Capstone #2 Final Report

Introduction

Question at hand

In this age of technological innovation, we are observing rapid advancements in fields such as computer vision. Computer vision can (and soon will) revolutionize aspects of daily life such as driving, shopping, and security. Our face recognition algorithms, for example, are quickly improving. While it may be helpful, face recognition has serious implications on privacy and surveillance. One cannot be certain about the intention of those who use this technology, which may motivate some people to find ways to maintain their anonymity. Is there a way to assist them by allowing them to create an anonymized version of their face?

Who would use it and why?

The users of this model would be people who would like to find ways to outsmart current face recognition algorithms. This project will attempt to extract features from both human and cat faces and provide an interface where one can adjust these features to create a hybrid face of their choosing. For instance, if we replace human ears with cat ears in a picture of someone's face, the face is still recognizable to humans, but perhaps not to computer vision algorithms. That way, people can still use a modified picture of their face for practical purposes while maintaining anonymity.

Data acquisition and cleanup

Acquiring the data

The cat faces were acquired from this Kaggle dataset. The dataset includes more than 9,000 images of cat faces along with the coordinates of certain facial features such as the eyes, ears, and mouth. Unfortunately, the pictures are not centered around the cats' faces, and they are not square, so some work needs to be done to transform the images into the format I need. In order to access the data, I simply downloaded the dataset from Kaggle. Each image had an accompanying catalog file (.cat extension) which had the location of each annotation. These images and catalog files were separated into seven folders.

The human faces were acquired from this dataset[1]. This dataset includes more than 100,000 images of celebrity faces. These images are already centered around the person's face and are square, so minimal work needs to be done to prepare the images.

[1] H.-W. Ng, S. Winkler. A data-driven approach to cleaning large face datasets.
Proc. IEEE International Conference on Image Processing (ICIP), Paris, France, Oct. 27-30, 2014.

In order to access the data, I needed to submit a request to the principal investigator of the research study that the images were originally prepared for. I was then able to download a folder that contained links to the images of the various celebrities' faces and the bounding boxes for the face in each image. The folder also had a script that downloaded the images and cropped them using the bounding boxes. The original and cropped images were then separated by celebrity. All of these steps were done automatically through the download script.

Image preprocessing

The cat faces were preprocessed using an algorithm that utilized the annotations to rotate, center, and crop the images around each cat's face. The steps to the algorithm are as follows:

1. Rotate the image such that the two eyes are level.
2. Establish the bounding box such that the upper limit is slightly above the tops of the ears and the lower limit is slightly below the mouth.
3. Crop the image into a square centered around the cat's face, adding a padding if the bounding box falls outside the image.

Rotating the image: The angle by which the picture needed to be rotated (θ) was calculated using the slope of the line connecting the two eyes. If the rotation angle was negative, the image was flipped before the rotation was applied. The image was then rotated counterclockwise by θ with the center of rotation taken to be the midpoint of the two eyes.

Creating the bounding box: The vertical distance between the mouth and the top of the higher of the two ears was calculated as *h*. The upper limit of the box was determined by adding a padding of 10% of h above the ears. Similarly, the lower limit of the box was determined by adding a padding of 20% of h below the mouth. The bottom of the box required additional padding so that the chin could be included in the cropped photo. Because we need our final image to be a square, the width of the bounding box was set equal to the height and the vertical center of the box was taken to be the midpoint of the two ears.

Cropping the picture: For some pictures, the bounding box fell outside the range of the picture itself. In order to account for this, a black border was added to the image before cropping. I calculated the distance from the bounding box to the image borders and determined if any part of the box fell outside the border. The size of the padding was taken to be the maximum amount of "spillover" so that we could retain as much of the image as possible. After accounting for spillover and adding a black border, the image was cropped and saved in a separate folder.

This process was repeated for every image until all of the images were processed. I was able to process all but two images, resulting in a final tally of 9,991 images. This is more than enough for the project. Figure 1 shows an example of the steps taken to clean the images.

Next, the human faces needed to be prepared. Most of this work was accomplished with the download script included with the dataset. The download script came with links to each of the images and the bounding boxes for the faces in each image. The script used the location of the

bounding box to crop the image and store the faces in a separate directory. Figure 2 below outlines this procedure.

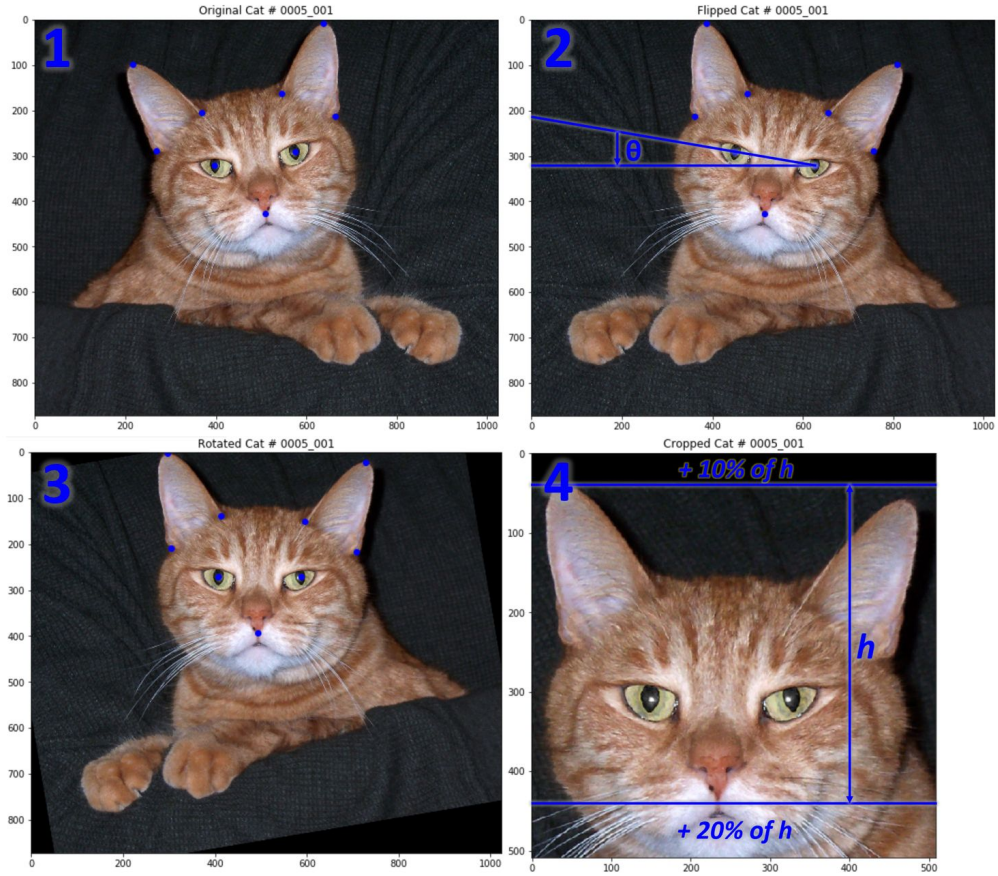


Figure 1: a depiction of the steps taken to clean the cat images. This series of steps was performed on the entire dataset, resulting in 9,991 cleaned images. Note the black bar on the top of image 4. This was added because the bounding box fell outside the range of the image; thus, a padding was included.

The human faces were organized by celebrity, with each celebrity in its own directory. Inside those directories, a subfolder titled “faces” held the processed images. In order to extract all of the faces into one single directory, I had to write a custom shell script to parse through all of the directories and move all the faces into a separate directory.

Removing outliers

As my algorithm went through the cat images, it printed the names of any cats that it could not process (only two). These cats’ faces are vertical, which means the rotation angle would be around 90°. The tangent of this angle is not well-defined, so attempting to rotate the image resulted in an error. Additionally, upon visual inspection I came across another outlier. In this case, the annotations provided were incorrect. These three images were removed from the dataset. We have already acquired close to 10,000 images, so losing three images would not be an issue. Figure 3 shows the three outliers that were removed.

Figure 2: a depiction of the steps taken to process the human faces. This procedure was completed by a download script that accompanied the dataset.

After extracting all of the human faces to a separate directory, we ended up with 46,850 images. This is a far cry from the 100,000+ images the original dataset claimed. One possible reason for this large discrepancy is that the images from the dataset were obtained using links to images online. By the time I used the download script to gather these images, I am sure many of these sites were taken down. In fact, the download script was written in Python 2. Since there was a significant amount of time between when the links were sourced and when I attempted to download the images, this could have resulted in the decreased number of images we were able to obtain. Regardless, 46,000 images is more than enough to proceed with the project.

Finally, I wrote a simple script that resized all images to a size of 256 x 256 pixels. In doing so, I discovered that several images had somehow become corrupted in the data cleaning process. After discarding the corrupted images, the final image count was 56,709 (cats and humans combined).

All of these images were then uploaded onto Google Drive so that a Google Colab notebook can easily use the data without having to rely on my local files.

Figure 3: the three cat images that were removed from the dataset. The eyes of images 1 and 3 are nearly vertical, so the tangent of the rotation angle could not be calculated. The annotations for image 2 were inaccurate and resulted in the ear being cropped instead of the face.

Building the neural network

Model concept and architecture[2]

The model I used for this problem is an autoencoder. The goal of this type of neural network is to compress input images into a low-dimensional representation (known as a "latent space") and decode the latent space to recreate the original image as closely as possible. This process is very similar to the compression algorithm used when converting an image to JPEG file format. JPEG images are created by compressing the input image just enough to where the original image can be recreated with only a minor loss in quality. The more the input image is compressed, the more degraded the JPEG image will be.

JPEG's compression algorithm typically uses a 10:1 compression ratio. As an example, the images used for this project are 256 x 256 pixels with three color channels, which yields an input dimension of 196,608. If we were to save the images in JPEG file format, they would be

[2] This project was based on CodeParade's YouTube video and its respective source code.

compressed into a 19,660-dimensional space. In comparison, my (initial) model's latent space has only 100 dimensions, which represents a compression ratio of 1,966:1. This large discrepancy is due to the fact that autoencoders recreate images based on features specific to the dataset it is trained on. JPEG's compression algorithm must learn a wide variety of features since it should be able to recreate any possible image. Our model, on the other hand, is only tasked with recreating cat faces and human faces; it does not have to learn nearly as many features and can thus compress to a much larger degree.

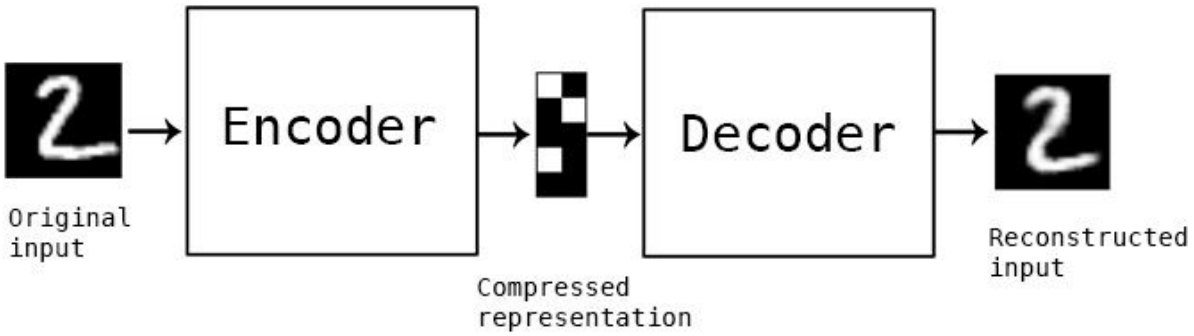


Figure 4: Basic concept behind autoencoders. Image taken from this Keras blog post about autoencoders.

Autoencoders come in a variety of types, each with their own function. The type I used is a variational autoencoder, which is a "generative" model that provides access to the values of the latent space vectors. The values of these vectors can then be manually adjusted and decoded using the decoder network, resulting in a "new" image based on features learned by the network. Since we only need the latent space and the decoder to generate these images, we can do away with the encoder entirely. Instead, we can simply embed the inputs directly into the latent space and decode the latent space values. Thus, our model architecture will consist of an "encoder" (embedding and flattening the inputs), the latent space (100 dimensions), and a decoder (2D convolutional transpose layers). Now we can start building and training the model.

Initial model

The initial model had a latent space of 100 dimensions and was trained on all 56,709 images. The input data was loaded in batches of 500 randomly sampled images (250 cats and 250 humans) and trained for 20 epochs per batch. The YouTube video I based my project on started with a dataset of 1,769 images, augmented them to make 35,380 samples, and trained the model for 2,000 epochs. As a result, their model saw 70,760,000 images over the course of its training period. I decided to have my model see the same number of images, so I attempted to train it for 7,076 batches, resulting in 141,520 epochs total.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1, 100)            50000
_________________________________________________________________
flatten_1 (Flatten)          (None, 100)               0
_________________________________________________________________
reshape_1 (Reshape)          (None, 1, 1, 100)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 3, 3, 64)          57664
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 5, 5, 64)          36928
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 7, 7, 32)          18464
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 15, 15, 32)        9248
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 31, 31, 16)        4624
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 63, 63, 16)        2320
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 127, 127, 8)       1160
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 256, 256, 3)       387
=================================================================
Total params: 180,795
Trainable params: 180,795
Non-trainable params: 0
```

Figure 5: Summary of initial neural network. This model is a sequential network consisting of an embedding layer and several convolutional transpose layers.

This model took many, many days of round the clock training on Google Colab's GPU engine, even after upgrading to Colab Pro. I monitored the loss and generated sample predictions throughout the duration of the training. At around 86% completion, I noticed that the predicted images were still very low-level. They were still completely brown/beige and green had just been introduced in the reconstructed images. The loss function also left much to be desired. The training and validation loss more or less started to converge, but it was still too high to make accurate predictions. I had expected to see much better performance by this point.

The main problem was that the variation in the dataset was far too high for the amount of training time I had planned for. Either I could continue training the model until it started to make accurate predictions, or I could scale back heavily; I opted for the latter since I was not sure how much extra time the model needed.

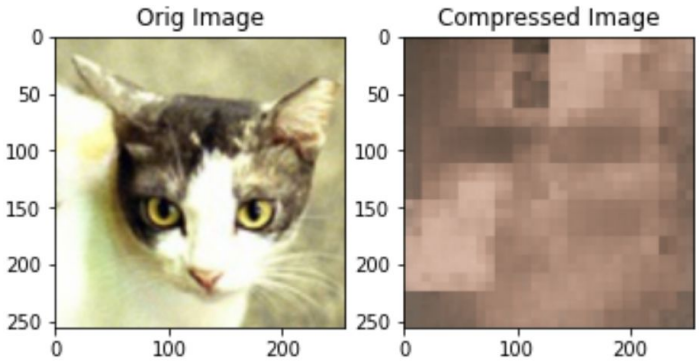


Figure 6: Example of a reconstructed image from the initial model. The model had just started to learn about eye position 86% of the way through training. Clearly, a new approach was needed.

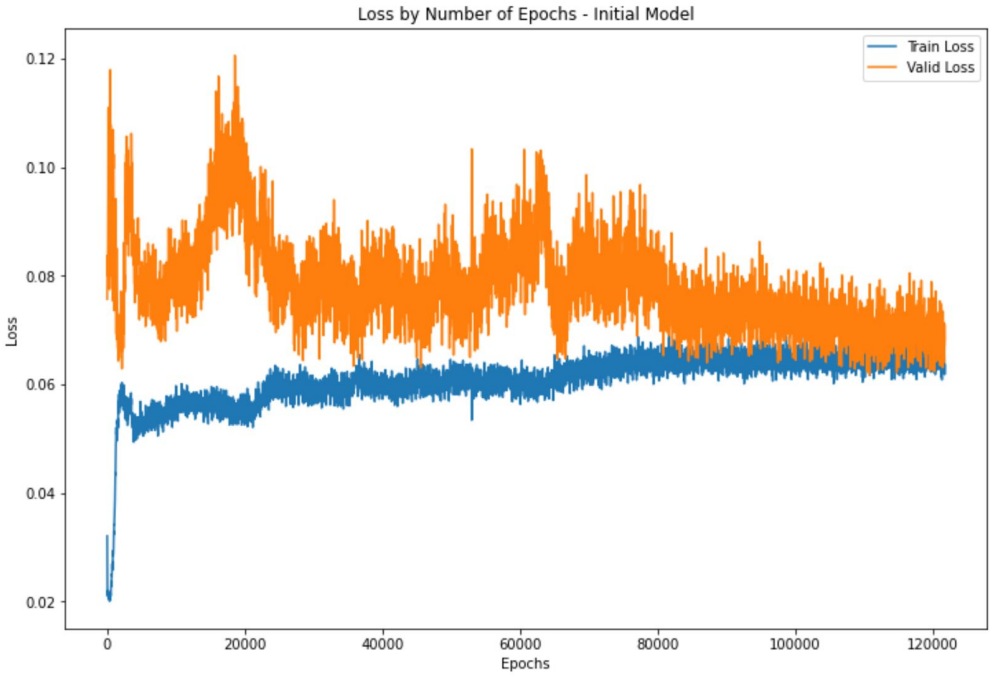


Figure 7: Plot of training and validation loss vs training epochs. They started to converge but the loss was still too high to make accurate predictions.

Reduced model

This time, I used only 200 images (100 cats and 100 humans) and trained the model for 20,000 epochs. Additionally, I decided to reduce the latent space from 100 dimensions to 25; I wanted each latent space vector to represent a specific feature, and 100 unique features from just 200 images seemed like too many. This represents a compression ratio of 7,864:1. The remaining model architecture was left unchanged.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1, 25)             5000
_________________________________________________________________
encoder_output (Flatten)     (None, 25)                0
_________________________________________________________________
decoder_input (Reshape)      (None, 1, 1, 25)          0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 3, 3, 64)          14464
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 5, 5, 64)          36928
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 7, 7, 32)          18464
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 15, 15, 32)        9248
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 31, 31, 16)        4624
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 63, 63, 16)        2320
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 127, 127, 8)       1160
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 256, 256, 3)       387
=================================================================
Total params: 92,595
Trainable params: 92,595
Non-trainable params: 0
```

Figure 8: Summary of reduced neural network. The latent space has been reduced to 25 dimensions.

This model only took around one hour to train and generated much better reconstructed images. The validation loss was very high compared to the training loss, which is clear evidence of overfitting. Overfitting was not a concern for me, however, since I was not using this model to generate brand new images; I was simply manipulating the existing model to explore the range of images it could provide. Once training was complete, I was satisfied with the current model and decided to proceed with building the application.

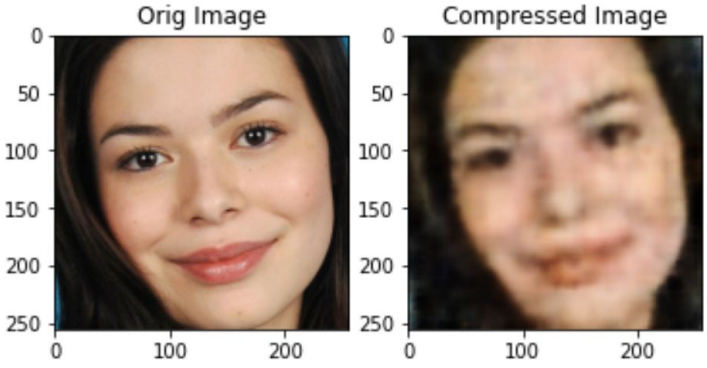


Figure 9: Example of a reconstructed image from the reduced model. The image is visibly degraded but still recognizable.

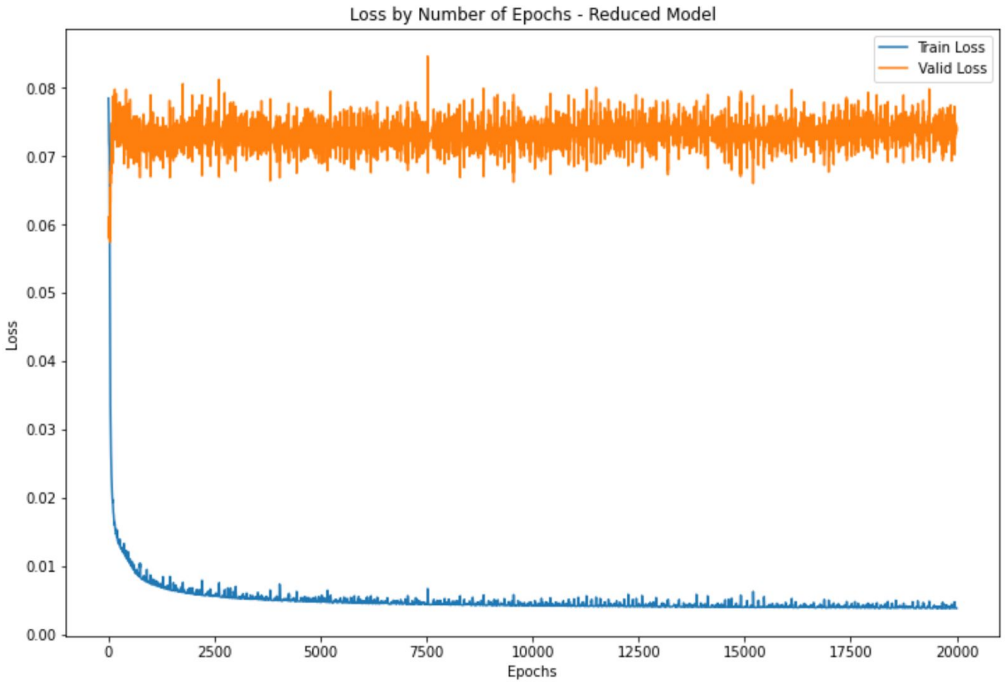


Figure 10: Plot of training and validation loss vs training epochs. This is clear evidence of overfitting but I proceeded with this model anyway.

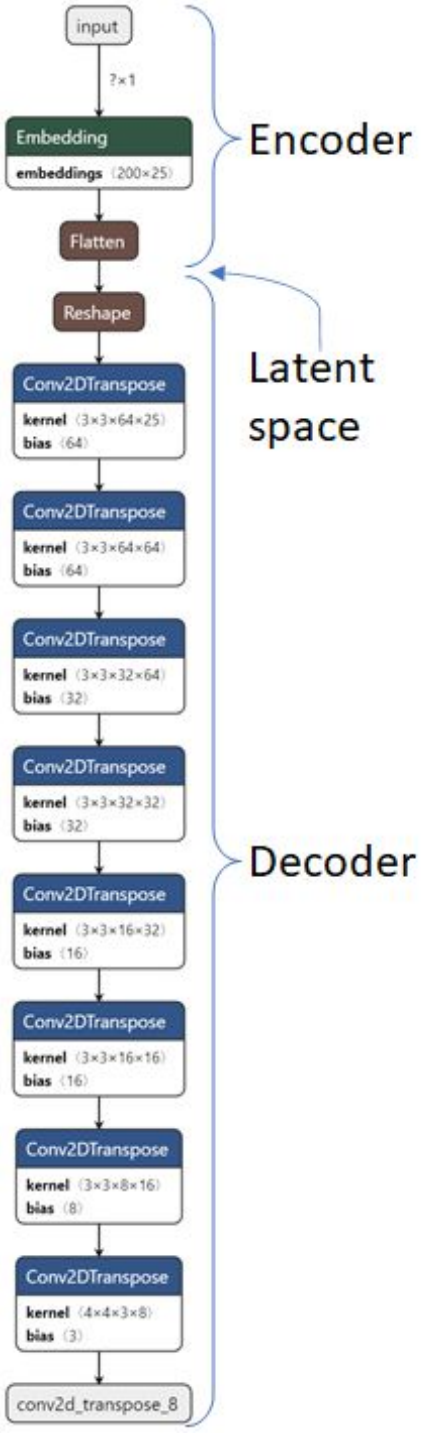
Creating the application

How it works

One the model is loaded, we need to separate the encoder and decoder halves at the 25-dimensional latent space layer in order to treat them as two separate functions. Since this model did not have an encoder, the embedding portion of the network became the “encoder.” The latent space vector was the output of the “encoder” function and the input of the “decoder” function. This allows us to directly adjust the values of the latent space vectors, feed it through the decoder, and observe the generated image.

The aim of accessing the latent space is to isolate 25 individual features that can be adjusted independently of one another. Collinearity among the vectors will not result in clearly defined features. Principal component analysis (PCA) was used to transform the 25-dimensional space into another 25-dimensional space where the components are linearly independent. These 25 components were then connected to sliders, allowing us to directly adjust their values. The new set of values were then transformed back into the original space and fed through the decoder to generate an updated image.

Unfortunately, adjusting the principal components did not yield any meaningful results. The image was changing based on the values of the sliders, but I was not able to formulate any



definitive conclusions about what each component represented. Perhaps the model was not performing well enough to be able to extract individual features, or maybe the reduced variability in the dataset caused the model to struggle to generalize the predictions. Either way, I changed the sliders so that they updated the latent space vectors directly. Despite the effects of collinearity among the vectors, I was able to generate some interesting images.

App interface

The app was created using Dash. The sliders control the values of the latent space vectors in terms of standard deviations away from the mean value of each component. When the app is initialized, all values are set to zero. The panel on the left can be used to toggle through various controls:

- Randomize: randomizing the value of each slider along a normal distribution centered at zero with standard deviation 1
- Reset: setting all sliders to zero
- Cat Average: setting all sliders to the average values of latent space vectors specific to cat images
- Human Average: setting all sliders to the average values of latent space vectors specific to human images
- Select a Face: displaying one of the 200 images from the original dataset and setting sliders to the appropriate positions

It seemed as though some sliders controlled certain features about an image, but they did not control those features independently. For example, one of the sliders seemed to adjust the vertical shading of an image, but it did not do so cleanly and ended up changing many other parts of the image along the way. Further work needs to be done in order to bring the application to a point where the features can be clearly identified and adjusted independently of one another.

Figure 11: (above) Depiction of how the model was split into the encoder and decoder functions.

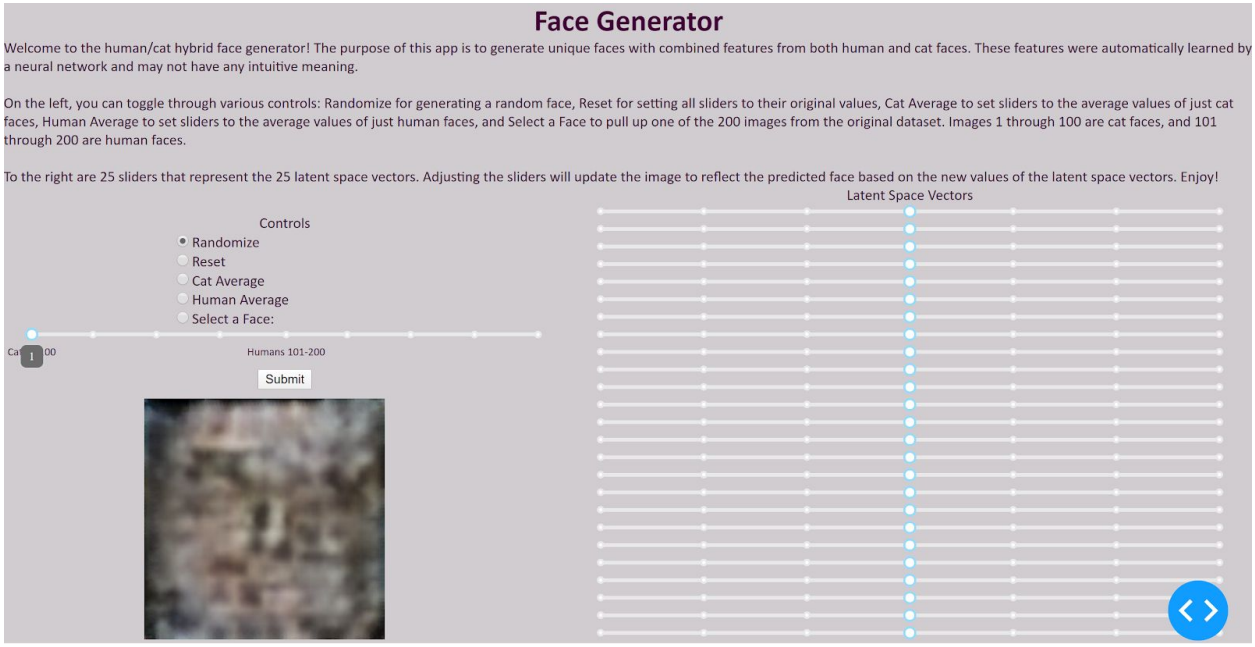


Figure 12: Interface of app immediately after launching. The picture displayed is the average value for each latent space vector over all images in the dataset.

Conclusion

While the model cannot be used to generate hybrid faces in its current state, it serves as a proof of concept for the use of variational autoencoders to generate new faces. In the future, I intend to redo the model architecture to create a symmetrical model that has distinct encoder and decoder halves. By doing so, the user can upload an image of their choice and the model can feed it through the encoder to generate a predicted image, even though it was not part of the original dataset. Also, once the performance of the model improves, I would change the sliders to control the principal components in order to extract meaningful features. Finally, once I am satisfied with the application, I can create a CI/CD pipeline to test and deploy the app. I would either use a Docker container and host it on a service such as Amazon Web Services, or use a Platform as a Service tool such as PythonAnywhere.

This project may not be ready for cybersecurity applications just yet, but it still shows us that we can find solutions to modern day problems in places we would not expect. As technology such as face recognition advances, so do tools to “cheat” the system, whatever the intention may be (apps of this kind should be used to scrub one’s identity for purely ethical purposes, of course). Our society seems to be growing both increasingly trusting and wary of the incorporation of technology into our daily lives. Hopefully, tools like this can give someone the peace of mind of knowing that out of all the ways we can be tracked, their face will not be one of them.