Deep Fake Detection

Ana Estrada Hamm, Yoseph Tamene, Danesh Badlani, Taylor Witherell, and Mitchell Roberts*
Washington and Lee University
Lexington, Virginia
estradahamma22@mail.wlu.edu,tameney22@mail.wlu.edu,badlanid22@mail.wlu.edu,witherellt21@mail.wlu.edu,
robertsm21@mail.wlu.edu

ABSTRACT

This research paper presents, explains, and analyzes the effectiveness of the involved steps in a process which uses a generative adversarial network (GAN) to create fake images of human faces, followed by the use of a convolutional neural network (CNN) to classify input images as real or fake. Both the GAN and the CNN were trained on real face images from the celeb_a dataset available through TensorFlow. We were able to create visually realistic fake images of faces using a relatively vanilla GAN architecture after tuning the hyperparameters. Our results showed that the CNN is able to perform binary classification to distinguish between real and fake images with 99.5% accuracy, offering hope that CNNs can provide a simple, fast solution to the rising risks that come along with advancements in GANs to create deep fake images.

In this paper, we introduce the issue, share learnings from related works, explain the data we used and the preprocessing steps we took, explain the model architectures of the GAN and the CNN, summarize key results, and conclusion.

CCS CONCEPTS

• **Deep Learning, Generative Adversarial Network, Convolutional Neural Network**;

KEYWORDS

Deep Learning, Generative Adversarial Network, Convolutional Neural Network, Face Generation, Image Classification, Deep Fakes, Binary classification, RMSprop, Vanilla GAN

ACM Reference format:
Ana Estrada Hamm, Yoseph Tamene, Danesh Badlani, Taylor Witherell, and Mitchell Roberts. 2021. Deep Fake Detection. In *Proceedings of Final Project, Lexington, VA, April 2021 (AI '21),* 11 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As Artificial Intelligence becomes increasingly advanced, there are inherent dangers and risks that come along with such progress. One of those dangers relates to the advances in generative adversarial networks (GAN). This poses a threat because humans are unable to tell the difference between the fake images and real ones, meaning that an individual with bad intentions could alter the view of reality by creating an image that places a real person in a false situation. This has implications not only for people's reputations and defamation of character issues, but also in the court of law, as evidence can be skewed to reflect a false reality. There is also the ability for bad actors to create fake faces to use in the creation of bot farms. These fake faces could be used to create fake profiles on social media platforms that generate propaganda that is both very realistic and harmful to the public because of how influential social media can be. This was seen in the last two US elections where bad actors were able to leverage the power of the internet and social media to spread misleading and false information. It is important to understand how these images are created and how to sift through fake versus real images. Figure 1 shows just how convincing deep fake images can be, as the left and right images are nearly impossible to discern which one is real and which is fake. With this degree of convincibility possible, the dangers of deep fakes are clear–fake, yet convincing media can be widely distributed, spreading misinformation that could sway opinions, tarnish reputations, and break down trust of information received on a viral level when in the hands of a bad actor.

Figure 1: Real vs Fake image of President Barack Obama. The real image is on the left [1]

However, perhaps the most promising solution to this problem, inherently created by artificial intelligence, is artificial intelligence itself. It is difficult to get a data set of fake images to train a neural net on because that would required the assignment of fake and real to be decided through careful analysis by a person, but GANs have been proven to be able to create realistic fake or augmented data that can be used in data augmentation efforts or to build a set

*Group of 3rd and 4th year students in CSCI-315: Artificial Intelligence, taught by Professor Watson

AI '21, April 2021, Lexington, VA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of fake images. This is especially useful because it is not always possible for humans to tell the difference between real and fake images to be able to label them appropriately. Though GANs have become extremely advanced in recent years, posing dangers but also leading to rapid advances related to video and image creation, deep learning scientists have another tool which is excellent for classifying images and learning specific features present in images. A fully trained convolutional neural network (CNN) can take an input image and accurately map it to a classification, in this case real or fake. Using a CNN could avoid or decrease the intensity of many of the dangers and risks of advances in GAN models, as determining that an image is fake could save a reputation, keep someone innocent out of jail, and protect the privacy of individuals.

In this paper, we explore, present, and test the efficacy of an approach for classifying GAN-generated and real celebrity images into two categories - real and fake - using a CNN. The aim of this research is to try to create images that look convincingly human. In order to show the effectiveness of deep learning models on a smaller scale production of fake images, we implemented a vanilla GAN that will help communicate just how advanced deep fakes can be when produced by only simple API calls and a few smart optimization practices. We are not aiming to create flawless fake faces, but we would like to create faces that one would see as a profile picture and not question if that person is real or not. We also wanted to expand on our generation of images and actually test a neural networks ability to tell the difference between fake and real images.

2 RELATED WORKS

Before exploring this topic through building our own model, we first consulted the related literature to learn more about methods that have been tried before for similar problems.

The first paper we found to be interesting and applicable to our topic is called *Detecting GAN generated Fake Images using Co-occurrence Matrices*, written by Nataraj et al. in 2019. In this research paper, the authors explain their methodology of taking input images created by two different GANs (cycleGAN and StarGAN), computing co-occurence matrices on the RGB channels of the image, and passing the signal to a multi-layer deep convolutional neural network [2]. Their method is summarized in Figure 2. The group found this method to be quite accurate at detecting GAN generated images, as it was able to achieve a training accuracy of 99.90% and validation accuracy of 99.40% on the cycleGAN images, and 99.43% and 99.39% on the StarGAN dataset. Testing accuracy was high as well, at 99.71% for cycleGAN images and 99.37% for the StarGAN images. With these results, it can be concluded that the method was successful in differentiating between GAN generated images and real ones.

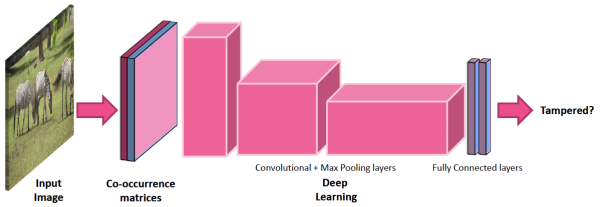

Figure 2: End-to-end framework to detect GAN images from Nataraj et al. 2019

The second paper we reviewed is titled *DeepFake Detection by Analyzing Convolutional Traces*, written by Guarnera et al. at the University of Catania in 2020. Their technique for detecting Deepfakes uses an Expectation Maximization (EM) algorithm, which extracts a set of local features which model the underlying convolutional generative process [3]. The paper tested a variety of GAN generated image sets against the baseline of the celeb_a dataset.

This technique was able to achieve a test accuracy of 92.67% for celeb_a tested against ATTGAN, 88.40% against GDWCT, 93.17% against STARGAN, 99.65% against STYLEGAN, and 99.81% against STYLEGAN2. The researchers also conducted a binary classification test that compared the celeb_a dastaset with all the images generated by all the networks. Figure 3 shows a visual representation of this binary classification problem using different kernel sizes. The results for this test were able to achieve a slightly lower, yet still impressive, 90.22% maximum accuracy, showing that this method is both effective when the GAN type is known and when it is unknown.

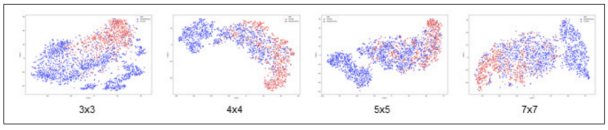

Figure 3: Two-dimensional t-SNE representation (CELEBA: red; DeepNetwork: blue) of a binary classification problem (with different kernel size): CELEBA Vs DeepNetworks. from Guarnera et al. 2020

Another paper we examined is called *Deep Fake Image Detection Based on Pairwise Learning*, written by Hsu et al. in 2020. The paper proposes using the contrastive loss for fake image detection. These researchers use several GANs (DCGAN, WGAP, WGAN-GP, LSGAN, and PGGAN) to generate fake–real image pairs [4]. Paraphrasing their approach: they use a two-streamed network structure allowing for pairwise information to be input. The network is then trained with pairwise learning for distinguishing between the features of fake and real images, with the model output being a binary classification of fake or real. This method was able to detect fake images with 98.6% accuracy when WGAN-GP was used for image

generation, 92.9% with DCGAN, 98.8% with WGAN, 94.7% with LSGAN, and 98.8% with PGGAN, showing it to be an effective method for fake image detection across various common GAN types.

The last paper we analyzed is called *Detecting Fake Images on Social Media using Machine Learning*, by AlShariah et al. at the Imam Mohammad Ibn Saud Islamic University (IMSIU) in 2019. This paper went into a more specific practical implication of the research topic. The research explains some of the downsides of using deep neural networks for detection of fake images, citing expensive computation, slow training speed, and dependencies on a large amount of data [5]. The paper then compares three potential approaches that could work better: Alexnet (a well-known CNN effective for image data), a classic CNN, and Alexnet with transfer learning. Their analysis concluded that, while each network provides a satisfactory approach to fake image detection, the approach using Alexnet was superior, as it was able to achieve 97% accuracy, followed by Alexnet using transfer learning, and finally the classic CNN. At 97% accuracy, the approach of using Alexnet to distinguish between real images and AI generated images is effective.

Our paper builds on the existing body of research through implementing our very own GAN to generate images and a CNN to classify the images as real or fake. We aim to support the results of the existing literature, while also offering a simple, yet effective approach to solving this crucial issue.

3 DATA

The data that we used was from the TensorFlow celeb_a data set. This is a data set that includes more than 200,000 images of celebrities at different angles and with all kinds of different features. When we were looking at the data set, we decided not to limit the number of facial features or other features that we took in because we wanted to have a lot variety available to the GAN. This also keeps the generated images from being too similar. These are the potential features that were present in the images that we selected: 5 o'clock shadow, arched eyebrows, attractive, bags, under eyes, bald, bangs, big lips, big nose, black hair, blond hair, blurry, brown hair, bush eyebrows, chubby, double chin, eyeglasses, goatee, gray hair, heavy makeup, high cheek bones, male, mouth slightly open, mustache, narrow eyes, no beard, oval face, pale face, pointy nose, receding hairline, rosy cheeks, sideburns, smiling, straight hair, wavy hair, wearing earrings, wearing lipstick, wearing necklace, wearing neck tie, and young. We wanted to, from the outset, include as many features as possible to ensure that we were not creating faces that were overly similar. This yielded some interesting results that will be discussed in the rest of the paper.

To train the GAN, we used 10,000 images from the data set. These images were prepossessed (see the Data Preprocessing section below) and then fed through the GAN. Importantly, we did not use the same images that we trained the GAN on in our data set that was used to test the CNN. The CNN used 10,000 different images from the celeb_a data set in combination with 10,000 images that were generated by the GAN. We were sure to exclude any images used in the training of the GAN because we believe that would have led to misleading results and is a form of data snooping. Data snooping is a phenomenon where some of the testing and training data blend together and cause highly accurate but misleading results. What this means for our data is that it would have led to an inaccurate testing of the CNN since the features of the 10,00 images used to train the GAN basically built the features of the faces of the GAN generated images. So it would have been more likely that the CNN that classified the images into fake and real would have gotten confused by the fake images.

We created the dataset for the CNN by combining the 10,000 real images through the process described above and the fake images that were generated using the GAN. We gave the real images a label of 1 and the fake images a label of 0.

Samples of the data we used can be found in a Google Drive folder that will be shared along with this paper.

4 DATA PREPROCESSING

Since we are using the same celeb_a dataset for both training the GAN and the CNN, we take the same steps when preprocessing the data for both.

We loaded the data from the TensorFlow celeb_a image library. Theses images were originally 178 by 208 pixels. Because we are feeding the images into convolutional neural networks, the images required minimal prepossessing. Convolutional neural networks, unlike feed forward neural networks (another deep learning model that can be useful for image classification), do not require as much prepossessing because convolutional neural networks are able to accept multiple channels of color. We apply the same transformation because both models use a CNN as their base.

The images were cropped to 128 by 128 pixels. Then, they were transformed into the Numpy arrays that represent the pixels of the images. The reason that we do this is because unlike humans, computers cannot actually see images and have to be fed the numerical information that represents the images. This is done by creating an array representation that stores the color values in a tuple of a given pixel in its location. The tuples' values include the value for red, green, and blue. The values are numbers 0-255 that represent the amount of each color that is present in the pixel. After creating the numerical representations of the numbers, we have to normalize the images by diving the pixels value by 255. We do this because it will give us a numerical representation of the color between 0 and 1. This is done so that in the model the relative difference between 0 to 255 does not skew the gradient or the weight. This is a technique to prevent exploding or vanishing gradients.

After the fake images are created by the GAN, they do not need to be cropped as they are already the correct size when they are created, but they still go through the same process as the real images mentioned above.

5 APPROACH

In developing both the GAN and the CNN, we took several iterative steps towards our final design of the model. We wanted to create a GAN that was not overly complicated, but could generate faces that were realistic from the celeb_a data. This involved using a

simpler GAN architecture that proved effective in creating lifelike faces. The same philosophy carried through with our CNN. The goal of our approach is to create models that get the job done and are complex enough, but are not overly complex to implement.

5.1 Generative Adversarial Network

5.1.1 Elements of Learning. The **Target Function** for this part of the project is a function that inputs real image data of celebrity faces and outputs fake, generated facial image data.

The **Training Data** which we are using is the CelebFaces Attributes Dataset (CelebA), which consists of more than 200K celebrity images, each with 40 attribute annotations. For our purposes, we are not interested in the attribute annotations, but rather use this dataset as our bank of "real" face images.

The **Hypothesis Set** is the set of all possible functions that could take real images as input and generate realistic images of faces.

Our chosen **Learning Algorithm** for this task is a generative adversarial network (GAN) because GANs are excellent for data generation tasks. A GAN is a deep learning algorithm that consists of 2 major components, a generator and discriminator, which work against each other to train themselves. The generator takes in a noise vector and attempts to create new data instances, while the discriminator distinguishes between real and fake images generated. The discriminator is a simple CNN, but the generator is essentially a reverse CNN. Our GAN uses RMSprop, an adaptive learning rate optimization technique; an initial learning rate of 0.0001; and no dropout layers.

The **Final Hypothesis** in this case is the fully trained GAN model, which generates realistic-looking fake images after training on real celebrity images.

5.1.2 Network Architecture. For the GAN, we began with a fairly vanilla network architecture. The base case for training utilized 20,000 epochs, RMSprop, a 0.0001 learning rate, and no dropout layers.

For the pre-processing of the training data, first we crop each image to have a 128 by 128 dimension. We then normalize each pixel's color value by dividing them by 255.

Once the pre-processing step is complete, we begin setting up the generator part of the GAN. First, we initialize the number of samples to 16, the latent dimension to 32, and the channels to 3. The number of samples is how many output images we would like once the GAN has been fully trained. The latent dimension is how big our latent space is going to be; they are simply the compressed representations from which our model takes a sample from a normal distribution (the noise) then uses to generate images. And finally, the three channels are the three colors in the RGB color space.

Then, like with other keras models, we begin with the Sequential() call. As parameters, we designate the input shape as the latent dimension so that the model builds continuously as we are adding layers. Then, we add a dense layer of size 128*16*16, which is the dimensionality of the output space, and apply the Leaky ReLU activation function to allow for small negative signals. We then reshape the output to 16 by 16 by 128.

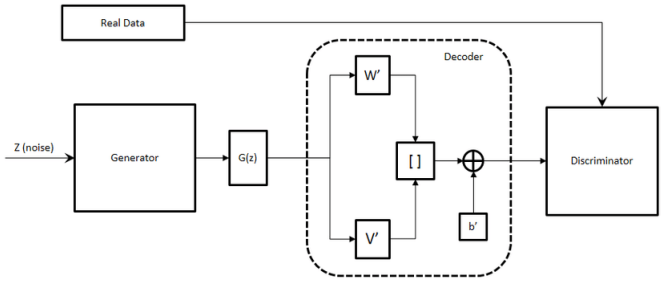

Figure 4: GAN Architecture

Next, we add a two-dimensional convolutional layer with 256 filters, kernel size of 5, and same padding. Same padding changes the padding around the image to fit the filter and stride with the image while balancing the padding by adding zeros around the edges of the image, as necessary. We again use the Leaky ReLU activation function.

Following the convolutional layer, we add a two-dimensional convolutional transpose layer with 256 filters, kernel size of 4, stride of 2, same padding, and Leaky ReLU activation. The transpose layer works in the opposite direction of a normal convolutional layer. It takes an input which is the shape of the output of the previous layer and maps it to the input shape of the previous layer. This is where the generator is trying to reconstruct data to match features of the input based on the relationships learned in the convolutional layer.

After the transpose layer, another two-dimensional convolutional transpose layer is added to further learn the complex hierarchical features involved in the mapping of signals to the input shape. Again, we use 256 filters, kernel size of 4, stride of 2, same padding, and Leaky ReLU activation.

Once again, we add the same transpose layer with the same parameters.

After the final transpose layer, we add two additional two-dimensional convolutional layers, each with 512 filters, kernel size of 5, same padding, and Leaky ReLU activation. We are using Leaky ReLU because it prevents the dying ReLU problem where a large number of ReLU units output zeros.

The final layer added in the generator part of the model is one last two-dimensional convolutional layer, with the number of filters set to the number of channels initialized earlier, kernel size of 7, hyperbolic tangent (tanh) activation, and same padding. We only use the tanh function in the output layer which scales the output to be between -1 and 1. We also print out a summary of the model.

Because a GAN also consists of a discriminator that takes an input and determines whether or not it fits a certain classification,

in this case “real" or “fake", we need to create another model architecture using the Keras Sequential() call. The discriminator has a shape of the image height, width, and number of channels.

This model does not use any transpose convolutional layers, since we are not interested in mapping the output back to a signal in the shape of the input. It is essentially a CNN. Thus, the first layer is a two-dimensional convolutional layer consisting of 256 filters, kernel size of 3 and Leaky ReLU activation.

We follow that layer with another convolutional layer, consisting of 256 filters, kernel size of 4, stride of 2, and Leaky ReLU activation.

Then, we add three identical layers to the previous one before flattening the output with the Keras Flatten() function, which we do to represent our complex hierarchical features as a flattened matrix, and feed them into a fully connected layer.

The fully connected layer is created using the Keras Dense() call with size of 1 and sigmoid activation. This essentially output a determination of “real" or “fake" based on the value of the output signal after being passed through the sigmoid function.

After both the generator and discriminator are setup, we set the optimizer to RMSprop with a learning rate of 0.001, clip value of 1.0, and decay of 1e-8. We apply gradient clipping with a clip value of 1.0 in order to prevent the exploding gradient problem. The decay value is the value by which we decrement the learning rate during training; this helps in the model learn more complex patterns and prevents it from memorizing noisy data.

We then compile the discriminator using the optimizer and binary cross entropy loss since we are working with a classification that can only take on one of two values, “real" or “fake".

Then, we create the actual GAN. Again, we use the Sequential call, with an input of shape latent dimension. We also pass the two models, our generator and discriminator, as parameters.

Next, the optimizer is set to RMSprop with a learning rate of 0.001, clip value of 1.0, and decay of 1e-8. Finally, we compile the GAN using the optimizer and binary cross entropy.

The next step is to actually train the GAN. We start by initializing a few variables. We set the number of epochs to 10000, the batch size to 16, RES_DIR to ‘./generated’, FILE_PATH to ‘%s/generated_%d.png’, and the save_dir to ‘./saved/’.

We only start the training if there is not a saved model found in the ‘./saved’ directory. To do this, we set the directory to RES_DIR. We also initialize the CONTROL_SIZE_SQRT, which will be used in the size of our control vectors, which are initialized using the numpy random.normal(). The size parameter is equal to the CONTROL_SIZE_SQRT squared by the latent dimension divided by 2.

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 32768)             1081344
_________________________________________________________________
leaky_re_lu_45 (LeakyReLU)   (None, 32768)             0
_________________________________________________________________
reshape_5 (Reshape)          (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_30 (Conv2D)           (None, 16, 16, 256)       819456
_________________________________________________________________
leaky_re_lu_46 (LeakyReLU)   (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_transpose_15 (Conv2DT (None, 32, 32, 256)       1048832
_________________________________________________________________
leaky_re_lu_47 (LeakyReLU)   (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_transpose_16 (Conv2DT (None, 64, 64, 256)       1048832
_________________________________________________________________
leaky_re_lu_48 (LeakyReLU)   (None, 64, 64, 256)       0
_________________________________________________________________
conv2d_transpose_17 (Conv2DT (None, 128, 128, 256)     1048832
_________________________________________________________________
leaky_re_lu_49 (LeakyReLU)   (None, 128, 128, 256)     0
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 128, 128, 512)     3277312
_________________________________________________________________
leaky_re_lu_50 (LeakyReLU)   (None, 128, 128, 512)     0
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 128, 128, 512)     6554112
_________________________________________________________________
leaky_re_lu_51 (LeakyReLU)   (None, 128, 128, 512)     0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 128, 128, 3)       75267
=================================================================
Total params: 14,953,987
Trainable params: 14,953,987
Non-trainable params: 0
```

Figure 5: Summary of GAN

5.1.3 Hyperparameter Tuning. Hyper parameters are variables that affect the model's ability to learn. So when the hyper parameters are tuned, it is affecting how the model will learn, but not what it learns. Hyper parameters include the number of epochs that the model is trained on, the learning rate, the learning rate optimizer, the architecture of the model, loss function, and many others.

We found, through initial testing of hyper parameters, that the number of epochs played a large role in determining the quality of the photos produced. We quickly realized how time consuming training the model was using tens of thousands of epochs, so we decided to prioritize the tuning of the epochs as opposed to other hyper parameters because of the results our initial testing yielded. But, before we settled on tuning epochs we looked at two other considerations that highly affect learning: the architecture of the model and the learning rate. We also looked at adding in dropout layers to the GAN to see if the images produced were more lifelike, but adding the dropout layers did not decrease the loss or increase the images' quality. We also tried to change the learning rate optimizer from RMSprop at 0.0001 to Adam at 0.001. The Adam optimizer yielded an exploding gradient problem that could be seen in the first 200 epochs of training. The Adam optimizer was not considered a good optimizer for this model so the RMSprop optimizer was used instead. We tested the RMSprop optimizer at values of 0.01, 0.001, and 0.0001. We found that the learning rate of 0.0001 was the best out of all of them in terms of creating images that were lifelike.

So, the research team decided to focus on training the GAN on different levels of epochs. The GAN was trained on 10,000, 20,000, 30,000, 40,000, and 50,000 epochs. 10,000 to 20,000 epochs yielded promising results in terms of the images that were produced, but

were ultimately outdone with the 30,000 to 40,000 epoch models. We found that 20,000 to 30,000 epochs were the desire number of epoch. Ultimately settling on using 30,000 epochs in the interest of time. 50,000 epochs and 40,0000 led to over-training of the model and the images became increasingly unrecognizable images of humans. We believe this was because the model was beginning to over-fit, and learning was starting to deteriorate as a result.

5.2 Convolutional Neural Network

5.2.1 Elements of Learning. For the Convolutional Neural Network, our **Target Function** is a hypothetical, non-linear function that perfectly maps real and GAN-generated input images to the correct classification as real or fake.

The **Training Data** which we are using is again the celeb_a dataset from TensorFlow. We are, however, using a different subset of the total set of images than were used to train the GAN, to avoid the data snooping issues.

The **Hypothesis Set** is the set of all non-linear functions that could map input images to output classifications as real or fake discovered through the architecture of the CNN.

We are implementing a Convolutional Neural Network as our **Learning Algorithm** because CNNs are excellent for image classification tasks. A CNN is a deep learning algorithm that assigns an importance to parts of an image, with the intention of being able to differentiate between the different parts of the image. Our CNN uses same padding, which changes the padding around the image to fit the filter and stride with the image while balancing the padding. Each Convolutional layer uses the relu activation function, which is a pixel by pixel operation that replaces all negative values with zeros. We are also using the he_uniform kernel initializer, which draws samples from a uniform distribution within a negative to positive limit based on the number of input units in the weight matrix.

The **Final Hypothesis** is the fully trained CNN model, which is validated using the testing set of data, comparing the actual labels to the labels predicted using the fully trained CNN. The final hypothesis closely approximates the target function, and takes an image and classifies it into one of two categories: real or fake.

5.2.2 Network Architecture. Next, we start to put together our CNN, starting by using the Sequential() call from keras to define the model. Once this is defined, we can start to add layers to our model. We add a 2D convolutional layer that takes the input shape of the images and 32 filters which is how many features the CNN will learn from and a 3 x 3 kernel size that is the dimension of the filter to extract the features from the image using a dot product. We also used the same padding so that there is no loss of data. This will ensure that the output dimension of the image after convolution is same as the input of the image. We used the ReLU activation function to add non-linearity to the model. It does pixel-by-pixel operation that replaces all negative values with zero. This layer also contains the 'he_uniform' kernel initializer, whose main purpose is to initialize the weights with uniform distribution. The number of filters are doubled after every 2 convolutional layers to make our model more robust. We double the number of filters because our representation of the data becomes increasingly more complex as we go through the neural net, and we increase them to make sure that we are capturing all the increasingly complex patterns.

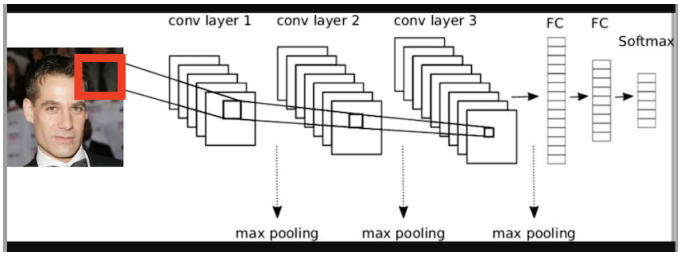

Figure 6: CNN Architecture

Then, we add a batch normalization layer, which, according to TensorFlow documentation, applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. The output is normalized with the mean and standard deviation of the current batch of inputs. A batch normalization layer stabilizes learning and reduces the number of epochs required to train the network. It is a regularization technique used to avoid overfitting of the model by normalizing the contributions to a layer for every mini-batch. It is a way to ensure that we have a balanced learning process and weights are not disappearing nor are some becoming more dominant than they need to be. It allows every layer of the network to learn independently.

The pooling layer in the network is used to reduce the size of the image while ensuring that important features are kept appropriately which helps to reduce the computation. Max-Pooling narrows down the scope and introduces a local invariance, meaning small changes in a local neighborhood do not change the result of max-pooling, according to our textbook. The reduced number of features also reduces the problem of overfitting. Here, we are using a 2x2 max pooling layer, which means we are taking the max values of 4 kernels each time we add a 2x2 max pooling layer.

Next, we add a dropout layer of 0.2, which means that we have a layer that randomly sets input units to 0 with a frequency of 0.2 during each training step. This is primarily useful for preventing overfitting issues, as the model is not able to rely solely on certain weights/"paths" through the network to map inputs to outputs.

Next, we add another convolutional layer, this one with 64 filters. Increasing the number of filters themselves in this layer also helps us gain more useful complex hierarchical features, followed by a batch normalization layer, another 2x2 max pooling layer, and a dropout layer with a rate of 0.2.

We then add yet another convolutional layer with 128 filters, even further increasing the number of complex hierarchical features the model can use to understand the data and relationship between input images and output classifications. Again, we follow this layer with a batch normalization layer, max pooling and dropout layers as previously used.

Now that we have an immense quantity of complex hierarchical features, we need to represent them as a flattened matrix, and

feed them into a fully connected layer. This is done with the keras Flatten() function. The flattening layer is added to help convert the image vector into 1D as the final fully dense layer requires the data to be passed in 1 dimension; so, this is an important step in the architecture.

Once the data is flattened, we can feed it into the fully connected dense layer. It is responsible for training the model, which uses the ReLU activation function with he_uniform kernel initializer. Using a ReLU activation function means that we are only going to focus on the features that add to our classification and do not take away form it. It uses 128 units of neurons because that is the last number of units used in the last convolutional layer. This basically connects all the dots. This layer is also followed by the same batch normalization and dropout layers as used before.

The last part of the model is adding the final dense layer (output layer), which is responsible for the actual classifications into the 2 different classes. The dense layer will consist of 1 node because it will classify if the image belongs to either real or fake since we did not one-hot encode our y-values. It uses the Sigmoid activation function because this results in a value between 0 and 1 which we can infer to be how confident it is of it being in one of the binary classes.

We set up early stop training, so that once certain accuracy metrics are achieved and the model is no longer improving, the model does not continue training on the data. Using a keras call again, we set our early stopping patience to 3, meaning we wait 3 epochs without significant change to the metric considered before stopping the training.

Next, we compile the model by using the Adam optimizer, which is a robust gradient-based optimization method suited to non convex optimization. Adam is the best learning rate optimizer because it combines both the RMS prop and optimization of momentum. It requires less tuning of the learning rate than the other learning rate optimization algorithms. This is a regularization technique that alters the hyper parameters. This will help the model stay away from the vanishing gradient problem. Naturally, since this application involves mapping to 2 different classes, we use binary cross entropy loss, with the metrics we are interested in tracking set to accuracy. The compile method from keras essentially configures the model for training.

We fit the model using 5 epochs with our early stop as the callback, and a validation set to be our testing set. The fit command trains the model on the training data. As it trains, it shows the metrics for accuracy, loss, validation accuracy, and validation loss after each epoch has run. Lastly, we generate plots of the model training accuracy and the model training loss against the number of training epochs to show how those metrics changed with each epoch.

5.2.3 Hyperparameter Tuning. In order to arrive at the best version of our CNN model as could be achieved, we tuned several of the hyperparamters along the development phase of the project before settling on the architecture outlined in the previous section. This section will describe the various approaches we attempted in the process of improving our model.

We began testing our CNN model with more layers such that it had 4 structures of convolutional, batch normalization, max pooling and dropout layers, but ended up reducing it to 3 architecture types of layers because we wanted to avoid the possibility of overfitting the model.

We also tuned the training and testing split to see how our model performed. We started by using the 90% and 10% split for training and testing respectively, where our model was giving a testing accuracy of about 96%. But, we wanted to ensure that the test size was larger and it could replicate similar or better results on more unseen data sets for our model to be generalizable. Hence, we ended up changing the training and testing split to 75% and 25% respectively while shuffling the data. This gave us the testing accuracy of 99% and also a more generalizable model.

The number of epochs was another hyperparameter that we tuned to check how our model was performing. We started off with fewer epochs such as 1 and tested different numbers such as 2, 5, 10 and 25. Our model was already performing above 90% within 1 epoch and reached at 99.6% accuracy for both training and testing at 5 epochs. We did not see any significant improvement in our accuracy and loss function after increasing the epochs from there, so we decided to use 5 for our model. This saves time and computational power to run this CNN by not using higher number of epochs.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 128, 32)      896
_________________________________________________________________
batch_normalization (BatchNo (None, 128, 128, 32)      128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 64, 64, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 64, 64, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 32, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 128)       73856
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 16, 16, 128)       0
_________________________________________________________________
flatten (Flatten)            (None, 32768)             0
_________________________________________________________________
dense (Dense)                (None, 128)               4194432
_________________________________________________________________
batch_normalization_3 (Batch (None, 128)               512
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 4,289,217
Trainable params: 4,288,513
Non-trainable params: 704
```

Figure 7: Summary of CNN

6 EVALUATION

In this section, we present the findings of our work and analyze the results. We split up the sections into evaluation of the GAN and evaluation of the classification of the CNN.

6.1 Generative Adversarial Network

Since we tuned the number of epochs in our GAN, these were the resulting images from the saved models. We generated 10 images per model to see how good the images were and if we and others found them convincingly human. In the generation of these images, there was a certain amount of variability in quality between one generated image to the next. The backgrounds are something that were highly variable as well as the hair. But focusing on the face structure, one can be convinced that some of the faces that were generated are real people. We were pleased with the results given our limited computing ability and the fact that we did not have months to train this model. As you will see in the images, the number of epochs did make a difference.

Figure 8: 50,000 epochs

Figure 9: 40,000 epochs

Figure 10: 30,000 epochs (The number of epochs we used)

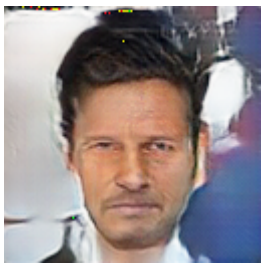

Figure 11: 20,000 epochs

Figure 12: 10,000 epochs

Figure 13 shows some of the examples of the fake faces generated by the GAN model that we used after running for 30,000 epochs. As it can be seen some of the fake images are very realistic and can easily confuse humans with the perception that they are real. This proves the efficacy of our GAN model that it has been successful in generating images that look very real.

Figure 13: Fake images generated using GAN

The GAN consists of two loss functions: discriminator and generator. The discriminator loss is going up with the number of epochs

which shows that our discriminator is successfully able to detect between fake and real images. Although the discriminator and generator (in our case, adversary loss) should converge, it seems that adversary loss also tends to increase with the number of epochs. This is because we have used a vanilla GAN, which can be noisy and unstable. However, our fake image generation turned out to be extremely good - hence, we should not discount the efficacy of our GAN model. The loss plot for our GAN can be viewed in Figure 14 while the loss plot in Figure 15 shows how the loss function worked when we tried the model with 10,000 epochs.

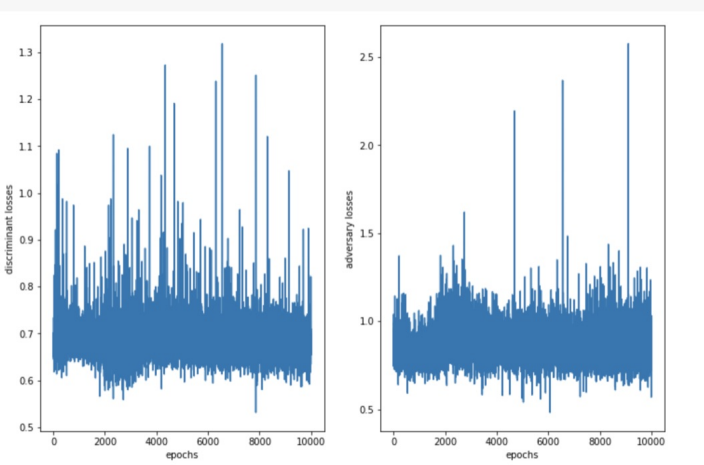

Figure 14: Loss plot for GAN 10,000

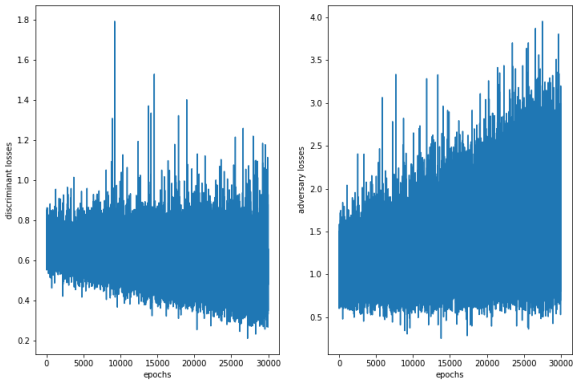

Figure 15: Loss plot for GAN 30,000

6.2 Convolutional Neural Network

The CNN is able to classify fake and real images with higher accuracy due to our robust use of different layers within our model. As can be seen in Figure 16, the accuracy had already crossed 99.5% on the 1st epoch. There is no significant change in the accuracy after the 2nd epoch. This proves that our model has performed extremely well with image classification.

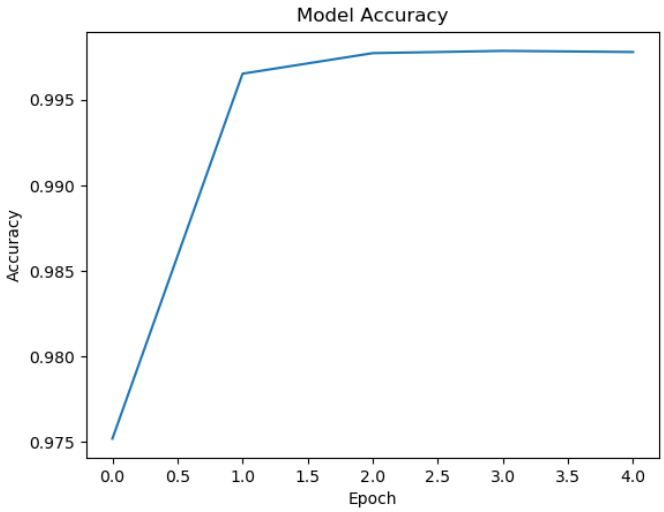

Figure 16: Training Accuracy for CNN

The loss plot in Figure 17 shows that the loss function in the training set (blue line) increases slightly until the 1st epoch and stays the same after that. This makes sense since our model is performing the same after the 1st epoch, as shown in our accuracy plot. Meanwhile, the loss decreases for the testing set (orange line) until the 1st epoch and then stays the same, showing our model is yielding great results after just the 1st epoch.

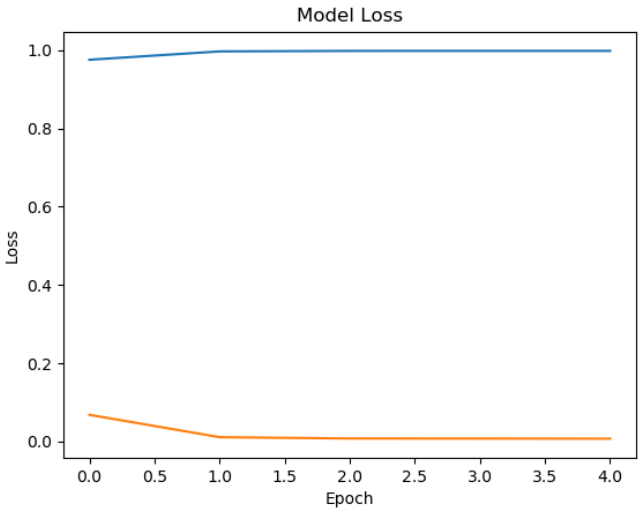

Figure 17: Loss for CNN

7 THREATS TO VALIDITY

While we are pleased with the results of both our GAN and CNN models, there are still threats to validity that should be included in the context of interpreting the results communicated in this research paper.

One of the main potential issues of this approach is the data that we chose to use. Due to the scarcity of readily available face image data, we chose to use the celeb_a data set available on TensforFlow. This dataset only includes images of celebrity faces, which can cause issue for our model, as it may not be highly generalizeable or representative of the full population. Thus, we could not necessarily apply our trained CNN to images of normal people and expect similar levels of accuracy. Celebrities tend to dress more nicely, have more distinctive features, and not fully represent all groups of the larger population, making it difficult to apply a model trained only on celebrities to detect real versus fake images of the broader population of civilians.

To account for this issue, we could spend more time and effort finding additional image data to use that includes many images of non-celebrities. For this project, the proof-of-concept was still achieved using celebrity data, but our model would be more robust with other image data included.

Additionally, if the GAN is not generating sufficiently realistic images of faces, it may be too easy for the CNN to pick out the fake images. In other words, if a human can tell the difference between the GAN-generated images and the real ones, it is not surprising that the CNN also had a high degree of accuracy when classifying them. We want a model that can differentiate between real and fake images when a human is unable to tell the difference.

This issue could be mitigated a number of ways. We could continue to tune the hyperparameters of the GAN until we are unable to tell the difference between real and fake images. Alternatively, we could reduce the amount of data the GAN trains on by cropping out some of the "noise" from the images–this includes distracting background features, excessive blank space, hands or large accessories being present in the image, or eliminating images where the person is not facing the camera head-on to simplify the features the model is attempting to learn. Had we implemented a more complex GAN, perhaps these considerations would not have a significant affect, but for our case, we must consider that they are playing a role in the model's ability to generate faces which are visually indistinguishable from real images.

Despite these potential threats to validity, we are confident in the process that led to the results we achieved with both the GAN and the CNN. Had we been able to iron out a few of these concerns, our model could have been somewhat more generalizable, but we still are convinced by these results that GANs are capable of generating fake images with a high degree of convincibility that they could be real, to the human eye. We are also still convinced that a CNN is an effective method of distinguishing between real and fake images.

8 CONCLUSIONS AND FUTURE WORK

The rapid innovations in image and video generation using GANs are extremely useful across industries that use image and video data. The ability to create data that humans cannot distinguish from real data is both incredibly powerful and somewhat daunting. With this technology becoming more widespread and easily accessible to those willing and interested to learn and use it, there are inherent risks that it will be used for ill-intentioned actions, like generating false images of real people doing scandalous activities that look real to the human eye. Additionally, fake social media accounts, fake videos, and other AI-generated media present a major issue for our generation. If we are to trust technology to be the powerful tool that it is designed to be, we need to not only be aware of what could go wrong, but also to have a plan to deal with it. Murphy's Law states that anything that can go wrong will go wrong, which can be extended to say that if technology has the ability to be used for evil or wrongdoing, it will be used for such purposes. As deflating as that statement is, it only becomes an issue if we are not prepared or knowledgeable on the proper means of mitigating or eliminating the unavoidable risks that come along with rapidly evolving technology. This paper is one attempt of many to understand both how fake images are generated and how to best detect and separate those images from real images.

In this paper, we presented a potential solution using a convolutional neural network to classify GAN generated images as real or fake. Our research group designed and trained a GAN on the celeb_a dataset, and then input the images into a CNN. We were able to successfully generate semi-realistic images of faces using the GAN. Some of the images were also realistic enough to confuse a human on whether or not they are real or fake. Furthermore, the CNN we designed and implemented for binary classification was correctly able to differentiate between real and fake images to an accuracy of 99.5% after just one epoch.

There are a few considerations to keep in mind with the interpretation of our results. Our CNN accuracy was high, at 99.5%, which may be a partial reflection on the quality of the GAN images we were able to create. While the GAN was very successful in creating realistic-looking images of faces, most of the images are still distinguishable as fake to the human eye, as becomes apparent when studying Figure 13. We consider this as a potential source of inflation for our CNN's accuracy. Because we included all of the features possible in the celeb_a dataset, that could have added to higher variability in the images created by the GAN.

While we are pleased with the success of our GAN to create realistic images of faces, future work could run additional tests on the CNN using image data created by other GANs to further validate that it works to a high degree of accuracy. Additionally, we could continue to tune the hyperparameters and number of epochs for the GAN to increase the similarities with real images.

Overall, this paper shows that even simple GANs are fully capable of generating realistic fake images and that CNNs can be used as an effective means of detecting fake images, even those that look visually indistinguishable from real images.

REFERENCES

[1] ABC Media gif of President Barack Obama https://abcmedia.akamaized.net/news/video/201809/obamaobamals.mp4

[2] Nataraj et al. 2019. *Detecting GAN generated Fake Images using Co-occurrence Matrices* https://arxiv.org/abs/1903.06836

[3] Guarnera et al. 2020. *DeepFake Detection by Analyzing Convolutional Traces* https://openaccess.thecvf.com/content_CVPRW_2020/papers/w39/Guarnera_DeepFake_Detection_by_Analyzing_Convolutional_Traces_CVPRW_2020_paper.pdf

[4] Hsu et al. 2020. *Deep Fake Image Detection Based on Pairwise Learning* https://www.mdpi.com/2076-3417/10/1/370/htm

[5] AlShariah et al. 2019. *Detecting Fake Images on Social Media using Machine Learning* https://thesai.org/Downloads/Volume10No12/Paper_24-Detecting_Fake_Images_on_Social_Media.pdf

[6] https://www.researchgate.net/figure/Block-diagram-GAN-training-module_fig1_324182115

[7] https://www.researchgate.net/figure/CNN-architecture-our-CNN-consists-of-three-convolutional-l fig1_306081277