Generating Dog Images with Deep Convolutional Generative Adversarial Networks

Chad Malla
SFU Computing Science
[cmalla@sfu.ca]

Krystal Nguyen
SFU Computer Engineering
[hbn2@sfu.ca]

https://www.bbc.co.uk/newsround/48673084

ABSTRACT

Massive amounts of valuable data that exists in the world is unlabeled (e.g. videos and images). A supervised machine learning task requires labeled data as input, learn some pattern and use inferences to predict labels for unseen data. The problem is for a lot of complex supervised classification tasks there is a lack of labelled data. This motivates the topic of this paper: Deep Convolutional Generative Adversarial Networks (DCGANS) (Radford et al., 2015) which aim to address the instability that come from the original GAN (Goodfellow et al., 2014) architecture. This paper will only focus on the architectural constraints mentioned in DCGAN paper by Radford applied to a fun problem of generating dog images, a recent Kaggle competition.

1. INTRODUCTION

We will start with a non-technical analogous example of what a generative adversarial network is doing. The story is about a counterfeiter named George who has zero knowledge about making fake currency bills. In fact, he has never seen money before, but chooses to make it his career. Dave, a detective, is responsible for discriminating between real and fake currency bills. Initially, George draws up something that is far from looking like legitimate currency. George passes this fake currency through a network of people carefully routed to be given to Dave. Dave is responsible for associating a numerical value to the real currency and fake currency he sees which he provides to another colleague of his. This colleague then takes the numbers associated with the fake currency to George. Yes, they are secretly working together to trick Dave. George makes some tweaks to improve how the currency should look like based off these numerical values. It doesn't improve much but George transfers through the network to get the fake currency to Dave. Dave doesn't have a clear distinction between a batch of real currency and fake anymore so using memory he tries to discriminate the fake currency from the real. He again associates values to the bills and this process continues as George tries to trick Dave whilst Dave discriminates.

George is a generator, a network responsible for making fake images from some noise. Dave is a discriminator, a network responsible for distinguishing between real images from a training dataset and fake images generated from the generator.

The paper will continue as follows:

1. Exploratory data analysis on the subset of the Stanford Image Dataset that contains pictures of only dogs
2. Listing the heuristics from *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* that provide stability to GANS
3. Walkthrough of our project, the continuous improvements to the network and the struggles with Tensorflow

2. EXPLORATORY DATA ANALYSIS

Since the dataset is an image dataset, the amount of exploratory data analysis was somewhat limited. Still we worked with the complementary *Annotation* folder that has an xml file associated with each picture. We were interested with some information [1] contained in each of the file, specifically the breed and image dimensions. To do this we used ElementTree from xml library to look for the xml tags that contained the information of interest. Refer to code to see how it was done. Once we got the data into Pandas dataframe, we were able to visualize it.

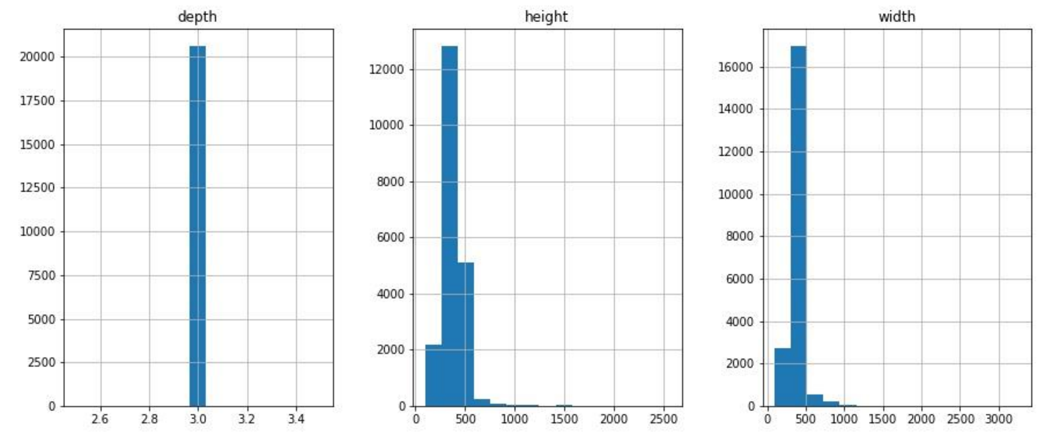
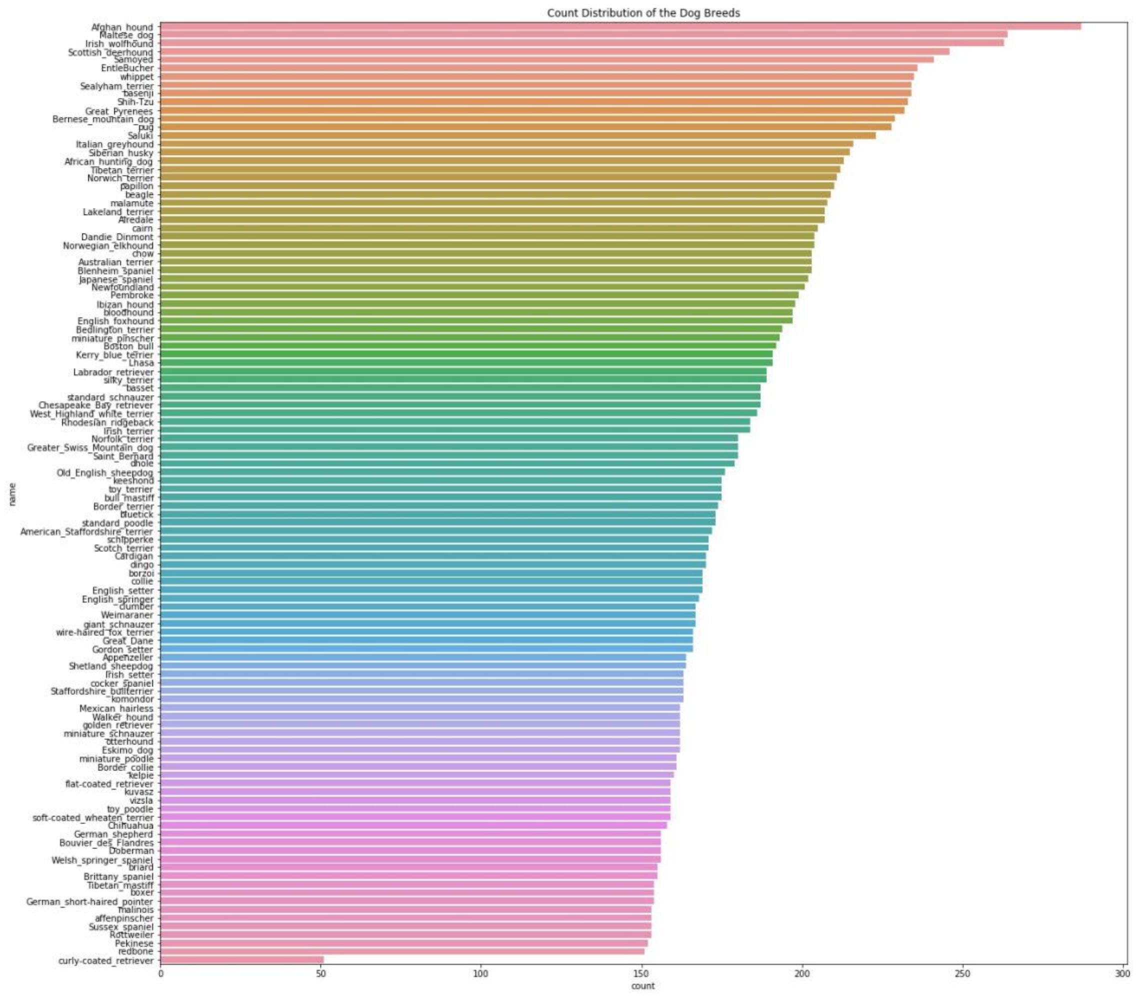


Figure 1 Distributions of image dimensions

Figure 1 shows that all the images are RGB (depth) but the sizes are different (height and width). This is a problem because the discriminator expects one size for its input. The preprocessing step required to resize every image to 64 by 64 which will be the same size as the generated images. The XML files also gave bounding box values which are dimensions of where exactly the dog is seen in the picture. We cropped each picture using these values in the preprocessing before resizing. However, the breed will be deterministic about what we should see as the results (e.g. images). To retrieve this information, we grouped the dataframe by the breed name and aggregated a count for each type. The following is count plot of the 112 different breed types.



The following 5 have the most pictures in the dataset:

BREED	COUNT
Afghan_hound	287
Maltese_dog	264
Irish_wolfhound	263
Scottish_deerhound	246
Samoyed	241

Our hypothesis is that the images that our generator outputs will mainly look like these particular dog breeds. Intuitively this makes sense because we are expecting to see dogs as the results because our dataset is dog pictures after using the bounding boxes. If we introduced a bias of a particular breed, then the model should learn to create images that look that breed.

3. GANS AND “HACKS” TO IMPROVE THEM

When Goodfellow first introduced Generative Adversarial Networks, they were found to be difficult to train. Both the generator and discriminator are networks that are adversaries to each other so improvements to one means the other suffers. This results with the generator not producing sensible outputs. They are blurry and suffered from checkerboard artifacts [1]. Radford proposed a set of heuristics or architectural constraints to the network to achieve better stability.

I will first explain the Generator and Discriminator in a DCGAN. The Generator is a network that will take a random vector of numbers from a distribution. For our project we used a normal distribution with a vector size of 100. The super interesting thing is that the Generator actually learns to create a picture of a dog given these random numbers through a network that reshapes the vector, puts the reshaped vector through transposed convolutional layers that upsamples and a final layer to flatten to create a 64 by 64 image. The discriminator is a regular convolutional neural network that essentially tries to classify the pictures as real or fake.

A deep convolutional generative adversarial network doesn’t have max pooling or fully connected layers but only uses convolutional and transposed convolutional layers. Transposed convolutional layers essentially do the opposite of a convolutional layer. In a convolutional neural network, the input will be an image in which the network downsamples to output a class value (e.g. MINIST dataset where the classes are [0-9]). The transposed convolutional neural network architecture looks like the following:

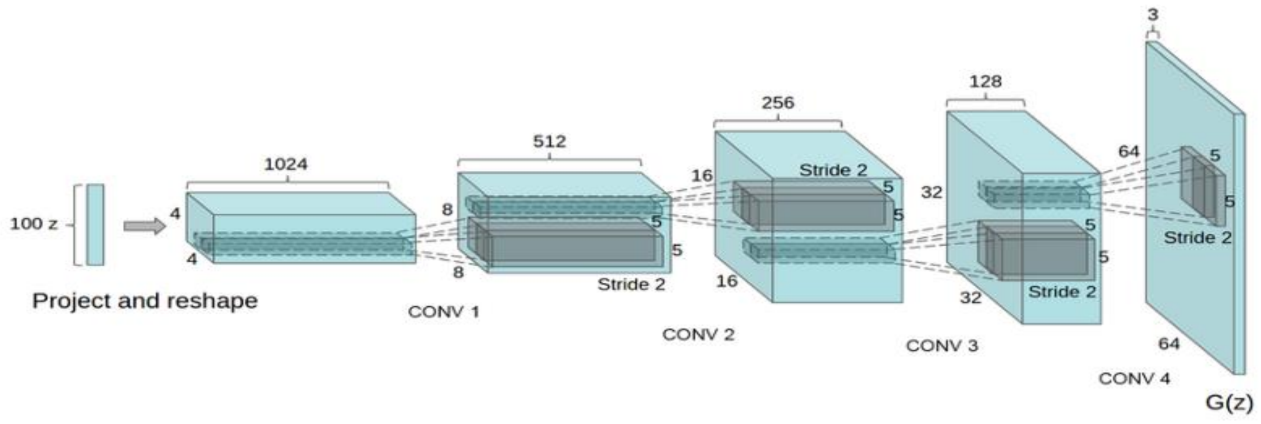


Figure 2 Radford et al., 2015

The following is a summary of the best practices of DCGAN that we applied to our project that are derived from Radford's paper.

1. Downsample using strided convolutions *(discriminator)*
2. Upsample using strided convolutions *(generator)*
3. Use LeakyReLu *(originally recommended ReLu for generator and LeakyReLu for discriminator but recent practice showed better results using LeakyReLu for both networks)*
4. Use Batch Normalization *(discriminator)*
5. Use Gaussian Weight Initialization
6. Use Adam SGD optimization
7. Scale the pixel values to [-1,1]

In addition to the heuristics from Radford, I applied some tips from Soumith Chintala, Facebook AI, by using normal distribution for the 100 random numbers that is provided as input for the generator, using separate batches of real and fake images and label smoothing.

The following explanations are paraphrased from *Unsupervised Representation Learning with Deep Convolutional Adversarial Networks, 2015.*

In a convolutional neural network (CNN) it is common to use deterministic *(no randomness)* spatial pooling functions such as max-pooling but not for GAN. Instead use strided convolutions, this allows the network to learn its own spatial downsampling (discriminator) and spatial upsampling (generator).

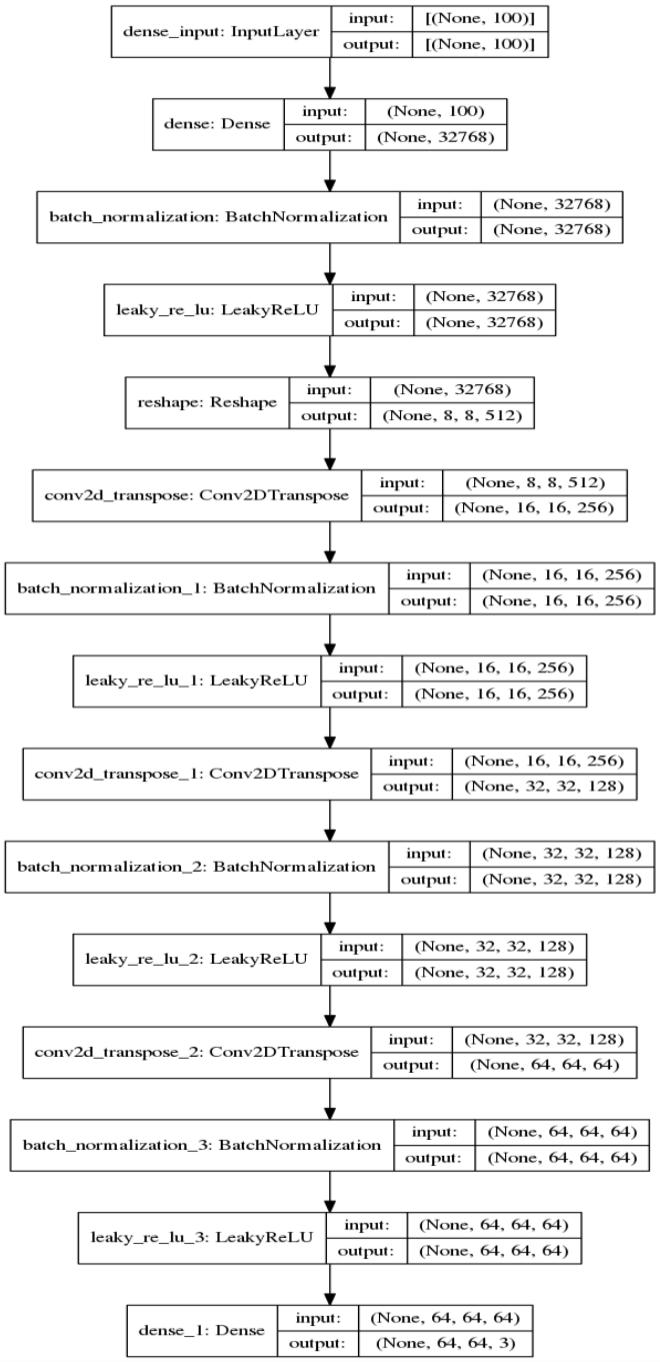
Applying batch normalization is normal in a CNN but for DCGAN it is recommended to apply to all layers except input layer and before output layer for the generator and discriminator. The gaussian weight initialization was recommended with mean 0 and standard deviation 0.2. Similarly, the other tips were drawn from experimenting with the model and manually examining the results for performance. Label smoothing is the idea of not using 1/0 but instead use intervals like [0.75,1] for real images and [0, 0.3] for fake images.

4. OUR PROJECT

This is a Kaggle competition and a link will be posted below for our kernel. We used Tensorflow/Keras along with various other helper libraries. We initially struggled because of the lack of experience with Tensorflow and Keras in addition to having zero knowledge about GANS. We experimented step by step changing little things at a time then waiting for training to finish.

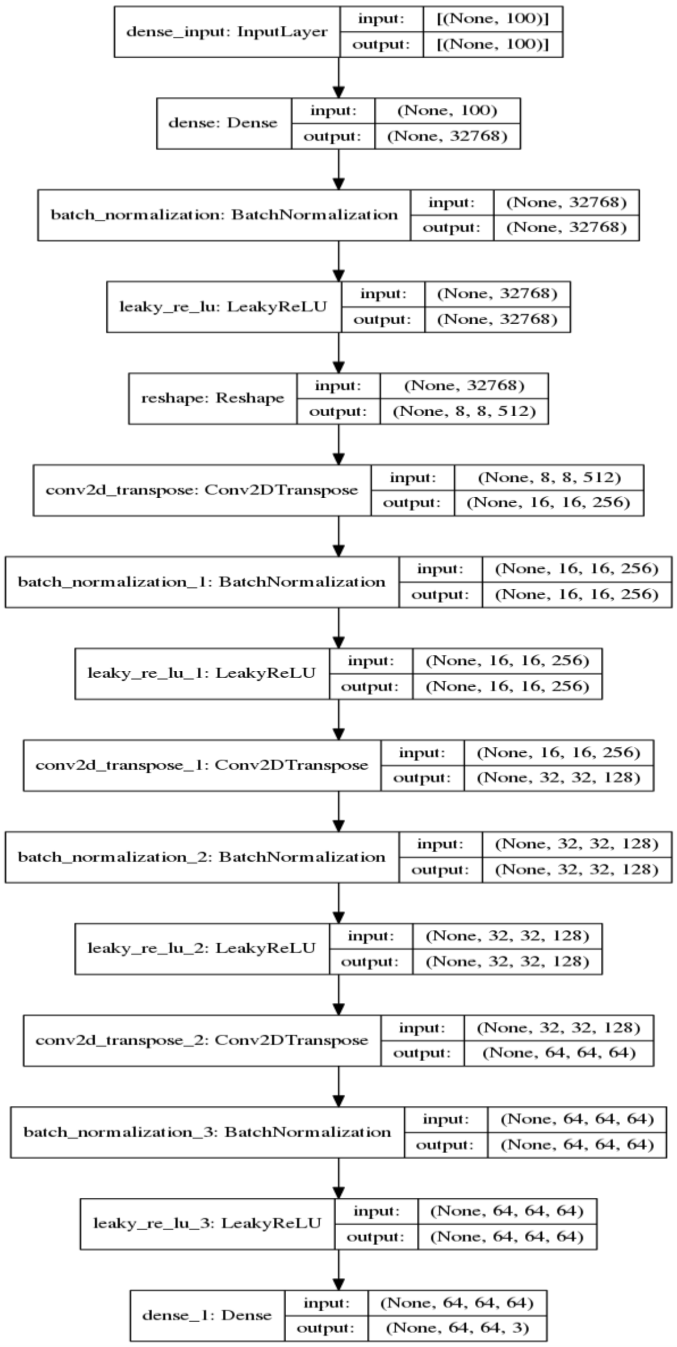
One thing that we changed not mentioned in the previous section are batch sizes. We evaluated our model by using Kaggle's public leaderboard scoring. We also introduced dropout layers in the discriminator of 30%. Then we experimented with using dropout layers in the generator with 50% and removed them from the discriminator. These experiments are random, and we don't know if they have a theoretical or mathematical foundation to them.

The following are images of the generator and discriminator networks we ended with.



dense_input: InputLayer input: [(None, 100)] output: [(None, 100)]
dense: Dense input: (None, 100) output: (None, 32768)
batch_normalization: BatchNormalization input: (None, 32768) output: (None, 32768)
leaky_re_lu: LeakyReLU input: (None, 32768) output: (None, 32768)
reshape: Reshape input: (None, 32768) output: (None, 8, 8, 512)
conv2d_transpose: Conv2DTranspose input: (None, 8, 8, 512) output: (None, 16, 16, 256)
batch_normalization_1: BatchNormalization input: (None, 16, 16, 256) output: (None, 16, 16, 256)
leaky_re_lu_1: LeakyReLU input: (None, 16, 16, 256) output: (None, 16, 16, 256)
conv2d_transpose_1: Conv2DTranspose input: (None, 16, 16, 256) output: (None, 32, 32, 128)
batch_normalization_2: BatchNormalization input: (None, 32, 32, 128) output: (None, 32, 32, 128)
leaky_re_lu_2: LeakyReLU input: (None, 32, 32, 128) output: (None, 32, 32, 128)
conv2d_transpose_2: Conv2DTranspose input: (None, 32, 32, 128) output: (None, 64, 64, 64)
batch_normalization_3: BatchNormalization input: (None, 64, 64, 64) output: (None, 64, 64, 64)
leaky_re_lu_3: LeakyReLU input: (None, 64, 64, 64) output: (None, 64, 64, 64)
dense_1: Dense input: (None, 64, 64, 64) output: (None, 64, 64, 3)

Figure 4: Generator



dense_input: InputLayer input: [(None, 100)] output: [(None, 100)]
dense: Dense input: (None, 100) output: (None, 32768)
batch_normalization: BatchNormalization input: (None, 32768) output: (None, 32768)
leaky_re_lu: LeakyReLU input: (None, 32768) output: (None, 32768)
reshape: Reshape input: (None, 32768) output: (None, 8, 8, 512)
conv2d_transpose: Conv2DTranspose input: (None, 8, 8, 512) output: (None, 16, 16, 256)
batch_normalization_1: BatchNormalization input: (None, 16, 16, 256) output: (None, 16, 16, 256)
leaky_re_lu_1: LeakyReLU input: (None, 16, 16, 256) output: (None, 16, 16, 256)
conv2d_transpose_1: Conv2DTranspose input: (None, 16, 16, 256) output: (None, 32, 32, 128)
batch_normalization_2: BatchNormalization input: (None, 32, 32, 128) output: (None, 32, 32, 128)
leaky_re_lu_2: LeakyReLU input: (None, 32, 32, 128) output: (None, 32, 32, 128)
conv2d_transpose_2: Conv2DTranspose input: (None, 32, 32, 128) output: (None, 64, 64, 64)
batch_normalization_3: BatchNormalization input: (None, 64, 64, 64) output: (None, 64, 64, 64)
leaky_re_lu_3: LeakyReLU input: (None, 64, 64, 64) output: (None, 64, 64, 64)
dense_1: Dense input: (None, 64, 64, 64) output: (None, 64, 64, 3)

Figure 3: Discriminator

5. Closing thoughts

At the start we mentioned we asked would we see more of the top 5 breeds as the output images from the generator than the other breeds. We can't actually conclude this because we couldn't train our DCGAN to get pictures that actually looked like dogs. We saw formations and colors which was really exciting but not enough to trick someone. The restrictions we faced was 1 GPU with a 9-hour time limit per commit. This includes preprocessing, the images were cropped using the bounding box values given in the XML files for each of the 20579 images, training (our final model with 250 epoch and batch size 32 is about 8 to 8.5 hours) and saving 10000 predictions to a folder. I think with more compute power the results could've been better.

One more thing to note is the batch size trade-off. The results have been better with lower batch size, lowest we went was with 32 and highest is 256. The trade-off is smaller batch size means longer training time. With batch size 256 each epoch runs on average 45 seconds. Batch size 32 runs 110 seconds on average. Measuring performance of GANS is hard because there isn't objective function we are trying to optimize. However, we can look at the loss graphs of both the generator and discriminator.

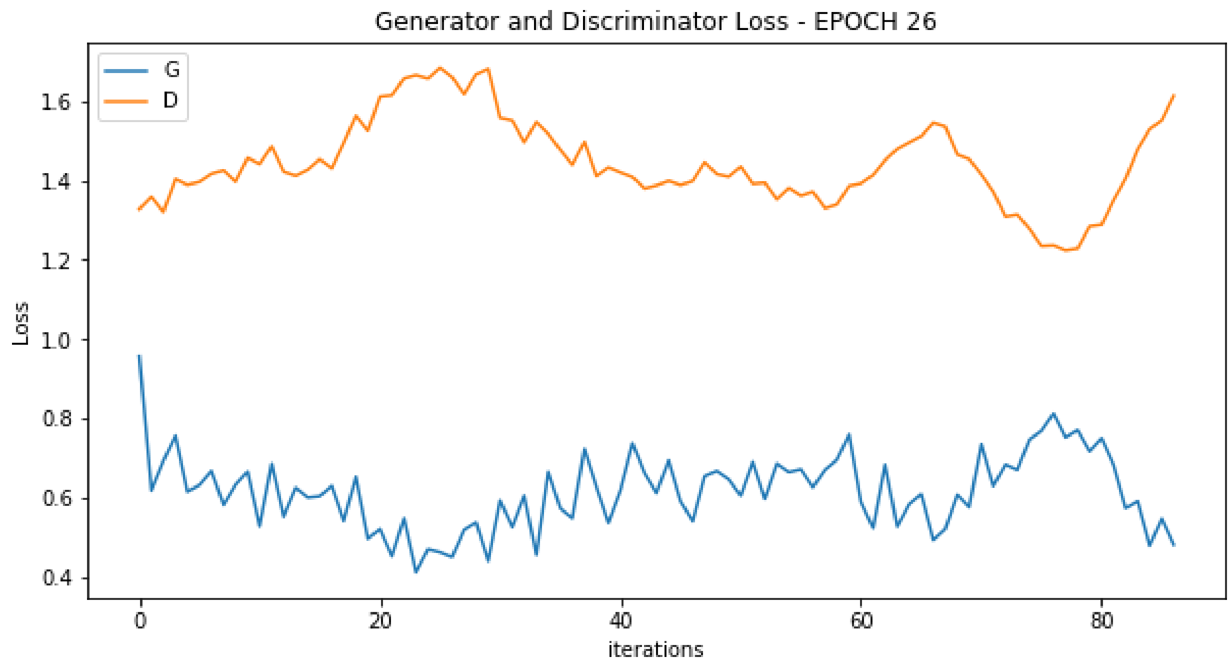


Figure 5: Batch size - 256

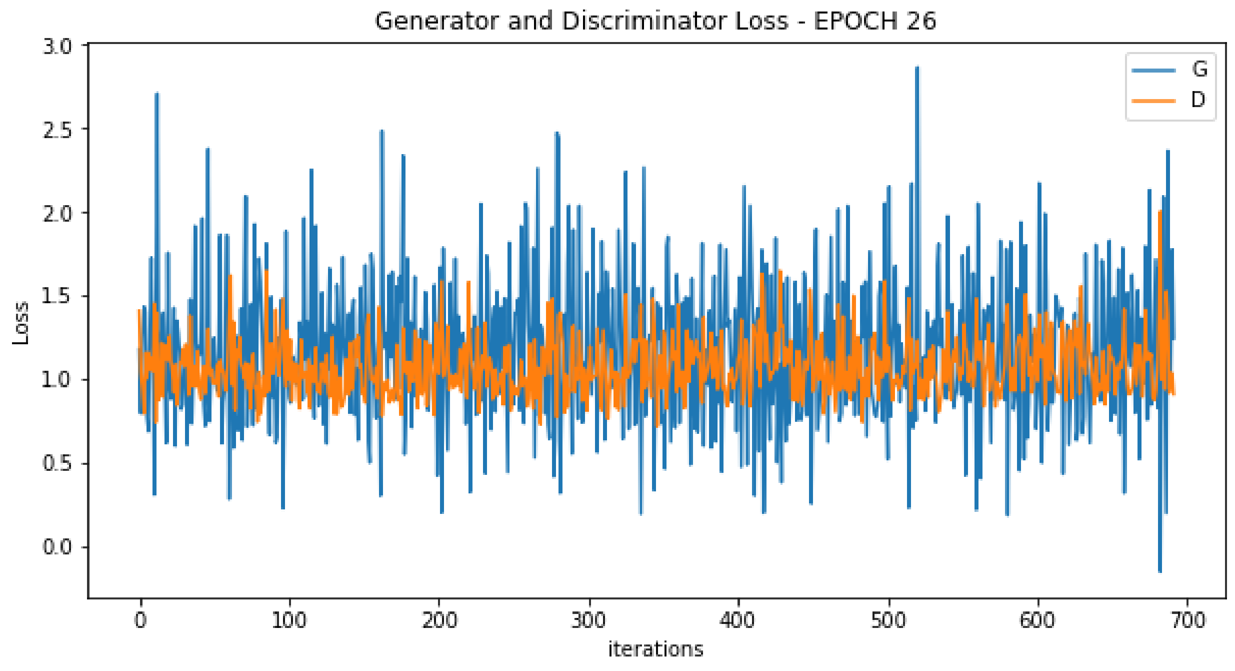


Figure 6: Batch size – 32

This doesn't confirm that our model is good, but the intuition is that having the individual network losses to converge is better. The second one looks like a better convergence than the first. Also, if we manually looked at the images, batch size of 32 had images at epoch 26 looking like what batch size 256 ended with after 200 epochs. We settled with 32 even though it means less epochs to train.

Things we would do differently is basically try a different generative method like memorizer GANS. Although those are probably against the competition rules for Kaggle, we would like to experiment with them personally to see if we can actually get dog pictures good enough to trick people.

Here is the link to our Kaggle kernel: https://www.kaggle.com/cmalla94/dcgan-generating-dog-images-with-tensorflow

6. REFERENCES

Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (Alec Radford, Luke Metz, Soumith Chintala et., 2015): https://arxiv.org/abs/1511.06434

[1] https://distill.pub/2016/deconv-checkerboard/

https://medium.com/activating-robotic-minds/up-sampling-with-transposed-convolution-9ae4f2df52d0

https://machinelearningmastery.com/how-to-code-generative-adversarial-network-hacks/