Facial Expression Recognition

Rahul Sridhar

rsridha2@uci.edu
ID: 41608676

June 15, 2017

I. Abstract

The problem of recognizing facial expressions from images has been gaining traction in the fields of Computer Vision and Machine Learning. Conventional Machine Learning and Image Classification techniques have traditionally under-performed in this particular task, especially when there are multiple facial expressions to choose from, some of which are difficult to distinguish for even humans. Using Kaggle's FER (Facial Expression Recognition) 2013 dataset, this paper explores the performance of an ensemble classifier of a simple, conventional K-Nearest Neighbor (K-NN) model, and a relatively more complex Convolutional Neural Network (CNN), and also compares its performance with state-of-the-art models, and suggests potential methods to improve on the current state-of-the-art, including the use of a recently developed generative class of neural networks called Generative Adversarial Networks (GAN), and in particular, InfoGAN, which additionally provides information that was used by the neural network to generate new data.

II. Introduction

The problem of recognizing facial expressions can be generally stated as follows: given an image containing a face, categorize the emotion that the face is trying to convey as belonging to one of several discrete categories - happy, sad, angry, etc.

The applications of facial expression recognition have been growing in number and variety. Robotics and Human Computer Interaction, health monitoring, customer-attentive marketing, behavioral sciences, video games, psychiatry and social media are a few domains in which facial expression recognition can add a lot of value. Apart from these, it can also be used in online chat rooms, television and educational software. [1][2]

The primary objectives of this project are multifold:

- Build a model that can predict the facial expression from an image reasonably well

- Suggest and implement novel hypotheses that can be tested and validated using relatively less complex models as a proof of concept so that it can be replicated with state-of-the-art models
- Gain a good amount of exposure to and experience in the end-to-end pipeline of creating and testing models geared towards performing an image classification task

III. Solution Design

Initial Solution Design

The initial set of hypotheses to be tested were going to be phased out in three parts:

1. Baseline classifier: Build a baseline model using either Convolutional Neural Networks (CNN), or K-nearest neighbors (K-NN)
2. Facial expression generation using InfoGAN [3]: Using a recent development in the machine learning world called Generative Adversarial Networks (GAN), and in particular, InfoGAN (Information GAN), I would like to identify the latent features that help discriminate between different facial expressions. Fundamentally, the way GANs will work here is that there will be a 'generator' that will generate images of faces with different facial expressions and there will be a competing 'discriminator' (adversary) that classifies the images it receives as either 'real' or 'fake'. The goal of the generator is to create as realistic faces (with different expressions) as possible and the goal of the discriminator is to distinguish between real faces (actual images of faces) and fake faces (ones that the generator generates) with a high accuracy. Both these models will be trained together – the point at which they come to an equilibrium is the point at which model training is typically terminated.
3. Feature augmentation to step 1 and build new classifier: Unlike traditional GANs, InfoGANs also help in identifying a set of features that aid the generator in generating new faces with different facial expressions. These will serve as new features to be augmented with the original CNN/K-NN model.

One of the hypotheses to be tested was that based on the final feature augmentation, there should be a boost in performance of the facial recognition model. Of course, the underlying hypothesis was that there are certain latent features in the image of a face that can serve as discriminators in the classification of facial expressions.

During the course of implementation, while testing the InfoGAN model (after building a baseline model using CNN), I realized that there were computational constraints that were too difficult to overcome. The InfoGAN model entails the training of three neural networks, which a MacBook Air with an i5 processor and 8 GB of RAM was not sufficient to handle; executing a few thousand iterations of the InfoGAN model required the program to be executed overnight. InfoGANs typically work well when they are executed for 100,000 or more iterations, which would have taken several days (an infeasible option).

Final Solution Design

Due to the above constraints, I modified my solution design to test a new set of hypotheses:

1. Baseline classifiers: Build two classification models using Convolutional Neural Networks (CNN), and K-nearest neighbors (K-NN). The K-NN model was built after performing dimensionality reduction using Principal Component Analysis (more details in subsequent sections)
2. Ensemble model: Build an ensemble classifier (of the above two models) using Gradient Boosting

The main hypothesis that is being tested is that a combination of two relatively weak models will result in a stronger model. If this hypothesis is proven to be true, in the future, a similar ensemble model can be built and tested on the state-of-the-art models.

IV. Data

The dataset is courtesy the Kaggle competition *"Challenges in Representation Learning: Facial Expression Recognition Challenge"* [4]. The data comprises of 48x48 grayscale images, and are split into three sets:

1. Training data: 28,709 images
2. Public test data: 3,589 images
3. Private test data: 3,589 images

There are totally 7 different types of facial expressions:

1. Angry (label = 0)
2. Disgust (label = 1)
3. Fear (label = 2)
4. Happy (label = 3)
5. Sad (label = 4)
6. Surprise (label = 5)
7. Neutral (label = 6)

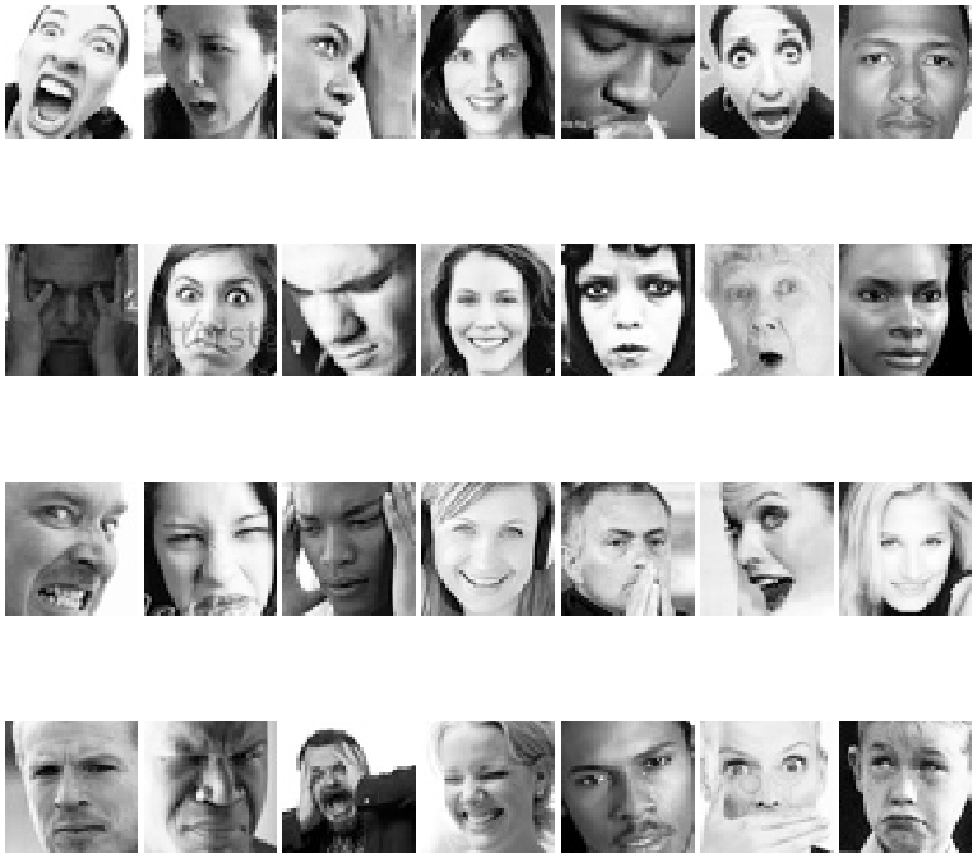


Figure 1: *Examples of faces from the Kaggle dataset. Each column contains 4 faces for each of the following emotions (in order): Anger, Disgust, Fear, Happy, Sad, Surprise, Neutral*

[4]

Figure 1 shows a few examples of faces of different kinds of classes. It is evident that unlike a few other image classification tasks that deal with face-related datasets, all the images are not front-facing, with quite a lot of inter-class and intra-class variations. The faces are also not homogeneous (in terms of ethnic diversity), not uniformly illuminated, vary in pose (left-side vs. right-side facing), and are also ambiguous in some cases in terms of the mapping between the class label and the actual expression of the face.

V. Implementation

Due to some of the reasons listed in the previous section, a fair amount of pre-processing was performed on the datasets. All the images were normalized and mean-centered before being fed to the algorithms. In some cases, I also experimented with random horizontal flips of the training images in order to improve the generalizability of the model.

The following subsections about the algorithms used in the solution are organized in the following manner: first, they briefly explain what each model or algorithm does, and then describe in detail the architecture and the final parameters used in the model building process, and how these parameters and architecture were arrived at.

Convolutional Neural Network (CNN)

CNNs are very similar to traditional ordinary, fully connected neural networks. They differ in the fact that they are primarily intended for use with image data, and the fact that they have special convolutional layers. Given images of size WxHxD (width, height and depth, which is equal to 3 if the images are in the RGB domain, and 1 if they are grayscale), each convolutional layer contains multiple filters or kernels of size typically much smaller than the dimensions of the image. These filters are essentially sliding windows that are convolved with small regions of each image and the outputs from each region are passed through a non-linear activation function and then sent to the next layer. Since the filters typically share weights (across the depth layer), the number of parameters that a CNN would have to learn is much lesser than a comparable fully connected neural network. This is so that features that are learned by the network from one region of the image can also be learned from another region. Usually, there are also special pooling layers inserted between convolutional layers that perform a type of down-sampling of their inputs by combining outputs from multiple regions of the image (either by averaging or taking the maximum). An example of a convolutional layer can be seen in Figure 2 [5].

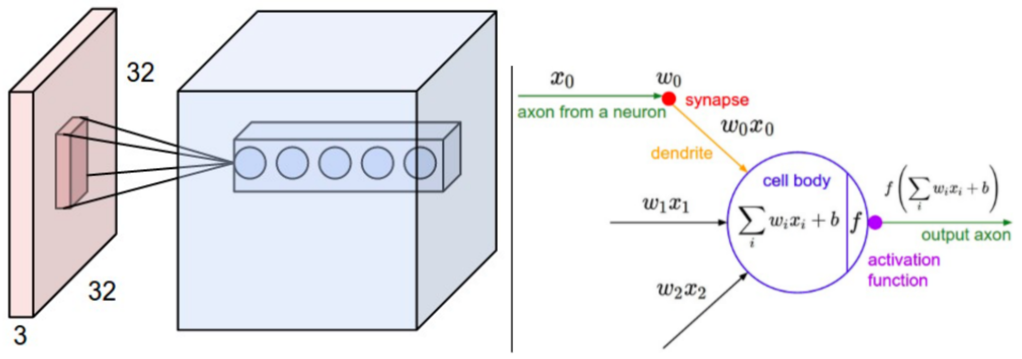


Figure 2: *Example of a convolutional layer* [5]

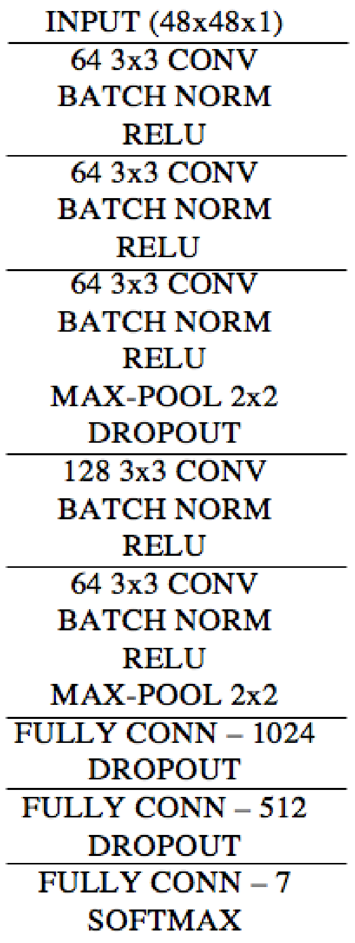
Initially, I used TensorFlow (Google's open source deep learning library) to create CNNs from scratch with different architectures and parameters. The following were the different parameters and hyper-parameters I kept tweaking to train the network on the training data and observed the performance (in terms of prediction accuracy) on the public and private test data:

- Number of convolutional layers (1 to 5)
- Number of fully connected layers (0 to 3)
- Number of neurons in each layer (32 to 128 in convolutional layers, 256 to 3072 in fully connected layers)
- Convolutional filter/kernel sizes (from 3x3 to 11x11)
- Dropout rate (probability of firing neurons of a particular layer varied between 0 to 0.75)
- Method of initializing weights (Random normal, Xavier [6], He [7])
- Regularization (L2)
- Batch size (64, 128, 256...)
- Learning rate of the optimizer (Adam [8], Stochastic Gradient Descent)
- Batch normalization (either done or not done)
- Max pooling (either done on a few particular layers or not done)
- Random horizontal flips of training images (either done or not done)
- Activation function (sigmoid or ReLU)
- Number of iterations (not epochs) of training the network (10,000 to more than 100,000)

Regardless of how any of the above parameters were changed, the generalization performance of the neural network was pretty poor. The test data accuracies always ranged between 20-25%. Even though that is still better than a random prediction (which would be just a bit more than 14%), and even though the overarching hypothesis of this project did not warrant that the individual, baseline models perform strongly, I felt that this performance was not nearly enough. Digging a little deeper into this issue, I realized that the problem was due to inadequate training of the network in terms of the number of iterations it ran for. Again, due to computational constraints, I could train the network for only a bit more than 100,000 iterations (which itself had to be done overnight).

I hence switched to a more efficient and scalable Python library called GraphLab, which uses a deep learning framework called MXNet. I had to make a few trade-offs by choosing this. Even though the time it took to train the network was much less, MXNet lacked flexibility in terms of the number and types of parameters that can be tuned. Finally, after a similar model building exercise as above, where I moved around between the realms of overfitting and underfitting, I created a CNN similar to the one shown in Figure 3. I was able to train the model for several epochs (passes through the entire training data) using MXNet within a couple of hours. The training error as a function of the number of epochs with this model can be viewed in

Figure 4.



INPUT (48x48x1)

64 3x3 CONV
BATCH NORM
RELU

64 3x3 CONV
BATCH NORM
RELU

64 3x3 CONV
BATCH NORM
RELU
MAX-POOL 2x2
DROPOUT

128 3x3 CONV
BATCH NORM
RELU

64 3x3 CONV
BATCH NORM
RELU
MAX-POOL 2x2

FULLY CONN – 1024
DROPOUT

FULLY CONN – 512
DROPOUT

FULLY CONN – 7
SOFTMAX

Figure 3: *CNN Architecture*

Generally, the impact of the parameters listed above on the predictive power of the model were as follows: training the model for a large number of epochs forced the model to overfit to the training data, and perform poorly on the test data. In order to counter this, I employed a technique similar to early stopping, wherein the neural network was trained for only a few epochs so that even though the performance on the training data was not great, it performed reasonably well on the test data. L2 regularization, dropout rate and random horizontal flips of the image had a similar impact. Increasing them (in case of the former two) or including them (in case of the latter) resulted in better generalizability of the model. As expected, increasing the number of convolutional/fully connected layers, and increasing the number of neurons in each layer increased the computational complexity and training time of the model (more parameters to learn). Having too few or too many layers/neurons resulted in underfitting and overfitting respectively. Xavier and He's weight initialization methods also empirically seemed to perform better than a random initialization

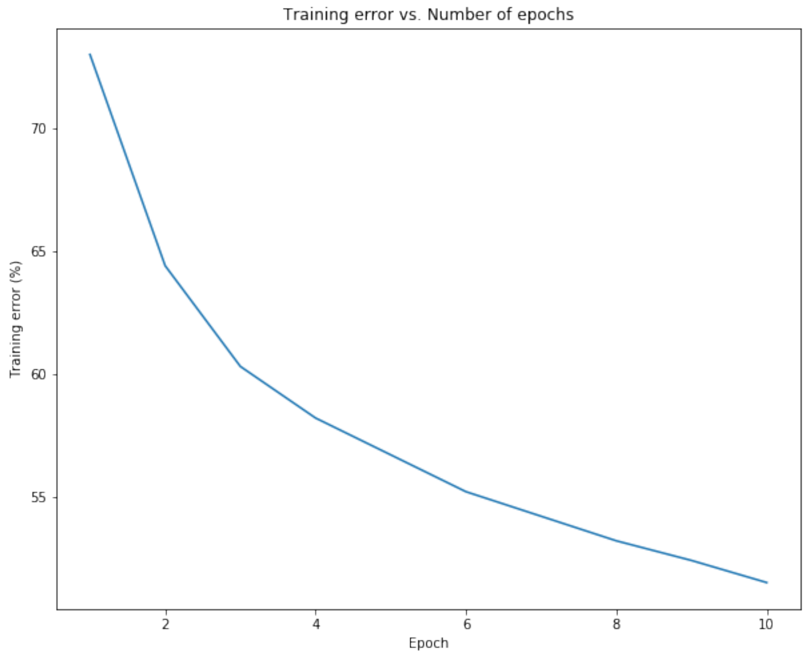


Figure 4: *CNN Training Error vs. Epoch*

of the weights. With regards to the activation function, empirically, ReLU seemed to perform better than the sigmoid function. One more thing that I observed was that even though increasing the number of convolution layers typically tends to increase the complexity of the network, having smaller filters (in the range of 3x3 or 4x4) offset the increased complexity to an extent. This is due to the fact that there are lesser parameters to learn, but at the same time, having more layers with smaller filters also gives the convolutional layers a receptive field equivalent to a network that has lesser layers with larger filters.

K-Nearest Neighbors (K-NN) and Principal Component Analysis (PCA)

K-NN is one of the more simplistic algorithms in the field of pattern recognition. Given a training data set along with its labels and a few test data, K-NN computes the pairwise distance between each of the images in the test data and the training data. The predicted label of the test image is the mode of the labels of the K closest neighbors of the test image in the training data. Closer neighbors have lower distances than farther neighbors. The distance function is pre-determined, and the Euclidean distance is typically the function of choice. To avoid ties, K is typically chosen as an odd number.

PCA is typically done as a prior step to reduce the dimensionality of the data. It is a statistical method that tries to find a potentially uncorrelated representation of the data. This method essentially performs an eigenvalue decomposition of the covariance matrix of the training data. The eigenvectors form a basis for the original image space. More the number of vectors or components, higher the variance of the data explained. This of course comes at a computational cost.

I reasoned that K-NN in the original space of the data (48*48 = 2304 dimensions) would neither be a feasible nor a meaningful exercise. The infeasibility is due to the computational complexity involved, and the lack of meaning is due to the fact that given the amount of data and the number of dimensions, two images being neighbors in such a high dimensional space would not amount to a lot because of the sheer distance between them. Hence, I performed PCA on the data to reduce its dimensionality. I then selected the top n principal components to build the K-NN model. The parameters that were tuned were:

- Number of components n (50 to 500) - the amount of variance in the data explained by these components varied between 83% to 97%
- Number of neighbors k (3 to 15)
- Distance function (Cosine Similarity or Euclidean)

As expected, increasing the number of components increased the computational complexity of the model and the tendency to overfit. With regards to the distance function, Cosine Similarity performed much better than Euclidean. One would expect this to be the case as well because Euclidean distance is susceptible to underperformance in the presence of outliers and is highly influenced by the magnitude of the data. Unlike it, the Cosine Similarity metric only measures the angle between two vectors. Arguably, it would be difficult to make a semantic interpretation when dealing with angles between PCA components. Regardless, intuitively and empirically, Cosine Similarity seems to be a better choice.

The final narrowed down set of parameters can be seen in Figure 5. For the implementation, I used Python's famous scikit-learn package to perform PCA, and GraphLab's fast and flexible implementation of the K-NN algorithm.

Ensemble model

A simple Gradient Boosted ensemble classifier was built on the resultant prediction labels from both the baseline models. Gradient Boosting is a simple model that is founded on the principle that an ensemble of weak classifiers (typically decision trees) can serve as a strong classifier and generalize well to unseen data. Each subsequent classifier of the ensemble tries to perform better than the previous one by trying to predict the residual error between the actual test label and the test label predicted by the previous classifier. As training progresses, lower is the overall error of the model. Hence, a few regularization controls have to be employed to make it less prone to overfitting.

Classifier	**Parameters**
PCA + K-NN	• # Components = 120 • k = 11 • Distance Function = Cosine Similarity
CNN	• # Epochs = 10 • (AdaM) Optimizer learning rate = 1e-4 • L2 regularization = 0.0005 • Batch size = 128 • Dropout = 0.7 (for fully connected layers) and 0.5 (for third convolution layer)
GB Ensemble	• # Trees = 50 • Maximum depth of tree = 5 • Minimum samples to split on = 5

Figure 5: *Tuned set of parameters for each model*

The following parameters were tuned in the model:

- Number of trees (50 to 500)
- Depth of the tree (3 to 5)
- Minimum number of samples to split a node of the tree
- Learning rate (0.1 to 0.9)

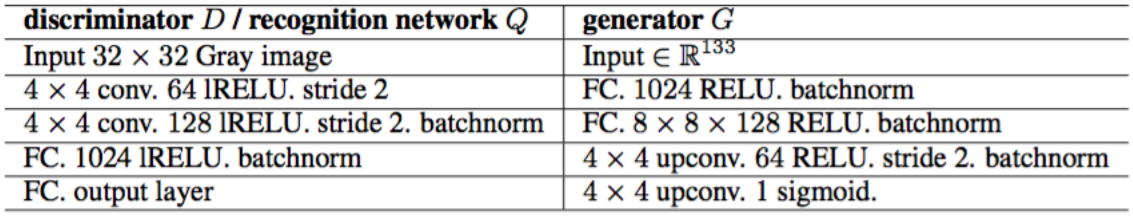
Given that the ensemble is built on a limited number of features (the labels from the two baseline classifiers), increasing the number of trees beyond a point increases the tendency to overfit and also stopped making a difference to the training accuracy. It however had a slightly detrimental impact on the generalizability of the model. The tree depth and the minimum number of samples to split nodes on had similar effects as well.

The final set of parameters for the ensemble model are shown in Figure 5. Again, Python's scikit-learn package was used for creating this model.

InfoGAN

Since the intuition behind InfoGAN was explained in Section III and since I did not end up using the InfoGAN model, this section has not been fleshed out in great detail. To add to the details provided in the previous section, InfoGAN primarily learns a disentangled representation of the data [3]. Prior to explaining what that means, here is a brief description of what a GAN does. In the case of a vanilla GAN, the generator network is first fed samples from a noisy feature vector, which it modifies based on the inputs from the discriminator network to produce data from a distribution that is as close to the original distribution of the data. The problem with this is that since there are no constraints as to how it generates the data and what kinds of features it learns, the final generative distribution is an "entangled" mixture of all the semantic features that make up the data. In the case of faces, these features could be the azimuth, hair style, emotion, etc. InfoGAN alleviates this problem by introducing a set of latent variables/codes which represent these semantic features. The loss function of InfoGAN contains a term to maximize the mutual information between the generated data distribution and the latent codes. This ensures that each of these latent codes intuitively correspond to some semantic feature of the underlying data distribution.

The architectures of the three neural networks that I implemented from scratch were inspired by the architectures used by the authors of the original paper [3] for testing InfoGAN on the CelebA dataset [9] (albeit modified to reflect the change in the sizes of the images). The original architecture can be seen in Figure 6.



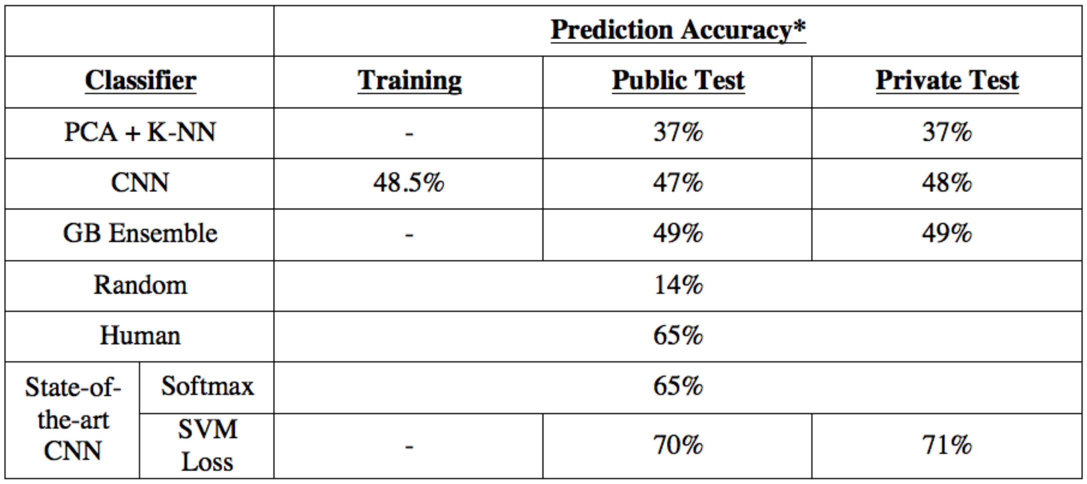
discriminator D / recognition network Q	**generator G**
Input 32×32 Gray image	Input $\in \mathbb{R}^{133}$
4×4 conv. 64 lRELU. stride 2	FC. 1024 RELU. batchnorm
4×4 conv. 128 lRELU. stride 2. batchnorm	FC. $8 \times 8 \times 128$ RELU. batchnorm
FC. 1024 lRELU. batchnorm	4×4 upconv. 64 RELU. stride 2. batchnorm
FC. output layer	4×4 upconv. 1 sigmoid.

Figure 6: *InfoGAN - architecture from the original paper* [3]

VI. Results and Comparison

The performance of the different models can be viewed in Figure 7. Both the individual baseline models (PCA+K-NN and CNN) perform much better than a model based on random predictions (which arguably is not a good benchmark, but is nevertheless a good sanity check). The ensemble model, with public and private test data accuracies of 49% each, performs marginally better than both of the individual models. Even though this improvement is not high enough to statistically validate the original hypothesis behind deciding to build an ensemble model, it does not outright reject the hypothesis either.

Some of the correct and incorrect predictions of the model can be seen in Figures 8 and 9. It appears as if the model gets the prediction right when the task is relatively straightforward whereas it gets confused when the labels are much more contentious and there are not a lot of variations between different classes of the image. This is even more evident from the confusion matrix, which is shown in Figure 10. Ideally, the diagonal elements must be as light as possible; the off-diagonal elements must



		Prediction Accuracy*		
Classifier		Training	Public Test	Private Test
PCA + K-NN		-	37%	37%
CNN		48.5%	47%	48%
GB Ensemble		-	49%	49%
Random		14%		
Human		65%		
State-of-the-art CNN	Softmax	65%		
	SVM Loss	-	70%	71%

Figure 7: *Results and Benchmarks (* Numbers are rounded)*
[10, 11]

be as dark as possible. The bottom-right-quadrant seems to be much better than the other quadrants in that aspect. As can also be seen here, the ensemble seems to be confusing between the 'Sad' and 'Neutral' expressions very often, which arguably, are not highly distinguished in terms of the variations in the images.

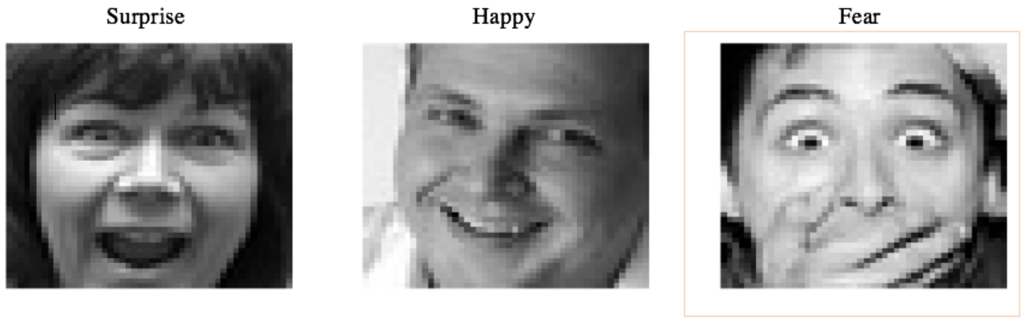


Figure 8: *Correctly predicted by the ensemble*

The best models that have been built for this data have performed better than this model. For example, the state-of-the-art models have accuracies of 60% [10] and 70% [11] on the test data. On the flip side, the ensemble model was to perform marginally better than a few other attempts at creating (deeper) CNNs and other models for this image classification task [2]. Even with this performance, the model was able to crack into the top 26 submissions for the original Kaggle competition. Given that over the course of the project, one of my goals was not to build individually strong models, but to test a hypothesis about the strength of an ensemble of weak classifiers, I consider this to be a great learning experience.

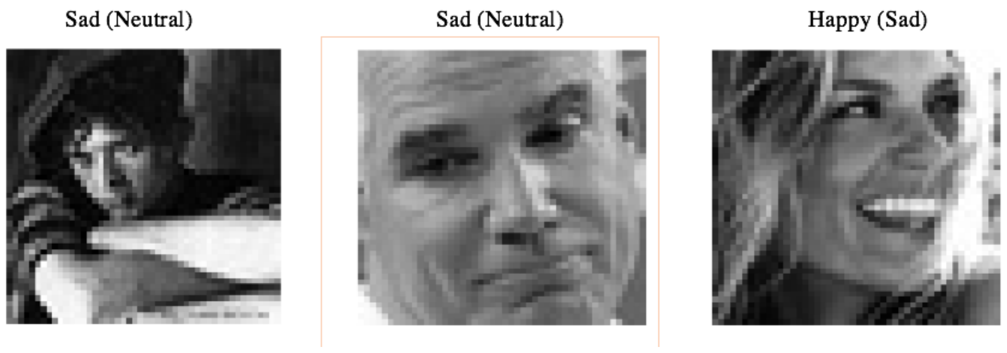


Figure 9: *Incorrectly predicted by the ensemble - Predicted labels are inside the parenthesis, actual labels are outside the parenthesis*

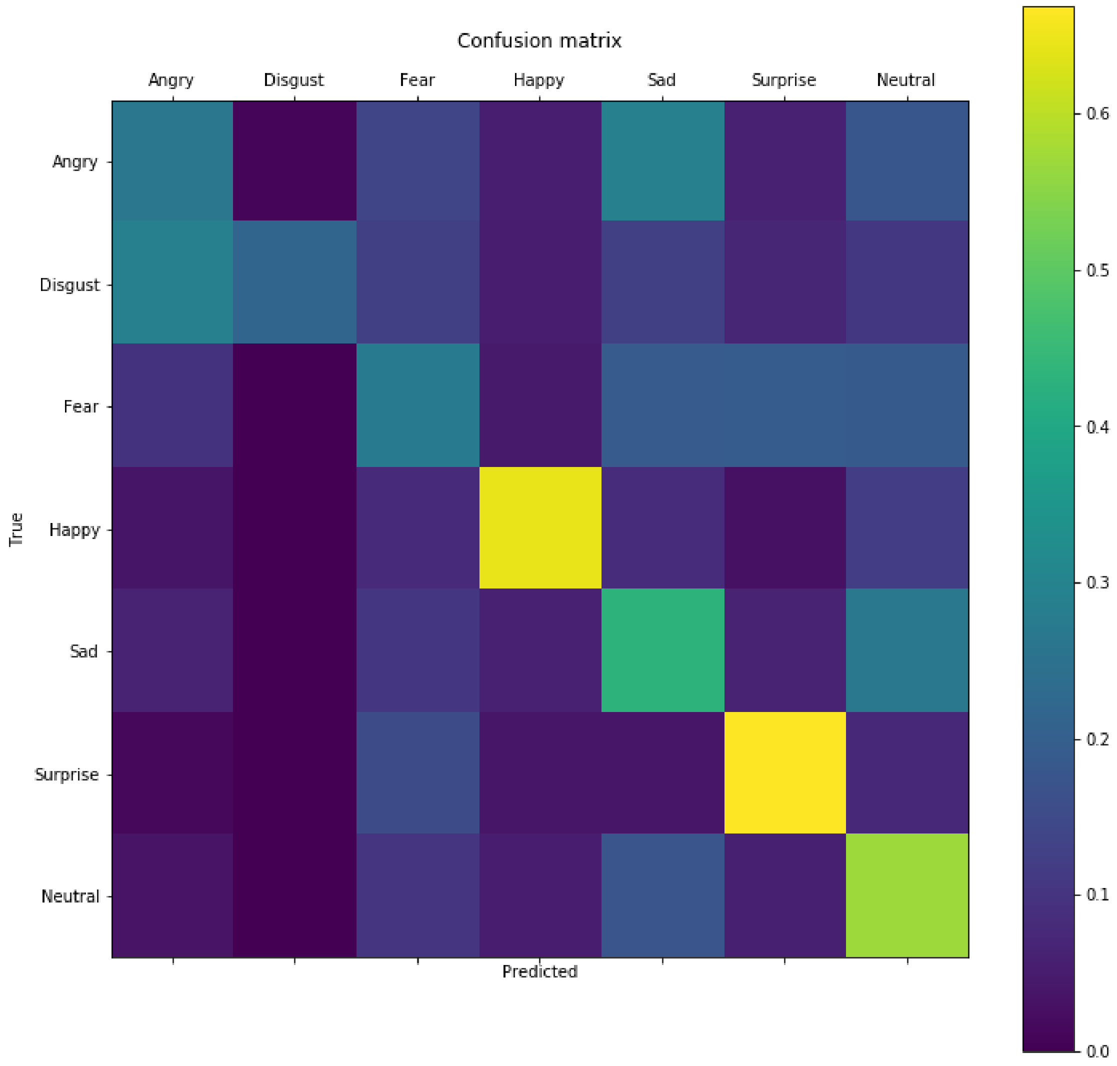


Figure 10: *Confusion Matrix*

VII. Discussion and Future Work

As mentioned earlier, the computational constraints were too huge to overcome in order to build a generative neural network (InfoGAN). In the ideal scenario, the underlying hypothesis behind the initial solution design would have been the better hypothesis to test. In future, it would be beneficial to test it with better and faster hardware (with GPUs or clusters, possibly). It would also be beneficial to test the underlying hypotheses from both the solution designs (feature augmentation using InfoGAN and building an ensemble) with the state-of-the-art models that have been built for this particular task and check if we can break through the current barriers to achieve a better performance.

In terms of the learning experience itself, I found this project to be highly fulfilling. Particularly, it was reassuring to see that certain hypotheses that had a rational basis seemed to be validated in practice. For example, the rationale behind choosing K-NN as a conventional baseline classifier was that given that the dataset is composed of faces with different expressions, it is not unreasonable to assume that faces of similar expressions will be ”near” to each other - if a laughing face in one picture is categorized as being happy, a laughing face in another face will also be categorized as being happy.

VIII. Conclusion

In this paper and the project, I have addressed the image classification task of recognizing one of seven ubiquitous facial expressions. I also designed and implemented a few different solution designs to tackle the same. With the help of a Gradient Boosted ensemble of two baseline classifiers (Convolutional Neural Network and K-nearest neighbor), I was able to achieve a test accuracy of 0.49. With more time and computational resources, I believe the suggested solution designs, if implemented using state-of-the-art models, would be able to perform much better than the state-of-the-art models themselves, whose accuracies are around 0.6 and 0.7.

References

[1] V. Bettadapura. Face expression recognition and analysis: The state of the art, 2012.

[2] A. Raghuvanshi and V. Choksi. Facial expression recognition with convolutional neural networks, 2016.

[3] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel. InfoGAN: Interpretable representation learning by information maximizing generative adversarial nets, 2016.

[4] Kaggle. Challenges in representation learning: Facial expression recognition challenge, 2013.

[5] Andrej Karpathy. http://cs231n.github.io/convolutional-networks/.

[6] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks, 2010.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[8] D. P. Kingma and J. Ba. AdaM: A method for stochastic optimization, 2015.

[9] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild, 2015.

[10] Goodfellow et al. Challenges in representation learning: A report on three machine learning contests, 2013.

[11] Y. Tang. Deep learning using linear support vector machines, 2013.