Homework 6

Xuzheng Lu

G34363475

GitHub Repository: https://github.com/LeanderLXZ/gan-aec

1 Variational Autoencoder (Keras)

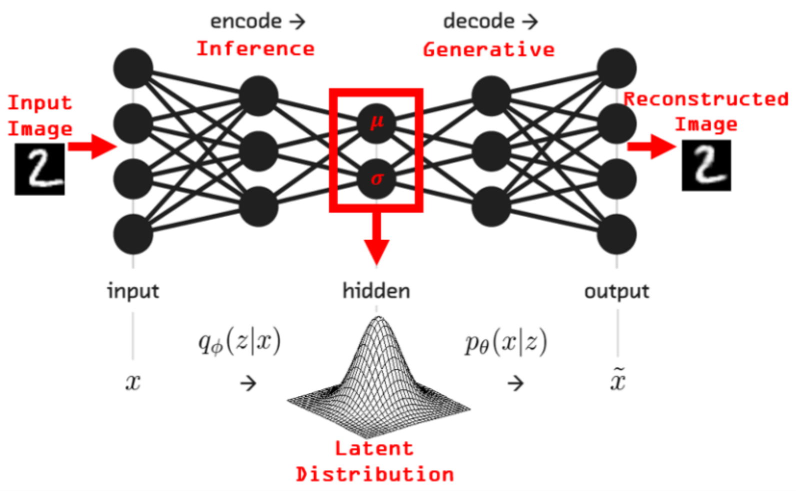
Standard autoencoders learn to generate compact representations and reconstruct their inputs well, but asides from a few applications like denoising autoencoders, they are fairly limited. The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, may not be continuous, or allow easy interpolation.

If the space has discontinuities and you sample/generate a variation from there, the decoder will simply generate an unrealistic output, because the decoder has no idea how to deal with that region of the latent space. During training, it never saw encoded vectors coming from that region of latent space.

1.1 Network Architecture

Variational Autoencoders (VAEs) have one fundamentally unique property that separates them from vanilla autoencoders, and it is this property that makes them so useful for generative modeling: their latent spaces are, by design, continuous, allowing easy random sampling and interpolation.

It achieves this by doing something that seems rather surprising at first: making its encoder not output an encoding vector of size *n*, rather, outputting two vectors of size n: a vector of means**,** **μ**, and another vector of standard deviations, **σ**.

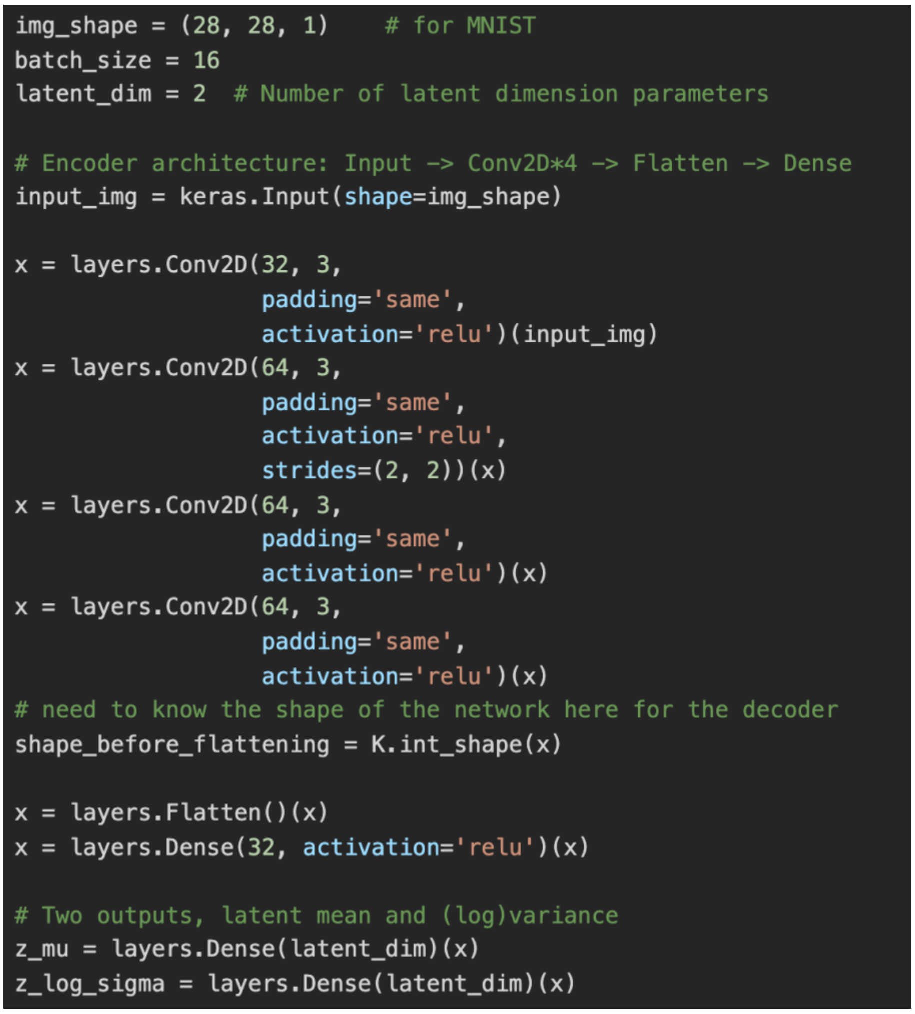


VAEs learn the parameters of the probability distribution that the data came from. These types of autoencoders have much in common with latent factor analysis. The encoder and decoder learn models that are in terms of underlying, unobserved latent variables. It's essentially an inference model and a generative model daisy-chained together.

VAEs learn about the distribution the inputs came from, we can sample from that distribution to generate novel data. VAEs can also be used to cluster data in useful ways.

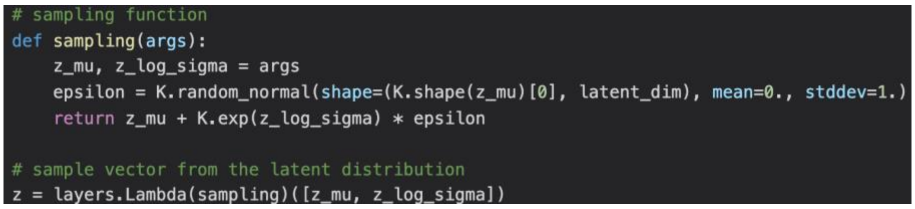
1.2 Code Implementation

Encoder:



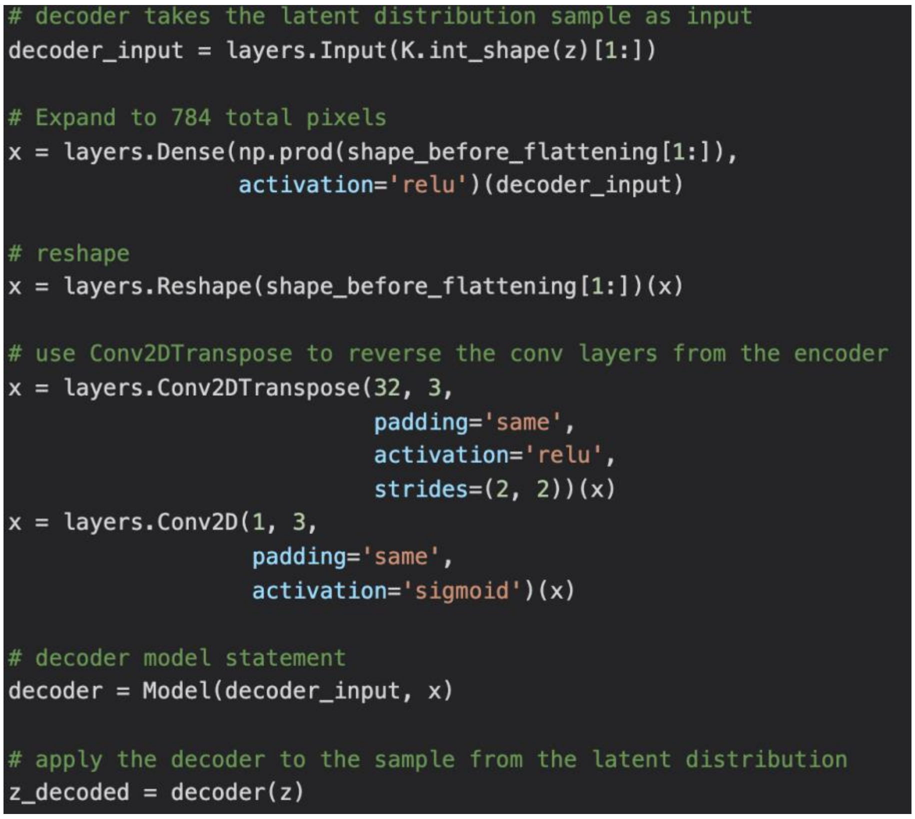
```python
img_shape = (28, 28, 1)    # for MNIST
batch_size = 16
latent_dim = 2  # Number of latent dimension parameters

# Encoder architecture: Input -> Conv2D*4 -> Flatten -> Dense
input_img = keras.Input(shape=img_shape)

x = layers.Conv2D(32, 3,
                  padding='same',
                  activation='relu')(input_img)
x = layers.Conv2D(64, 3,
                  padding='same',
                  activation='relu',
                  strides=(2, 2))(x)
x = layers.Conv2D(64, 3,
                  padding='same',
                  activation='relu')(x)
x = layers.Conv2D(64, 3,
                  padding='same',
                  activation='relu')(x)
# need to know the shape of the network here for the decoder
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

# Two outputs, latent mean and (log)variance
z_mu = layers.Dense(latent_dim)(x)
z_log_sigma = layers.Dense(latent_dim)(x)
```

Sampling layer:



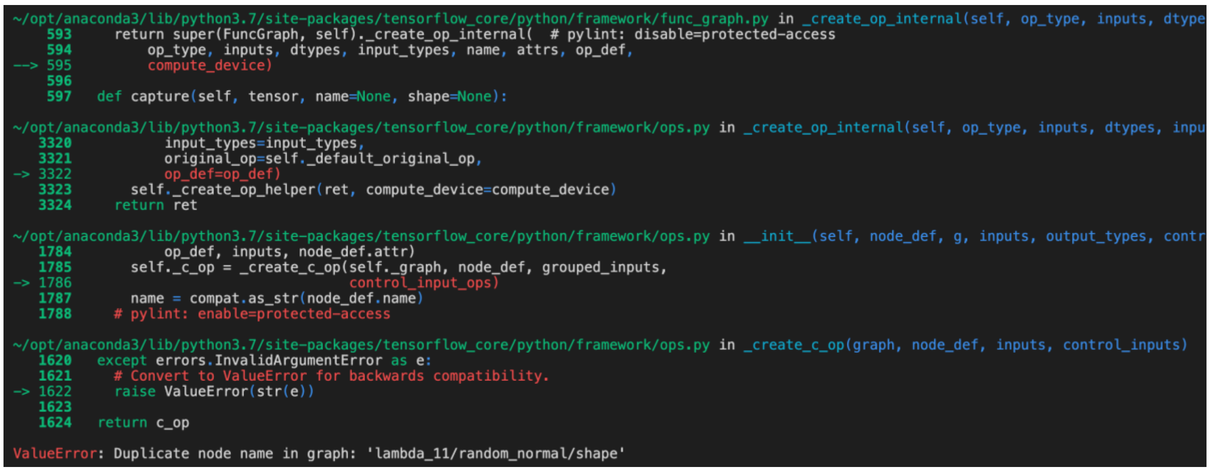
```python
# sampling function
def sampling(args):
    z_mu, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mu)[0], latent_dim), mean=0., stddev=1.)
    return z_mu + K.exp(z_log_sigma) * epsilon

# sample vector from the latent distribution
z = layers.Lambda(sampling)([z_mu, z_log_sigma])
```

Decoder:



```python
# decoder takes the latent distribution sample as input
decoder_input = layers.Input(K.int_shape(z)[1:])

# Expand to 784 total pixels
x = layers.Dense(np.prod(shape_before_flattening[1:]),
                 activation='relu')(decoder_input)

# reshape
x = layers.Reshape(shape_before_flattening[1:])(x)

# use Conv2DTranspose to reverse the conv layers from the encoder
x = layers.Conv2DTranspose(32, 3,
                           padding='same',
                           activation='relu',
                           strides=(2, 2))(x)
x = layers.Conv2D(1, 3,
                  padding='same',
                  activation='sigmoid')(x)

# decoder model statement
decoder = Model(decoder_input, x)

# apply the decoder to the sample from the latent distribution
z_decoded = decoder(z)
```

I tried to run the code. However, there is an error shown as follows. I tried hours to fix this, including reinstall the anaconda, python, keras, and tensorflow. Yet, I still cannot run the code, and I don't know why.

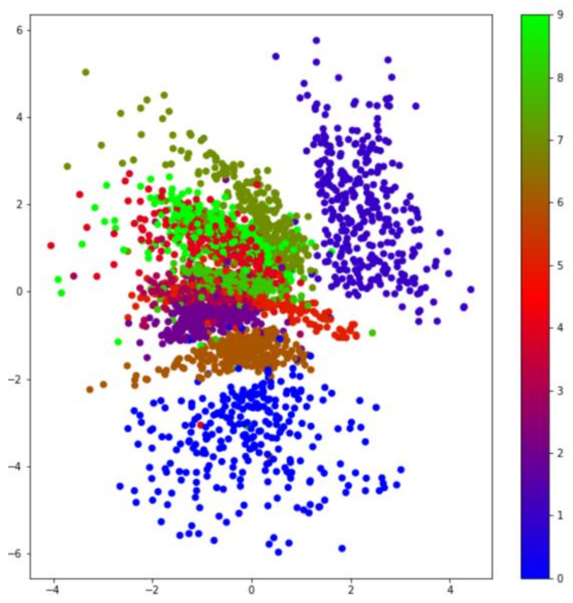


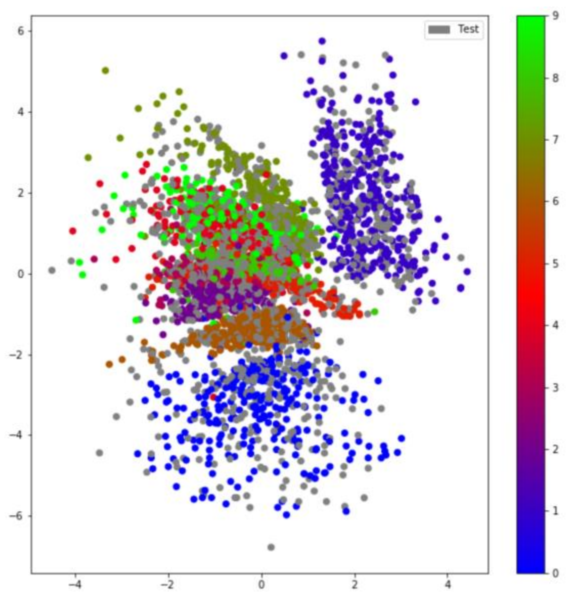
```
~/opt/anaconda3/lib/python3.7/site-packages/tensorflow_core/python/framework/func_graph.py in _create_op_internal(self, op_type, inputs, dtype
    593     return super(FuncGraph, self)._create_op_internal(  # pylint: disable=protected-access
    594         op_type, inputs, dtypes, input_types, name, attrs, op_def,
--> 595         compute_device)
    596 
    597   def capture(self, tensor, name=None, shape=None):

~/opt/anaconda3/lib/python3.7/site-packages/tensorflow_core/python/framework/ops.py in _create_op_internal(self, op_type, inputs, dtypes, inpu
   3320           input_types=input_types,
   3321           original_op=self._default_original_op,
-> 3322           op_def=op_def)
   3323       self._create_op_helper(ret, compute_device=compute_device)
   3324     return ret

~/opt/anaconda3/lib/python3.7/site-packages/tensorflow_core/python/framework/ops.py in __init__(self, node_def, g, inputs, output_types, contr
   1784           op_def, inputs, node_def.attr)
   1785       self._c_op = _create_c_op(self._graph, node_def, grouped_inputs,
-> 1786                                 control_input_ops)
   1787       name = compat.as_str(node_def.name)
   1788     # pylint: enable=protected-access

~/opt/anaconda3/lib/python3.7/site-packages/tensorflow_core/python/framework/ops.py in _create_c_op(graph, node_def, inputs, control_inputs)
   1620   except errors.InvalidArgumentError as e:
   1621     # Convert to ValueError for backwards compatibility.
-> 1622     raise ValueError(str(e))
   1623 
   1624   return c_op

ValueError: Duplicate node name in graph: 'lambda_11/random_normal/shape'
```

1.3 Results

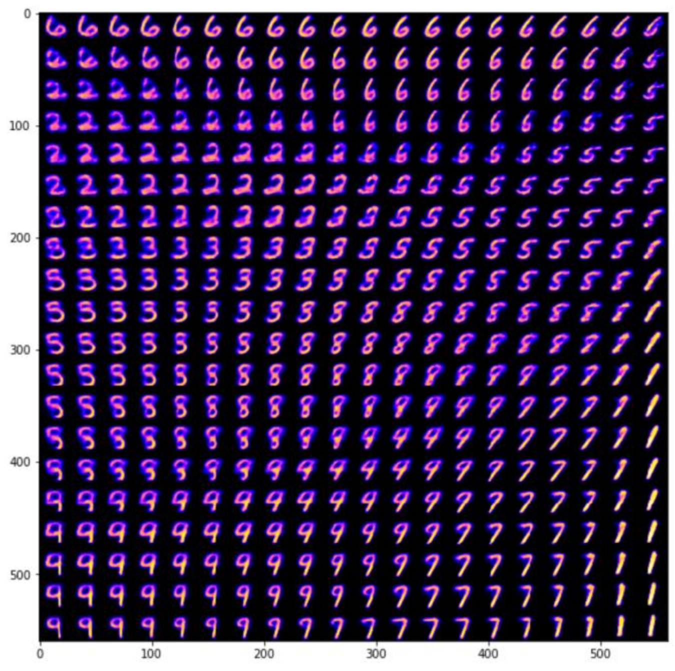
1.3.1 Clustering of digits in the latent space

We can make predictions on the validation set using the encoder network. This has the effect of translating the images from the 784-dimensional input space into the 2-dimensional latent space. When we color-code those translated data points according to their *known* digit class, we can see how the digits cluster together.





1.3.2 Sample digits



2 Convolutional Autoencoder (TensorFlow)

The code of previous project is not working on my PC, so I implemented another Autoencoder in TensorFlow.

An autoencoder network is actually a pair of two connected networks, an encoder and a decoder. An encoder network takes in an input, and converts it into a smaller, dense representation, which the decoder network can use to convert it back to the original input. The goal of the autoencoder is to find a small representation of the input data.

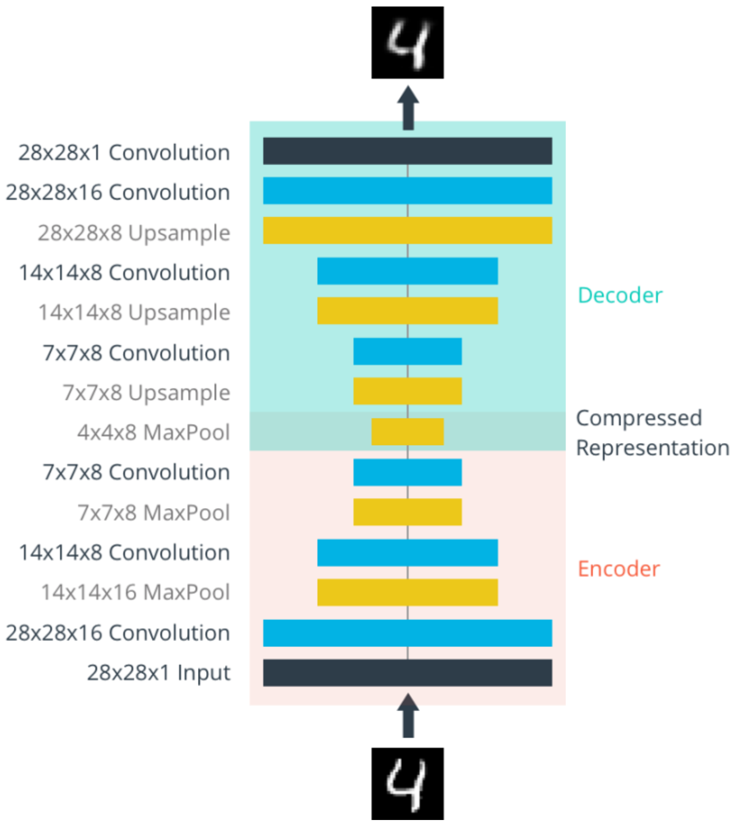
I built a convolutional autoencoder in TensorFlow to compress the MNIST dataset. With autoencoders, I pass input data through an encoder that makes a compressed representation of the input. Then, this representation is passed through a decoder to reconstruct the input data.

2.1 Network Architecture

The **encoder part** of the network is a typical convolutional pyramid. Each convolutional layer will be followed by a max-pooling layer to reduce the dimensions of the layers.

The **decoder part** of the network needs to convert from a narrow representation to a wide reconstructed image. The decoder takes the output of the encoder, generated form a 4x4x8 max-pool layer, as the input. The goal is to output a 28x28x1 image, so it needs to work back up from the narrow decoder input layer.

The structure of the convolutional autoencoder network is as follows.



Here the original images have the size of 28x28 = 784, and the final encoder layer has the size of 4x4x8 = 128. So, the encoded vector is roughly 16% the size of the original image.

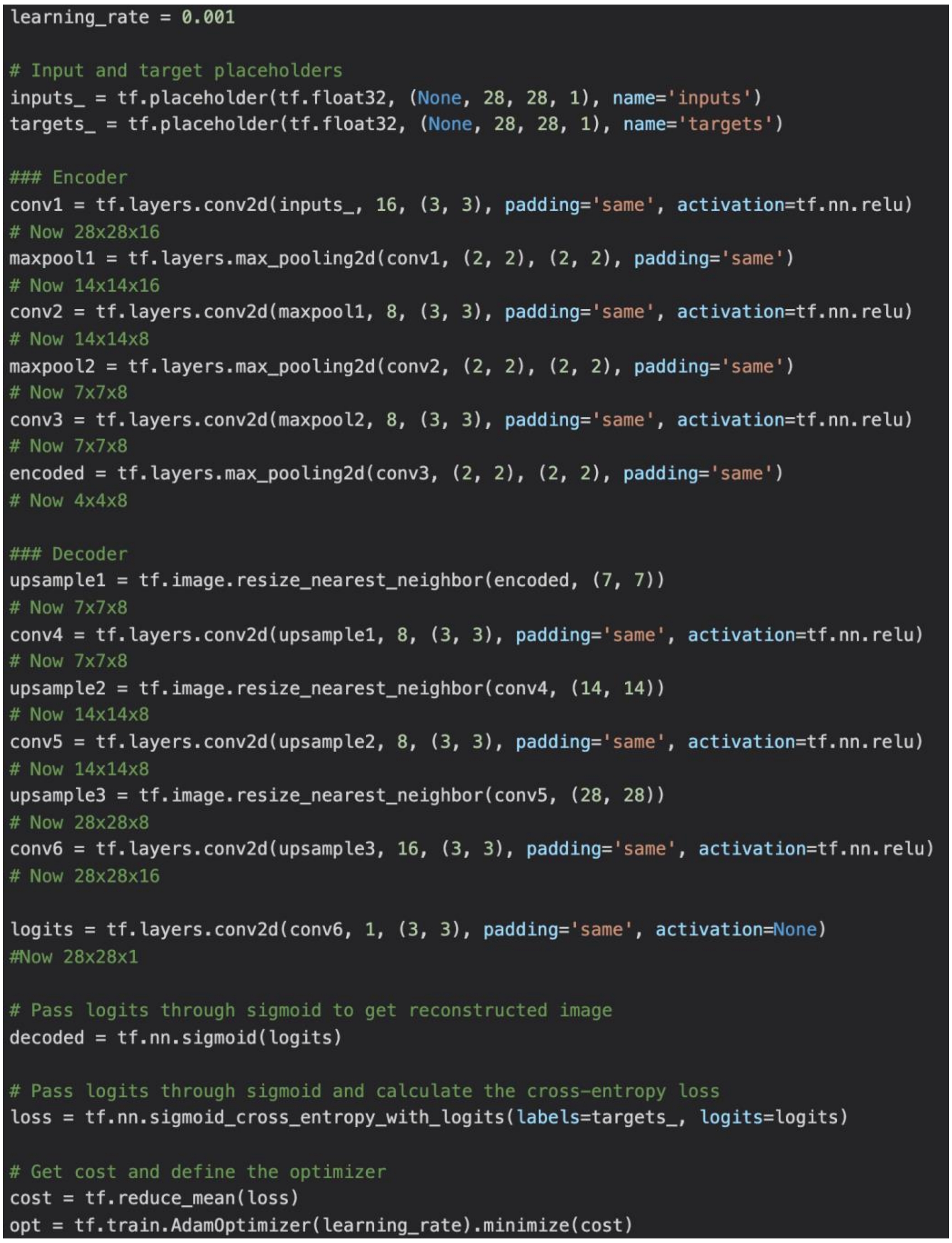
For convolutional layers, I used `tf.layers.conv2d`. I used `tf.layers.max_pooling2d` as the max-pool layers, to the reduce the width and height. A stride of 2 will reduce the size by a factor of 2.

The decoder has several "Upsample" layers, we can use transposed convolution layers to increase the width and height of the layers. They work almost exactly the same as convolutional layers, but in reverse. However, transposed convolution layers can lead to artifacts in the final images, such as checkerboard patterns. This is due to overlap in the kernels which can be avoided by setting the stride and kernel size equal.

In this article (https://distill.pub/2016/deconv-checkerboard/), the authors show that these checkerboard artifacts can be avoided by resizing the layers using nearest neighbor or bilinear interpolation (upsampling) followed by a convolutional layer. Therefore, instead of the transposed convolution layers,   I used `tf.image.resize_images` followed by a convolution,

2.2 Code Implementation

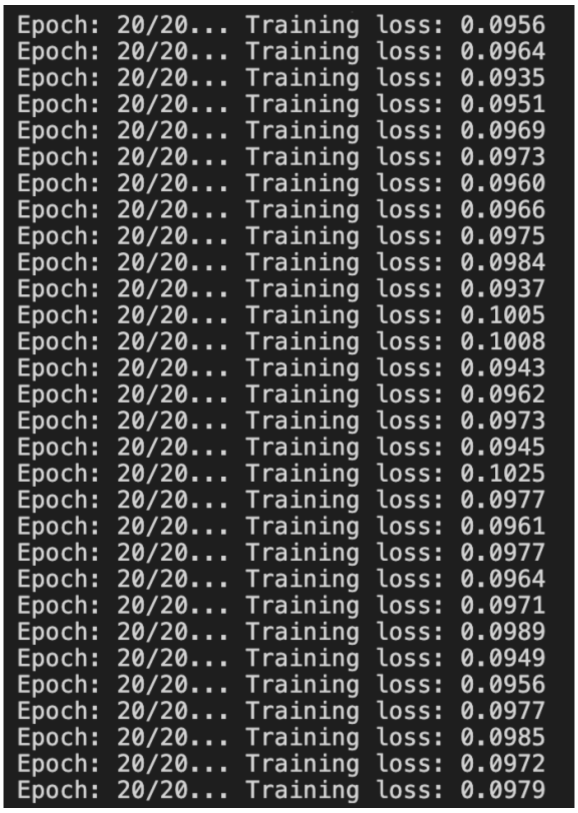
The code is shown below:



```
learning_rate = 0.001

# Input and target placeholders
inputs_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='inputs')
targets_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='targets')

### Encoder
conv1 = tf.layers.conv2d(inputs_, 16, (3, 3), padding='same', activation=tf.nn.relu)
# Now 28x28x16
maxpool1 = tf.layers.max_pooling2d(conv1, (2, 2), (2, 2), padding='same')
# Now 14x14x16
conv2 = tf.layers.conv2d(maxpool1, 8, (3, 3), padding='same', activation=tf.nn.relu)
# Now 14x14x8
maxpool2 = tf.layers.max_pooling2d(conv2, (2, 2), (2, 2), padding='same')
# Now 7x7x8
conv3 = tf.layers.conv2d(maxpool2, 8, (3, 3), padding='same', activation=tf.nn.relu)
# Now 7x7x8
encoded = tf.layers.max_pooling2d(conv3, (2, 2), (2, 2), padding='same')
# Now 4x4x8

### Decoder
upsample1 = tf.image.resize_nearest_neighbor(encoded, (7, 7))
# Now 7x7x8
conv4 = tf.layers.conv2d(upsample1, 8, (3, 3), padding='same', activation=tf.nn.relu)
# Now 7x7x8
upsample2 = tf.image.resize_nearest_neighbor(conv4, (14, 14))
# Now 14x14x8
conv5 = tf.layers.conv2d(upsample2, 8, (3, 3), padding='same', activation=tf.nn.relu)
# Now 14x14x8
upsample3 = tf.image.resize_nearest_neighbor(conv5, (28, 28))
# Now 28x28x8
conv6 = tf.layers.conv2d(upsample3, 16, (3, 3), padding='same', activation=tf.nn.relu)
# Now 28x28x16

logits = tf.layers.conv2d(conv6, 1, (3, 3), padding='same', activation=None)
#Now 28x28x1

# Pass logits through sigmoid to get reconstructed image
decoded = tf.nn.sigmoid(logits)

# Pass logits through sigmoid and calculate the cross-entropy loss
loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=targets_, logits=logits)

# Get cost and define the optimizer
cost = tf.reduce_mean(loss)
opt = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

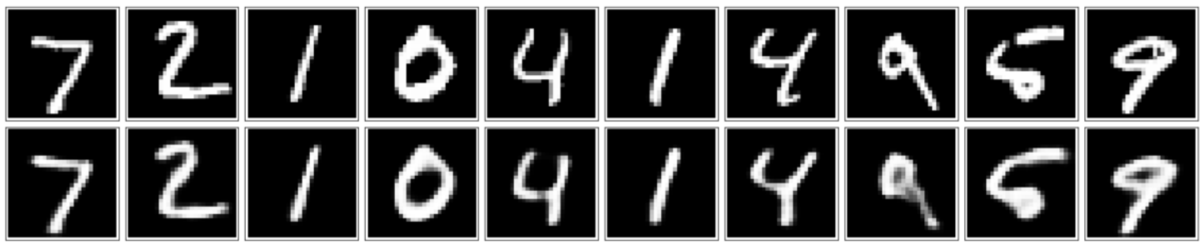
2.3 Training & Evaluation

In traning process, I pass the images into the network as 28x28x1 arrays.



```
Epoch: 20/20... Training loss: 0.0956
Epoch: 20/20... Training loss: 0.0964
Epoch: 20/20... Training loss: 0.0935
Epoch: 20/20... Training loss: 0.0951
Epoch: 20/20... Training loss: 0.0969
Epoch: 20/20... Training loss: 0.0973
Epoch: 20/20... Training loss: 0.0960
Epoch: 20/20... Training loss: 0.0966
Epoch: 20/20... Training loss: 0.0975
Epoch: 20/20... Training loss: 0.0984
Epoch: 20/20... Training loss: 0.0937
Epoch: 20/20... Training loss: 0.1005
Epoch: 20/20... Training loss: 0.1008
Epoch: 20/20... Training loss: 0.0943
Epoch: 20/20... Training loss: 0.0962
Epoch: 20/20... Training loss: 0.0973
Epoch: 20/20... Training loss: 0.0945
Epoch: 20/20... Training loss: 0.1025
Epoch: 20/20... Training loss: 0.0977
Epoch: 20/20... Training loss: 0.0961
Epoch: 20/20... Training loss: 0.0977
Epoch: 20/20... Training loss: 0.0964
Epoch: 20/20... Training loss: 0.0971
Epoch: 20/20... Training loss: 0.0989
Epoch: 20/20... Training loss: 0.0949
Epoch: 20/20... Training loss: 0.0956
Epoch: 20/20... Training loss: 0.0977
Epoch: 20/20... Training loss: 0.0985
Epoch: 20/20... Training loss: 0.0972
Epoch: 20/20... Training loss: 0.0979
```

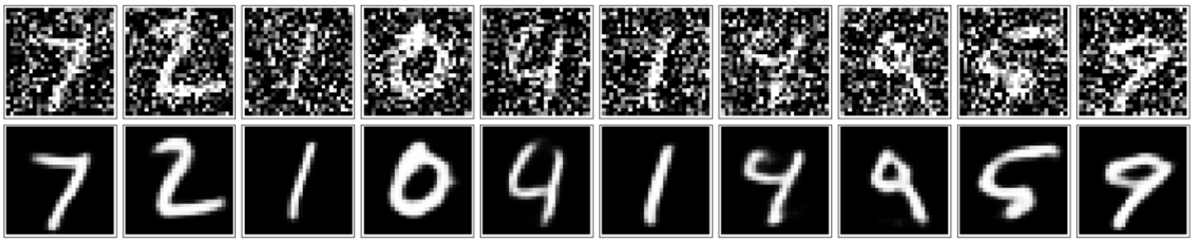
The output of the test set is as follows:



As shown above, the network can rebuild the images efficiently and precisely.

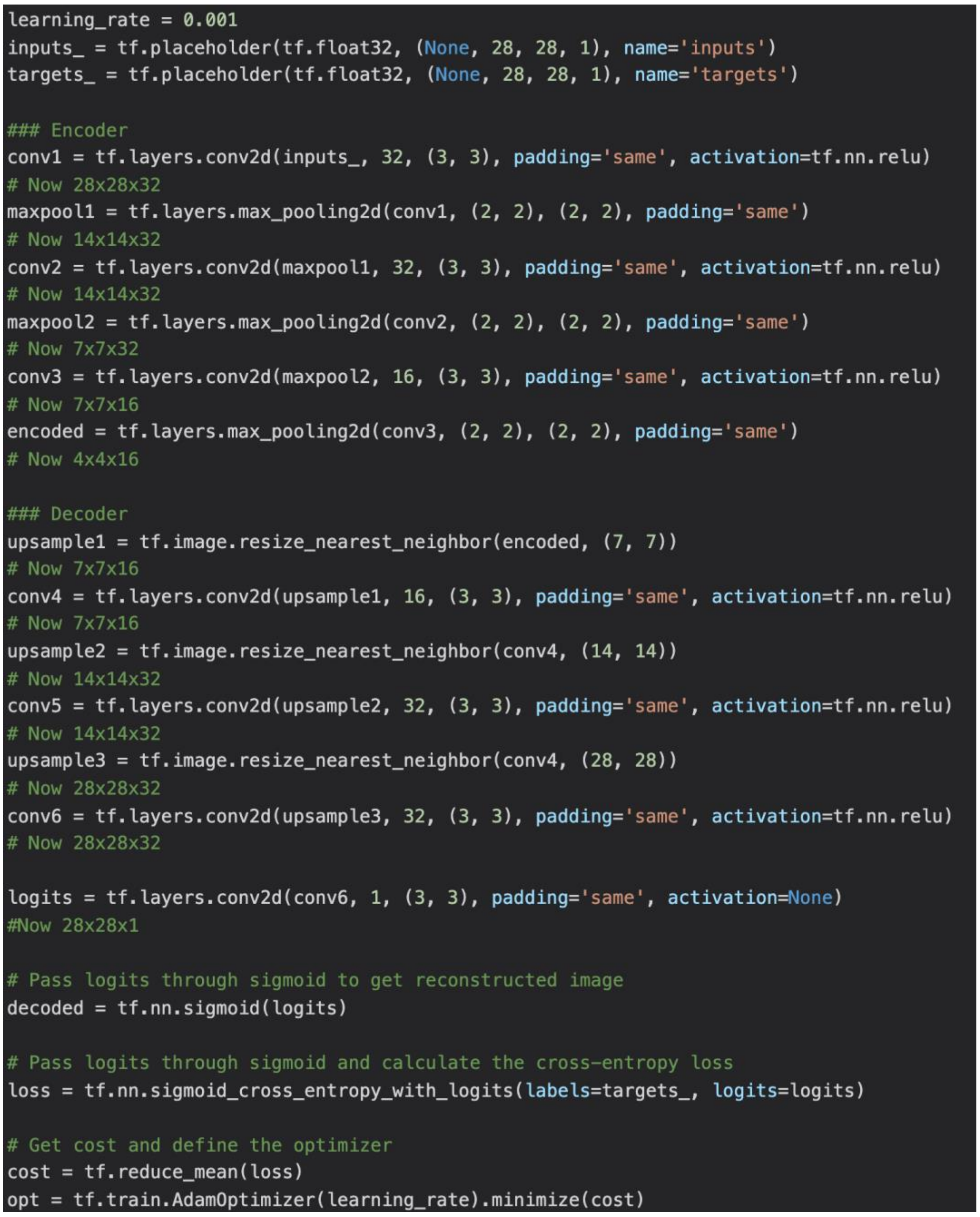
2.4 Denoising

Autoencoders can be used to denoise images quite successfully just by training the network on noisy images. I can create some the noisy images by adding Gaussian noise to the training images, then clipping the values to be between 0 and 1. I used noisy images as input and the original, clean images as targets. Here's an example of the noisy images I generated and the denoised images.



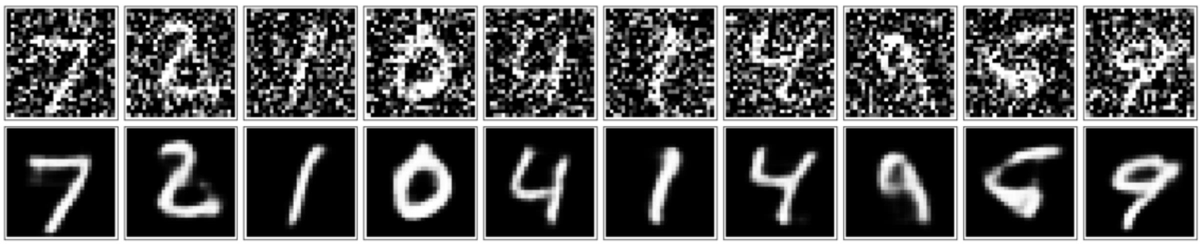
Since this is a harder problem for the network, I used deeper convolutional layers and more feature maps. I used 32-32-16 for the depths of the convolutional layers in the encoder, and the same depths going backward through the decoder.

The code is shown as follows:



```
learning_rate = 0.001
inputs_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='inputs')
targets_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='targets')

### Encoder
conv1 = tf.layers.conv2d(inputs_, 32, (3, 3), padding='same', activation=tf.nn.relu)
# Now 28x28x32
maxpool1 = tf.layers.max_pooling2d(conv1, (2, 2), (2, 2), padding='same')
# Now 14x14x32
conv2 = tf.layers.conv2d(maxpool1, 32, (3, 3), padding='same', activation=tf.nn.relu)
# Now 14x14x32
maxpool2 = tf.layers.max_pooling2d(conv2, (2, 2), (2, 2), padding='same')
# Now 7x7x32
conv3 = tf.layers.conv2d(maxpool2, 16, (3, 3), padding='same', activation=tf.nn.relu)
# Now 7x7x16
encoded = tf.layers.max_pooling2d(conv3, (2, 2), (2, 2), padding='same')
# Now 4x4x16

### Decoder
upsample1 = tf.image.resize_nearest_neighbor(encoded, (7, 7))
# Now 7x7x16
conv4 = tf.layers.conv2d(upsample1, 16, (3, 3), padding='same', activation=tf.nn.relu)
# Now 7x7x16
upsample2 = tf.image.resize_nearest_neighbor(conv4, (14, 14))
# Now 14x14x32
conv5 = tf.layers.conv2d(upsample2, 32, (3, 3), padding='same', activation=tf.nn.relu)
# Now 14x14x32
upsample3 = tf.image.resize_nearest_neighbor(conv4, (28, 28))
# Now 28x28x32
conv6 = tf.layers.conv2d(upsample3, 32, (3, 3), padding='same', activation=tf.nn.relu)
# Now 28x28x32

logits = tf.layers.conv2d(conv6, 1, (3, 3), padding='same', activation=None)
#Now 28x28x1

# Pass logits through sigmoid to get reconstructed image
decoded = tf.nn.sigmoid(logits)

# Pass logits through sigmoid and calculate the cross-entropy loss
loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=targets_, logits=logits)

# Get cost and define the optimizer
cost = tf.reduce_mean(loss)
opt = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

The result of the validation of the denoising is as below:



I added noise to the test images and passing them through the autoencoder. It does a suprisingly great job of removing the noise, even though it's sometimes difficult to tell what the original number is.

3 Generative Adverserial Networks (TensorFlow)

Generative adversarial networks (GANs) are algorithmic architectures that use two neural networks, pitting one against the other (thus the “adversarial”) in order to generate new, synthetic instances of data that can pass for real data. They are used widely in image generation, video generation and voice generation.

3.1 Network Architecture

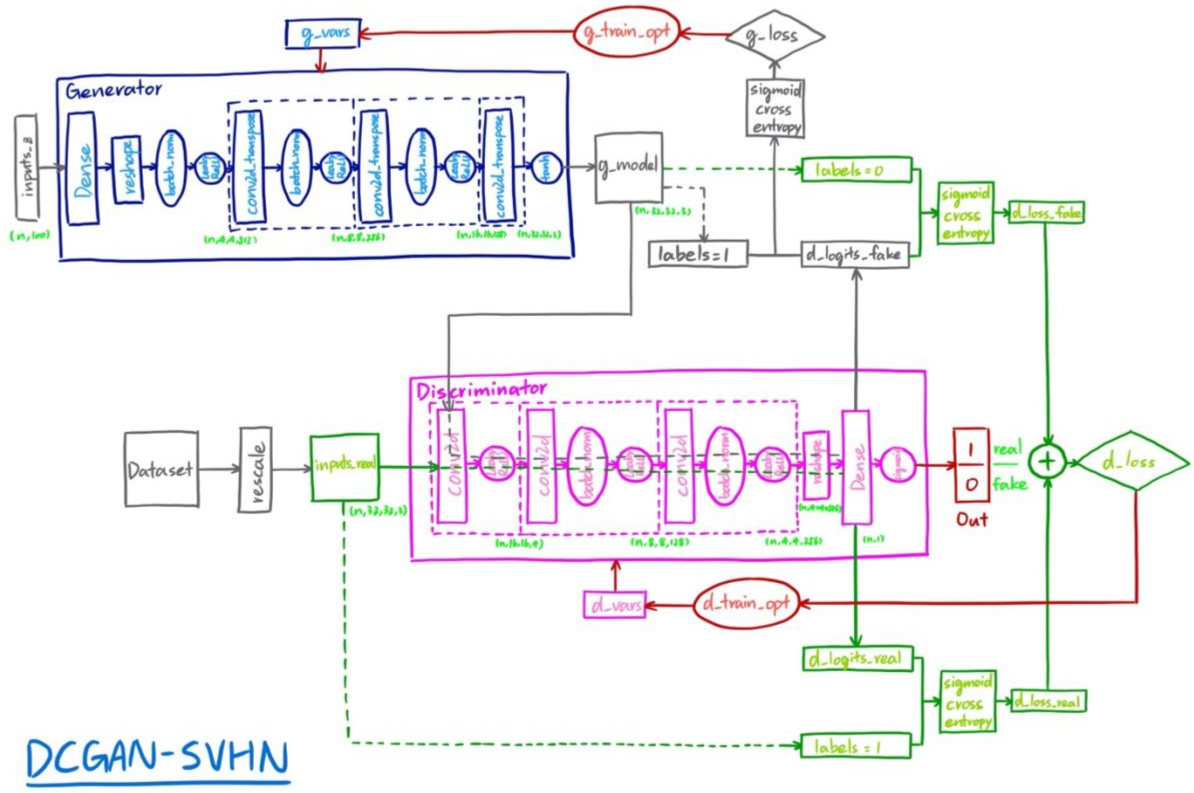
One neural network, called the generator, generates new data instances, while the other, the discriminator, evaluates them for authenticity; i.e. the discriminator decides whether each instance of data that it reviews belongs to the actual training dataset or not.

Meanwhile, the generator is creating new, synthetic images that it passes to the discriminator. It does so in the hopes that they, too, will be deemed authentic, even though they are fake. The goal of the generator is to generate passable hand-written digits: to lie without being caught. The goal of the discriminator is to identify images coming from the generator as fake.

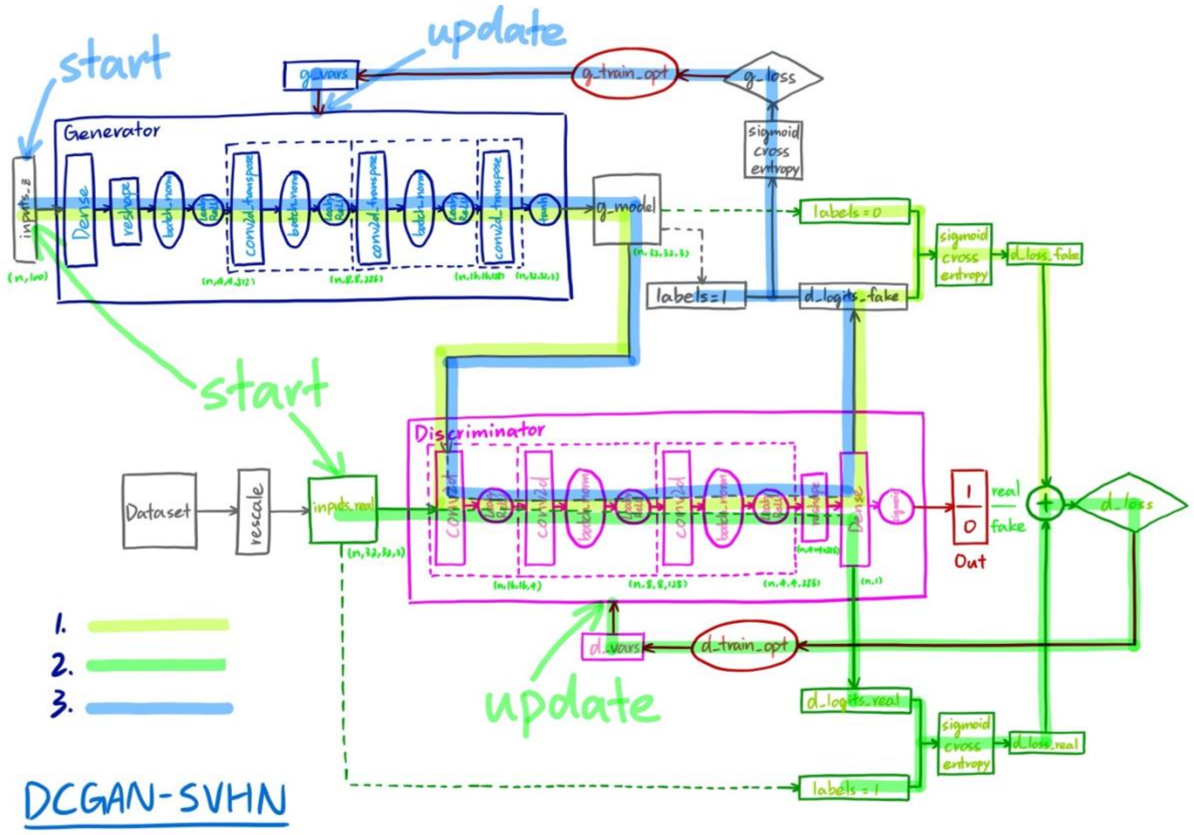
The steps a GAN takes:

- The generator takes in random numbers and returns an image.
- This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.
- The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

Here is the architecture graph I understand:



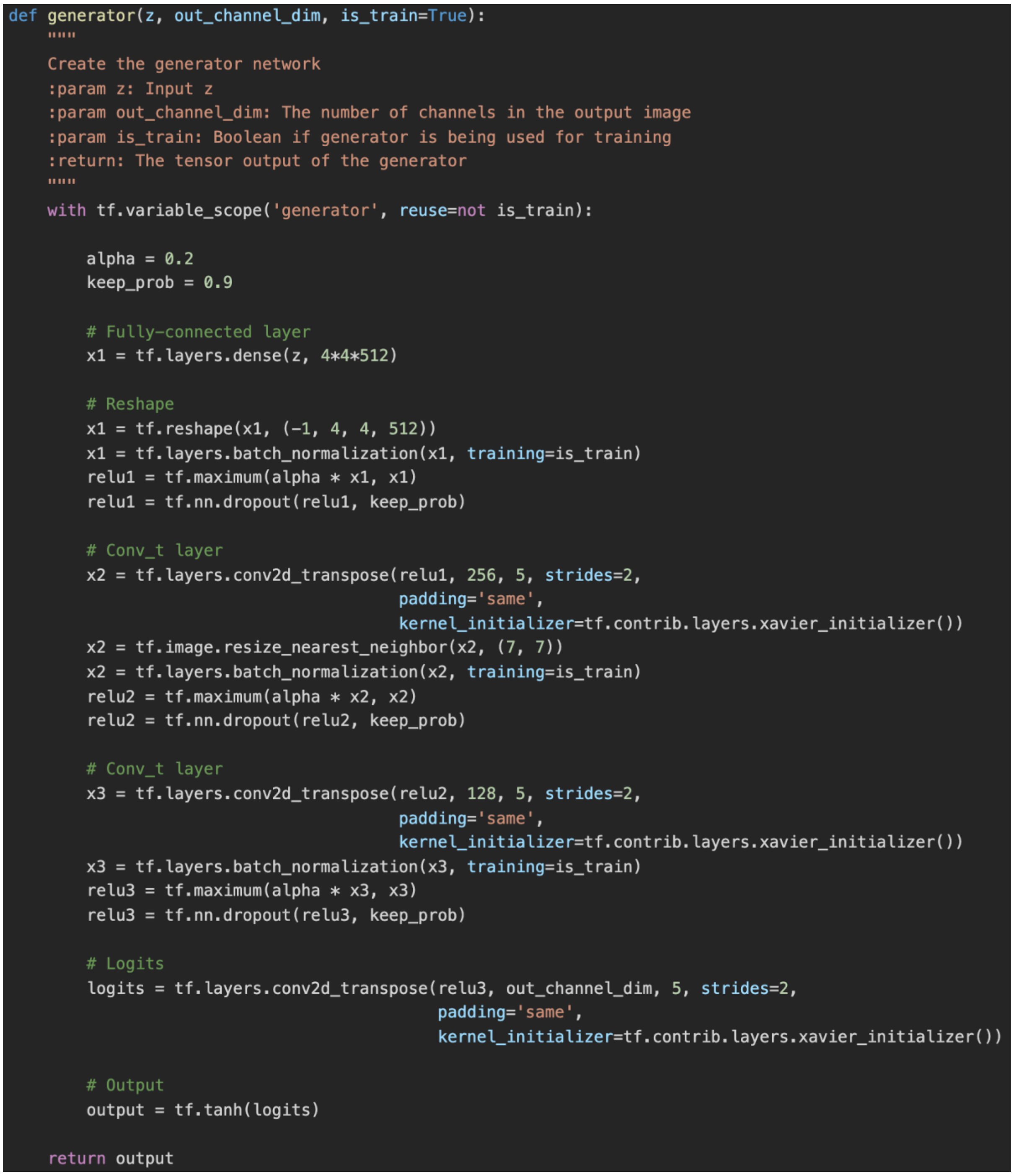
The update process is as folliwing:



3.2 Code Implementation

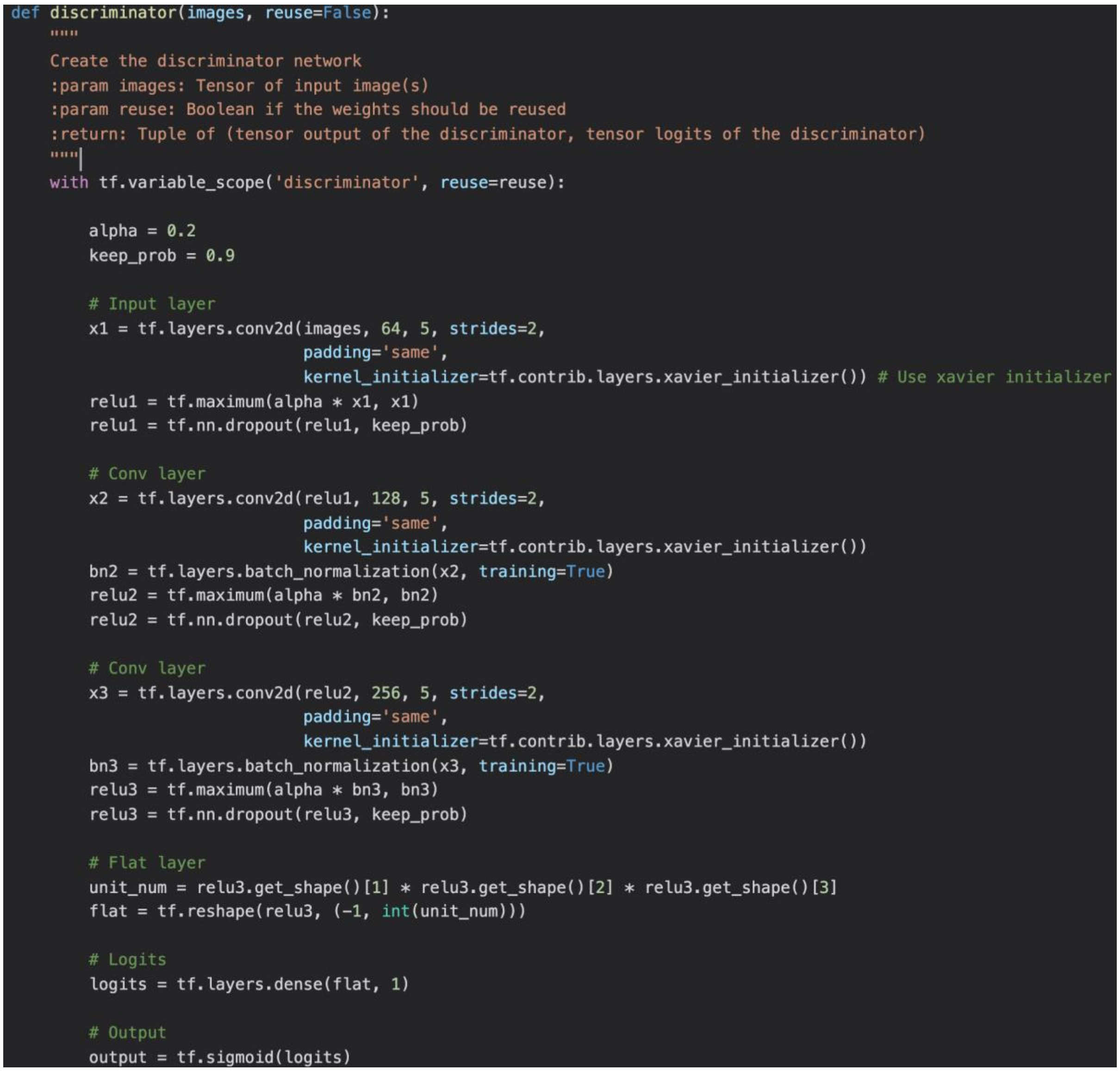
I implemented the GAN in TensorFlow because I'm more familiar with it than PyTorch.

Generator:



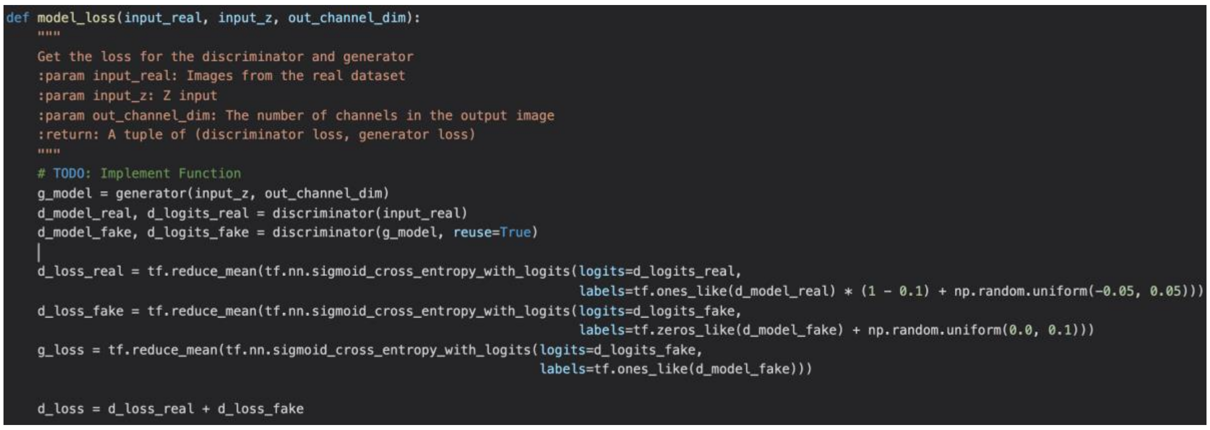
```python
def generator(z, out_channel_dim, is_train=True):
    """
    Create the generator network
    :param z: Input z
    :param out_channel_dim: The number of channels in the output image
    :param is_train: Boolean if generator is being used for training
    :return: The tensor output of the generator
    """
    with tf.variable_scope('generator', reuse=not is_train):

        alpha = 0.2
        keep_prob = 0.9

        # Fully-connected layer
        x1 = tf.layers.dense(z, 4*4*512)

        # Reshape
        x1 = tf.reshape(x1, (-1, 4, 4, 512))
        x1 = tf.layers.batch_normalization(x1, training=is_train)
        relu1 = tf.maximum(alpha * x1, x1)
        relu1 = tf.nn.dropout(relu1, keep_prob)

        # Conv_t layer
        x2 = tf.layers.conv2d_transpose(relu1, 256, 5, strides=2,
                                        padding='same',
                                        kernel_initializer=tf.contrib.layers.xavier_initializer())
        x2 = tf.image.resize_nearest_neighbor(x2, (7, 7))
        x2 = tf.layers.batch_normalization(x2, training=is_train)
        relu2 = tf.maximum(alpha * x2, x2)
        relu2 = tf.nn.dropout(relu2, keep_prob)

        # Conv_t layer
        x3 = tf.layers.conv2d_transpose(relu2, 128, 5, strides=2,
                                        padding='same',
                                        kernel_initializer=tf.contrib.layers.xavier_initializer())
        x3 = tf.layers.batch_normalization(x3, training=is_train)
        relu3 = tf.maximum(alpha * x3, x3)
        relu3 = tf.nn.dropout(relu3, keep_prob)

        # Logits
        logits = tf.layers.conv2d_transpose(relu3, out_channel_dim, 5, strides=2,
                                            padding='same',
                                            kernel_initializer=tf.contrib.layers.xavier_initializer())

        # Output
        output = tf.tanh(logits)

    return output
```

Discriminator:



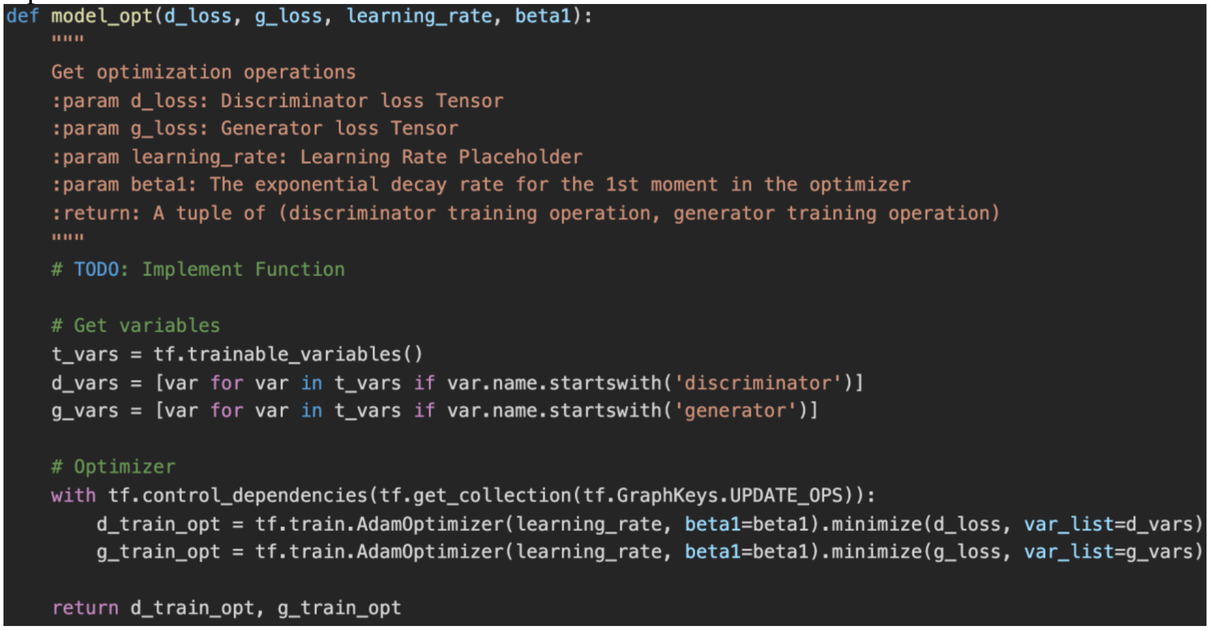
```python
def discriminator(images, reuse=False):
    """
    Create the discriminator network
    :param images: Tensor of input image(s)
    :param reuse: Boolean if the weights should be reused
    :return: Tuple of (tensor output of the discriminator, tensor logits of the discriminator)
    """
    with tf.variable_scope('discriminator', reuse=reuse):

        alpha = 0.2
        keep_prob = 0.9

        # Input layer
        x1 = tf.layers.conv2d(images, 64, 5, strides=2,
                              padding='same',
                              kernel_initializer=tf.contrib.layers.xavier_initializer()) # Use xavier initializer
        relu1 = tf.maximum(alpha * x1, x1)
        relu1 = tf.nn.dropout(relu1, keep_prob)

        # Conv layer
        x2 = tf.layers.conv2d(relu1, 128, 5, strides=2,
                              padding='same',
                              kernel_initializer=tf.contrib.layers.xavier_initializer())
        bn2 = tf.layers.batch_normalization(x2, training=True)
        relu2 = tf.maximum(alpha * bn2, bn2)
        relu2 = tf.nn.dropout(relu2, keep_prob)

        # Conv layer
        x3 = tf.layers.conv2d(relu2, 256, 5, strides=2,
                              padding='same',
                              kernel_initializer=tf.contrib.layers.xavier_initializer())
        bn3 = tf.layers.batch_normalization(x3, training=True)
        relu3 = tf.maximum(alpha * bn3, bn3)
        relu3 = tf.nn.dropout(relu3, keep_prob)

        # Flat layer
        unit_num = relu3.get_shape()[1] * relu3.get_shape()[2] * relu3.get_shape()[3]
        flat = tf.reshape(relu3, (-1, int(unit_num)))

        # Logits
        logits = tf.layers.dense(flat, 1)

        # Output
        output = tf.sigmoid(logits)
```

Loss:



```python
def model_loss(input_real, input_z, out_channel_dim):
    """
    Get the loss for the discriminator and generator
    :param input_real: Images from the real dataset
    :param input_z: Z input
    :param out_channel_dim: The number of channels in the output image
    :return: A tuple of (discriminator loss, generator loss)
    """
    # TODO: Implement Function
    g_model = generator(input_z, out_channel_dim)
    d_model_real, d_logits_real = discriminator(input_real)
    d_model_fake, d_logits_fake = discriminator(g_model, reuse=True)

    d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_real,
                                                                         labels=tf.ones_like(d_model_real) * (1 - 0.1) + np.random.uniform(-0.05, 0.05)))
    d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                                                         labels=tf.zeros_like(d_model_fake) + np.random.uniform(0.0, 0.1)))
    g_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logits_fake,
                                                                    labels=tf.ones_like(d_model_fake)))

    d_loss = d_loss_real + d_loss_fake
```

Optimizer:

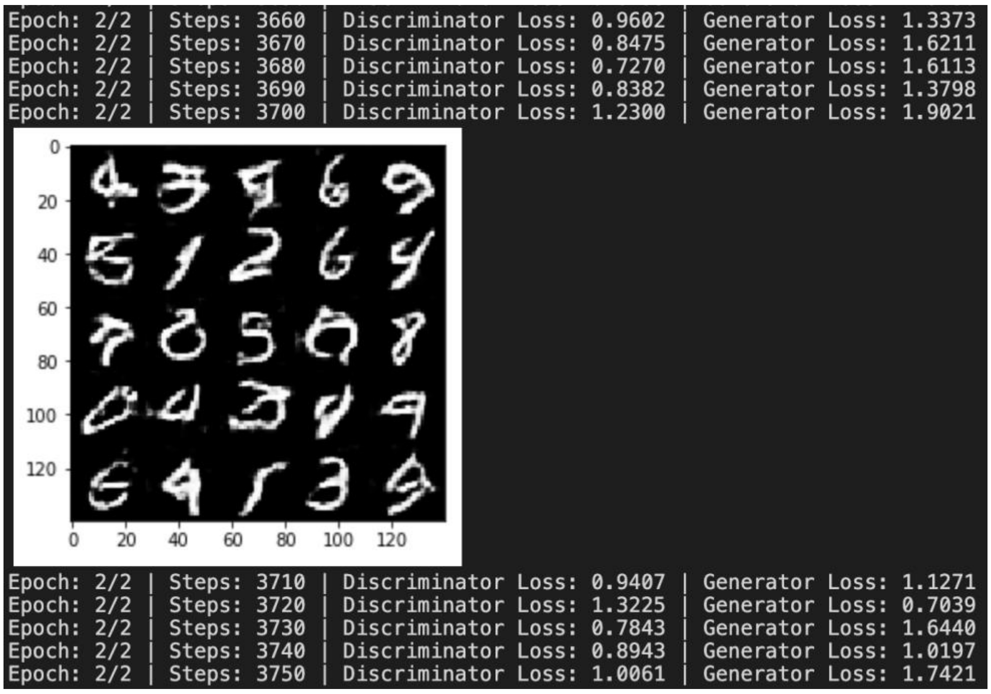


```python
def model_opt(d_loss, g_loss, learning_rate, beta1):
    """
    Get optimization operations
    :param d_loss: Discriminator loss Tensor
    :param g_loss: Generator loss Tensor
    :param learning_rate: Learning Rate Placeholder
    :param beta1: The exponential decay rate for the 1st moment in the optimizer
    :return: A tuple of (discriminator training operation, generator training operation)
    """
    # TODO: Implement Function

    # Get variables
    t_vars = tf.trainable_variables()
    d_vars = [var for var in t_vars if var.name.startswith('discriminator')]
    g_vars = [var for var in t_vars if var.name.startswith('generator')]

    # Optimizer
    with tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        d_train_opt = tf.train.AdamOptimizer(learning_rate, beta1=beta1).minimize(d_loss, var_list=d_vars)
        g_train_opt = tf.train.AdamOptimizer(learning_rate, beta1=beta1).minimize(g_loss, var_list=g_vars)

    return d_train_opt, g_train_opt
```

3.3 Training & Evaluation

3.3.1 MNIST – Hand Witten Digits Dataset

I used the MNIST dataset for digits generation.

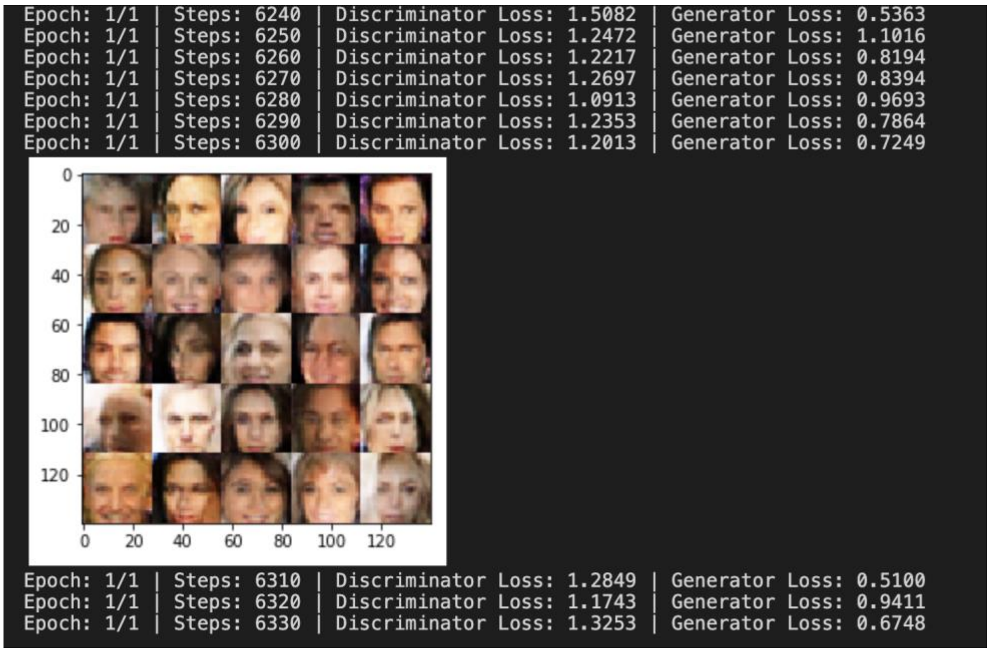


```
Epoch: 2/2 | Steps: 3660 | Discriminator Loss: 0.9602 | Generator Loss: 1.3373
Epoch: 2/2 | Steps: 3670 | Discriminator Loss: 0.8475 | Generator Loss: 1.6211
Epoch: 2/2 | Steps: 3680 | Discriminator Loss: 0.7270 | Generator Loss: 1.6113
Epoch: 2/2 | Steps: 3690 | Discriminator Loss: 0.8382 | Generator Loss: 1.3798
Epoch: 2/2 | Steps: 3700 | Discriminator Loss: 1.2300 | Generator Loss: 1.9021
```

```
Epoch: 2/2 | Steps: 3710 | Discriminator Loss: 0.9407 | Generator Loss: 1.1271
Epoch: 2/2 | Steps: 3720 | Discriminator Loss: 1.3225 | Generator Loss: 0.7039
Epoch: 2/2 | Steps: 3730 | Discriminator Loss: 0.7843 | Generator Loss: 1.6440
Epoch: 2/2 | Steps: 3740 | Discriminator Loss: 0.8943 | Generator Loss: 1.0197
Epoch: 2/2 | Steps: 3750 | Discriminator Loss: 1.0061 | Generator Loss: 1.7421
```

3.3.2 CelebA – Human Faces Dataset

I used the CelebA dataset for face generation. It took around 20 minutes on my GPU server to run one epoch.



```
Epoch: 1/1 | Steps: 6240 | Discriminator Loss: 1.5082 | Generator Loss: 0.5363
Epoch: 1/1 | Steps: 6250 | Discriminator Loss: 1.2472 | Generator Loss: 1.1016
Epoch: 1/1 | Steps: 6260 | Discriminator Loss: 1.2217 | Generator Loss: 0.8194
Epoch: 1/1 | Steps: 6270 | Discriminator Loss: 1.2697 | Generator Loss: 0.8394
Epoch: 1/1 | Steps: 6280 | Discriminator Loss: 1.0913 | Generator Loss: 0.9693
Epoch: 1/1 | Steps: 6290 | Discriminator Loss: 1.2353 | Generator Loss: 0.7864
Epoch: 1/1 | Steps: 6300 | Discriminator Loss: 1.2013 | Generator Loss: 0.7249
```

```
Epoch: 1/1 | Steps: 6310 | Discriminator Loss: 1.2849 | Generator Loss: 0.5100
Epoch: 1/1 | Steps: 6320 | Discriminator Loss: 1.1743 | Generator Loss: 0.9411
Epoch: 1/1 | Steps: 6330 | Discriminator Loss: 1.3253 | Generator Loss: 0.6748
```

As shown above, the network could roughly generate the human faces.