Final Project Report: Detecting Fake Human Face Images

Team Members: Alberto Ibarra, Jared (Jia Le) Lim, Jonathan Chang

Google Drive for reproduction (mount in Colab)**:** link (need Berkeley email to access)

Github repo for reproduction : link (same analysis files as GDrive, no data due to Github limit on file size)

Project motivation: Generative AI, particularly GANs (Generative Adversarial Networks) have led to an explosion of highly realistic, AI-generated faces on the internet. Unfortunately it's been used in a variety of concerning ways including the creation of fake online personas for scams, propaganda and online influence campaigns, and even state espionage. Thus, the ability to detect and flag these fake faces is important in improving trust and safety on social media and elsewhere online.

Datasets: Our primary dataset is this 140k Faces dataset that we found on Kaggle. It combines NVIDIA's 70k real faces, which were taken from Flickr and 70k fake faces generated by StyleGAN. To test generalizability of our models, we also used this Photoshop dataset.

Data processing and training setup: Because training CNN models on a large dataset of images is very computationally intensive, we wanted to utilize Google Colab's GPU resources. The data pipeline we used was: 1) download Kaggle datasets to zip files, 2) upload those zip files GDrive, 3) mount the Drive to CoLab, 4) unzip the dataset into the Colab's local environment to train/validation/test folders, and 5) read the folders into Keras ImageDataGenerator (allows us to efficiently convert batches of image data into tensors and do transformations on the images, which is helpful for larger datasets), and 6) fit models with images in the generator.

Machine Learning model implementation and hyperparameters tuning:

1. Feed Forward Neural Networks:

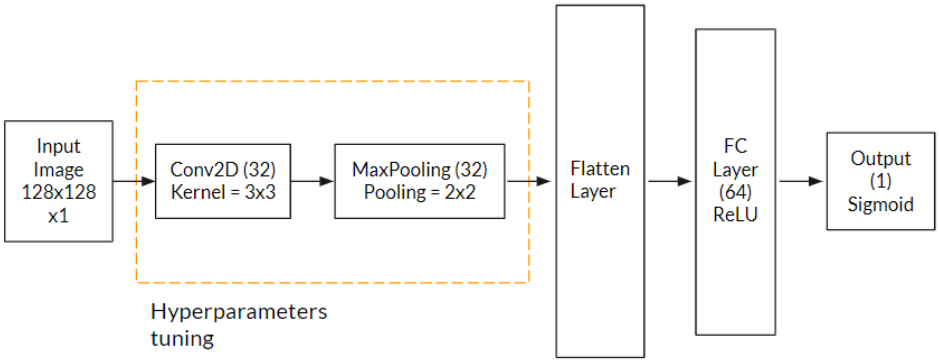
Using the ImageDataGenerator, the images were preprocessed to have a size of 128x128 pixels and grayscale for ease of running our Feed Forward Neural Network. The reason for this particular model was to have a baseline neural network to compare it to more complex machine learning models, such as Convolutional Neural Networks and the Densenet CNN. For this particular neural network, we had a simplified architectural model that our input data was passed through a fully connected layer (with ReLu) and passed it along to our one unit output that utilized the sigmoid activation function. Moreover, earlier in the analysis, we were able to conduct hyperparameter tuning for this particular model that involved a range of parameters:

- Activation Function ('relu','sigmoid','tanh','softmax')
- Dense Units (32, 64, 128, 256, 512)
- Epoch Amounts (1, 5, 10,15, 20)
- Learning Rate (0.01, 0.001, 0.0001)
- Batch Sizes (25)

Based on the trial of hyperparameters, we utilized the accuracy and ROC-AUC to determine the best model.

2. Convolutional Neural Networks (from scratch):

After the images are preprocessed, they are fed into our convolutional neural network (CNN) for real/fake classification. The following is the basic architecture of our CNN, which involves the Conv2D layer in Keras to convolve the input image into a number of feature maps, followed by MaxPooling to reduce the dimension. The convolution and pooling layers are repeated multiple times, each with a different combination of filter, kernel and pooling size. Then the convolved data is flattened before passing through a fully connected layer with 64 units, and eventually outputting the probability for the 'real' class using sigmoid activation function.



For hyperparameters, we focused on tuning those of the convolutional and pooling layers, namely the filter size, kernel size and max pooling size. We also experimented with different numbers of pairs of Conv2D and MaxPooling layers and the results of hyperparameters tuning are illustrated in the next section.

3. Densenet CNN:

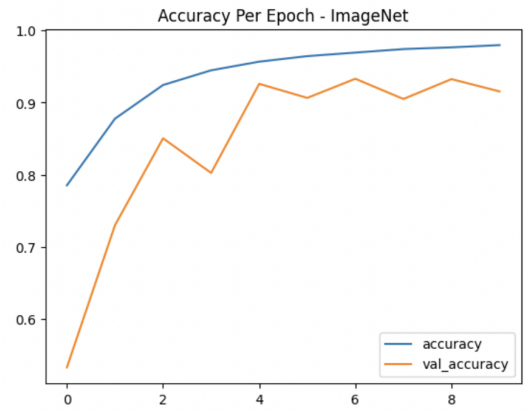
In order to improve upon our CNN results, we then implemented models using the Densenet architecture, which adds more direct connections between each of the layers (including skip connections) in order to solve the vanishing gradient problem, increase feature propagation, and utilize fewer parameters. Specifically we use Densenet121, which utilizes 121 layers of neurons. Our base model runs the Densenet121 model out of the box on grayscale 128x128 pixel images, just adding one more pooling layer and one dense layer with a sigmoid activation function to classify into one of the two classes. For tuning, we focused on changing image size, adding in RGB colors, and then using the ImageNet pretrained weights.

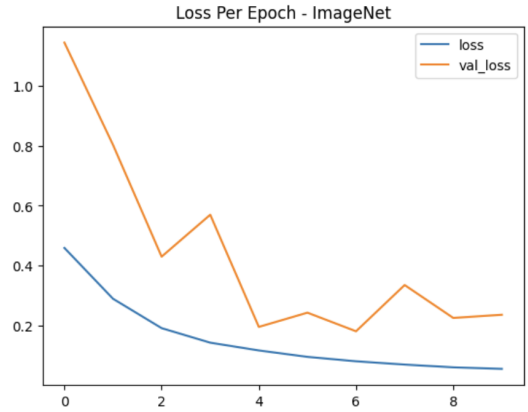
Results:

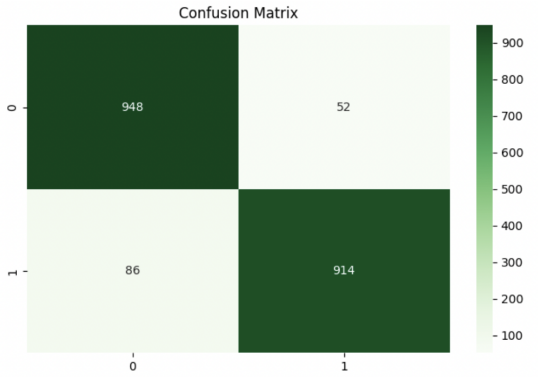
To ensure fair comparison of model performances, we are using 10 epochs for all the models we trained. The model performance is evaluated with metrics including area under ROC curve, accuracy, precision and recall. We are primarily using ROC-AUC instead of accuracy as the main metric to avoid the accuracy paradox due to imbalance training data. Overall, the best performing model was Densenet w/ImageNet weights, with 0.987 ROC-AUC, followed by our tuned Densenet model (ROC-AUC 0.946), tuned CNN model (ROC-AUC 0.909), and lastly

tuned Feed Forward NN (ROC-AUC 0.787). For each of the models, we plot out the accuracy/loss functions of the train and validation sets at each epoch, calculate and plot the confusion matrix, and calculate the ROC-AUC score, along with average precision score. We can see that this particular model achieves good validation performance after around 5 epochs.

Training, prediction plots for Densenet w/ImageNet Model








```
ROC-AUC Score: 0.980759
AP Score: 0.9802147027544307
```

	precision	recall	f1-score	support
fake	0.92	0.95	0.93	1000
real	0.95	0.91	0.93	1000
accuracy			0.93	2000
macro avg	0.93	0.93	0.93	2000
weighted avg	0.93	0.93	0.93	2000

1. *Feed Forward Neural Networks:*

Table below shows the results of 4 models with distinct hyperparameters.

Specification (all 10 epochs)	Avg train time per epoch	ROC-AUC	Accuracy	Avg Precision	Avg Recall
Model 1: Dense units 128, 128x128 pixels, grayscale, learning rate=0.001	~65s	0.7573	0.72	0.73	0.72
Model 2: Dense units 128, 128x128 pixels, grayscale, learning rate=0.0001	~61s	0.7873	0.68	0.72	0.68
Model 3: Dense units 64, 128x128 pixels, grayscale, learning rate=0.001	~60s	0.499	0.5	0.25	0.25
Model 4: Add color (RGB), 128x128 pixels, learning rate = 0.001	~72s	0.5	0.5	0.25	0.25

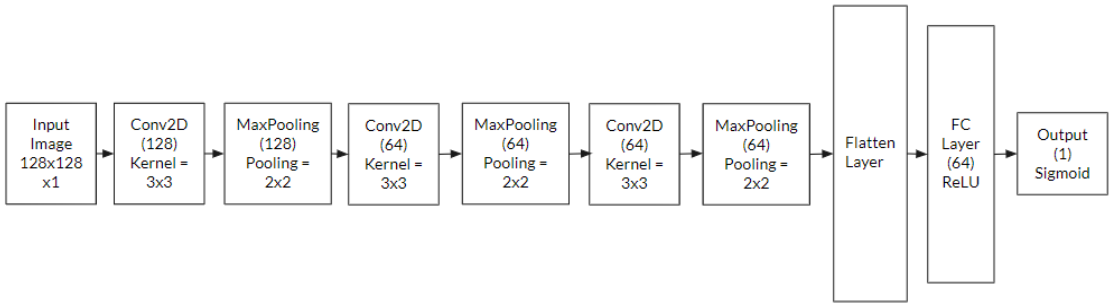
Model 2 is the best using the ROC-AUC performance criteria. It is interesting to see that the change of learning rate appeared to change the ROC-AUC result, in which we utilized the smallest hyperparameter of the learning rate (0.0001).

2. *Convolutional Neural Networks from scratch:*

Table below shows the results from 6 CNN models with different hyperparameters.

Specification (all 10 epochs and 128x128x1)	Avg train time per epoch	ROC-AUC	Accuracy	Avg Precision	Avg Recall
CNN-1 (64 filters x1)	~45s	0.806	0.73	0.73	0.73
CNN-2 (32 filters x1 + 64 filters x1)	~45s	0.854	0.79	0.79	0.79
CNN-3 (32 filters x2 + 64 filters x1)	~45s	0.904	0.83	0.83	0.83
CNN-3.1 (6x6 kernel, 4x4 pooling)	~45s	0.892	0.81	0.81	0.81
CNN-4 (32 filters x1 + 64 filters x2)	~45s	0.896	0.80	0.81	0.80
CNN-5 (128 filters x1 + 64 filters x2)	~45s	0.909	0.82	0.83	0.82

With our best model we are able to achieve 0.909 ROC-AUC, the best model is using 1 pair of Conv2D and MaxPooling with 128 filters and 2 pairs with 64 filters as shown below:



3. *Densenet CNN:*

Our base model improves upon the best CNN specification (ROC-AUC of .92). Tuning image size and adding color both increase ROC-AUC, but the change in image size drastically increases training time. Our best model is w/ImageNet pretrained weights, which achieves a ROC-AUC of 0.987 and average precision and recall of 0.92.

Specification (all 10 epochs)	Avg train time per epoch	ROC-AUC	Accuracy	Avg Precision	Avg Recall
Base Model: DenseNet, 128x128 pixels, grayscale	~84s	0.920	.82	.84	.82
Change image size (256x256 pixels), grayscale	~240s	0.944	0.88	0.88	0.88
Add color (RGB), 128x128 pixels	~85s	0.946	0.88	0.88	0.88
ImageNet pretrained weights,128x128 pixels, RGB	~85s	0.987	0.92	0.92	0.92

Where the model does poorly:

Our best model (DenseNet w/ImageNet) misclassified 138 out of the 2000 out-of-sample test images. Because we don't have any labels besides fake and real, we can only visually inspect the images to identify some patterns. If we found a dataset that also had feature labels on images, we would be able to conduct this exercise in a more systematic way. However, from our visual inspection, we are able to identify several plausible patterns where the model struggles. For fake images that are misclassified as real, we see many instances of: 1) people with sunglasses or glasses, 2) people wearing hats or hoods, 3) dark backgrounds without good subject/background separation. For real people that are misclassified as fake, they tend to have: 1) hats, headbands, or colored hair, 2) uniformly colored (or blurry / high bokeh) backgrounds, 3) dark backgrounds, and 4) obstructions in front of face (e.g. lipstick, microphone).

Fake images misclassified as real (left), example real images misclassified as fake (right)

Extra analysis on Generalizability to Photoshopped images

We wanted to evaluate the generalizability of the model to other forms of image manipulation. We focused on Photoshopped images since that's another popular method of manipulation. We used this Photoshop dataset developed by a Yonsei University research group. Using our best model, we then tested on 2000 images (1000 photoshopped, 1000 real). The test performance for Photoshopped data was extremely poor, with an ROC-AUC of .5, which is as good as random. Thus, it's clear that our GAN trained model does not generalize at all to Photoshopped images. For next steps, it could be interesting to create a combined dataset to train a more generalized model.

Team member contributions:

We all did work to search for data, set up the ML data processing and training pipeline on Colab.

Alberto's work: I worked on the baseline models, in which I reviewed K-means, Naive baseline, and the Feed Forward Neural Network. For this project, I focused around the Feed Forward Neural Network and hypertuned a variety of parameters, which included the dense units, pixel size, epochs, color scheme (e.g. grayscale, RGB), and activation functions. I also showcased the ROC-AUC for each of the models for this specific Neural Network.

Jared's work: I worked on the CNN from scratch model and tuned different parameters including the network structure, the kernel size, pooling size and filter size, and documented the result in CSV. I also tested the effect of different epochs on the model performance. I reviewed other algorithms including K-means and provided some basic code for the team to start with.

Jonathan's work: I worked to set up our dataset to mount and unzip from Google Drive and did research to select the Densenet CNN framework to try to improve on our CNN from scratch models. I then trained the 4 different Densenet specifications: a base model, larger image sizes, RGB color, and utilizing ImageNet weights. Using our best model (Densenet w/ImageNet weights), I then printed out all the misclassified images to try to figure out what our model was classifying poorly. Finally, I ran the extra analysis to see how well our best model would fit the Photoshop data set.