on the exponentially decaying averages of both the past gradients and the past squared gradients. Adam generally converges faster than SGD. We found that Adam decreased the loss function in a more stable pattern while training our model and actually decreased the training time overall. PyTorch does offer several other variations of the Adam algorithm (AdamW, NAdam, Adamax, etc.) but we did not feel that any of the variations would make a significant difference. We decided to continue using Adam as our optimization algorithm in the final version of our model for its computational efficiency and stable minimization of our loss function.

Empirical Results

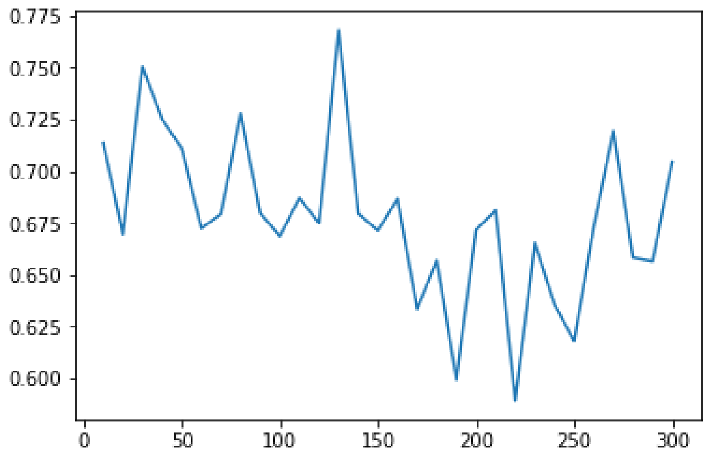
First Iteration:

The following output is our first attempt at a running neural network. It was trained on half the data for 10 epochs, using stochastic gradient descent as its optimizer. Notice how the loss function fluctuates to extremes (like we saw in class with the MNIST dataset). After some research, we found Adam, which works as an extension to SGD. This one was suggested many places online for its efficiency and ability to fix the kinds of fluctuation issues we saw.

Epoch 0 start.



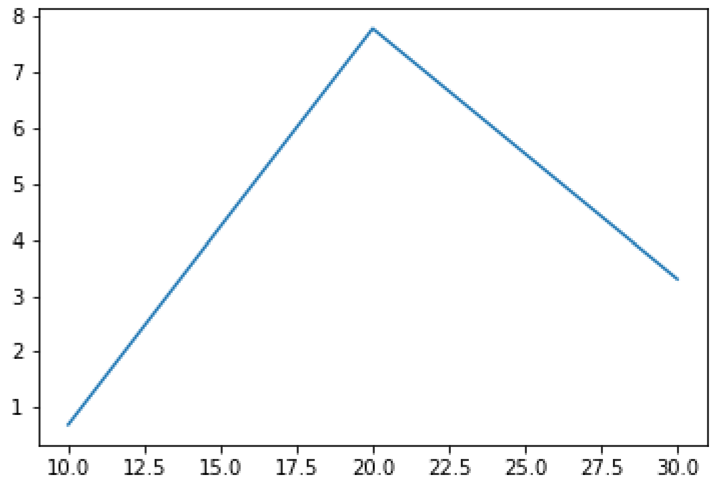
Epoch 9 start.



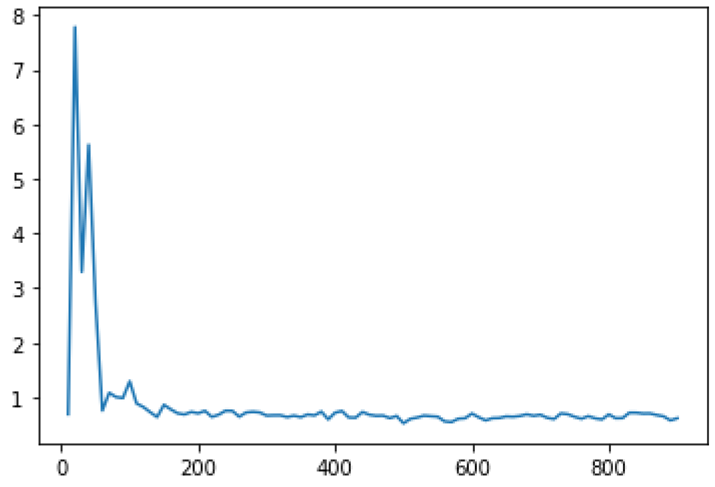
Second Iteration:

With the ADAM optimizer and the accuracy calculation working, we got the loss function to decrease in a stable manner, and go from 0.49 (random, as we feed half the data as related people and half the data as unrelated people) to 0.63, which though not great, is certainly learning something. This was computed on half the available data (because our K-fold cross validation was not making a computational difference yet), and with only 30 epochs. The examples on Kaggle we observed ranged from 67%-80% accurate, so the low 63% with little data and as a first real attempt isn't too worrying.

```
Epoch:  0  start.
Accuracy of the network on the 1734 val pairs in k group  0 : 49 %
```



```
Epoch:  29  start.
Accuracy of the network on the 1734 val pairs in k group  0 : 63 %
```

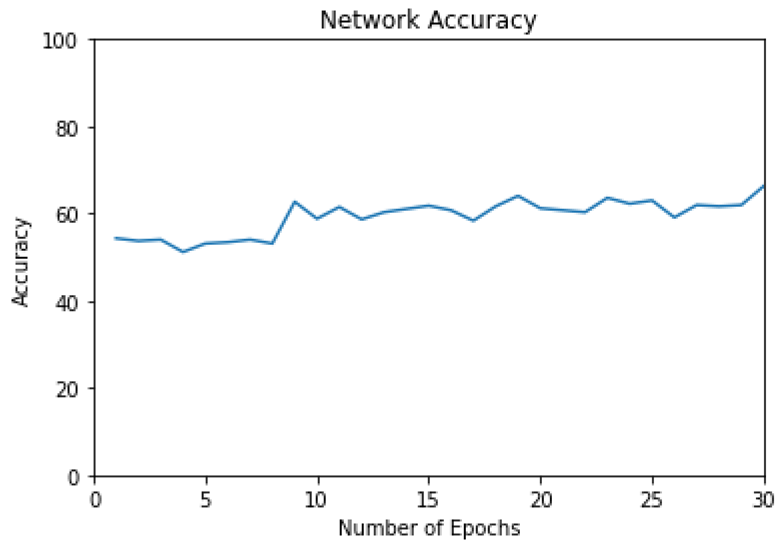


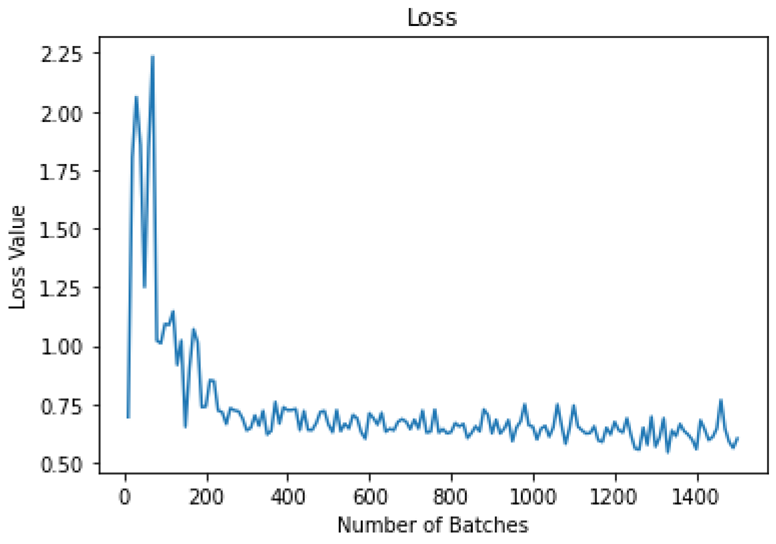
Third Iteration:

In the third attempt, we increased K to 4 (probably where it will stay, as we only have three thousand data samples so breaking it up much smaller might be too small of a validation set). We also used K to evaluate an average of each model's accuracy. This got an average accuracy of 60%, which makes sense because the sample size is far smaller. K-fold cross-validation makes the individual models less accurate on the validation data, but hopefully more accurate on the test data. We didn't save the output for this particular run, and since there have been significant structural changes since this iteration, our word will have to be good enough for this one.

Fourth Iteration:

In the fourth attempt, we implemented the graphing of accuracy per epoch in addition to the loss function. We also began testing the network on test data, not simply evaluating the validation accuracy. We also ran K-fold cross validation with K=4 for the remaining iteration, but found no folds as outliers, so we're fairly confident our model is not being passed data in a suspicious order. This makes sense, because we are shuffling both the images and relationships data and sampling it randomly. The following is the last epoch of our fourth attempt, which achieved an accuracy of 66% on test data utilizing all the training data.





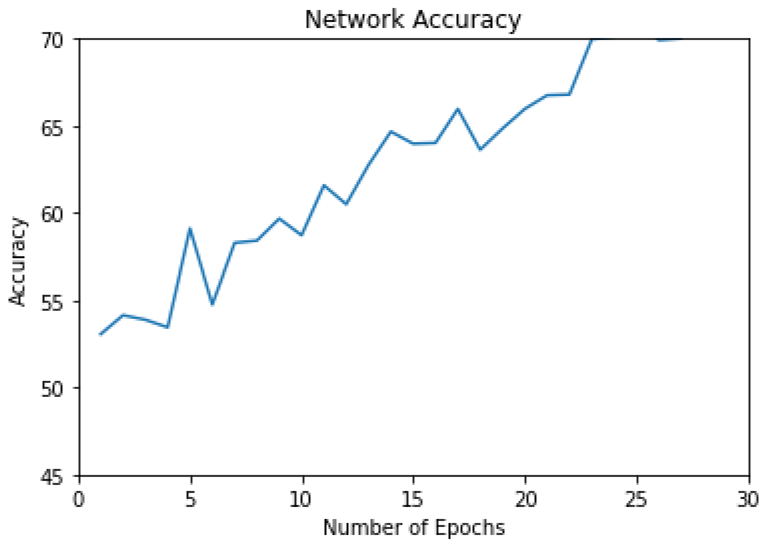
```
Accuracy: 66.26686656671664 %
```

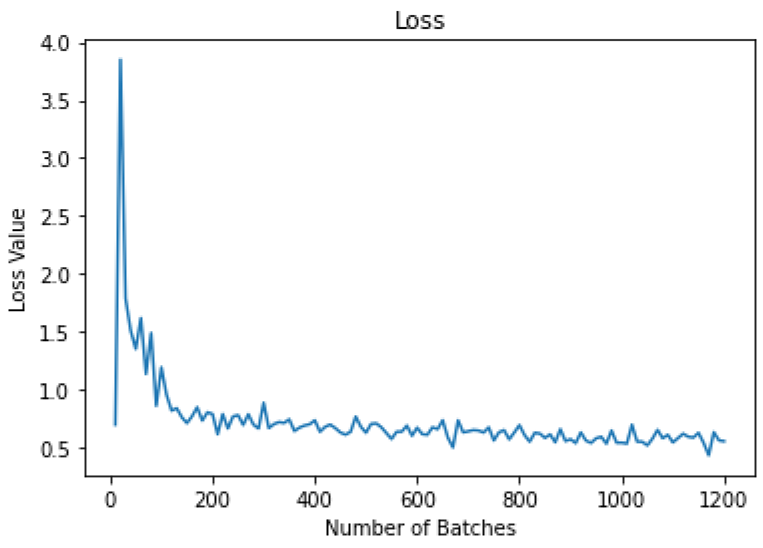
Fifth Iteration:

After some consideration, we decided to include our fifth attempt because while it was a major misstep, it was a vital part of our process. As one can see from our graph, we didn't even expect this model to improve beyond 70% accuracy based on what we had seen from our own model and from the majority of models implemented for this kaggle competition. Short aside: the highest we saw for any implementation was 80%, but this was far and above the best, so our goal was anywhere form 60-70 to be in the typical range and prove that at least our network is learning to a statistically significant degree. All that said, you can imagine our surprise at the model not only achieving, but maintaining a 72% accuracy for the last 7 epochs. However, in class we recalled learning to always be skeptical of surprisingly good results, and upon further investigation it turns out we were letting the validation data bleed into the training data. We had just changed our data source from the given training set to a subset of the test data. The training and testing facial images were split properly, but our model trains on relationships, not on faces. Since we were randomly sampling from the same relationship data, there was obviously some overlap. The good news about this, is once we fixed it in the next iteration, we were much more confident in the result, because now we had seen what happens when there is leakage between training and validation sets, and we saw the accuracy decrease.

One more note about this iteration: as one would expect, this model generalized worse than a model with distinct training and validation sets. Since we saw a 72% accuracy on validation, but only 56% on test, this problem was easy to diagnose.

```
Accuracy of the network on the 2301 val pairs in k group  0 : 72 %
```



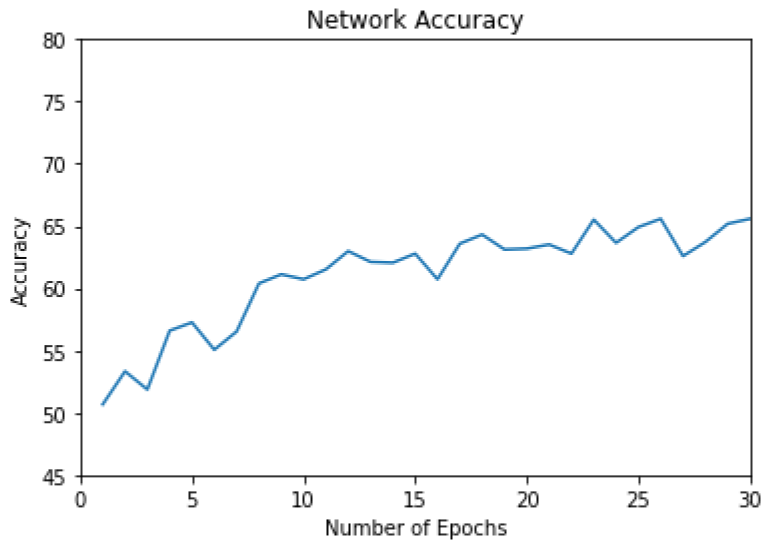


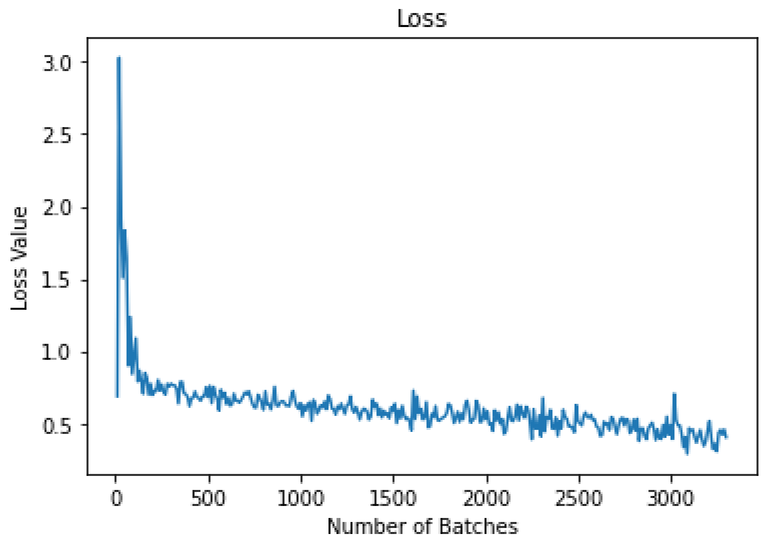
```
K-FOLD CROSS VALIDATION RESULTS FOR 0 FOLDS

--------------------------------------------------------------------------------

Fold 0: 72.2729248152977 %

Average: 72.2729248152977 %
```

Seventh and Final Iteration:

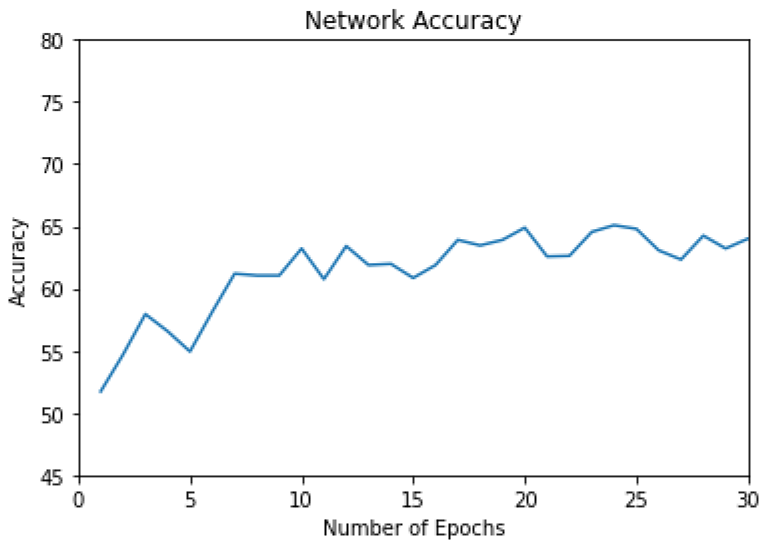
This is our final model. For it, we used a portion of the test data (and separated it from the test data to avoid our previous issue), rather than the given training data. The difference is that the original training data is organized by family and person, with several images per person. This means we have many images, but fewer relationships. In the test data, the reverse is true. We have individual images without classification for family or person, and over 50 thousand relationships. We were curious what would happen if we instead used the testing data, as there's far more of it. If in the future we were to get more data, it would be far easier to work from a master list of relationships and put all the images in the same list without classifying them. This model ended up, with a few tweaks, being similarly accurate to the fifth iteration when only using 20% of the available data. However, with all the K-folds, this took multiple hours to run, and any more would cause Google Colab to time out. In the future, if we had a better GPU or more time to use a weaker one, we could use all the available data, which should be far more accurate.

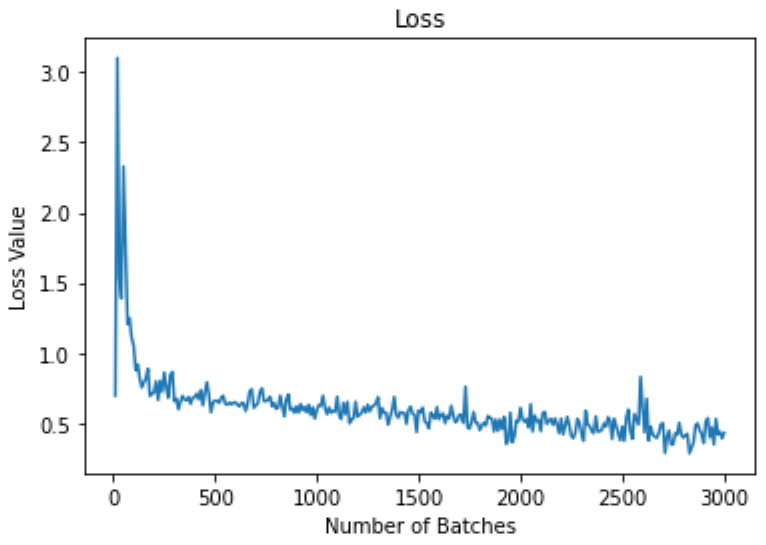
Below, first is the full model, and second is the K-fold cross-validation output.





```
K-FOLD CROSS VALIDATION RESULTS FOR 0 FOLDS
--------------------------------------------------------------------------------
Fold 0: 65.5792276964048 %

Average: 65.5792276964048 %


Accuracy of the network on the 2028 val pairs in k group  3 : 64 %
```





```
Accuracy of the network on the 2028 val pairs in k group  3 : 64 %

K-FOLD CROSS VALIDATION RESULTS FOR 4 FOLDS
--------------------------------
Fold 0: 61.94067370537959 %
Fold 1: 63.00050684237202 %
Fold 2: 64.33734939759036 %
Fold 3: 64.00394477317555 %

Average: 63.320618679629376 %
```

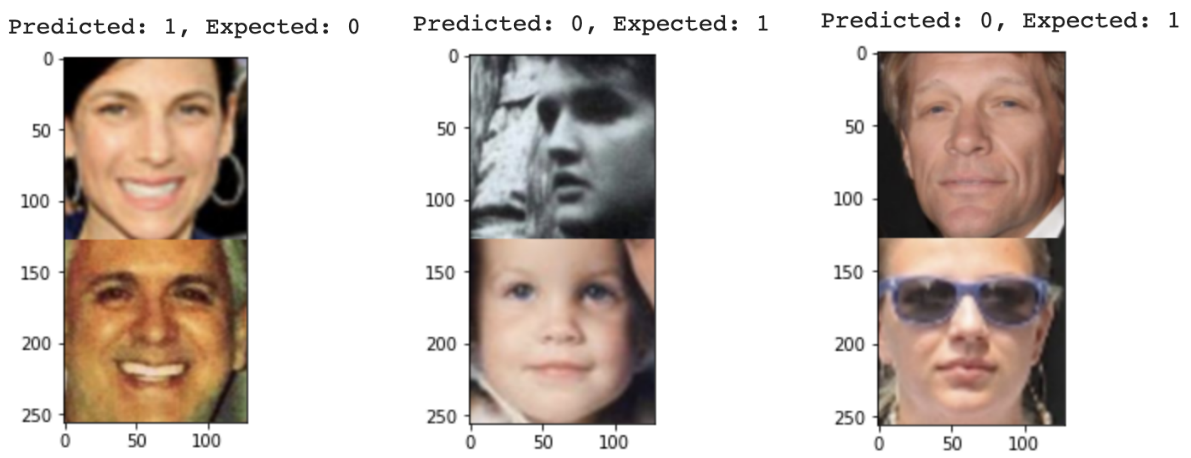
Analysis

In order to analyze our model's rates of error, we tracked the percentage of false positives and false negatives produced by our model on the testing data. A false positive indicates that our model incorrectly classified two images as related, and a false negative indicates that our model incorrectly classified two images as unrelated. The rates for our model's accuracy are as follows:

True Positive:	**70.02%**
True Negative:	**59.46%**
False Positive:	**40.54%**
False Negative:	**29.98%**

Our model was better at correctly identifying related individuals than unrelated individuals (70.02% vs. 59.46%), but it was also more likely to classify two unrelated images as related than classify two related images as unrelated (40.54% vs. 29.98%). Ideally, the rate of false positives and false negatives would be lower than our model yielded. This is something that future work on the project should aim to improve.

In the pursuit of deeper analysis, when the model made an error, we displayed the two incorrectly classified images with the predicted and expected values. Three examples are shown below, where the first pair was incorrectly classified as related and the other two pairs were incorrectly classified as unrelated.


Predicted: 1, Expected: 0
Predicted: 0, Expected: 1
Predicted: 0, Expected: 1
0
50
100
150
200
250
0
50
100

Compared to a dataset like MNIST, it was very difficult to find any pattern in our model's errors. The facial images in our testing data were so diverse in colors and features that we could not visually derive any specific characteristics that appear to influence our model in one direction or the other. With more time, a more comprehensive analysis could be performed.

Discussion

Moving forward, it may be worthwhile to examine the racial and ethnic makeup of the provided dataset and analyze whether any unequal representation exists. Any bias in the data could be a detriment to the reliability of our model. In a project like this, dealing with facial classification or recognition, it is important from an ethical standpoint to ensure that every demographic is properly represented. Otherwise, we run the risk of our model performing well on certain demographics and poorly on others.

A future variation of this project could expand upon the classification of blood-related faces and focus specifically on ancestor/descendent relationships. It could be interesting to implement a network that determines whether a person is a descendant of another person based on their facial images, effectively building a sort of ancestral tree.

One of the major difficulties we encountered over the course of this project was handling the data provided to us. We found the given dataset severely lacking in both the volume of data and the overall organization. Initially, we spent a significant amount of time just trying to figure out the file structure and the composition of each file. It was also a challenge to learn which files were actually meant to be used together, as there appeared to be several different sets of training and testing data. As previously discussed, we ended up using the designated testing data to train our model once we realized that the given training dataset was not even close to large enough.

If we had more time, we would have liked to utilize much more data to train our model. Although the testing data we used to train was more substantial than the provided training data, it was still less than we had hoped to use. Training with more data could potentially have improved our model's performance. With more time, we also could have tried training with a higher number of epochs. We had to reduce the number of epochs from 100 to 30 for the sake of saving time, as 100 epochs with multiple K-folds would have taken far too long to complete.

We would advise future CS4100 students to get started *early*. The end of the semester comes quicker than you'd think! Even if the majority of the work doesn't get done until the last few weeks, it is definitely beneficial to try and make small steps towards your goal as soon as you've decided on a project topic. This way, you have time to pivot if you encounter any roadblocks. We tried our best to follow our week-by-week plan, and in the end, it did keep us mostly on track with our project progress. Finally, don't be too intimidated. A project like this can seem overwhelming at first, especially when you're learning something completely new. We broke the problem down into smaller, more manageable tasks, which made our overarching goal feel much more achievable.

Sources

Conceptual Sources:

- https://www.baeldung.com/cs/ml-relu-dropout-layers
- https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/
- https://machinelearningmastery.com/k-fold-cross-validation/
- https://www.machinecurve.com/index.php/2021/02/03/how-to-use-k-fold-cross-validation-with-pytorch/
- https://towardsdatascience.com/8-simple-techniques-to-prevent-overfitting-4d443da2ef7d

Examples referenced:

- https://github.com/bainjamain/pytorch-examples/blob/master/notebooks/1_NeuralNetworks/9_siamese_nn.ipynb
- https://datahacker.rs/019-siamese-network-in-pytorch-with-application-to-face-similarity/
- https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148
- https://tianyusong.com/2017/11/16/pytorch-implementation%E2%80%8B-siamese-network/
- https://www.kaggle.com/jiangstein/a-very-simple-siamese-network-in-pytorch
- https://www.kaggle.com/ankiuci/modified-siamese-network-pytorch

Documentation Sources:

- https://pytorch.org/docs/stable/
- https://pandas.pydata.org/docs/