Northeastern SMILE Lab - <u>Recognizing Faces in the Wild</u>

Jadon Laforest - jjlaforest@wpi.edu
Oli Thode - omthode@wpi.edu
Timothy Goon - twgoon@wpi.edu

Introduction

Our project aimed to tackle Northeastern SMILE Lab's "Recognizing Faces in the Wild" Kaggle competition. This competition is focused on developing automatic kinship classifiers based on pairs of images, focused on the faces of the individuals. Our goal was to create a shallow and deep approach to this problem of kinship classification. The competition provided training data with around 1000 families, all with at least 2 family members, and at least one image of each member's face. From this data, we were able to create positive and negative examples of kinship- either by taking images from the same family or a different one- and train our models. Fully trained, our machines took in two images and outputted a probability that the two people pictured were related.

The problem this competition focuses on is something important to solve and interesting to study. Although it may seem straightforward to detect if someone is related, it is not something that is used in practice often. This is mainly due to three reasons that were outlined in the competition documentation: a lack of image databases that reflect true data distributions from families around the world, many hidden factors in familial facial relationships, and the need for more complicated models than current facial recognition and object classification can accomplish. It is also interesting how skilled humans are at recognizing faces, but how it still can be difficult for us to tell if two people are related. Developing a highly accurate machine could clue researchers in to what facial similarities or differences make kinship obvious.

Methods

Creating Training & Testing Sets

Before we really started testing some of our ideas and techniques out, we first had to develop our training and testing sets from the data we were given. The competition provided a huge folder of families with images of their members within each respective subfolder. It also came with a csv file with related members. Using these in tandem, we were able to cross each image in each member folder with the corresponding related member's images (according to the csv). This resulted in a massive set of positive examples, each of which was a pair of images. At this point, we decided to use around 10% of the examples, since the full set was over 40GB. To create our negative examples, we grabbed 2 random members from any family folder, checked to make sure they were not in the same family, and then crossed all their images if they were not related. We then combined the positive and negative examples into the training set. We used the same exact method for forming the testing set. We ensured that the number of positive and negative examples in our training set was exactly even, making our baseline approximately 50%.

Initial Data Manipulation

To make data preprocessing easier (and less resource heavy), we stacked each pair of images on top of eachother, so that only one image was needed for each example. This also ensured only one weight vector was used; one that spanned over both images. This led to an important step performed later, where we flipped the images to encourage symmetric discoveries.

Because some of the images in the set were already grayscale, we also converted all the RGB photos into grayscale. We also reduced the image resolution from the original 224x224 to 128x128 pixels. This made the data easier to work with, took up a lot less space, and made it faster to train.

Shallow Model: Softmax Regression

Our first thought for our shallow model was to implement softmax regression. We used sklearn's built in Logistic Regression method (found here) and fit our data to it. We found out very quickly that with just our initial data, this wouldn't work very well. The process quickly filled our RAM, trained very slowly, and didn't give us any promising results. On our first run we got about ~0.47 accuracy and an ROC of ~0.45 using the stacked images. From here, we knew we needed to change our problem and how we dealt with it.

Instead of training directly on the images, we instead looked into libraries that pulled out facial features, or face embeddings. We found a pre-trained facenet model by Hiroki Taniai that seemed to do exactly what we wanted. Now, instead of training on stacked pairs of face images, we started to train on stacked pairs of face embeddings. The performance instantly improved, both resource and results wise. Training did not suck up all our computer resources and it trained much faster than our initial trials. Our ROC and accuracy instantly shot up to around 0.55. We knew now that grabbing the features with Facenet was much better than training on the raw images themselves. We also saw that it was much faster and easier to train on too. At this point, we were curious to compare our results to the deeper model.

Deep Model: Dense Neural Network

Using the same data we used for the shallow model, we began working on our deep model. Using Keras, we created a 4 layer neural network (NN). We used ReLU for our hidden layer activation functions and softmax for the final layer. We used the Adam optimizer and a loss function of categorical cross entropy. Training this on the set with stacked face embeddings, we were able to get results very close to what we were getting in the shallow model. Watching the results closely, we saw that after about 3 epochs, our NN converged to 100% accuracy on the training data. Clearly, this was overfitting the data and not giving us the best results. This was even more obvious when testing with more epochs- which gave us very bad (~0.51 ROC) results compared to what we were getting with the shallow model. One idea to reduce overfitting and give us more generalized solutions was attempting to encourage symmetric weights.

Encouraging Symmetric Weights

One thing we really wanted to try was encouraging both of our models to recognize symmetrical patterns. To do this, we modified our training set. Everytime a set of images/face embeddings were stacked, we also wanted to add the opposite stacking to the sets. For example, if we stacked embedding 1 on top of embedding 2 and threw it into the training set, we stacked embedding 2 on top of embedding 1 and threw it in as well.

With the changes to our training data, we saw noticeable changes in both our shallow and deep models. It was clear that this approach was a good technique, and we kept it as a permanent feature in our training set. At this point, we also noticed that using a subset of our data seemed to be the bottleneck of our models, especially the deep model (since it continued to converge early). To solve this, we just remade our training set with 30% of the data provided on Kaggle.

Revisiting Models with More Data

With more data in our training set, we went back to training and optimizing our models. We quickly noticed that they took much longer to train and that the deep model did not converge nearly as fast as before. Our shallow model results with more data were around the same or a bit worse than we were getting on the smaller data set, which we very much expected. Initially, our results on the deep model were a bit lower than what we were getting with the smaller data set. We believe one reason for this was because we had not yet optimized (or even changed) our hyperparameters from the smaller training set. More data also meant it was harder for the machine to learn correctly- at least in the time/epochs we were training with at first. With more data, we then wanted to find the best way to get consistent, or generalized results.

Attempting a Generalized Solution: Random Noise

From the suggestion of our professor, we decided to try a couple of methods that may improve our training and results. The first of which was adding a matrix of gaussian noise onto the training embeddings. This was done in hopes of making a more generalized solution. With the default parameters we set, it seemed like the noise was making our final results worse than without the noise. After a bit of optimization, we found that it helped improve the performance of our model by a small amount. Over multiple tests, we noticed that the noise was in fact generalizing our solutions and giving us better results overall on the testing set. Satisfied with our attempts so far, the last thing we wanted to determine was the best possible results with different hyperparameters.

Optimizing Hyperparameters

We performed a grid search algorithm to determine the best hyperparameters for training our deep model. The hyperparameters we worked with were: number of epochs, batch size, number of neurons in the two hidden layers, learning rate, standard deviation of the random noise, and activation function. After extended testing, we determined the best hyperparameters and set them as the defaults for all future training. Below were our final hyperparameter values:

- Epochs: 20
- Batch size: 64
- Number of neurons in both in both hidden layers: 15
- Learning rate: 0.0001
- Activation function: scaled exponential linear
- Standard deviation of random noise: .5

Table of Results

Note: All techniques were tested on our own testing set (formed in the same way as Kaggle testing set, but on unused training data). These results are shown below.

Model	**Techniques**	**CE Loss**	**Accuracy**	**ROC**
SVM (10% Data)	2 Stacked Images	1.16	0.53	0.45
(10% Data)	Face Embeddings	0.72	0.44	0.48
(10% Data)	Weight Symmetry	0.69	0.54	0.55
(30% Data)	More Data	1.09	0.49	0.55
(30% Data)	Final Iteration (With Random Noise)	0.84	0.61	0.67
Dense 3 layer NN (10% Data)	Face Embeddings	0.89	0.46	0.43
(10% Data)	Weight Symmetry	0.72	0.56	0.59
(30% Data)	More Data	0.74	0.49	0.49
(30% Data)	Adding Random Noise (Default parameters)	0.83	0.44	0.43
(30% Data)	Final Iteration (Hyperparameter Optimization)	0.83	0.61	0.70

Conclusions

Throughout this project, we tried a lot of different techniques and strategies, some of which worked much better than others. We were able to learn from each new iteration of our models to improve and optimize our next solutions.

One of our first implemented features, the stacking of the pairs of images, initially caused more trouble than it was worth. This was mainly due to the sizes of the stacked images taking up a lot of resources and causing training to be extremely slow and sometimes causing system crashes. To help, we implemented the use of face embeddings using a pretrained Facenet model, which helped both our performance and results. It was much easier to train and significantly improved our shallow model accuracy.

Implementing the weight symmetry technique also had a positive impact on our results. It did not slow our training, and our accuracy was much better for both the shallow and deep models.

Adding more data to our training set was a necessary step, but something that caused both of our models to return worse results. This, of course, was before any hyperparameter optimization, so these results were quite expected. Making a more generalized solution with the addition of random noise also seemed like a worthwhile addition, even though our initial results showed that it gave worse results- especially for the deep model. Both of these shortcomings were quickly resolved when we did all our hyperparameter optimizations. With more epochs and larger batch sizes, and a few other tweaked parameters, we were able to achieve much better results than before, and ones that could not be matched by our shallow model.

After all our implementations and techniques, we found that our deep model performed marginally better than the shallow, and gave more consistent results for back to back training. With more time we would have loved to implement an ensemble of similar deep models to create a better overall model.

References

Hiroki Taniai, ‘Keras Facenet,’ 2018, https://github.com/nyoki-mtl/keras-facenet

- We used a Pre-trained facenet model by Hiroki Taniai to create embeddings of important facial features instead of just training on the faces we were originally given.

Jason Brownlee, ‘How to Develop a Face Recognition System Using FaceNet in Keras,’ 2019, https://machinelearningmastery.com/how-to-develop-a-face-recognition-system-using-facenet-in-keras-and-an-svm-classifier/

- We also referenced some code to create embeddings from the facenet model using a tutorial