Image Generation using PixelCNN, PixelRNN, VaE, GAN

Group Members

Akshay Agrawal(19CH10008)
Jay Dutonde(19CH30008)

Abstract

A GAN is a type of neural network that is able to generate new data from scratch. You can feed it a little bit of random noise as input, and it can produce realistic images of bedrooms, or birds, or whatever it is trained to generate. One thing all scientists can agree on is that we need more data. The advancement in Machine Learning today is because of more computational power and more data. The data collection part is often expensive and sometimes we don't have enough data to train the model properly, here we can use several Generative Models to generate artificial data. Till now Machine Learning was mainly used to get rid of rule based models but with the help of Generative Models we can enter the creative world and ask Model to generate Art Images, Human Images etc which it has not seen. With the help of VaE we can achieve Lossless Data transmission. This can be achieved by first encoding the data into smaller dimensional Latent space and transmitting it and further decoding it with the decoder block into the other side and hence achieving faster data Transmission. In this project we are aiming to build various generative models trained on MNIST dataset and compare the performances of the output.

METHODOLOGY

This section gives details about the design / architecture of various generative models. The data preprocessing and the loss function used to train the models.

DATA

The data is the famous MNIST dataset which contains 60,000 images of handwritten digits (0-9).
http://yann.lecun.com/exdb/mnist/

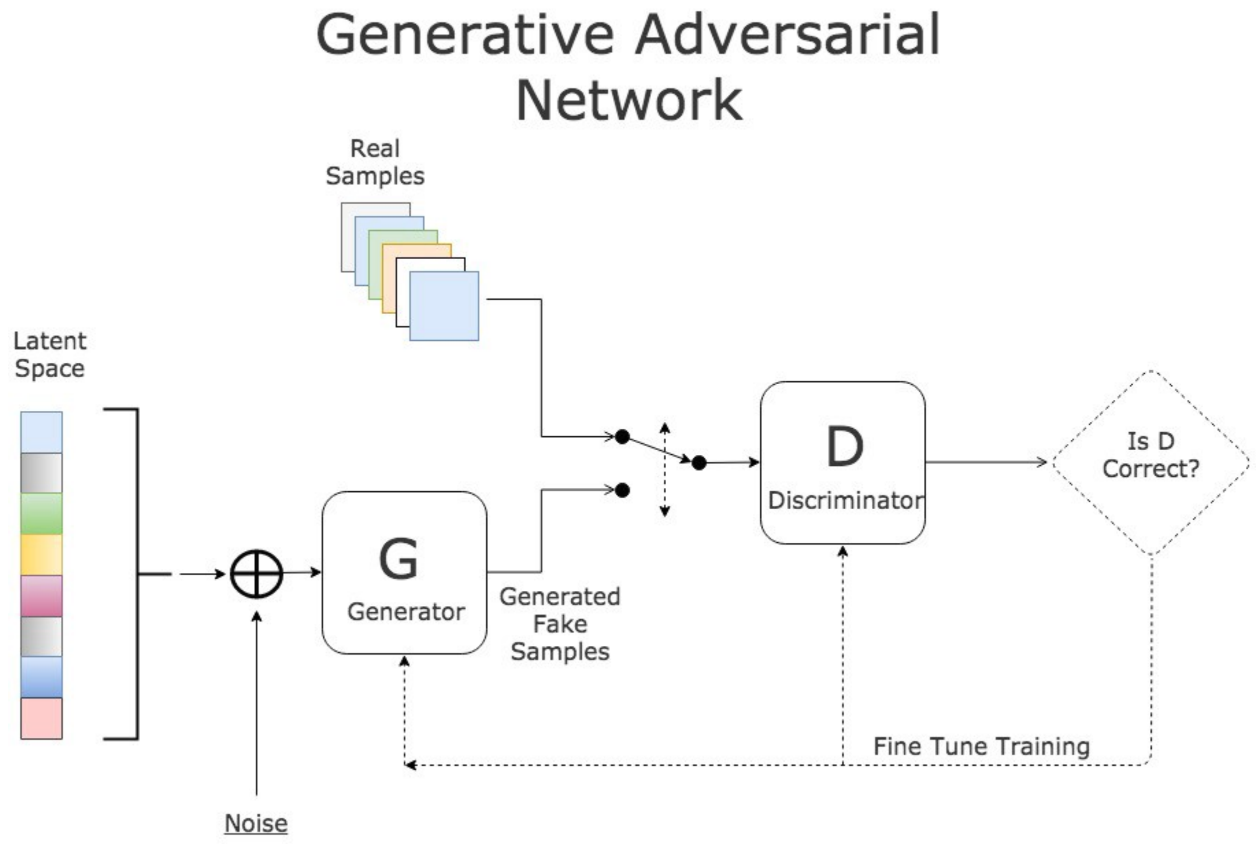
ARCHITECTURES

1) **Vanilla GAN**

Generative Adversarial Networks

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio

We studied the above paper and built our custom Generator and Discriminator using only Dense Layers.



Preprocessing

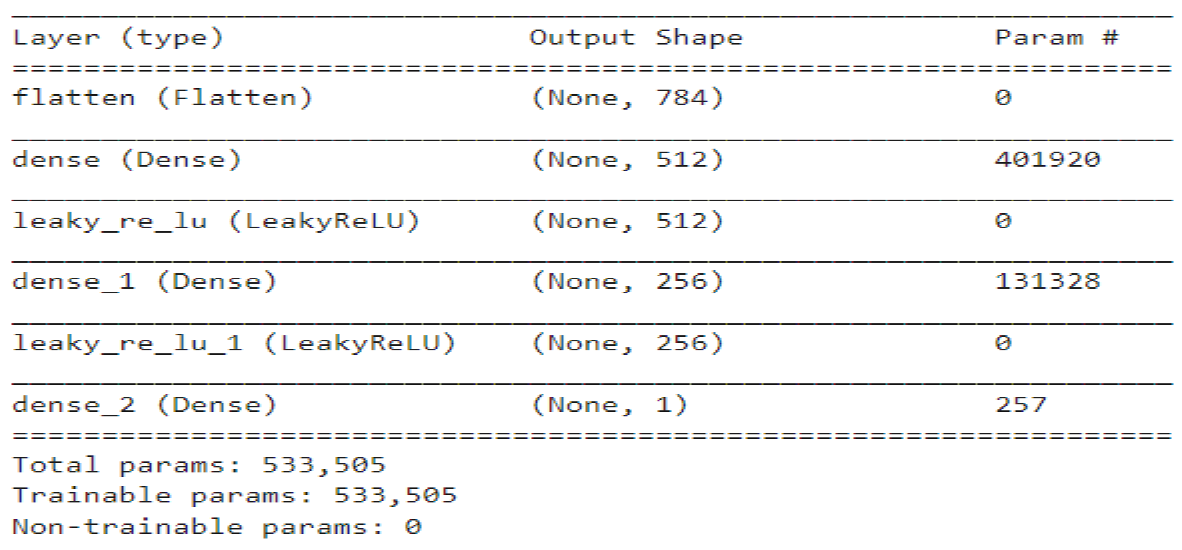
We used OpenCV to read and preprocess the image. Each image was normalized around 0. After transformation each pixel contains values [-1 1]. Some implementations used to transform image pixels in range [0 1]. We used the first one.

Generator Architecture

```python
def build_generator():
  noise_shape = (100, ) #1D array of size 100 (latent vector / noise)

  #Define your generator network
  #Here we are only using Dense layers. But network can be complicated based
  #on the application. For example, you can use VGG for super res. GAN.

  model = Sequential([
                      Dense(256, input_shape = noise_shape),
                      LeakyReLU(alpha = 0.2),
                      BatchNormalization(momentum = 0.8),
                      Dense(512),
                      LeakyReLU(alpha=0.2),
                      BatchNormalization(momentum = 0.8),
                      Dense(1024),
                      LeakyReLU(alpha=0.2),
                      BatchNormalization(momentum = 0.8),
                      Dense(np.prod(img_shape), activation = 'tanh'),
                      Reshape(img_shape)
  ])

  model.summary()

  noise = Input(noise_shape)
  gen_img = model(noise)

  return Model(noise, gen_img)
```

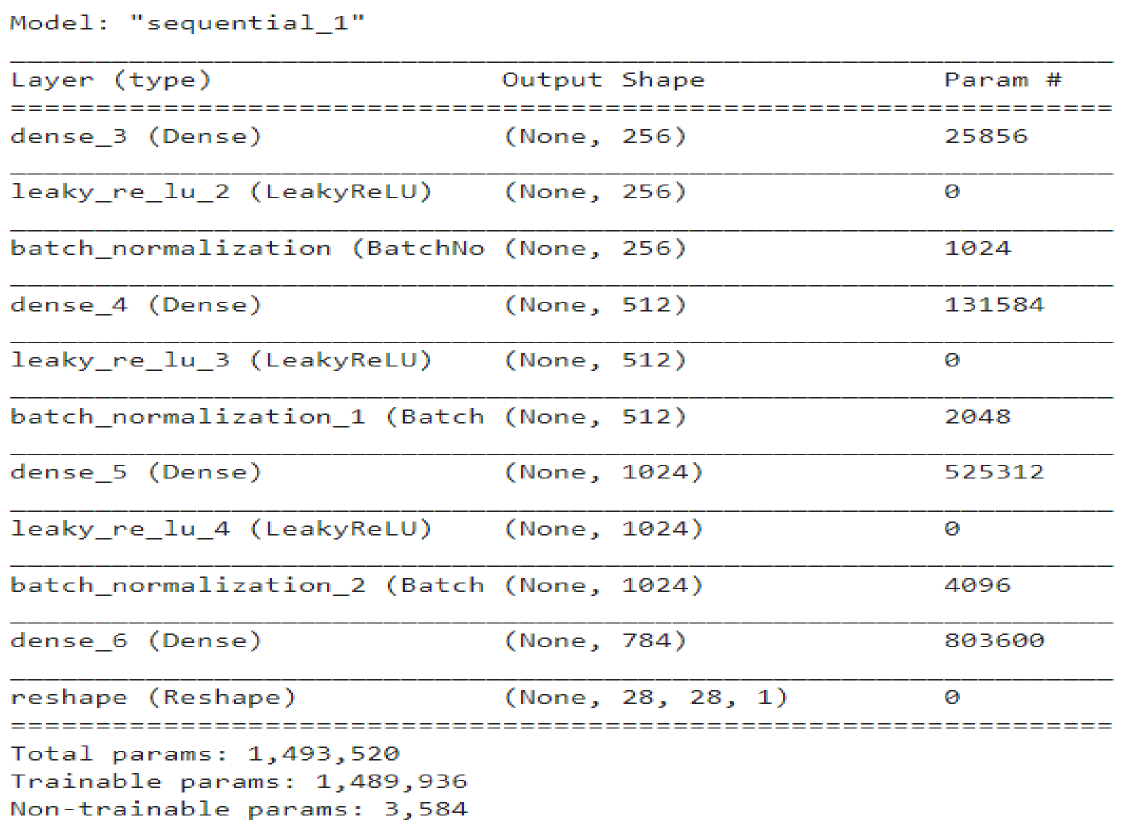
Our Generator taken **100d noise** as input and generates **28*28 images** as output. We used **Leaky ReLU** activation function with **alpha = 0.2** to solve the **dying ReLU problem** and also at the end of each layer we used **Batch Normalisation** with **momentum = 0.8**. At the output we used the tanh **activation** function because our input was scaled to [-1 1]. For training we used **Adam Optimizer with learning late, momentum = 0.0002, 0.5.**

Discriminator



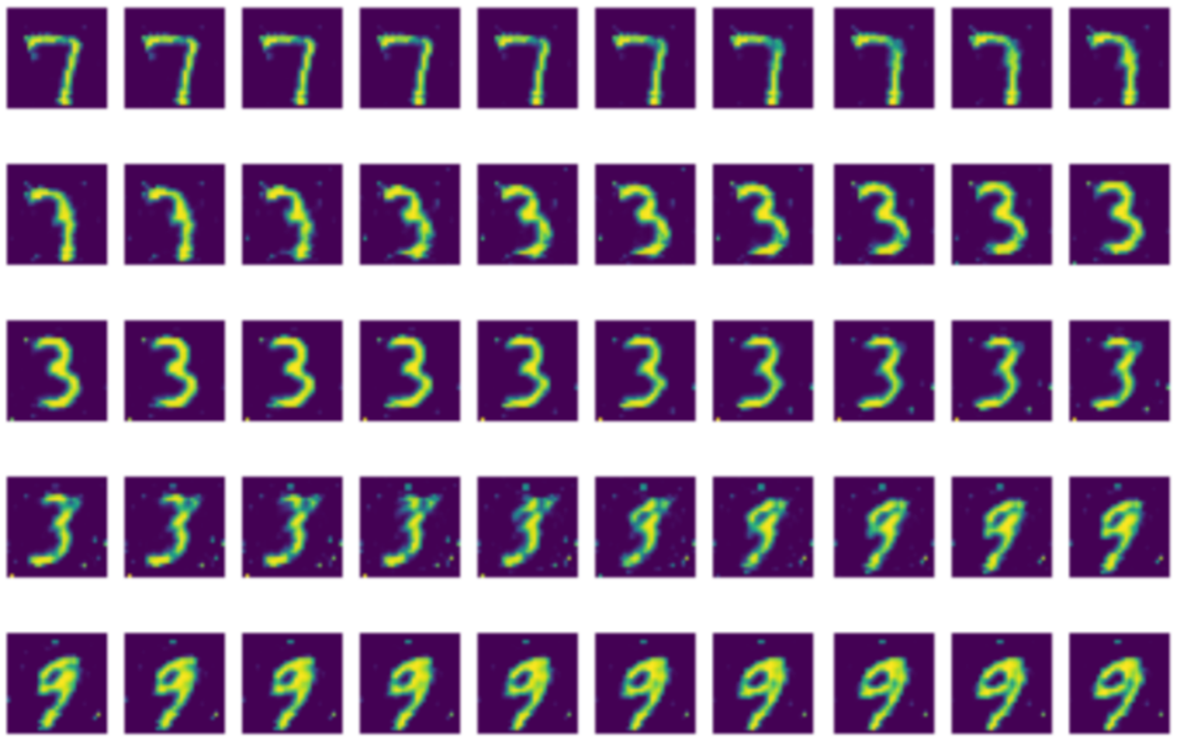
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 512)               401920
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 533,505
Trainable params: 533,505
Non-trainable params: 0
```

Generator



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 256)               25856
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 256)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 256)               1024
_________________________________________________________________
dense_4 (Dense)              (None, 512)               131584
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 512)               0
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
dense_5 (Dense)              (None, 1024)              525312
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 1024)              0
_________________________________________________________________
batch_normalization_2 (Batch (None, 1024)              4096
_________________________________________________________________
dense_6 (Dense)              (None, 784)               803600
_________________________________________________________________
reshape (Reshape)            (None, 28, 28, 1)         0
=================================================================
Total params: 1,493,520
Trainable params: 1,489,936
Non-trainable params: 3,584
```

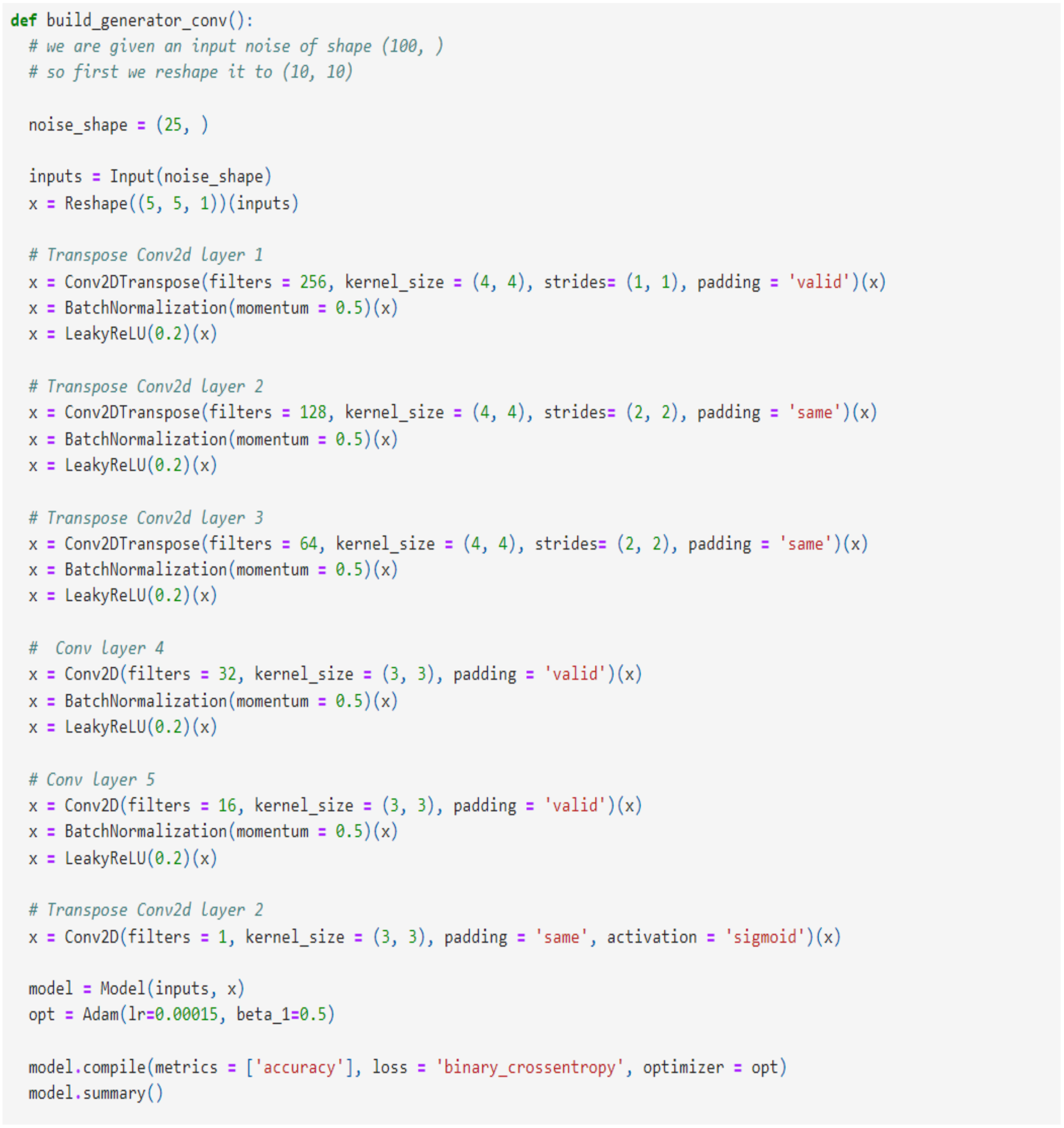
Sample Output of Vanilla GAN



2) DCGAN

We built our custom DCGAN where the discriminator contains **Conv2D** layers and the generator contains **transposed Convolutional layers** (Conv2DTranspose).

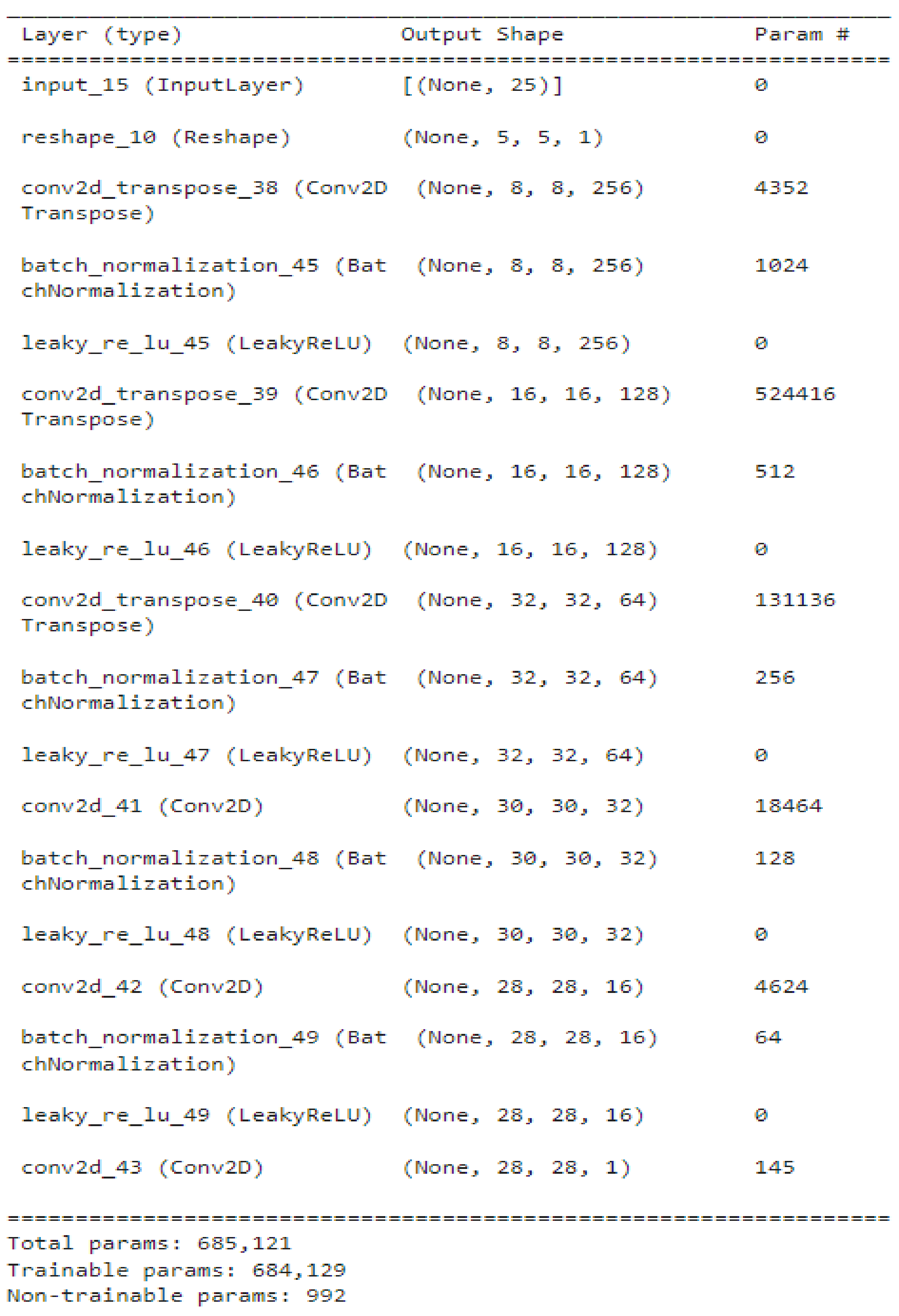
Generator Architecture



```
def build_generator_conv():
  # we are given an input noise of shape (100, )
  # so first we reshape it to (10, 10)

  noise_shape = (25, )

  inputs = Input(noise_shape)
  x = Reshape((5, 5, 1))(inputs)

  # Transpose Conv2d Layer 1
  x = Conv2DTranspose(filters = 256, kernel_size = (4, 4), strides= (1, 1), padding = 'valid')(x)
  x = BatchNormalization(momentum = 0.5)(x)
  x = LeakyReLU(0.2)(x)

  # Transpose Conv2d Layer 2
  x = Conv2DTranspose(filters = 128, kernel_size = (4, 4), strides= (2, 2), padding = 'same')(x)
  x = BatchNormalization(momentum = 0.5)(x)
  x = LeakyReLU(0.2)(x)

  # Transpose Conv2d Layer 3
  x = Conv2DTranspose(filters = 64, kernel_size = (4, 4), strides= (2, 2), padding = 'same')(x)
  x = BatchNormalization(momentum = 0.5)(x)
  x = LeakyReLU(0.2)(x)

  #  Conv Layer 4
  x = Conv2D(filters = 32, kernel_size = (3, 3), padding = 'valid')(x)
  x = BatchNormalization(momentum = 0.5)(x)
  x = LeakyReLU(0.2)(x)

  # Conv Layer 5
  x = Conv2D(filters = 16, kernel_size = (3, 3), padding = 'valid')(x)
  x = BatchNormalization(momentum = 0.5)(x)
  x = LeakyReLU(0.2)(x)

  # Transpose Conv2d Layer 2
  x = Conv2D(filters = 1, kernel_size = (3, 3), padding = 'same', activation = 'sigmoid')(x)

  model = Model(inputs, x)
  opt = Adam(lr=0.00015, beta_1=0.5)

  model.compile(metrics = ['accuracy'], loss = 'binary_crossentropy', optimizer = opt)
  model.summary()
```

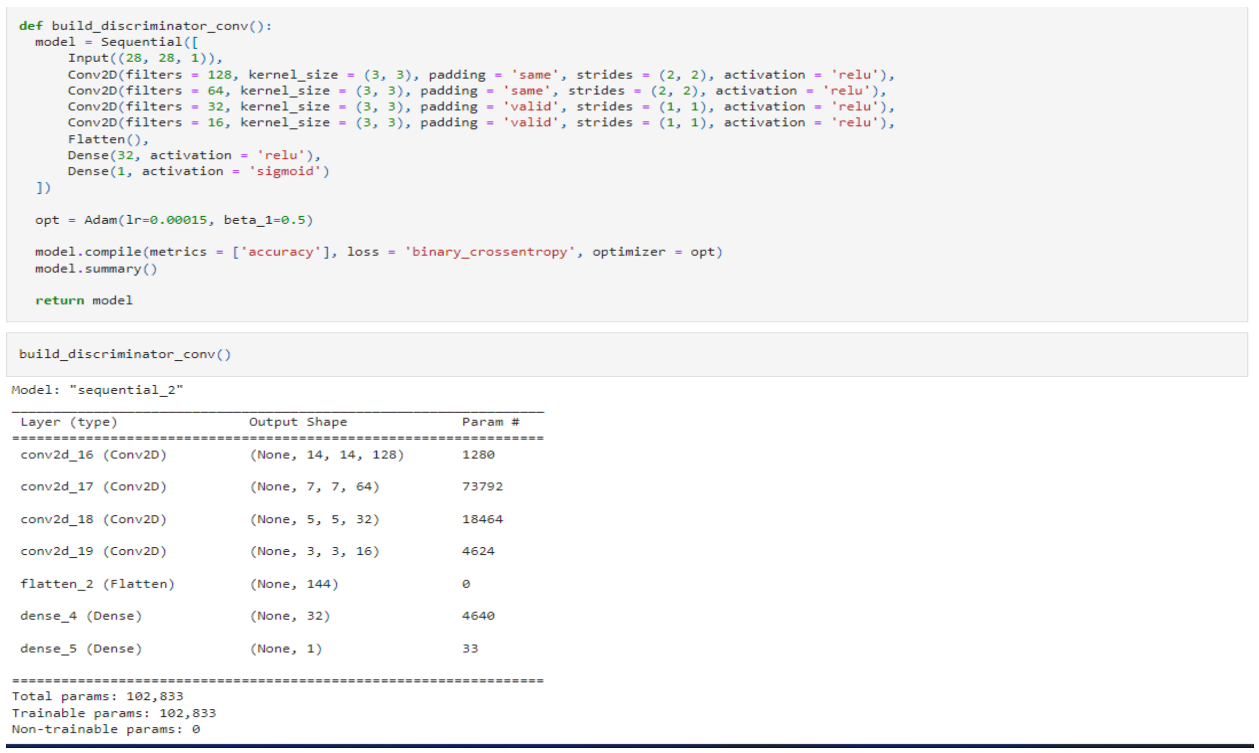
Our Generator taken **25d noise** as input and generates **28*28 images** as output. We used **Leaky ReLU** activation function with **alpha = 0.2** to solve the **dying ReLU problem** and also at the end of each layer we used **Batch Normalisation** with **momentum = 0.5**. At the output we used the sigmoid **activation** function because our input was scaled to [0 1]. For training we used **Adam Optimizer with learning late, momentum = 0.00015, 0.5.** We changed the transformation technique and some hyperparameters to notice the change in output if any.

Generator Architecture Summary



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_15 (InputLayer)       [(None, 25)]              0         
                                                                 
 reshape_10 (Reshape)        (None, 5, 5, 1)           0         
                                                                 
 conv2d_transpose_38 (Conv2D  (None, 8, 8, 256)        4352      
 Transpose)                                                      
                                                                 
 batch_normalization_45 (Bat  (None, 8, 8, 256)        1024      
 chNormalization)                                                
                                                                 
 leaky_re_lu_45 (LeakyReLU)  (None, 8, 8, 256)         0         
                                                                 
 conv2d_transpose_39 (Conv2D  (None, 16, 16, 128)      524416    
 Transpose)                                                      
                                                                 
 batch_normalization_46 (Bat  (None, 16, 16, 128)      512       
 chNormalization)                                                
                                                                 
 leaky_re_lu_46 (LeakyReLU)  (None, 16, 16, 128)       0         
                                                                 
 conv2d_transpose_40 (Conv2D  (None, 32, 32, 64)       131136    
 Transpose)                                                      
                                                                 
 batch_normalization_47 (Bat  (None, 32, 32, 64)       256       
 chNormalization)                                                
                                                                 
 leaky_re_lu_47 (LeakyReLU)  (None, 32, 32, 64)        0         
                                                                 
 conv2d_41 (Conv2D)          (None, 30, 30, 32)        18464     
                                                                 
 batch_normalization_48 (Bat  (None, 30, 30, 32)       128       
 chNormalization)                                                
                                                                 
 leaky_re_lu_48 (LeakyReLU)  (None, 30, 30, 32)        0         
                                                                 
 conv2d_42 (Conv2D)          (None, 28, 28, 16)        4624      
                                                                 
 batch_normalization_49 (Bat  (None, 28, 28, 16)       64        
 chNormalization)                                                
                                                                 
 leaky_re_lu_49 (LeakyReLU)  (None, 28, 28, 16)        0         
                                                                 
 conv2d_43 (Conv2D)          (None, 28, 28, 1)         145       
                                                                 
=================================================================
Total params: 685,121
Trainable params: 684,129
Non-trainable params: 992
```

Discriminator Architecture and Summary

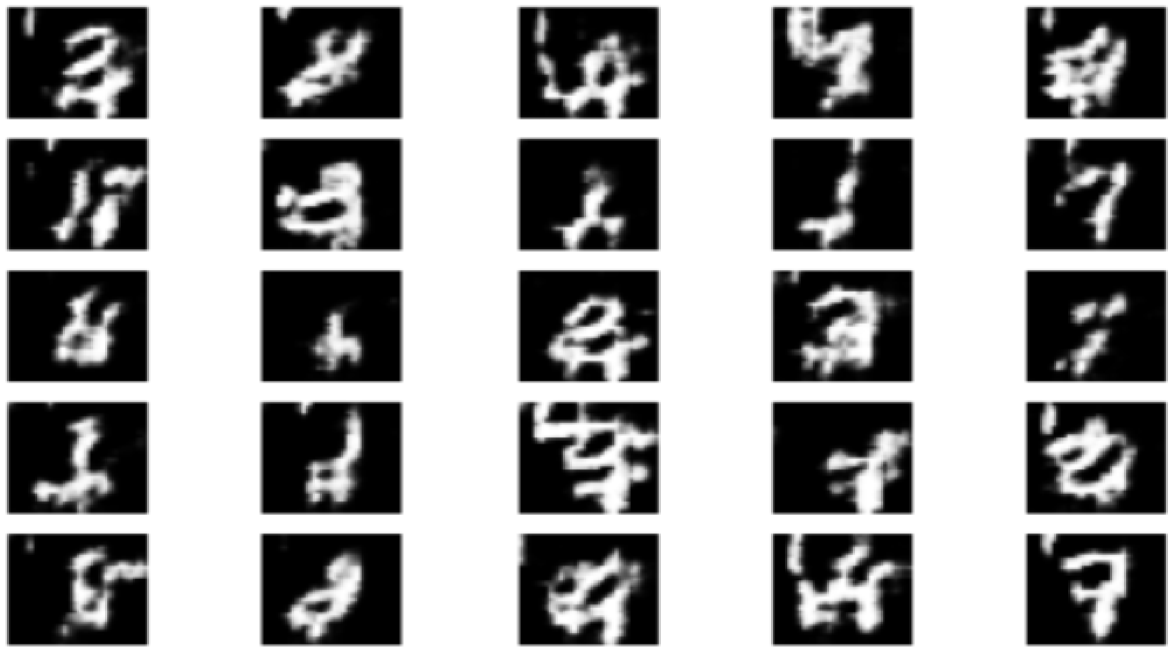


```
def build_discriminator_conv():
  model = Sequential([
      Input((28, 28, 1)),
      Conv2D(filters = 128, kernel_size = (3, 3), padding = 'same', strides = (2, 2), activation = 'relu'),
      Conv2D(filters = 64, kernel_size = (3, 3), padding = 'same', strides = (2, 2), activation = 'relu'),
      Conv2D(filters = 32, kernel_size = (3, 3), padding = 'valid', strides = (1, 1), activation = 'relu'),
      Conv2D(filters = 16, kernel_size = (3, 3), padding = 'valid', strides = (1, 1), activation = 'relu'),
      Flatten(),
      Dense(32, activation = 'relu'),
      Dense(1, activation = 'sigmoid')
  ])

  opt = Adam(lr=0.00015, beta_1=0.5)

  model.compile(metrics = ['accuracy'], loss = 'binary_crossentropy', optimizer = opt)
  model.summary()

  return model
```

```
build_discriminator_conv()
```

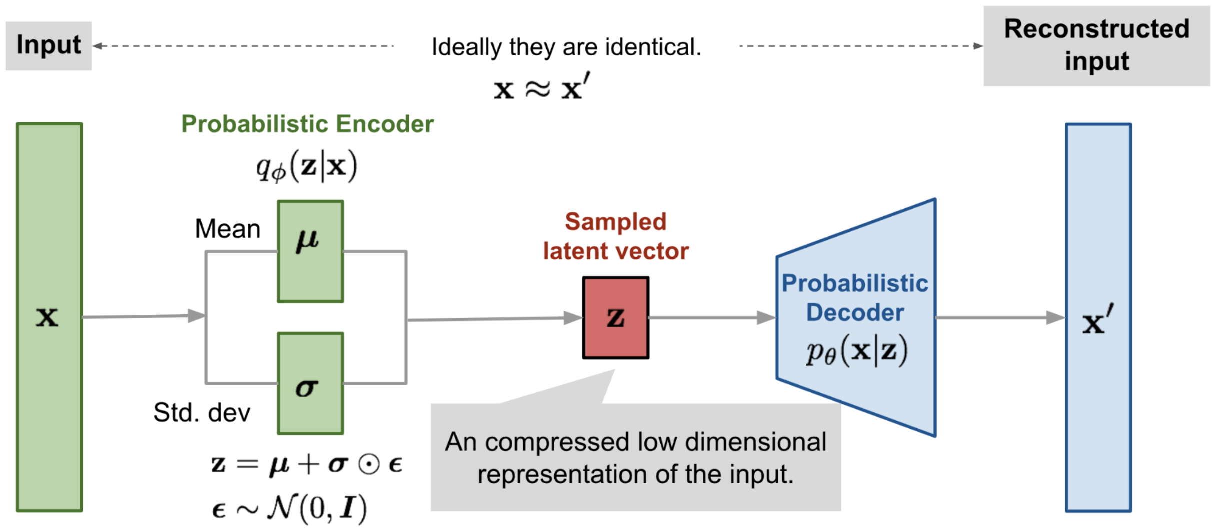
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_16 (Conv2D)          (None, 14, 14, 128)       1280

 conv2d_17 (Conv2D)          (None, 7, 7, 64)          73792

 conv2d_18 (Conv2D)          (None, 5, 5, 32)          18464

 conv2d_19 (Conv2D)          (None, 3, 3, 16)          4624

 flatten_2 (Flatten)         (None, 144)               0

 dense_4 (Dense)             (None, 32)                4640

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 102,833
Trainable params: 102,833
Non-trainable params: 0
```

Sample Output of DCGAN



3) VAE

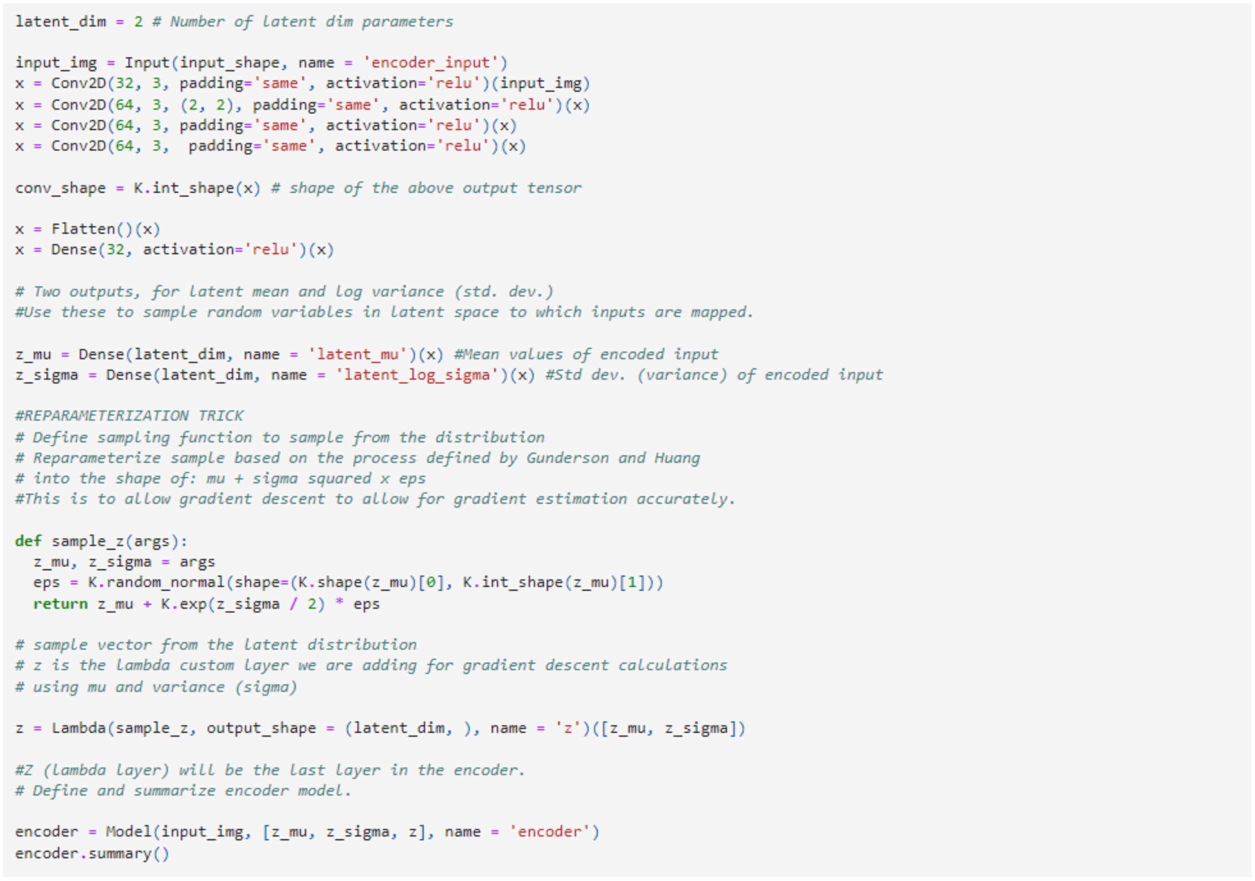
We studied the paper **An Introduction to Variational Autoencoders** Diederik P. Kingma, Max Welling and built our custom encoder and decoder.


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
Mean
$\boldsymbol{\mu}$
$\mathbf{x}$
$\boldsymbol{\sigma}$
Std. dev
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

Preprocessing

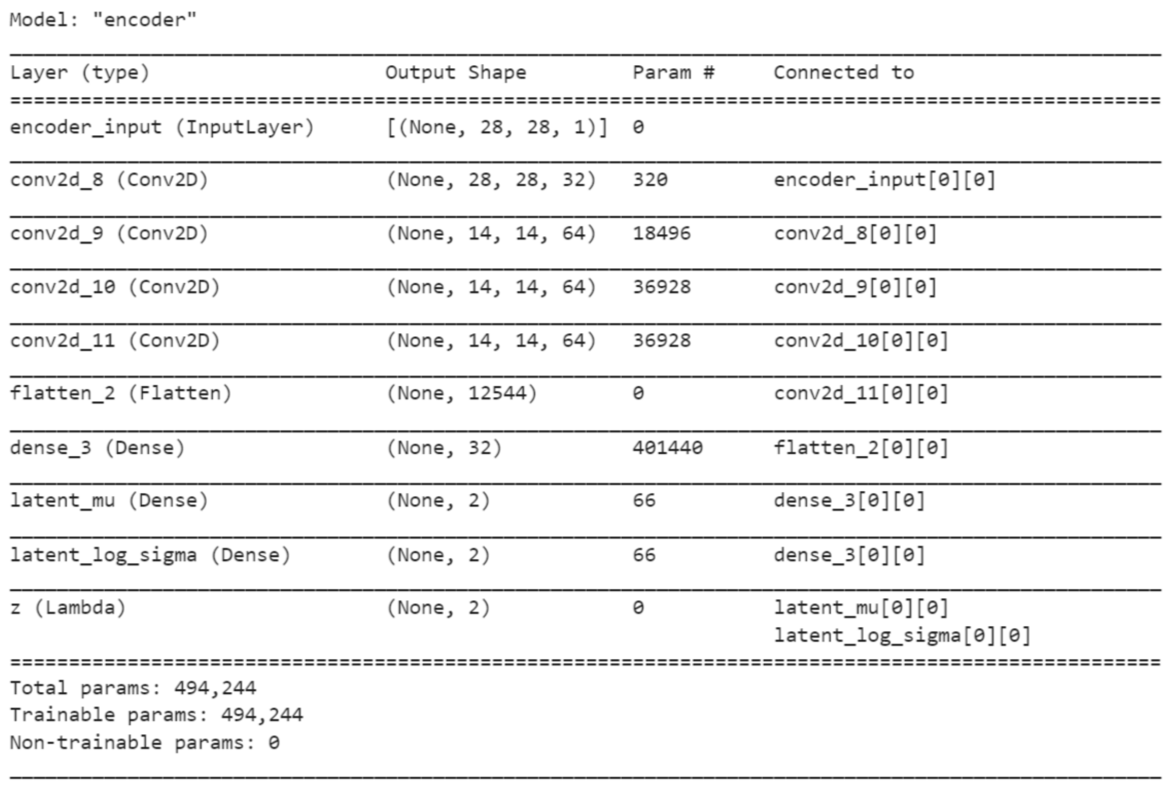
We scaled the image pixels in range [0 1]. For every (i, j) of image X we apply X[i][j] = X[i][j]/255.

Encoder Architecture



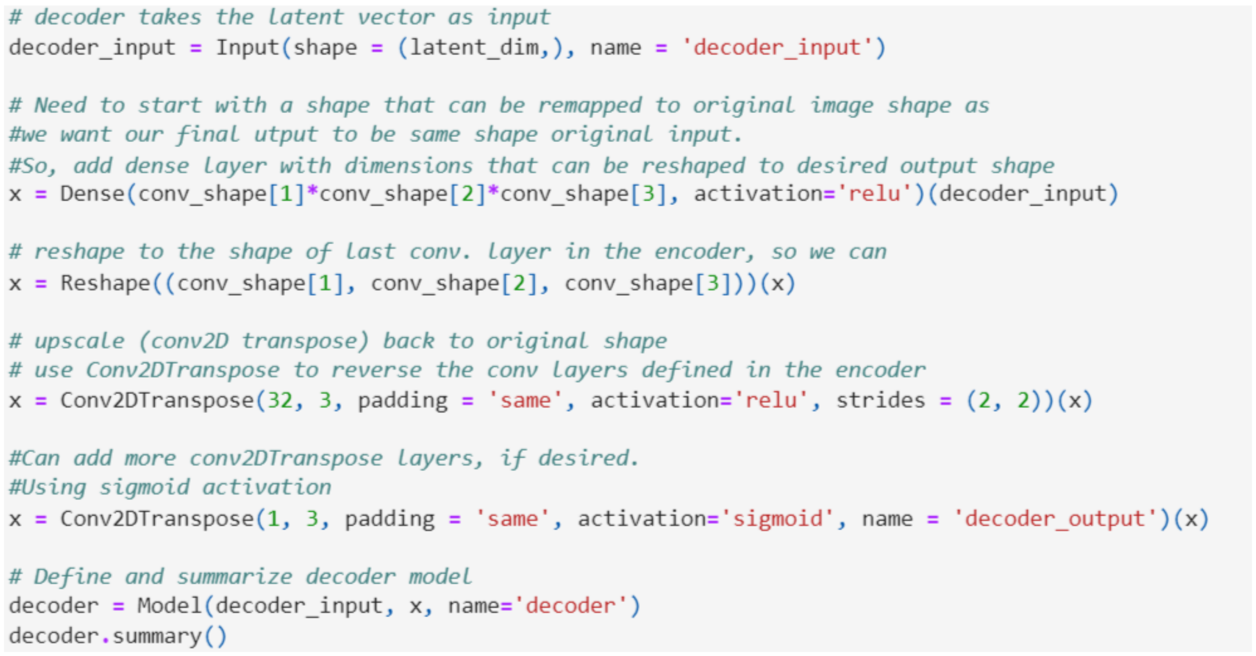
```python
latent_dim = 2 # Number of latent dim parameters

input_img = Input(input_shape, name = 'encoder_input')
x = Conv2D(32, 3, padding='same', activation='relu')(input_img)
x = Conv2D(64, 3, (2, 2), padding='same', activation='relu')(x)
x = Conv2D(64, 3, padding='same', activation='relu')(x)
x = Conv2D(64, 3,  padding='same', activation='relu')(x)

conv_shape = K.int_shape(x) # shape of the above output tensor

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

# Two outputs, for latent mean and log variance (std. dev.)
#Use these to sample random variables in latent space to which inputs are mapped.

z_mu = Dense(latent_dim, name = 'latent_mu')(x) #Mean values of encoded input
z_sigma = Dense(latent_dim, name = 'latent_log_sigma')(x) #Std dev. (variance) of encoded input

#REPARAMETERIZATION TRICK
# Define sampling function to sample from the distribution
# Reparameterize sample based on the process defined by Gunderson and Huang
# into the shape of: mu + sigma squared x eps
#This is to allow gradient descent to allow for gradient estimation accurately.

def sample_z(args):
  z_mu, z_sigma = args
  eps = K.random_normal(shape=(K.shape(z_mu)[0], K.int_shape(z_mu)[1]))
  return z_mu + K.exp(z_sigma / 2) * eps

# sample vector from the latent distribution
# z is the lambda custom layer we are adding for gradient descent calculations
# using mu and variance (sigma)

z = Lambda(sample_z, output_shape = (latent_dim, ), name = 'z')([z_mu, z_sigma])

#Z (lambda layer) will be the last layer in the encoder.
# Define and summarize encoder model.

encoder = Model(input_img, [z_mu, z_sigma, z], name = 'encoder')
encoder.summary()
```

Here we use **latent dimension** (dimension of latent variable) = **2**, In encoder we **4 Conv2D layers and 2 dense layers**. We also applied a reparameterization technique to allow our model to be trained using backpropagation. The implementation can be seen in the above image.



```
Model: "encoder"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
encoder_input (InputLayer)      [(None, 28, 28, 1)]  0

__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 28, 28, 32)   320         encoder_input[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 14, 14, 64)   18496       conv2d_8[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 14, 14, 64)   36928       conv2d_9[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 14, 14, 64)   36928       conv2d_10[0][0]
__________________________________________________________________________________________________
flatten_2 (Flatten)             (None, 12544)        0           conv2d_11[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                 (None, 32)           401440      flatten_2[0][0]
__________________________________________________________________________________________________
latent_mu (Dense)               (None, 2)            66          dense_3[0][0]
__________________________________________________________________________________________________
latent_log_sigma (Dense)        (None, 2)            66          dense_3[0][0]
__________________________________________________________________________________________________
z (Lambda)                      (None, 2)            0           latent_mu[0][0]
                                                                 latent_log_sigma[0][0]
==================================================================================================
Total params: 494,244
Trainable params: 494,244
Non-trainable params: 0
__________________________________________________________________________________________________
```

Decoder Architecture



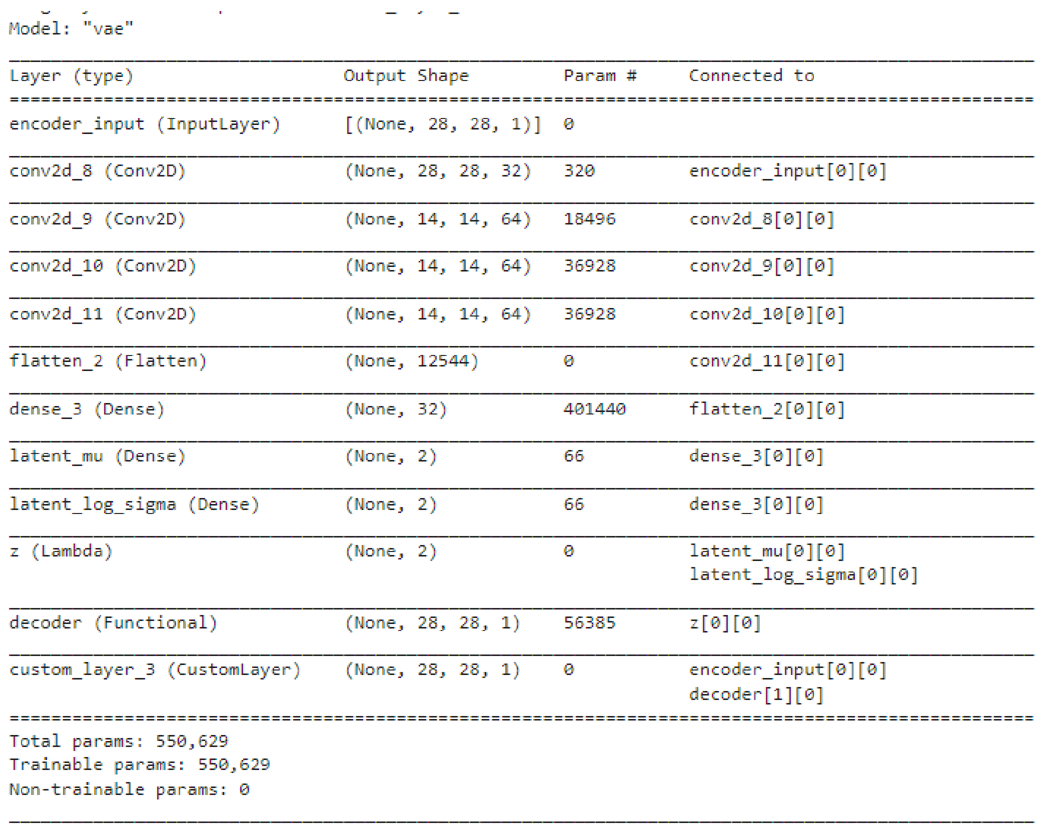
```
# decoder takes the latent vector as input
decoder_input = Input(shape = (latent_dim,), name = 'decoder_input')

# Need to start with a shape that can be remapped to original image shape as
#we want our final utput to be same shape original input.
#So, add dense layer with dimensions that can be reshaped to desired output shape
x = Dense(conv_shape[1]*conv_shape[2]*conv_shape[3], activation='relu')(decoder_input)

# reshape to the shape of last conv. layer in the encoder, so we can
x = Reshape((conv_shape[1], conv_shape[2], conv_shape[3]))(x)

# upscale (conv2D transpose) back to original shape
# use Conv2DTranspose to reverse the conv layers defined in the encoder
x = Conv2DTranspose(32, 3, padding = 'same', activation='relu', strides = (2, 2))(x)

#Can add more conv2DTranspose layers, if desired.
#Using sigmoid activation
x = Conv2DTranspose(1, 3, padding = 'same', activation='sigmoid', name = 'decoder_output')(x)

# Define and summarize decoder model
decoder = Model(decoder_input, x, name='decoder')
decoder.summary()
```

We used a custom loss function which is weighted sum of **Reconstruction loss and KL Divergence loss** with weights [1 -5e-4].

Loss Function

```python
class CustomLayer(keras.layers.Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)

        # Reconstruction loss (as we used sigmoid activation we can use binarycrossentropy)
        recon_loss = keras.metrics.binary_crossentropy(x, z_decoded)

        # KL divergence
        kl_loss = -5e-4 * K.mean(1 + z_sigma - K.square(z_mu) - K.exp(z_sigma), axis=-1)
        return K.mean(recon_loss + kl_loss)

    # add custom loss to the class
    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        return x

# apply the custom loss to the input images and the decoded latent distribution sample
y = CustomLayer()([input_img, z_decoded])
# y is basically the original image after encoding input img to mu, sigma, z
# and decoding sampled z values.
#This will be used as output for vae

# =================
# VAE
# =================
vae = Model(input_img, y, name='vae')

# Compile VAE
vae.compile(optimizer='adam', loss=None)
vae.summary()
```

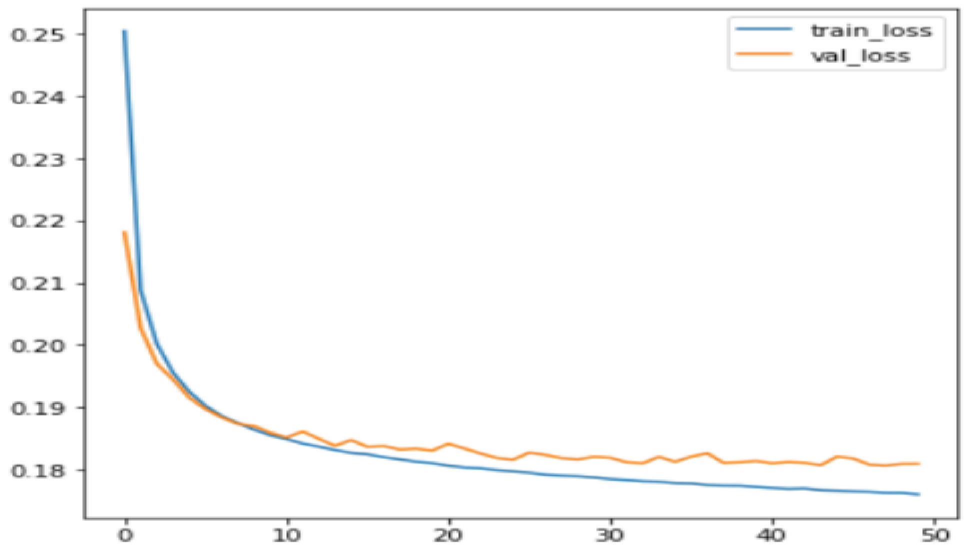
Complete Model Summary (including Encoder, Decoder, Loss Layer).



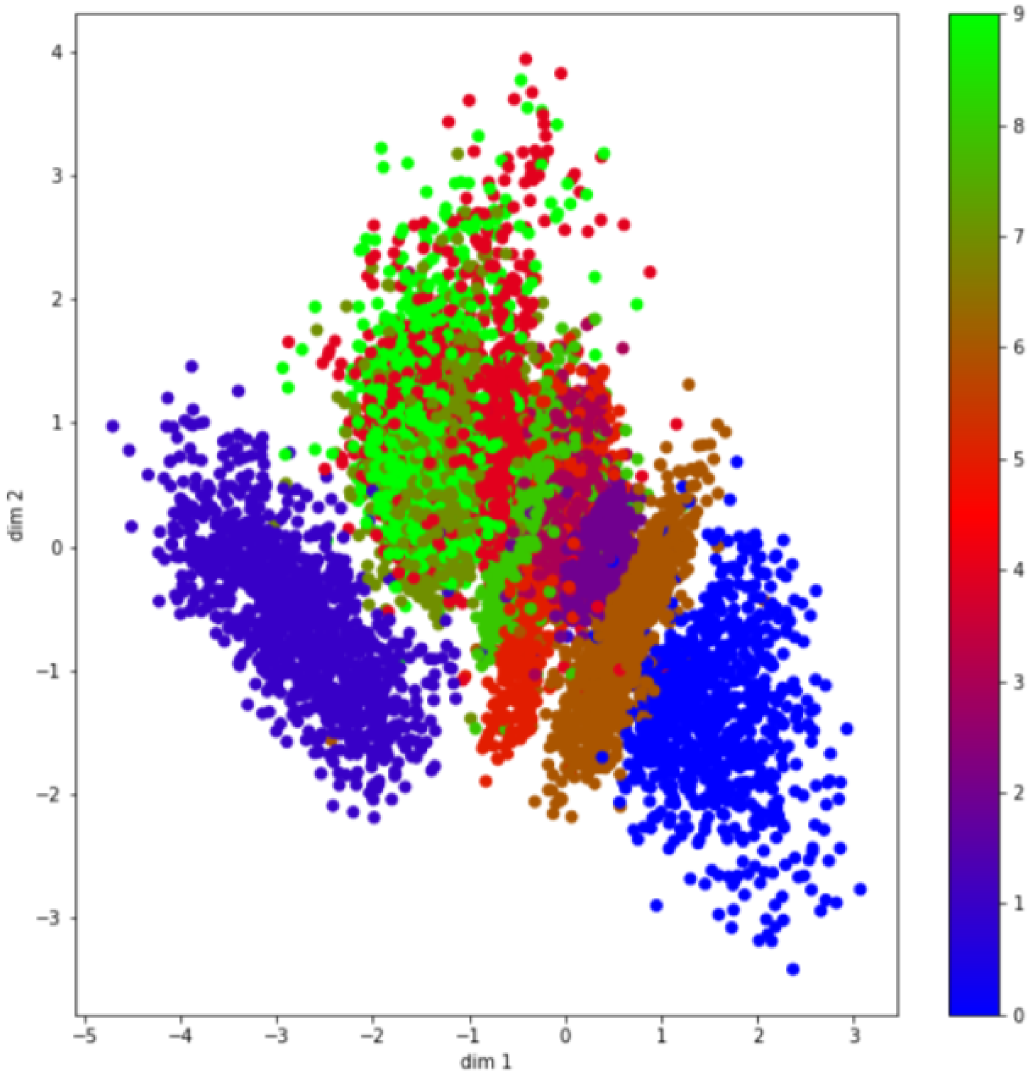
```
Model: "vae"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
encoder_input (InputLayer)      [(None, 28, 28, 1)]  0
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 28, 28, 32)   320         encoder_input[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 14, 14, 64)   18496       conv2d_8[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 14, 14, 64)   36928       conv2d_9[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 14, 14, 64)   36928       conv2d_10[0][0]
__________________________________________________________________________________________________
flatten_2 (Flatten)             (None, 12544)        0           conv2d_11[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                 (None, 32)           401440      flatten_2[0][0]
__________________________________________________________________________________________________
latent_mu (Dense)               (None, 2)            66          dense_3[0][0]
__________________________________________________________________________________________________
latent_log_sigma (Dense)        (None, 2)            66          dense_3[0][0]
__________________________________________________________________________________________________
z (Lambda)                      (None, 2)            0           latent_mu[0][0]
                                                                 latent_log_sigma[0][0]
__________________________________________________________________________________________________
decoder (Functional)            (None, 28, 28, 1)    56385       z[0][0]
__________________________________________________________________________________________________
custom_layer_3 (CustomLayer)    (None, 28, 28, 1)    0           encoder_input[0][0]
                                                                 decoder[1][0]
==================================================================================================
Total params: 550,629
Trainable params: 550,629
Non-trainable params: 0
__________________________________________________________________________________________________
```

Loss curves

We used Adam optimizer for training.



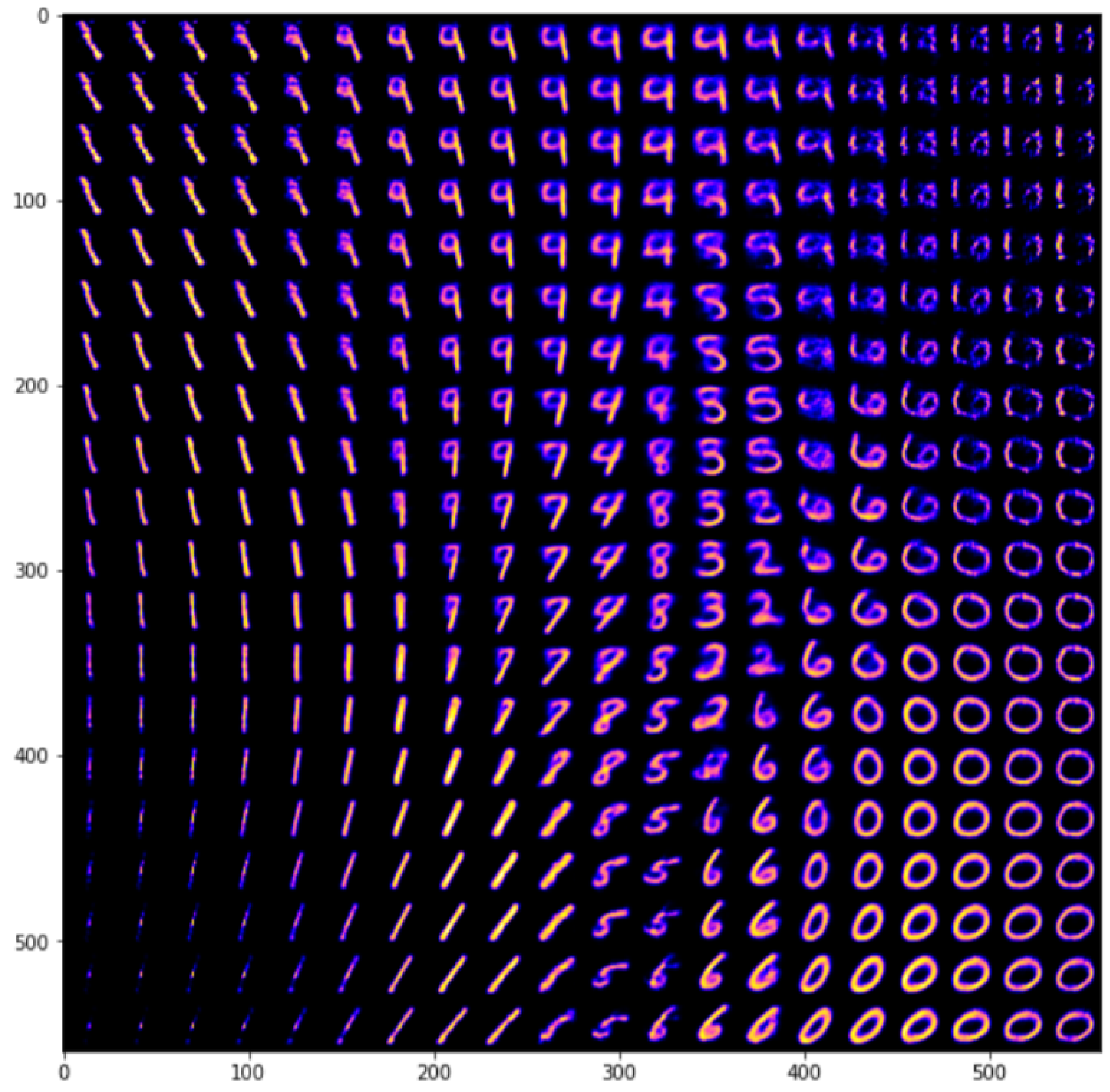
Spatial position of each number with respect to [Z1, Z2] 2d dimensional variables



We can see different numbers form different clusters in space.

Sample Output

The output axis contains Z1, Z2 and in the below image we can see the change in the image by changing each latent variables from range [0, 500].



4) Pixel RNN

We studied the paper **Pixel Recurrent Neural Networks** Aaron van den Oord, Nal Kalchbrenner, Koray Kavukcuoglu and referencing it built our custom model.

An effective approach to model such a network is to use probabilistic density models (like Gaussian or Normal distribution) to quantify the pixels of an image as a product of conditional distributions. This approach turns the modeling problem into a sequence problem wherein the next pixel value is determined by all the previously generated pixel values

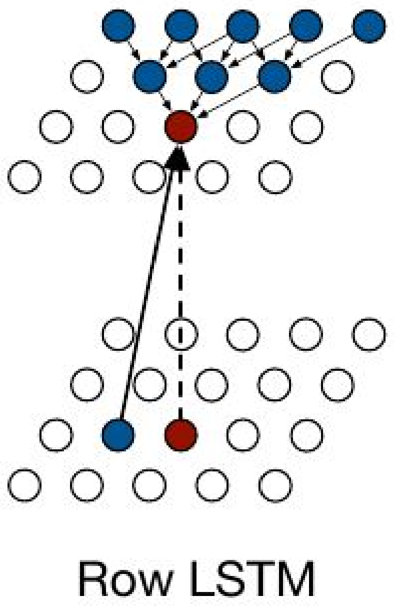
Model Architecture: There are four different architectures which can be used, namely :Row LSTM, Diagonal BiLSTM, a fully convolutional network and a Multi Scale network.The network consists of upto 12 layers of two dimensional LSTMs. Two types of LSTM layers that are used are,

1. **Row LSTM** : The first layer is a 7x7 convolution that uses the mask of type A. It is followed by the input to state layer which is a 3x1 convolution that uses mask of type B and a 3x1 state to state convolution layer which is not masked. The feature map is then passed through a couple of 1x1 convolution layers consisting of ReLU and of mask type B.
2. **Diagonal BiLSTM** : The only difference between its architecture and Row LSTM's is in the input to state and the state to state layers. It has a 1x1 convolution input to state layer with mask type B and a 1x2 convolution state to state layer without mask.

Row LSTM

Hidden state(i,j) = Hidden state(i-1,j-1)+ Hidden state(i-1,j+1)+ Hidden state(i-1,j)+ p(i,j)

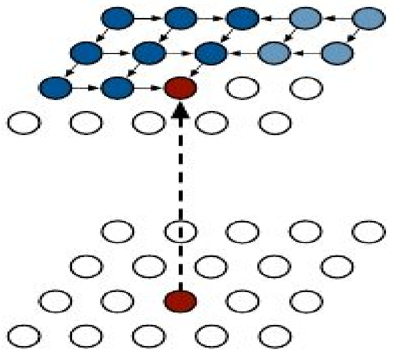
This processes the image row to row from top to bottom computing features of the whole row at once. It captures a fairly triangular region above the pixel. However it isn't able to capture the entire available region.



Row LSTM

Diagonal BiLSTM: pixel(i, j) = pixel(i, j-1) + pixel(i-1, j).

The receptive field of this layer encompasses the entire available region. The processing goes on diagonally. It starts from the top corner and reaches the opposite corner while moving in both directions.


Diagonal BiLSTM

Residual connections (or skip connections) are also used in these networks to increase convergence speed and propagate signals more directly through the network.

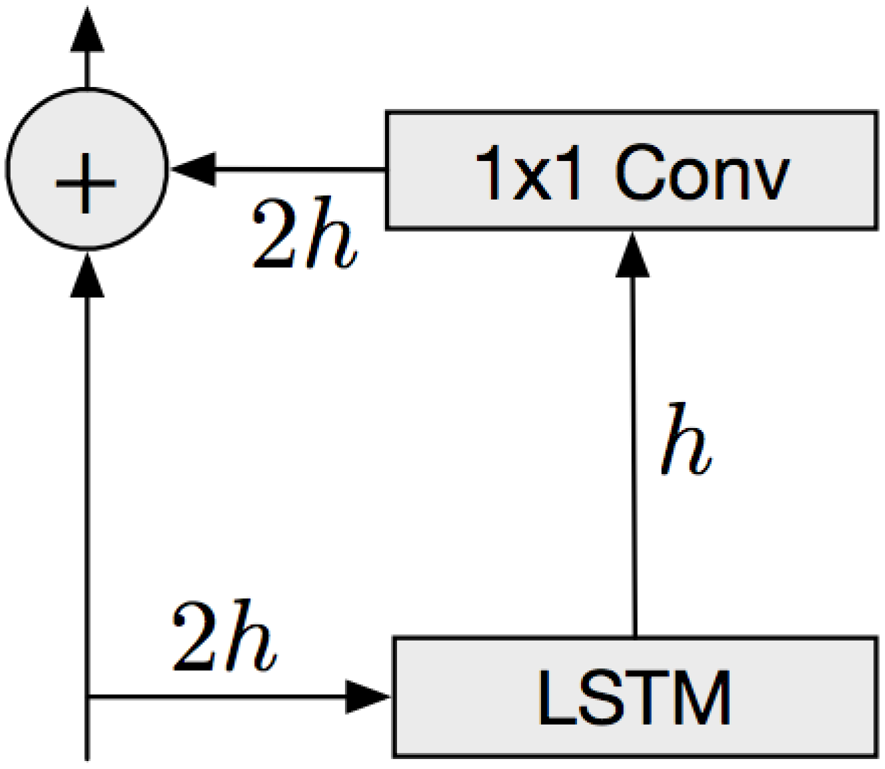


FIGURE 5 : Residual block for PixelRNNs. ‘h’ refers to the number of parameters.

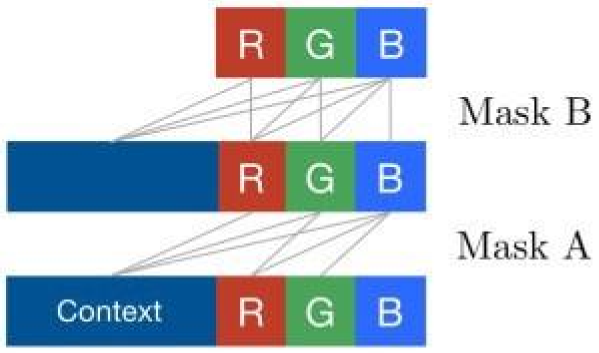
Masked convolutions :

The features for each input position in every layer are split into three parts each one corresponding to a color(RGB). For computing the values of G channel we need the value of the R channel along with values of all previous pixels. Similarly, B channel requires information of both R and G channels. To restrict the network to adhere to these constraints we apply masks to convolutions.

We use two types of masks :

1. **Type A** : this mask is only applied to the first convolutional layer and restricts connections to those colors in current pixels that have already been predicted.
2. **Type B** : this mask is applied to other layers and allows connections to predicted colors in the current pixels.

Masks are an important part of network which maintain the number of channels in the network.



Connectivity inside a masked convolution

Loss Function and Evaluation metric : Here, negative log likelihood (NLL) is used as the loss and evaluation metric as the network predicts(classifies) the values of pixel from values 0–255

Sample Output of PixelRNN

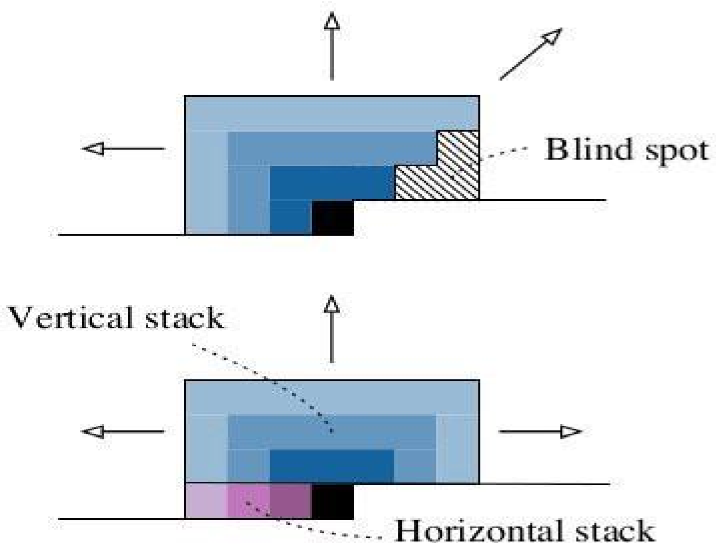
5) Pixel CNN

The main drawback of PixelRNN is that training is very slow as each state needs to be computed sequentially. This can be overcome by using convolutional layers and increasing the receptive field. PixelCNN uses standard convolutional layers to capture a bounded receptive field and compute features for all pixel positions at once. It uses multiple convolutional layers that preserve the spatial resolution. However, pooling layers are not used. Masks are adopted in the convolutions to restrict the model from violating the conditional dependence.

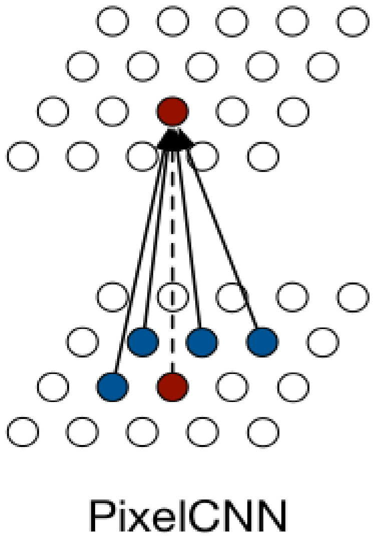
We referenced the paper **Conditional Image Generation with PixelCNN Decoders**

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu

PixelCNN lowers the training time considerably as compared to PixelRNN. However, the image generation is still sequential as each pixel needs to given back as input to the network to compute next pixel.The major drawback of PixelCNN is that it's performance is worse than PixelRNN. Another drawback is the presence of a Blind Spot in the receptive field. The capturing of receptive field by a CNN proceeds in a triangular fashion. It causes several pixels to be left out of the receptive field, as shown in the figure below. Since convolutional networks capture a bounded receptive field (unlike BiLSTM) and computes features for all pixels at once, these pixels aren't dependent on all previous pixels which is undesirable. The convolutional layers are unable to completely process the receptive fields thus leading to a slight miscalculation of pixel values. The pixels left out constitute the blind spot.



Blind spot in a PixelCNN and its solution in Gated PixelCNN



input-to-state and state-to-state mapping for PixelCNN

Sample Output Of Pixel CNN

Conclusion

Our Study explores various generative models and their performance on generating images of digits 0-9. We found Vanilla GAN to be better in the output than DCGAN and also DCGAN was inconsistent in training (producing good results in the midway of the training then at the end diverges). Ideally DCGAN should perform better but as MNIST is just 28*28 image vanilla GAN also performs well. DCGAN will perform well for high dimensional images and good hyperparameter tuning. VAE also performed well with nice sample images also we showed how images change with change in latent variable this shows we have good control in Latent variable. VAE images were good but lacked some sharpness. PIXEL RNN performed quite good and produced crisp and clean Outputs on the other hand PIXEL CNN produced very bad output images.

References

[1] **Generative Adversarial Networks**

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio

[2] **An Introduction to Variational Autoencoders**

Diederik P. Kingma, Max Welling

[3] **Conditional Image Generation with PixelCNN Decoders**

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu

[4] **Pixel Recurrent Neural Networks**

Aaron van den Oord, Nal Kalchbrenner, Koray Kavukcuoglu