Irfan Ahmed

ICP 8

Due Date: 10/19/2020 (But I asked for extension due to family emergency, which was approved.)

Explanation on adding more layers:

There are pros and cons to adding additional layers. The only way to know what will work best is to try it out. By adding more layers, I did not see a significant improvement in my output. In theory the more convolutional layer a model contains the slower it will be, but by adding more kernels per layer can help mitigate the slower process and using GPU to speed things up. After adding more layers one problem I encountered was my output for decoder did not match my input for encoder. One of the ways to fix this issue is by adjusting the strides variable. The stride argument was set to 2 but after playing around with random numbers for each layer in the decoder cell, I was able to match the input and output to 28 X 28 X 1. But by adding the additional layer and changing the stride value my output did not come out that great. The output was not that great which did not make too much of sense. Stride is often used for down sample the output channel. To fix this problem after adding additional layer I opted to using the cropping method to make sure my input and output matched so I could run train and compile my data.

a. What you learned in the ICP

One of the main take ways for this ICP was understanding how Autoencoders work. Autoencoders is a feedforward neural network where the input is the same as the output. The input is compressed into a lower dimensional code and then the output is reconstructed from this representation. To build the autoencoder model we need to define the encoding method, decoding method, and a loss function which compares the output with the target. The three components are encoder, code, and decoder. The encoder produces the code, the decoder reconstructs the input. One important thing to note about Autoencoders is that they are data specific, meaning they can only compress data like what Autoencoders have been trained on. Also, the output will not be the same as the input, rather a degraded representation. Lastly, it is an unsupervised method since we do not need explicit labels to train on. Variational Autoencoder is just an probabilistic approach.

b. ICP description what was the task you were performing

1. Imported the required libraries/packages and loaded data from MNIST dataset
2. Reduced the data size to avoid any colab errors. The training data set was from [0:10000] and the test [0:1000].
3. The data model configuration variables were initialized, such as img_width, img_heigth, no_epochs, batch size, validation_split, verbosity, latenet_dim and num_channels.
4. The data was reshaped based on the input train shape and input test shape along with the img height and width. The input train and test were Parsed numbers as floats and the test and train normalized.
5. The encoder was defined and total of Conv2D layers were created. After each layer it was normalized and at the end.
6. Then the shape of the final Conv2D output was retrieved.
7. Reparameterization function created
8. The reparameterization function was passed into Lambda. This is to ensure that the correct gradient is computed during the backwards pass based on the values for mu and sigma.
9. Performed Encoder instantiation (taking inputs through the input layer).
10. The decoder was defined with total of 4 Conv2DTransposer layers.
11. Performed instantiate decoder. This takes the inputs from the decoder input layer and outputs what the output will by the output layer.
12. Created VAE.
13. Compiled and trained the data.
14. Viewed the outputs as scatter plot and the images.

c. Challenges that you faced

One of the difficulties I faced was figuring out how to add more layers. After figuring that out I encountered another layer which was incompatible shapes. What that meant was my input and output shapes did not match. But after trial and error I found out that I would need to adjust the strides for my decoder. But that did not give me the best result. I ended up using cropping method to obtain the best results.

d. Screen shots that shows the successful execution of each required step of your code

Importing the requires libraires

```
import keras
import tensorflow as tf
from keras.layers import Conv2D, Conv2DTranspose, Input, Flatten, Dense, Lambda, Reshape
from keras.layers import BatchNormalization
from keras.models import Model
from keras.datasets import mnist
from keras.losses import binary_crossentropy
from keras import backend as K
import numpy as np
import matplotlib.pyplot as plt
```

MNIST dataset will be utilized for training the autoencoder. The dataset consistes of 28 X 28 pixel images. The goal is for our autoencoder to learn the distribution of the written digits across 2D latent space.

```
[2] #load MNIST dataset
    (input_train_1, target_train_1), (input_test_1, target_test_1) = mnist.load_data()
```

Looking at the data

```
[3] print(input_train_1.shape)
    print(input_test_1.shape)
    print(target_train_1.shape)
    print(target_test_1.shape)
```

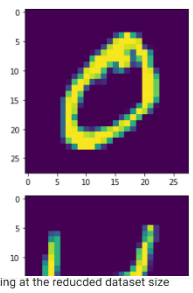
```
(60000, 28, 28)
(10000, 28, 28)
(60000,)
(10000,)
```

Reducing the data size

```
input_train=input_train_1[0:10000]
target_train=target_train_1[0:10000]
input_test=input_test_1[0:1000]
target_test=target_test_1[0:1000]
```

Looking at the first 5 images

```
[5] for i in range(10):
      pr_img = input_train[i]
      pr_img = np.array(pr_img, dtype='float')
      pixels = pr_img.reshape(28,28)
      plt.imshow(pixels)
      plt.show()
```



Looking at the reducded dataset size

```
print(input_train.shape)
print(input_test.shape)
print(target_train.shape)
print(target_test.shape)
```

```
(10000, 28, 28)
(1000, 28, 28)
(10000,)
(1000,)
```

```
[7] print(input_train.shape[1])
```

```
28
```

```
[8] print(input_train.shape[2])
```

```
28
```

Model configuration: setting configuration parameters for data and model. The width is obtained from the training data. The height and width are 28, and the batch size is set to 128 samples per batch. verbosity is set to true (be means of 1). This allows the output to display everything on the screen.

```
[9] #configuration of data and model
    img_width, img_height = input_train.shape[1], input_train.shape[2]
    batch_size = 128
    no_epochs = 100
    validation_split = 0.2
    verbosity = 1
    latent_dim = 2
    num_channels = 1
```

```
[10] print(input_train.shape[0])

     10000
```

The data is reshaped to refelct the traning or test data. (X, 28,28,1), where X is the number of smaples in trainning or test

```
#reshape data
input_train = input_train.reshape(input_train.shape[0], img_height, img_width, num_channels)
input_test = input_test.reshape(input_test.shape[0], img_height, img_width, num_channels)
input_shape = (img_height, img_width, num_channels)

#parse numbers as floats
input_train = input_train.astype('float32')
input_test = input_test.astype('float32')


#normalize data
input_train = input_train / 255
input_test = input_test / 255
```

Encoder seup

```
# Encoder Definition
i       = Input(shape=input_shape, name='encoder_input')
# layer 1
cx      = Conv2D(filters=8, kernel_size=3, strides=2, padding='same', activation='relu')(i)
cx      = BatchNormalization()(cx)
#layer 2
cx      = Conv2D(filters=16, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
cx      = BatchNormalization()(cx)
#layer 3
cx      = Conv2D(filters=16, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
cx      = BatchNormalization()(cx)
#layer 4
cx      = Conv2D(filters=16, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
cx      = BatchNormalization()(cx)



x       = Flatten()(cx)
x       = Dense(20, activation='relu')(x)
x       = BatchNormalization()(x)
mu      = Dense(latent_dim, name='latent_mu')(x)
sigma   = Dense(latent_dim, name='latent_sigma')(x)
```

We can see that first layer is the input layer, which takes in the input_shape = (282, 28,1) and name = 'enocder_input'

Then the 2D convolutional layer, or Connv2D. This will learn 8 filters by deploying 3 x 3 kernel. Uses padding 'same' and ReLU activation.

Batch Normaliztion is applied. This ensures that the outputs of the Conv2D layer that are input to the next Conv2D layer have a steady mean and variance.

A Conv2D layer is added again. It learns 16 filters and the rest is equal to the first Conv2D layer

Bath is normalized again

The Flatten layer is added, it purpose it flatten the multi-dimensional data from the convolutional layer into one dimensional shape.

The Flatten layer is added, it purpose it flatten the multi-dimensional data from the convolutional layer into one dimensional shape.

The Dense layer is added with 20 output neurons.

BatchNormaliztino is added again

The final two layers mu and sigma. The first output means value u of the encoded input and the second one outputs the stddevs.

Will need to retrieve the shape of the final Conv2D output

```
[13] #get shape for Conv2D
     conv_shape = K.int_shape(cx)
```

```
[14] #Define sampling with reparametirzaton
     def sample_z(args):
       mu, sigma = args
       batch     = K.shape(mu)[0]
       dim       = K.int_shape(mu)[1]
       eps       = K.random_normal(shape=(batch, dim))
       return mu + K.exp(sigma / 2) * eps
```

This is to enusre correct gradients are computed during the backwards pass based on our values for mu and sigma

```
[15] z       = Lambda(sample_z, output_shape=(latent_dim, ), name='z')([mu, sigma])
```

```
[16] print(z.shape)

     (None, 2)
```

Encoder instantiation, this will take inputs through the input layer i and outputting values genereated by mu, sigma and z layers

```
#instantiate encoder
encoder = Model(i, [mu, sigma, z], name='encoder')
encoder.summary()
```

```
Model: "encoder"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
encoder_input (InputLayer)      [(None, 28, 28, 1)]  0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 14, 14, 8)    80          encoder_input[0][0]
__________________________________________________________________________________________________
batch_normalization (BatchNorma (None, 14, 14, 8)    32          conv2d[0][0]
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 7, 7, 16)     1168        batch_normalization[0][0]
__________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 7, 7, 16)     64          conv2d_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 4, 4, 16)     2320        batch_normalization_1[0][0]
__________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 4, 4, 16)     64          conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 2, 2, 16)     2320        batch_normalization_2[0][0]
__________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 2, 2, 16)     64          conv2d_3[0][0]
__________________________________________________________________________________________________
flatten (Flatten)               (None, 64)           0           batch_normalization_3[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 20)           1300        flatten[0][0]
__________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 20)           80          dense[0][0]
__________________________________________________________________________________________________
latent_mu (Dense)               (None, 2)            42          batch_normalization_4[0][0]
__________________________________________________________________________________________________
latent_sigma (Dense)            (None, 2)            42          batch_normalization_4[0][0]
__________________________________________________________________________________________________
z (Lambda)                      (None, 2)            0           latent_mu[0][0]
                                                                 latent_sigma[0][0]
==================================================================================================
Total params: 7,576
Trainable params: 7,424
Non-trainable params: 152
__________________________________________________________________________________________________
```

Creating decoder

Decoder begins with an input layer the decoder_layer and taks input witht the shape of (latent_dim). Conv2DTranspose layer is to unsample the point in latent space adding BatchNormaliztion and conv2dTranspose. Still have 8 filters with shape of (28,28,8) Final step Conv2DTranspose layer which does nothing to the width and height of the data, but makes sure that the numbber of filters learns equals num_channels

```
# Decoder Definition
d_i   = Input(shape=(latent_dim, ), name='decoder_input')

x     = Dense(conv_shape[1] * conv_shape[2] * conv_shape[3], activation='relu')(d_i)

x     = BatchNormalization()(x)

x     = Reshape((conv_shape[1], conv_shape[2], conv_shape[3]))(x)
#layer 1
cx    = Conv2DTranspose(filters=64, kernel_size=3, strides=2, padding='same', activation='relu')(x)
cx    = BatchNormalization()(cx)
#layer 2
cx    = Conv2DTranspose(filters=32, kernel_size=3, strides=2, padding='same', activation='relu')(cx)
cx    = BatchNormalization()(cx)
cx    = tf.keras.layers.Cropping2D(cropping=((0,1),(0,1)))(cx)
#layer 3
cx    = Conv2DTranspose(filters=16, kernel_size=3, strides=2, padding='same',  activation='relu')(cx)
cx    = BatchNormalization()(cx)
#layer 4
cx    = Conv2DTranspose(filters=8, kernel_size=3, strides=2, padding='same',  activation='relu')(cx)
cx    = BatchNormalization()(cx)

o     = Conv2DTranspose(filters=num_channels, kernel_size=3, activation='sigmoid', padding='same', name='decoder_output')(cx)
```

```
# Instantiate decoder
decoder = Model(d_i, o, name='decoder')
decoder.summary()
```

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   [(None, 2)]               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                192
_________________________________________________________________
batch_normalization_5 (Batch (None, 64)                256
_________________________________________________________________
reshape (Reshape)            (None, 2, 2, 16)          0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 4, 4, 64)          9280
_________________________________________________________________
batch_normalization_6 (Batch (None, 4, 4, 64)          256
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 8, 8, 32)          18464
_________________________________________________________________
batch_normalization_7 (Batch (None, 8, 8, 32)          128
_________________________________________________________________
cropping2d (Cropping2D)      (None, 7, 7, 32)          0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 14, 14, 16)        4624
_________________________________________________________________
batch_normalization_8 (Batch (None, 14, 14, 16)        64
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 28, 28, 8)         1160
_________________________________________________________________
batch_normalization_9 (Batch (None, 28, 28, 8)         32
_________________________________________________________________
decoder_output (Conv2DTransp (None, 28, 28, 1)         73
=================================================================
Total params: 34,529
Trainable params: 34,161
Non-trainable params: 368
_________________________________________________________________
```

```
# Instantiate VAE
vae_outputs = decoder(encoder(i)[2])
vae         = Model(i, vae_outputs, name='vae')
vae.summary()
```

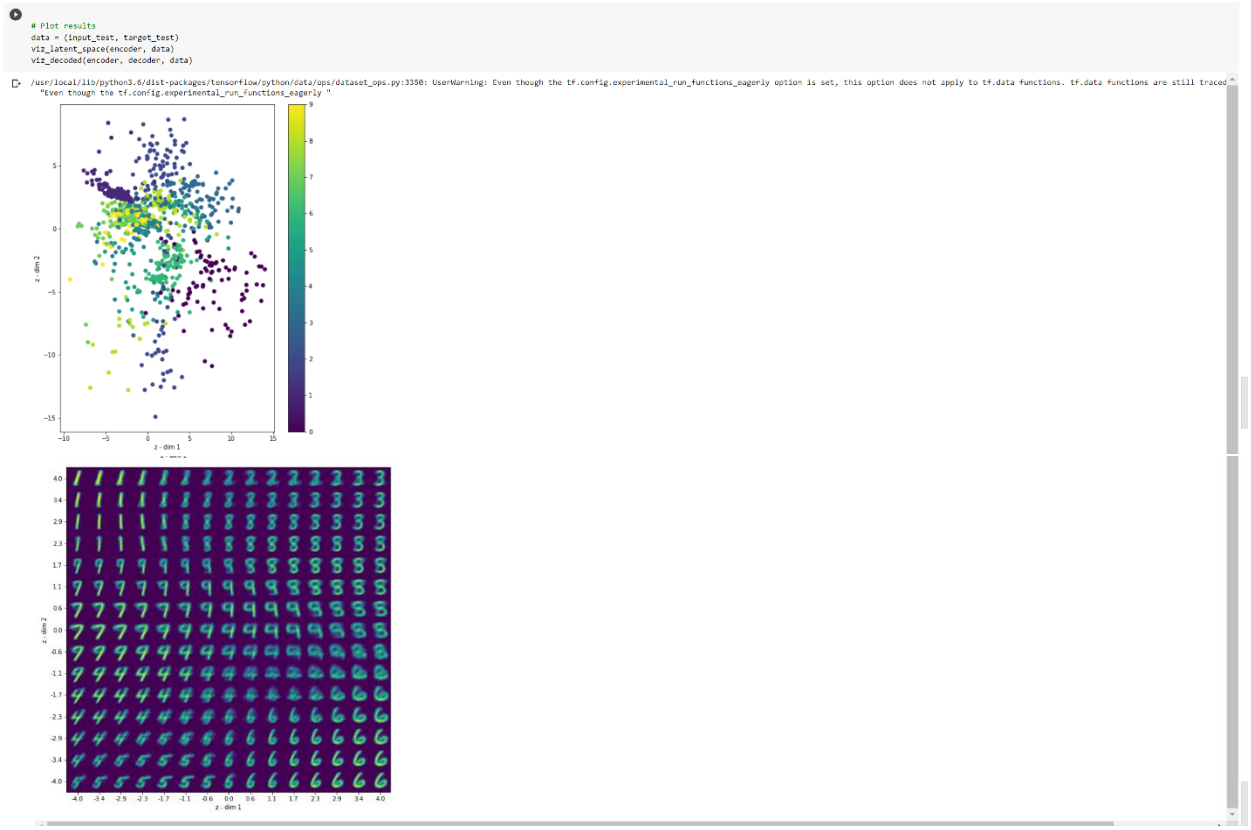
```
Model: "vae"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   [(None, 28, 28, 1)]       0
_________________________________________________________________
encoder (Functional)         [(None, 2), (None, 2), (N 7576
_________________________________________________________________
decoder (Functional)         (None, 28, 28, 1)         34529
=================================================================
Total params: 42,105
Trainable params: 41,585
Non-trainable params: 520
_________________________________________________________________
```

```
tf.config.run_functions_eagerly(True)
# Compile VAE
vae.compile(optimizer='adam', loss='binary_crossentropy')

# Train autoencoder
vae.fit(input_train, input_train, epochs =no_epochs, batch_size = batch_size, validation_split = validation_split)
```

```
Epoch 1/100
  1/63 [..............................] - ETA: 0s - loss: 0.2072/usr/local/lib/python3.6/dist-packages/tensorflow/python/data/ops/dataset_ops.py:3350: UserWarning: Even though the tf.config.experimental_run_functions_eagerly option is set, this option does not
  "Even though the tf.config.experimental_run_functions_eagerly "
63/63 [==============================] - 3s 48ms/step - loss: 0.2129 - val_loss: 0.2169
Epoch 2/100
63/63 [==============================] - 3s 48ms/step - loss: 0.2076 - val_loss: 0.2139
Epoch 3/100
63/63 [==============================] - 3s 47ms/step - loss: 0.2043 - val_loss: 0.2086
Epoch 4/100
63/63 [==============================] - 3s 47ms/step - loss: 0.2028 - val_loss: 0.2072
Epoch 5/100
63/63 [==============================] - 3s 47ms/step - loss: 0.2005 - val_loss: 0.2039
Epoch 6/100
63/63 [==============================] - 3s 46ms/step - loss: 0.1990 - val_loss: 0.2015
Epoch 7/100
63/63 [==============================] - 3s 48ms/step - loss: 0.1976 - val_loss: 0.2051
Epoch 8/100
63/63 [==============================] - 3s 49ms/step - loss: 0.1970 - val_loss: 0.2019
Epoch 9/100
63/63 [==============================] - 3s 48ms/step - loss: 0.1955 - val_loss: 0.1989
Epoch 10/100
63/63 [==============================] - 3s 47ms/step - loss: 0.1944 - val_loss: 0.1997
Epoch 11/100
63/63 [==============================] - 3s 47ms/step - loss: 0.1930 - val_loss: 0.1969
Epoch 12/100
63/63 [==============================] - 3s 47ms/step - loss: 0.1922 - val_loss: 0.1965
Epoch 13/100
63/63 [==============================] - 3s 47ms/step - loss: 0.1922 - val_loss: 0.1969
Epoch 14/100
63/63 [==============================] - 3s 47ms/step - loss: 0.1909 - val_loss: 0.1974
Epoch 15/100
63/63 [==============================] - 3s 47ms/step - loss: 0.1906 - val_loss: 0.1953
Epoch 16/100
63/63 [==============================] - 3s 48ms/step - loss: 0.1897 - val_loss: 0.1938
Epoch 17/100
63/63 [==============================] - 3s 47ms/step - loss: 0.1893 - val_loss: 0.1946
Epoch 18/100
63/63 [==============================] - 3s 48ms/step - loss: 0.1889 - val_loss: 0.1951
```

```
# Results visualization
# Credits for original visualization code: https://keras.io/examples/variational_autoencoder_deconv/


def viz_latent_space(encoder, data):
  input_data, target_data = data
  mu, _, _ = encoder.predict(input_data)
  plt.figure(figsize=(8, 10))
  plt.scatter(mu[:, 0], mu[:, 1], c=target_data)
  plt.xlabel('z - dim 1')
  plt.ylabel('z - dim 2')
  plt.colorbar()
  plt.show()
```

```
[27] def viz_decoded(encoder, decoder, data):
      num_samples = 15
      figure = np.zeros((img_width * num_samples, img_height * num_samples, num_channels))
      grid_x = np.linspace(-4, 4, num_samples)
      grid_y = np.linspace(-4, 4, num_samples)[::-1]
      for i, yi in enumerate(grid_y):
          for j, xi in enumerate(grid_x):
              z_sample = np.array([[xi, yi]])
              x_decoded = decoder.predict(z_sample)
              digit = x_decoded[0].reshape(img_width, img_height, num_channels)
              figure[i * img_width: (i + 1) * img_width,
                     j * img_height: (j + 1) * img_height] = digit
      plt.figure(figsize=(10, 10))
      start_range = img_width // 2
      end_range = num_samples * img_width + start_range + 1
      pixel_range = np.arange(start_range, end_range, img_width)
      sample_range_x = np.round(grid_x, 1)
      sample_range_y = np.round(grid_y, 1)
      plt.xticks(pixel_range, sample_range_x)
      plt.yticks(pixel_range, sample_range_y)
      plt.xlabel('z - dim 1')
      plt.ylabel('z - dim 2')
      # matplotlib.pyplot.imshow() needs a 2D array, or a 3D array with the third dimension being of shape 3 or 4!
      # So reshape if necessary
      fig_shape = np.shape(figure)
      if fig_shape[2] == 1:
        figure = figure.reshape((fig_shape[0], fig_shape[1]))
      # Show image
      plt.imshow(figure)
      plt.show()
```



e. Output file link if applicable

https://github.com/UMKC-APL-BigDataAnalytics/icp8-irfancheemaa

f. Video link (YouTube or any other publicly available video platform)

https://umkc.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=8583ae36-141d-4cb9-bc59-ac5a00572ee7

g. Any inside about the data or the ICP in general

Had hard time explain how the output was affected by adding more layers to the encoder and decoder. Overall, enjoyed the ICP.