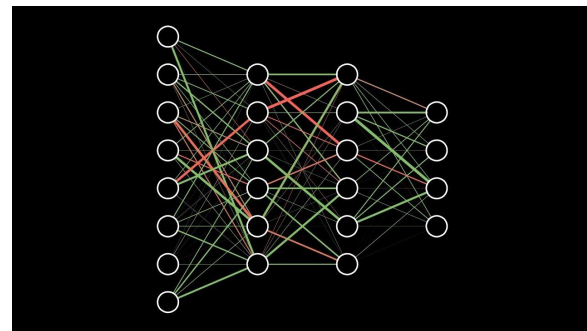


# Deep Convolutional GAN

Pooneet Thaper  
Professor Grimmelmann  
CSC 59929

# Artificial Neural Networks

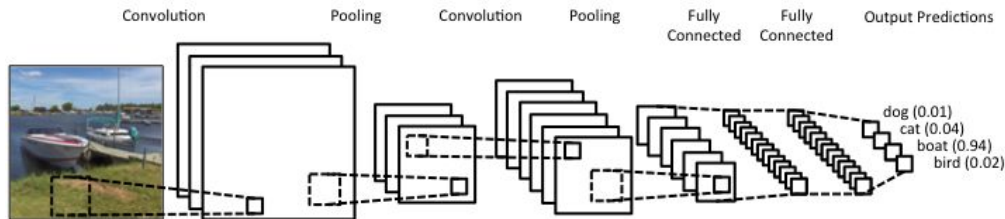
- An Artificial Neural Network (ANN) is a model based vaguely on the arrangement of neurons in the human brain
- Consists of layers of neurons that receive weighted information from previous layers, process this input, and output their activations to following layers
- Utilized by doing forward propagating input to get output
- Optimized by using gradient descent and back propagating errors to update weights



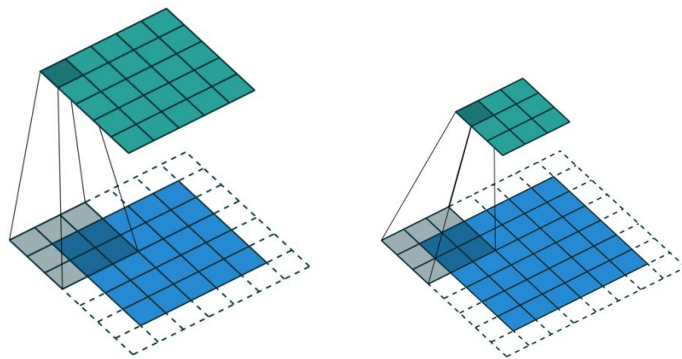
Source: Youtube/3Blue1Brown

# Convolution

- Convolution (in ANNs) is an application of the same weights and biases across a (usually) 2 dimensional tensor (ex: image)
- Takes a dot product of the values in sliding window area and, in doing so, extracts spatial features from the data for further processing
- Currently the state of the art in many computervision areas



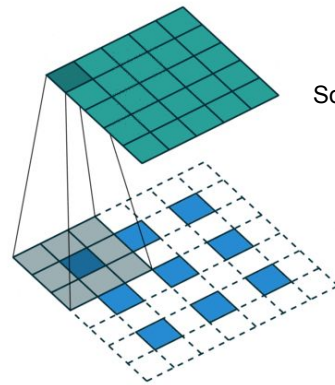
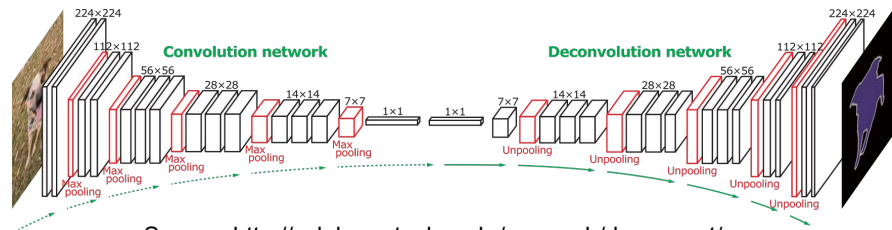
Source: <https://www.clarifai.com/technology>



Source: [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)

# Transpose Convolution

- Transpose Convolution is a convolution operation that is often used in conjunction with convolution for when you want to “undo” the spatial changes from the convolution (ex. an autoencoder)
- Are often used to perform upsampling of input data to higher resolution images, for example



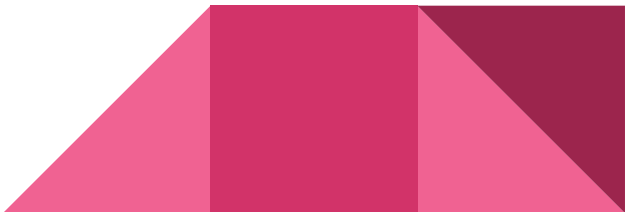
Source: [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)

# TensorFlow



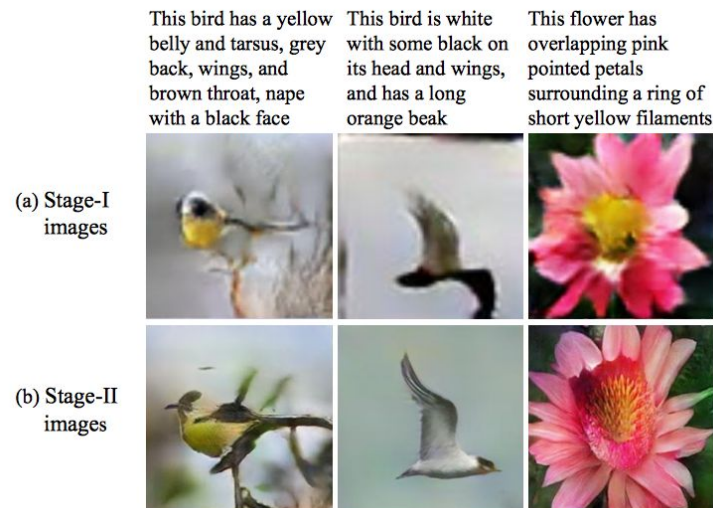
- TensorFlow is a machine learning library which was open source by Google in 2015 and has since been very popular
- It utilizes symbolic programming (define now, run later) to create computation graphs that can be use for, among other things, deep learning
- Very useful since it handles many of the intricacies of deep learning including automatic gradient calculation, built in functions that make defining deep learning models easier, and easy optimization
- Allows distributed computing over heterogeneous hardware (including GPU, CPU, and TPU)
- Further abstraction provided by Keras

# Generative Adversarial Networks

- A Generative Adversarial Network (GAN) is an unsupervised learning technique that has two neural networks train by competing against each other
  - Discriminator aims to do a binary classification of the input images as being from either the original data distribution (“real”) or being generated by the generator (“fake”)
  - Generator aims to produce data that mimics the distribution of the original data from an input vector of (Gaussian) noise
  - Both of these models can be trained by gradient descent based on their performance and their respective loss functions
- 

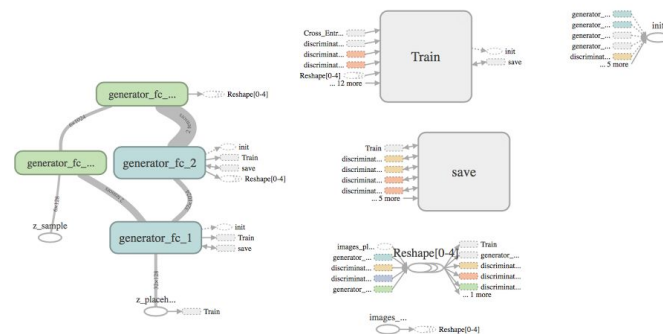
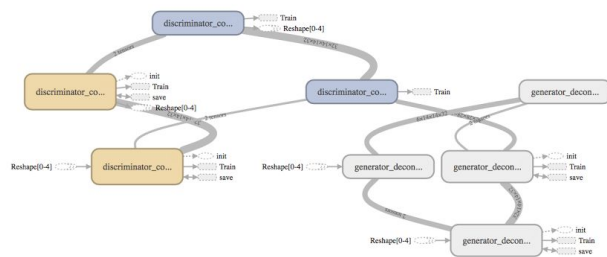
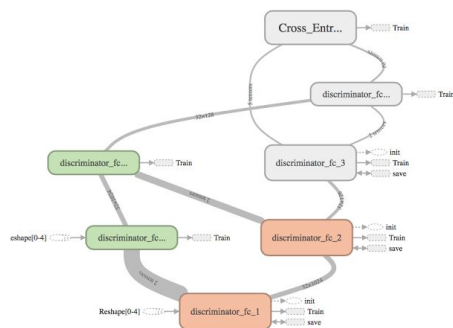
# Why you should care about GANs

- This technique is able to extract an understanding of the original data in an unsupervised manner
- Allow computers to be “creative” and generate data, including images, paintings, and predictions
- New way of training machine learning models adversarially in a zero sum game



Source: Arxiv 1616.03242v1

# What does a GAN look like in TensorFlow

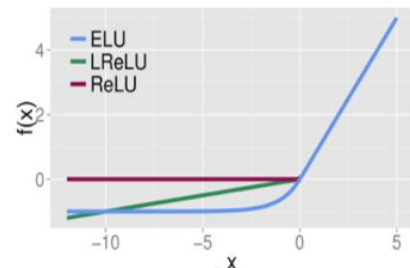




# Discriminator: CNN

- Discriminator used here is a convolutional neural network which takes in `batch_size` number of images and classifies them as either real or fake
- Consists of two convolutional layers followed by three fully connected layers with all but the last layer using an Elu activation
- The loss function used is the cross entropy between the predicted value and either 0 for original images and 1 for generated images

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)],$$



# Generator: Transpose CNN

- Generator use here is a transpose convolutional neural network with two fully connected layers followed by two transpose convolutional layers
- Takes in batch\_size input vectors of 128 normally distributed random numbers and produces batch\_size 28x28x1 output “images”
- The loss of this network is based on the performance of the discriminator on the output of this network
  - Cross entropy is used similarly to that of the discriminator but with inverted labels
    - i.e. Labels for fake images are 1



# What does this look like in code?

```
def discriminator(images, reuse_variables=None):
    if not reuse_variables:
        print('Initializing Discriminator')
    else:
        print('Reusing Discriminator')
    with tf.variable_scope(tf.get_variable_scope(), reuse=reuse_variables):
        d_conv_1 = conv_layer(input=images, kernel_size=5, channels_in=1, channels_out= 32, stride=2,
                               name='discriminator_conv_1', reuse_variables=reuse_variables)
        d_conv_2 = conv_layer(input=d_conv_1, kernel_size=5, channels_in=32, channels_out= 64, stride=2,
                               name='discriminator_conv_2', reuse_variables=reuse_variables)
        flattened = tf.reshape(d_conv_2, [-1, 7*7*64])
        d_fc_1 = fc_layer(input=flattened, size_in=7*7*64, size_out=1024, activation='elu', name='discriminator_fc_1',
                           reuse_variables=reuse_variables)
        d_fc_2 = fc_layer(input=d_fc_1, size_in=1024, size_out=128, activation='elu', name='discriminator_fc_2',
                           reuse_variables=reuse_variables)
        d_fc_3 = fc_layer(input=d_fc_2, size_in=128, size_out=1, activation='linear', name='discriminator_fc_3',
                           reuse_variables=reuse_variables)
    return d_fc_3
```

<https://github.com/PooneetThaper/GAN-Implementation>

```
def generator(z, reuse_variables=None):
    if not reuse_variables:
        print('Initializing Generator')
    else:
        print('Reusing Generator')
    with tf.variable_scope(tf.get_variable_scope()):
        g_fc_1 = fc_layer(input=z, size_in=128, size_out=1024, activation='elu', name='generator_fc_1',
                           reuse_variables=reuse_variables)
        g_fc_2 = fc_layer(input=g_fc_1, size_in=1024, size_out=7*7*64, activation='elu', name='generator_fc_2',
                           reuse_variables=reuse_variables)
        reshaped = tf.reshape(g_fc_2, [-1, 7, 7, 64])
        g_deconv_1 = deconv_layer(input=reshaped, kernel_size=5, channels_in=64, channels_out= 32, stride=2,
                                   name='generator_deconv_1', activation='elu', reuse_variables=reuse_variables)
        g_deconv_2 = deconv_layer(input=g_deconv_1, kernel_size=5, channels_in=32, channels_out= 1, stride=2,
                                   name='generator_deconv_2', activation='sigmoid', reuse_variables=reuse_variables)
    return g_deconv_2
```

# Code (continued)

```
# Defining the graphs, loss, optimizers, and summaries
tf.reset_default_graph()
```

```
# Creating placeholders for input to graph
z_placeholder = tf.placeholder(tf.float32, [batch_size, 128], name = 'z_placeholder')
images_placeholder = tf.placeholder(tf.float32, [batch_size, 784], name = 'images_placeholder')
images_reshaped = tf.reshape(images_placeholder, [-1, 28, 28, 1])
tf.summary.image('Input_images', images_reshaped, 6)
```

```
# Initializing Generator and Discriminators
Generator = generator(z_placeholder)
Discriminator_real = discriminator(images_reshaped)
Discriminator_fake = discriminator(Generator, reuse_variables=True)
```

```
# Defining loss functions for networks
with tf.name_scope('Cross_Entropy'):
    D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=Discriminator_real,
                                                                           labels=tf.zeros_like(Discriminator_real)))
    D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=Discriminator_fake,
                                                                           labels=tf.ones_like(Discriminator_fake)))
    G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=Discriminator_fake,
                                                                           labels=tf.zeros_like(Discriminator_fake)))

tf.summary.scalar('D_loss_real', D_loss_real)
tf.summary.scalar('D_loss_fake', D_loss_fake)
tf.summary.scalar('G_loss', G_loss)
```

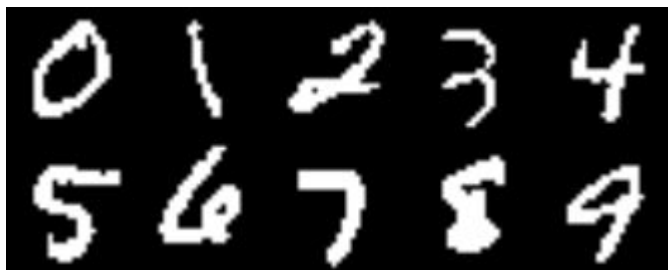
```
# Getting the variables for each network to update only certain weights during each optimization
tvars = tf.trainable_variables()
D_vars = [var for var in tvars if 'discriminator_' in var.name]
G_vars = [var for var in tvars if 'generator_' in var.name]
```

```
# Defining the optimizer (Adam adaptive learning rate and momentum optimizer)
with tf.name_scope('Train'):
    D_train_real = tf.train.AdamOptimizer(learning_rate).minimize(D_loss_real, var_list=D_vars)
    D_train_fake = tf.train.AdamOptimizer(learning_rate).minimize(D_loss_fake, var_list=D_vars)
    G_train = tf.train.AdamOptimizer(learning_rate).minimize(G_loss, var_list=G_vars)
```

```
# Defining graph for generating a sample image
z_sample = tf.placeholder(dtype = tf.float32, shape = [6, 128], name='z_sample')
sample_images = generator(z_sample, True)
tf.summary.image('Generated_images', sample_images, 6)
```

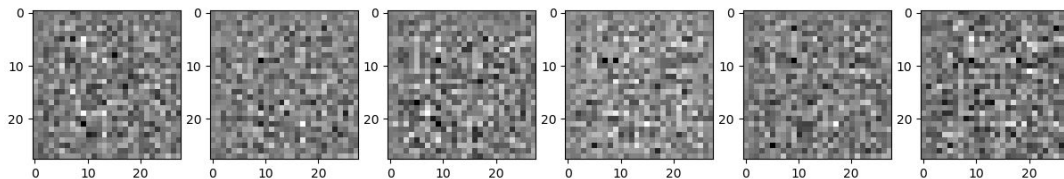
# Dataset: MNIST Handwritten Digits

- As a preliminary trial, I chose to utilize the MNIST Handwritten digit dataset
- This dataset consists of 60,000 training samples and 10,000 tests samples that each consist of a 28x28 grid of grayscale values (0 to 1)
- This is often used in computervision as a sample dataset



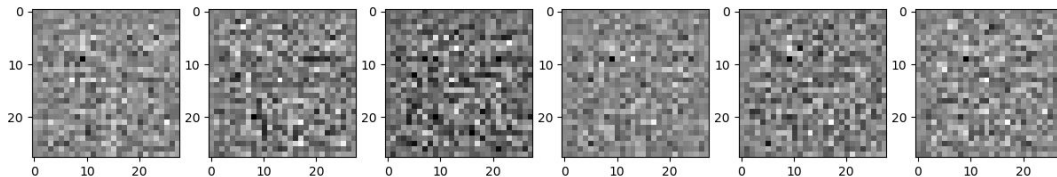
# Initial Results

- The TensorFlow graph was executed on a GTX 1070 for GPU acceleration and images were shown using Matplotlib every 10,000 batches
- There doesn't seem to be much happening here in terms of handwritten digits that are clearly visible



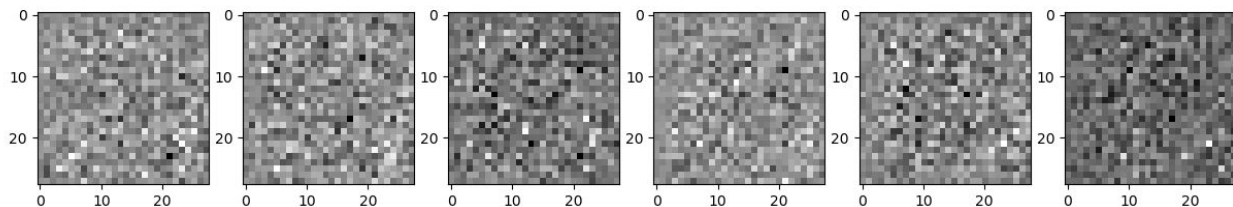
Output at 200,000 batches (~1.5 hours)

Output at 980,000 batches (~8.5 hours)



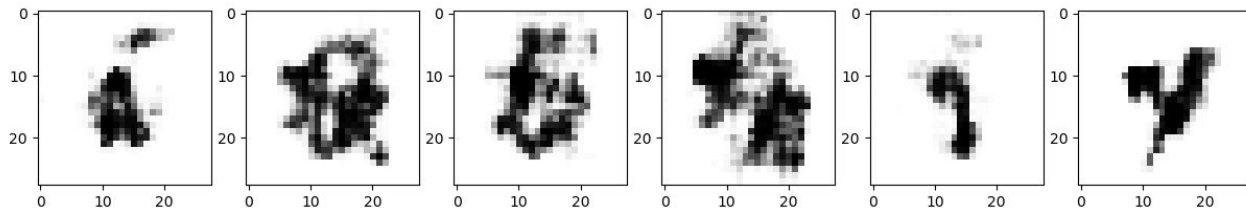
# Further Exploration

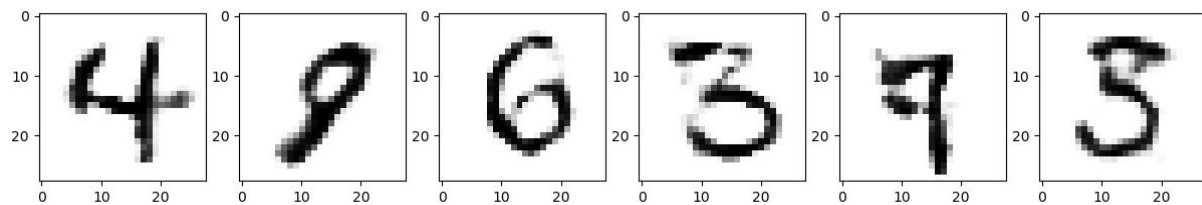
- After some debugging, the learning rate was reduced from 0.03 to 0.0001 and dropout was added to each layer in each network with a 0.75 probability of keeping the result of each neuron (except the last generator layer)



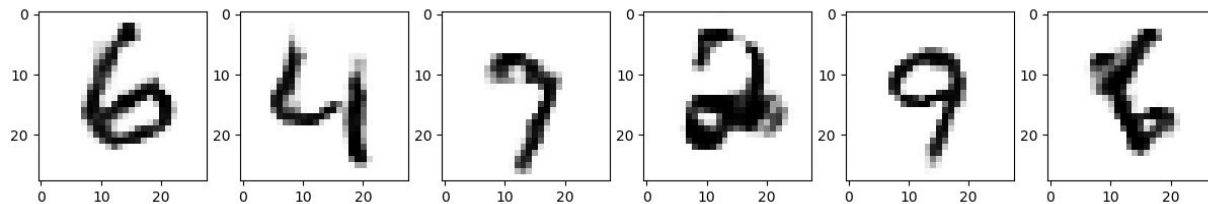
Output before training

Output at 200,000 batches

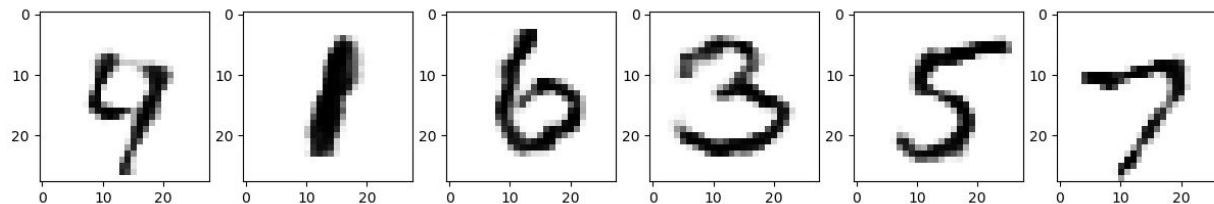




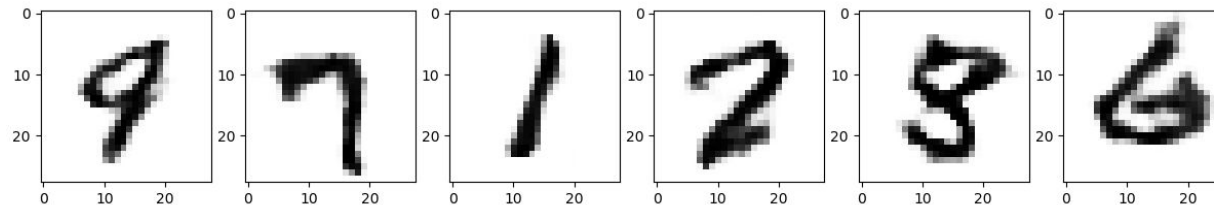
Output at 560,000 batches



Output at 790,000 batches

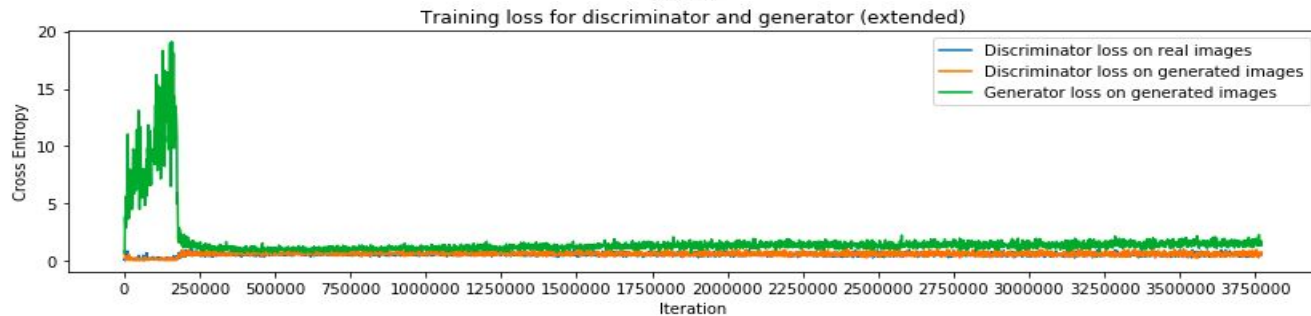
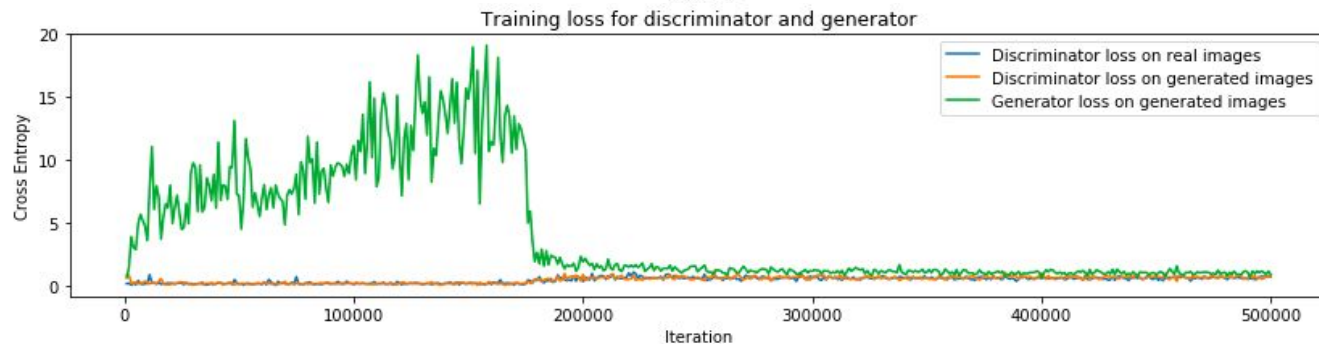
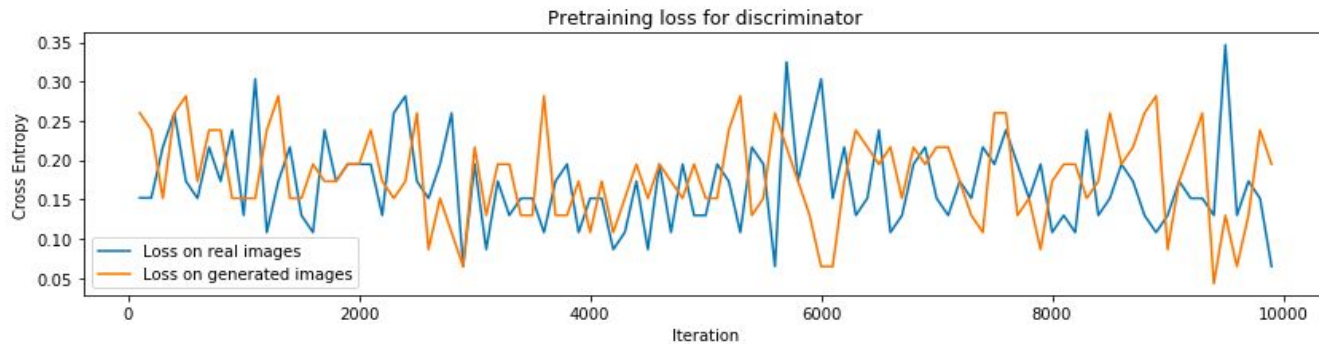


Output at 940,000 batches



Output at 1,120,000 batches





# Discussion

- Generative Adversarial Networks are notoriously difficult to train and can fail to train for a wide variety of reasons
  - Not enough time given/lack of computational power
  - One network becomes overwhelmingly good and overtakes the other
  - The generator figures out a weakness in the discriminator and exploits it by producing very similar results for any random input
  - Unstable training (not continuously improving but rather, given the adversarial aspect, passing the “advantage” back and forth)
  - Bad hyperparameter choice :(

