ENSC 424: Detecting images generated with a Generative Adversarial Network

Kyle Smolko: 301337775
Roy Ataya: 301161512
John Ko: 301234320

Abstract

Generative Adversarial Networks (GANs) are an effective method of creating fake images that are nearly indistinguishable from the real thing. In this report, we create a tool that can be used to identify images of human faces that are generated using a specific type of GAN called a StyleGAN, and study its effectiveness on uncompressed (PNG) images, and compressed (JPEG) images with a quality factor of 75. We achieve this by training two Convolutional Neural Networks on the Red, Green, and Blue co-occurrence matrices of each image, one on a dataset of 6000 PNG images of human faces, and one on a dataset of 24,000 JPEG images of human faces. Our results show an accuracy of over 90% when tested on uncompressed PNG images, but we also found that the effects of compression reduce our accuracy to ~76% for JPEG images.

Introduction

Deepfakes are artificial content, usually in the form of pictures and videos of a person [1]. They are hard to distinguish from real images and videos. Generative Adversarial Networks (GANs), are used to create deepfake images. GANs have two components: The generator, which learns to create an output, and the discriminator, which learns to find differences between the output and real data. The generator and discriminator are trained against each other in order to get an optimal value of loss between the generator and discriminator [2]. This means the GAN model has found some optimal convergence where it cannot be improved. If this is the case, the model will now create images that are indistinguishable from the real thing.

StyleGANs are a specific type of GAN that we focus on. StyleGANs are a GAN with some added features. These features include Bilinear Upsampling, a Mapping Network, some additional Gaussian Noise in each block, Mixing Regularization, and removal of latent vector input to the generator [3].

The reason for deepfake detection research is because of the possible nefarious use cases. GAN generated images can be used for online disinformation campaigns, where one person can pretend to be many people with many different avatars or profile images. As such, there are those that try to develop ways to detect them. One such method used to detect GAN generated images is to train a deep learning model using the co-occurrence matrices of both real and fake images [4]. This method is a more generalized method of detecting deepfakes, but there are other methods that rely on detecting artifacts.

Project Description

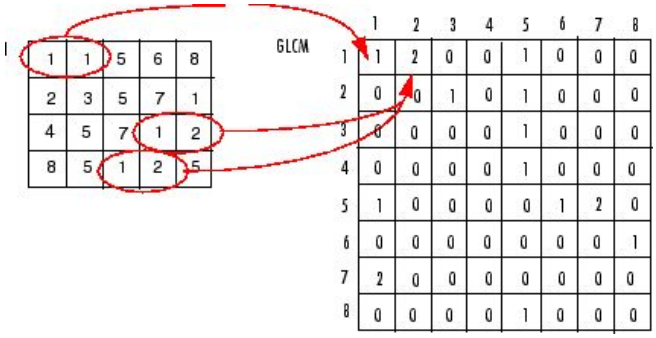
Goal

Our goal for our project was to identify GAN generated images using established methods mentioned previously. We focus on images generated using a StyleGAN, as described above. In particular, we trained a deep learning model using the co-occurrence matrices of real and fake images to detect deepfakes. We limited our scope to images of human faces only. We collected data on the success rate of our trained model (using a large dataset of test images), and we also provided an interface for another user to input their own images, which can be classified by our program. Our goal was to obtain an optimal accuracy for classification of PNG images, but we also decided to study the effects of JPEG compression on the accuracy of this process.

Theoretical Background

Co-occurrence Matrices

Co-occurrence matrices are matrices that describe the number of times a pair of values occur at a given offset from each other in another matrix [5]. An example is shown in Figure 1 below



https://www.mathworks.com/help/images/ref/graycomatrix.html

Figure 1: A co-occurrence matrix (right) of a matrix (left) that can store values 1 - 8, with the offset = 1

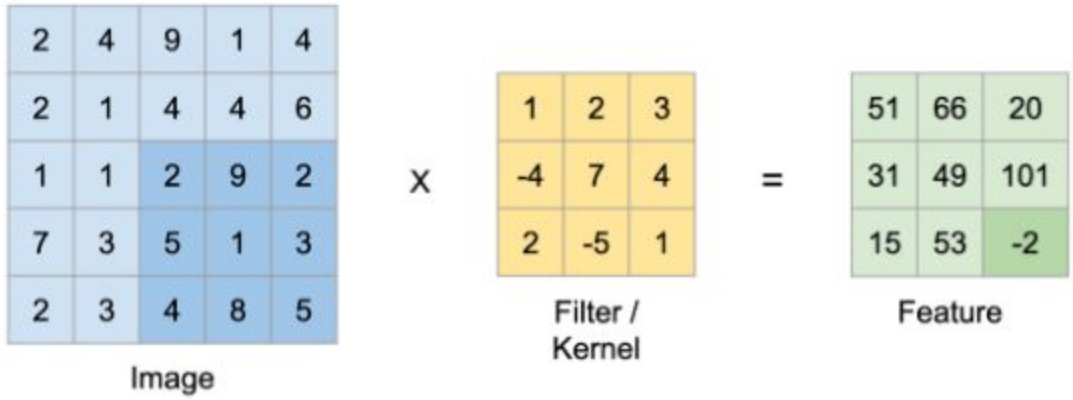
In the example above, since the values “1” and “1” occur only once at an offset of 1 space from each other, the value “1” gets placed in the co-occurrence matrix at position (1, 1). Likewise, since the values “1” and “2” occur twice at an offset of 1 space from each other, the value “2” gets placed into the co-occurrence matrix at position (1, 2).

One study that we have read suggests that there is a pattern hidden within these co-occurrence matrices of images that can identify whether or not they were generated using a GAN. This is the idea that we have used to base our project off of.

Model

Convolutional Layer

A convolutional layer is a very important component of any Convolutional neural network. One can think of this as a feature detector. It is the application of a filter onto an input image or tensor. A filter can be thought of as a 3D structure composed of 2D kernels. One of these kernels slides across an input matrix, where the area of the kernel is convolved with an area of the input [6]. The output is stored in a feature map and the size of the feature map depends on the stride length.



https://towardsdatascience.com/convolution-neural-networks-a-beginners-guide-implementing-a-mnist-hand-written-digit-8aa60330d022

Figure 2: Result of a 3x3 Kernel performing convolutions on a 5x5 input matrix

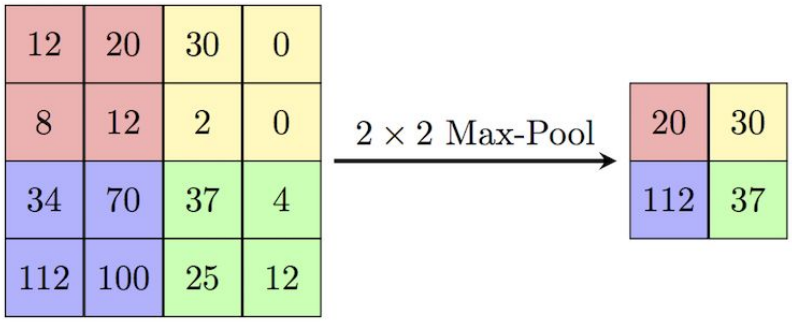
In Figure 2 above, we have a 3x3 kernel and a 5x5 input that results in a 3x3 feature map. The stride length in this example is 1. If the stride length was 2, the feature map would be 2x2 [6]. Furthermore, given a specified filter size, n, we would have an output of n feature maps, since we are doing kernel convolution with n different kernels.

RELU Activation Layer

Readers will notice some convolutional layers in our model, discussed later, will have an associated RELU activation function. This activation function is applied to the output feature map of a convolutional layer specified. The values of the feature map has a RELU activation applied. This means if the value is greater than or equal to zero then the value doesn't change. If the value is less than zero the value is changed to zero [6].

Max Pooling Layer

Max-pooling layers always come after convolutional ones. This layer will down sample a feature map, thereby reducing computational costs [7]. By pooling the features, it also reduces overfitting [7]. Like a convolutional layer, max pooling has a kernel. By default in Keras this kernel has a stride length of 2. So the kernel separates the feature map into areas where the maximum value of the area is taken, resulting in another feature map of smaller size.



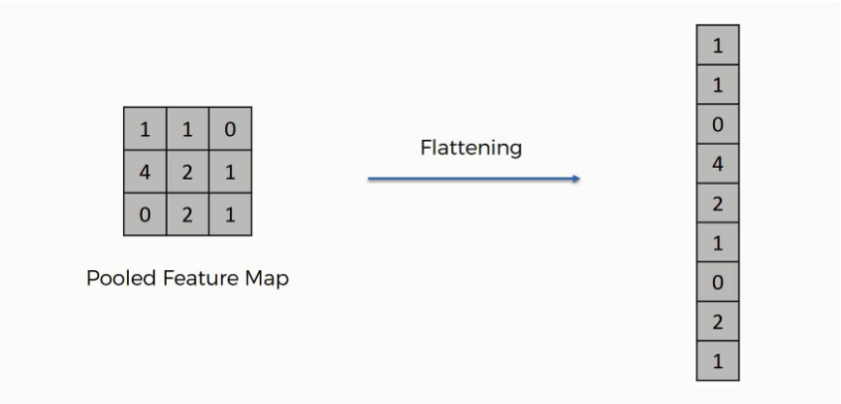
https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png

Figure 3: Graphical representation of the 2x2 Max-Pool of a 4x4 Matrix

In Figure 3 above, a 2x2 kernel, with stride length 2 separates the 4x4 feature map into 4 regions. The maximum value of each region is taken and put into a new feature map of reduced size in the case above 2x2.

Flatten Layer

Flatten will reshape an input feature map into a column array. Keras documentation requires this reshape to occur, so that the dense layers can take this array as an input [8]. Although Keras' documentation states this is done implicitly, it's not, hence a flatten layer is required. For example, a 3x256x256 tensor becomes a column array of 196,608 elements.

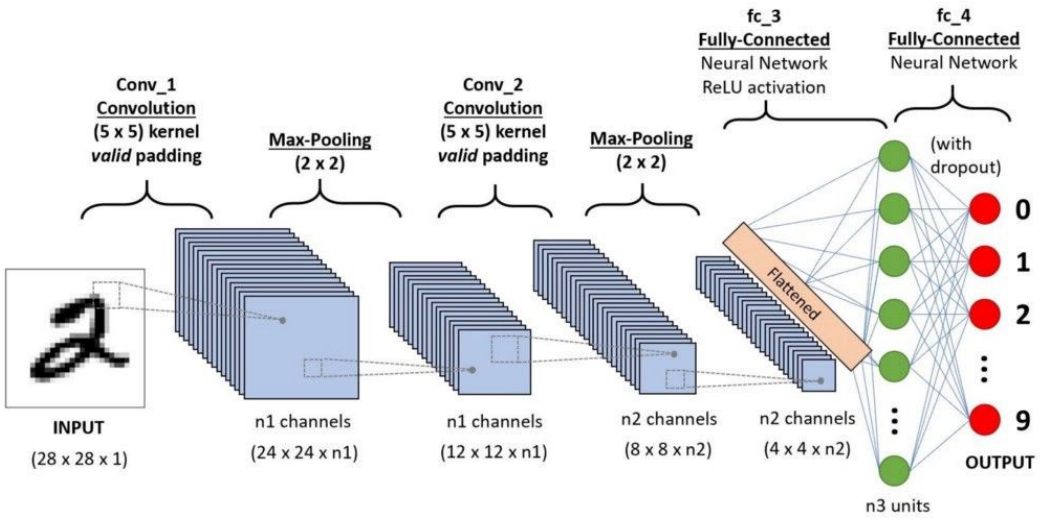


https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Figure 4: Graphical representation of the flattening of a 3x3 Matrix

Dense Layer

A dense layer is a fully connected layer that connects all inputs to all outputs. It performs classification based on all the features extracted from the layer beforehand, such as the convolutional layer and max pooling layers [9].



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Figure 5: Graphical representation of a dense layer with 10 outputs

Note: The example in Figure 5 above has 10 classes, but our model has 2. This shows all the layers and their place in the convolutional neural network. More specifically, the example illustrates where the dense layer(s) is in the model.

Optimizer and Loss Function

An optimizer updates the weights of the model via back-propagation and tries to minimize the loss function [10]. The loss function tells the optimizer if it's going in the right or wrong direction. In general, the optimizer does an iterative technique called gradient descent. The analogy is that if there is a mountainous terrain, we are trying to find the valleys. The valleys correspond to the global/local minimas, which we want to steer towards. Meaning the outputs of our model compared to desired output is similar, hence the loss is minimized. In regards to steering towards a minimum, a negative gradient is iteratively recalculated. A technique that will be relevant to other sections in this report is the learning rate of the optimizer. It may be that your local gets stuck in local minimums, so an option would be to increase the learning rate [10]. The learning rate dictates the size of steps taken to these valleys. If a learning rate is high it takes larger steps. This is a double edged sword, as you will go to a minimum faster, however, there is a chance to overshoot this global or optimal minimum [10]. A slower learning rate also has its issues. It will take longer to calculate negative gradients, but it may result in better steps towards optimal minimums. While the general principles are similar, the Adam optimizer algorithm, which is the optimizer we used, does things slightly different. Readers will see the learning rate is a relevant hyperparameter that needs to be tuned to get desired results. Tuning of the learning rate, with respect to the model proposed in this report, will be shown in a later section, specifically for our jpg model.

Dropout Layers

Dropout layers, and other regularization techniques, were not mentioned in the paper our model was based on. However, we introduced these layers to our model, as we found some overfitting occurring. A dropout is applied to a layer, where it will randomly set several output features of the layer during training to zero [11]. Keras allows users to control the dropout probability. The idea, made known by Hinton, is to introduce noise so the feature maps produced will have patterns removed that happen to be not of consequence [11]. Doing so helps combat overfitting in the model.

Implementation Details

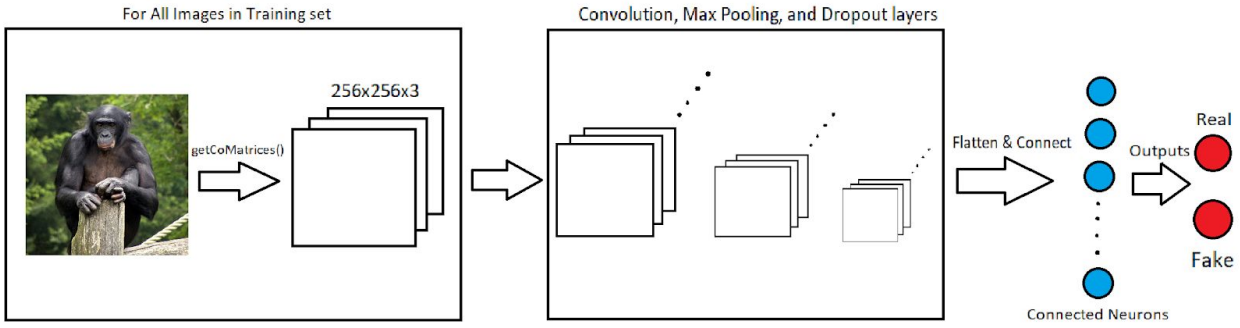


Figure 6: Illustration of our implementation

Co-occurrence Matrices

For our purposes, we extracted the co-occurrence matrices of each image in the Red, Green, and Blue (RGB) colour channels within the getCoMatrices function that we wrote. This means each image will give us three co-occurrence matrices. Since the RGB colour channels can hold values between 0 and 255, each co-occurrence matrix will be of size 256x256. That means for each image, we create a tensor of size 3x256x256 that we can then feed into our model for training.

Creating the Model

Our model was based on the model proposed in the paper, “Detecting GAN generated Fake Images using Co-occurrence Matrices” [4]. The authors proposed a neural network architecture composed of 6 convolutional layers, where 3 of them were paired with RELU activation functions. Moreover, the model has 3 max pooling layers, 1 flatten layer and 2 dense layers. One deviation from the original model includes adding 4 dropout layers to help with overfitting. The dropout layers were placed after max-pooling layers and between dense layers, as this placement was recommended by various literature [12]. As a result, our model had less overfitting. Moreover, we chose an optimizer, a loss function and hyperparameters that differ from the paper as they worked well with our model, or because they were never specified. Changed hyperparameters include the learning rate, epoch, and batch size, which we determined using trial and error. Our model uses the Adam optimizer, paired with a categorical cross-entropy loss function. However, the cross-entropy loss function reduces to a binary function as our model has a binary output. Our group tried to use the binary cross entropy loss function, which also required an extra dense layer that has 1 neuron, combined with a sigmoid layer, but we had better results with the latter. This model is created by our createModel function using Keras. The exact specifications of the model

layers is shown in table 1 below. The layers in the table follow sequentially exactly like our model.

Conv2D(32, (3,3), activation = 'relu', input_shape=(256,256,3))	This is a 2D convolutional layer, where the first parameter, 32, is the size of the filter. The (3,3) is the kernel size, so we have 32, 3x3 kernels. In keras you can have an activation layer immediately follow the convolutional layer by having it in the function parameter, like the relu activation in our example. The input shape can be specified, although this is used for the start of the model. We used the defaults values for padding and stride length, which is 0 and 1 for the latter.
Conv2D((32, (5,5)))	Has a filter of 32, 5x5 kernels. Default stride and padding.
MaxPooling2D((2,2))	This is a max pooling layer where the function parameter is the size of the kernel in this case 2x2. For all max pooling layers we used the default stride length of 2.
Dropout(0.1)	The argument inside the dropout layer is the probability of dropping out occurring, in this 10%.
Conv2D((64, (3,3), activation='relu'))	Has a filter of 64, 3x3 kernels and paired with a relu activation function. Default stride and padding.
Conv2D((64, (5,5)))	Has a filter of 64, 5x5 kernels. Default stride and padding
MaxPooling2D((2,2))	Same as above max pooling layer
Dropout(0.2)	Dropout with 20% probability
Conv2D((128, (3,3), activation='relu'))	Has a filter of 128, 3x3 kernels and paired with a relu activation function. Default stride and padding
Conv2D((128, (5,5)))	Has a filter of 128, 5x5 kernels. Default stride and padding
MaxPooling2D((2,2))	Same as above max pooling layer
Dropout(0.25)	Dropout with 25% probability
Flatten()	Reshapes feature map for the dense layer.
Dense(256)	A fully connected layer with the argument that specifies the number of neurons, in this case 256 neurons.
Dropout(0.5)	Dropout with 50% probability
Dense(256)	Same as described above.

Table 1: List of layers in our model

Dataset Generation

Deep learning requires generation of datasets and labels to be run through a model. We created a function called genDs to do this. Readers may ask why our group did not use Keras' function to generate datasets. This was done because Keras did not have a data augmentation option for creating grey co-occurrence matrices. GenDs takes arguments for folder paths, real and fake image folders, to create training and validation datasets with corresponding sets of labels. This function also makes sure the dataset type and shape will be compatible with the function that trains our model. Once the function is done generating the datasets it returns four lists: A list of training tensors, a list of training labels, a list of validation tensors, and a list of validation labels.

Training the Model

Generated datasets are passed to the plotAccuracy function, so training can begin. Training is done using Keras' fit function. The fit function allows users to adjust hyperparameters such as batch size and number of epochs. Batch size is the number of images that will be run through the model at once. Epoch is the number of passes the entire training set has completed. Tuning these parameters may result in better performance. For our PNG model, we used a batch size of 32, which is the default value provided by Keras' fit function, and we used 50 epochs, as we noticed that the accuracy of our model levels off after that point. For our JPEG model, we used a batch size of 40, and 50 epochs, as we noticed that more epochs caused our model to begin overfitting, dropping the validation accuracy.

Test Harness

We have included a tool called "test_script.py" that can be used by our users (such as the Professor, TAs, or other students) which takes the models supplied by our training script, and uses it to classify a set of images within a supplied folder. The script uses two models: One for PNG images, and one for JPEG images. Depending on whether or not the images within the supplied folder is a PNG or JPEG, the script will select one of the two models to test that image, and provide an output of whether the image is "Real" (not generated using a StyleGAN) or "Fake" (generated using a StyleGAN).

All images to be tested are expected to be of size 1024x1024, as our dataset that we trained with contained images of only this size. Images of a different size may have co-occurrence matrices that are quite different. As such, our test script resizes any images that are not this size to be 1024x1024.

Results

PNG model results

For our PNG model we used 6000 images which are 3000 fakes and 3000 real, we used an equal number of images for fake and real faces to avoid biasing our model one way or the other. The datasets are broken down into 80% training and 20% validation split, to ensure that we add adequate amount of images for training with a respectable amount for validation that our model worked properly. Additionally, we used 20 images for testing, which were all unrelated to our training and validating sets. In the end we achieved a total accuracy for our model of 90.5% on our validation dataset with 50 epochs and a batch size of 32. After saving our model we used a decoupled test dataset of 2000 images to test our model and we obtained the following results: 90% accuracy on fake images, 94.6% accuracy on real images, and an overall accuracy of 92.3% accuracy.

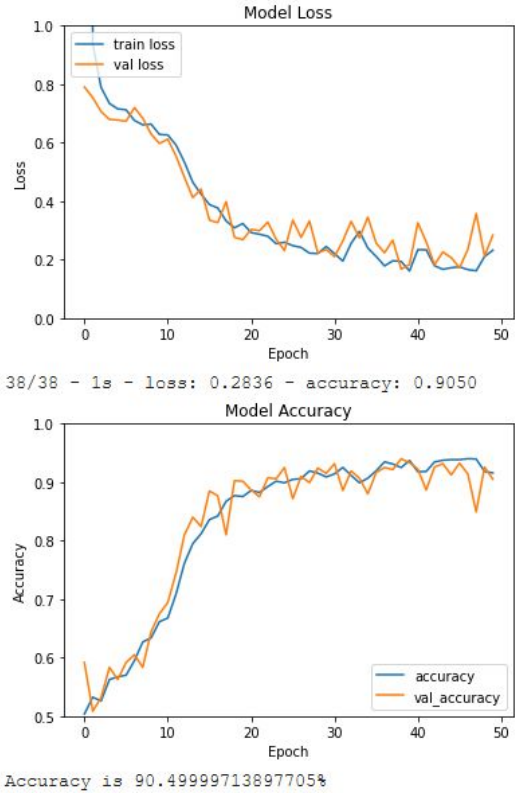
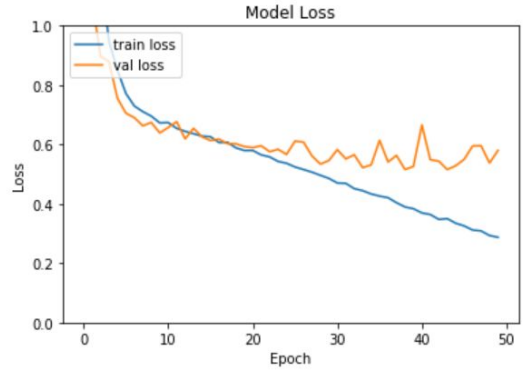


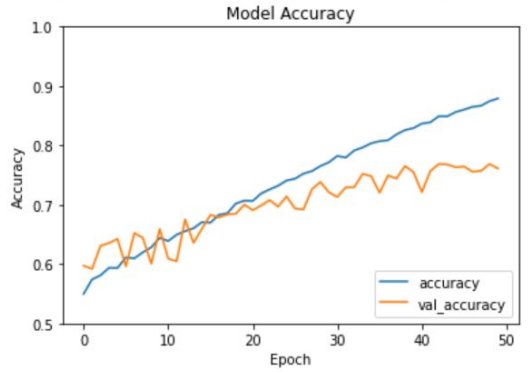
Figure 7: Validation loss and accuracy of our model for Uncompressed PNG images

JPEG model results

For our JPEG model we used 24000 compressed images (which we converted from PNGs to JPEGs using our PNG_to_JPEG script) which are 12000 fakes and 12000 real, we used an equal number of images for fake and real faces to avoid biasing our model one way or the other. Each of the JPEG images had a quality factor of 75. The datasets are broken down into 80% training and 20% validation split, to ensure that we add adequate amount of images for training with a respectable amount for validation that our model worked properly. Additionally, we used 2000 images for testing, which were all unrelated to our training and validating sets. In the end we achieved a total accuracy for our model of 76.1% with 50 epochs, a batch size of 40, and a learning rate of 7.5e-6. After saving our model we used a decoupled test dataset of 2000 images to test our model and we obtained the following results: 85.5% accuracy on fake images, but 69.6% accuracy on real images, as the effects of compression made it more difficult to find a pattern that defines an image as real.




```
150/150 - 4s - loss: 0.5799 - accuracy: 0.7610
```




```
Accuracy is 76.10416412353516%
working model fit
```

Figure 8: Validation loss and accuracy of our model for compressed JPEG images

Conclusion

Contributions

Task	Person	Contribution %
Proposal	Roy/John/Kyle	33/33/33
Research	Roy/John/Kyle	33/33/33
Dataset	Kyle	100
Environment Setup	Roy	100
getCoMatrices()	Kyle	100
trainModel()	Kyle/John	50/50
genDS()	John	100
PlotAccuracy()	Kyle	100
DataAugmentation()	Roy	100
png_to_jpg()	Roy	100
test_script.py	John/Roy/Kyle	33/33/33
Presentation	Roy/John/Kyle	33/33/33
FInal Report	Roy/John/Kyle	33/33/33
README	Kyle	100

Table 2: Contributions of each member

Summary

One method used to detect GAN generated images is to train a deep learning model using the co-occurrence matrices of both real and fake images. Our goal for our project was to identify GAN generated images using established methods mentioned previously. We focused on images generated using a StyleGAN. In particular, we trained a deep learning model using the co-occurrence matrices of real and fake images to detect deepfakes. With PNG images we achieved an overall accuracy for our model of 90.5%; and for our test data we obtained 90% accuracy on fake images, 94.6%

accuracy on real images, and an overall accuracy of 92.3% accuracy. With JPEG images we achieved a total accuracy for our model of 76.1%; and for our test data we obtained the following results: 85.5% accuracy on fake images, and 69.6% accuracy on real images.

Further Exploration

In the future we would like to improve our JPEG model so that it is on par with our PNG model. This will require us to tune our parameters more finely. Additionally, we would like to train one model that would work with multiple image formats, which was not computationally feasible for us at this time as this would require a much larger dataset.

References

[1] “Introduction — Generative Adversarial Networks — Google Developers,” Google. [Online]. Available: https://developers.google.com/machine-learning/gan. [Accessed: 16-Oct-2020].

[2] F. Farnia and A. Ozdaglar, “GANs May Have No Nash Equilibria,” arXiv.org, 21-Feb- 2020. [Online]. Available: https://arxiv.org/abs/2002.09124. [Accessed: 16-Oct-2020].

[3] T. Karras, S. Laine, and T. Aila, “A Style-Based Generator Architecture for Generative Adversarial Networks,” 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Mar. 2019.

[4] L. Nataraj, T. M. Mohammed, B. S. Manjunath, S. Chandrasekaran, A. Flenner, J. H. Bappy, and A. K. Roy-Chowdhury, “Detecting GAN generated Fake Images using Co-occurrence Matrices,” Electronic Imaging, vol. 2019, no. 5, 2019.

[5] “Create a Gray-Level Co-Occurrence Matrix - MATLAB & Simulink,” MathWorks. [Online]. Available: https://www.mathworks.com/help/images/create-a-gray-level-co-occurrence-matrix.html [Accessed: 7-Dec-2020].

[6] A. Gibson and J. Patterson, "Deep Learning: A Practitioner's Approach," Sebastopol, O'Reilly Media, 2017, pp. 250-265.

[7] A. Gibson and J. Patterson, "Deep Learning: A Practitioner's Approach," Sebastopol, O'Reilly Media, 2017, pp. 266-267.

[8] A. Gibson and J. Patterson, "Deep Learning: A Practitioner's Approach," Sebastopol, O'Reilly Media, 2017, pp. 614-616.

[9] A. Gibson and J. Patterson, "Deep Learning: A Practitioner's Approach," Sebastopol, O'Reilly Media, 2017, pp. 268.

[10] F. Chollet, "Deep Learning With Python," Shelter Island N.Y, Manning Publications, 2018, pp. 48-52.

[11] F. Chollet, "Deep Learning With Python," Shelter Island N.Y, Manning Publications, 2018, pp. 104-111.

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research,* 2014.

Software

We used Python 3.8.6 as our programming language (Note: Tensorflow currently does not support Python 3.9 yet). The specific libraries we used are Tensorflow 2.3.0 with Keras 2.4.3 for creating our dataset and training our model. We also used OpenCV 4.1.2 and Scikit-image 0.16.2, and Pillow 8.0.1 for processing our images and extracting the co-occurrence matrices from them. We also used Numpy 1.18.5 for our data structures and Matplotlib 3.3.3 for plotting our results.

Our dataset was sourced from the Flickr Faces HQ (https://github.com/NVlabs/ffhq-dataset) dataset for our real images, and from the Nvidia style deepfake dataset (https://github.com/NVlabs/stylegan) for our fake images.