VAE

Variational autoencoder [1] models inherit autoencoder architecture, but use variational approach homework, we will implement VAE and quantitatively measure the quality of the generated sample

[1] Auto-Encoding Variational Bayes, Diederik P Kingma, Max Welling 2013 https://arxiv.org/abs/13

[2] Improved techniques for training gans, Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Rad Neural Information Processing Systems

[3] A note on inception score, Shane Barratt, Rishi Sharma 2018 https://arxiv.org/abs/1801.01973

PART I. Train a good VAE model

Setup

```
import tensorflow as tf
if tf.__version__ < '2.0.0':
    tf.enable_eager_execution()
tf.executing_eagerly()

import numpy as np
import os

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# A bunch of utility functions

def show_images(images):
    # images reshape to (batch_size, D)
    images = np.reshape(images, [images.shape[0], -1])
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
    sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
```

```
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(img.reshape([sqrtimg,sqrtimg]))
    return

def preprocess_img(x):
    return 2 * x - 1.0

def rel_error(x,y):
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def count_params(model):
    """Count the number of parameters in the current TensorFlow graph """
    param_count = np.sum([np.prod(p.shape) for p in model.weights])
    return param_count
```

Dataset

We will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each p digit on black background (0 through 9). This was one of the first datasets used to train convolutic standard CNN model can easily exceed 99% accuracy.

Heads-up: Our MNIST wrapper returns images as vectors. That is, they're size (batch, 784). If you \ resize them to (batch,28,28) or (batch,28,28,1). They are also type np.float32 and bounded [0,1].

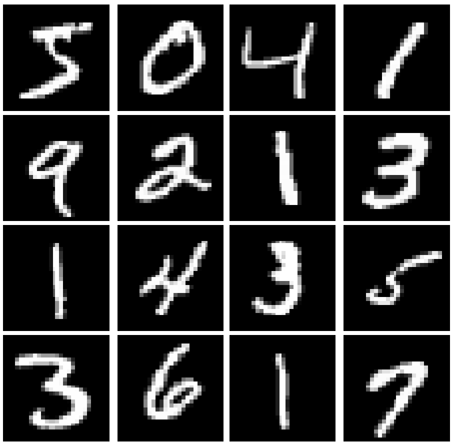
```
class MNIST(object):
    def __init__(self, batch_size, shuffle=False):
        """
        Construct an iterator object over the MNIST data

        Inputs:
        - batch_size: Integer giving number of elements per minibatch
        - shuffle: (optional) Boolean, whether to shuffle the data on each epoch
        """
        train, _ = tf.keras.datasets.mnist.load_data()
        X, y = train
        X = X.astype(np.float32)/255
        X = X.reshape((X.shape[0], -1))
        self.X, self.y = X, y
        self.batch_size, self.shuffle = batch_size, shuffle

    def __iter__(self):
        N, B = self.X.shape[0], self.batch_size
        idxs = np.arange(N)
        if self.shuffle:
            np.random.shuffle(idxs)
        return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))

# show a batch
mnist = MNIST(batch_size=16)
show_images(mnist.X[:16])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```



```
X_DIM = mnist.X[0].size
num_samples = 100000
num_to_show = 100

# Hyperparamters. Your job to find these.
# TODO:
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
num_epochs = 100
batch_size = 100
Z_DIM = 5
learning_rate = 5e-4
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Encoder

Our first step is to build a variational encoder network $q_\phi(z \mid x)$.

Hint: You should use the layers in `tf.keras.layers` to build the model. Use four FC layers. All fully c
For initialization, just use the default initializer used by the `tf.keras.layers` functions.

The output of the encoder should thus have shape `[batch_size, 2*z_dim]`, and contain real number
diagonal log variance $\log \sigma(x_i)^2$ of each of the `batch_size` input images. Note, we want to make i
stability.

WARNING: Do not apply any non-linearity to the last activation.

```
def q_phi(z_dim=Z_DIM, x_dim=X_DIM):
  model = tf.keras.models.Sequential([
    # TODO: implement architecture
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    tf.keras.layers.Dense(392, activation="relu", use_bias=True, input_shape=(x_dim,)),
    tf.keras.layers.Dense(196, activation="relu", use_bias=True),
    tf.keras.layers.Dense(128, activation="tanh", use_bias=True),
    tf.keras.layers.Dense(2 * z_dim,  use_bias=True)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  ])
  return model
```

```
# TODO: implement reparameterization trick
def sample_z(mu, log_var):
  # Your code here for the reparameterization trick.
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  samples = None
  #print(mu.shape)
  z = tf.random.normal(tf.shape(mu))
  s = tf.math.exp(0.5 * log_var)
  samples = mu + s * z
  return samples
  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

Decoder

Now to build a decoder network $p_\theta(x \mid z)$. You should use the layers in `tf.keras.layers` to constru connected layers should include bias terms. Note that you can use the tf.nn module to access acti initializers for parameters.

In this exercise, we will use Bernoulli MLP decoder where $p_\theta(x \mid z)$ is modeled with multivariate B Gaussian distribution we discussed in the lecture, as following (see Appendix C.1 in the original pa

$\log p(x \mid z) = \sum_{i=1} x_i \log z_i + (1 - x_i) \log(1 - z_i)$

Note, the output of the decoder should have shape `[batch_size, x_dim]` and should output the unn

WARNING: Do not apply any non-linearity to the last activation.

```
def p_theta(z_dim=Z_DIM, x_dim=X_DIM):
  model = tf.keras.models.Sequential([
    # TODO: implement architecture
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    tf.keras.layers.Dense(128, activation="tanh", use_bias=True, input_shape=(z_dim,)),
    tf.keras.layers.Dense(196, activation="relu", use_bias=True),
    tf.keras.layers.Dense(392, activation="relu", use_bias=True),
    tf.keras.layers.Dense(x_dim, use_bias=True)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
  ])
  return model
```

Loss definition

Compute the VAE loss.

1. For the reconstruction loss, you might find `tf.nn.sigmoid_cross_entropy_with_logits` or `tf.kera`
2. For the kl loss, we discussed the closed form kl divergence between two gaussians in the lec

```
def vae_loss(x, x_logit, z_mu, z_logvar):
  recon_loss = None
  kl_loss = None # negative value
```

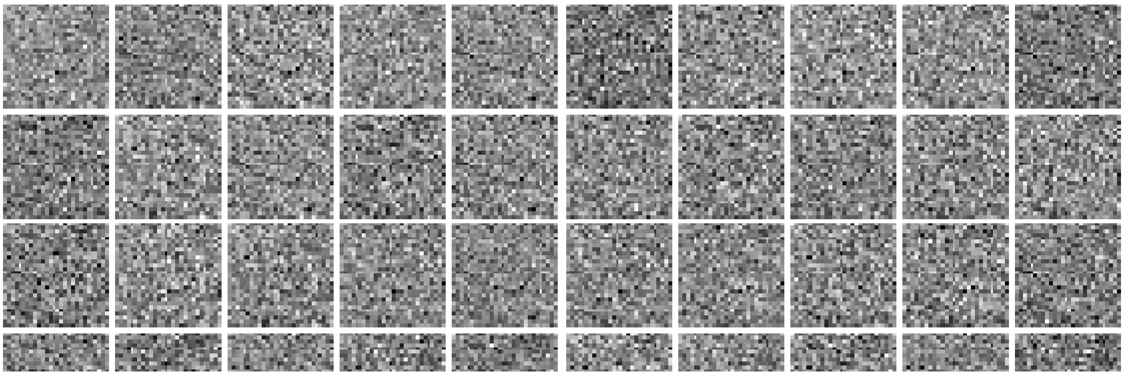
```
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #print(x, x_logit)
    #bce = tf.keras.losses.BinaryCrossentropy(from_logits=True, reduction=tf.keras.losses.Reduction.N
    recon_loss = tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(x, x_logit), axis=1)
    #entropy = bce(x, x_logit)
    #recon_loss = tf.reduce_sum(entropy)
    #print(entropy.shape, recon_loss.shape)
    temp =  1 + z_logvar - tf.square(z_mu) - tf.math.exp(z_logvar)
    kl_loss = -0.5 * (tf.reduce_sum(temp, axis = 1))
    #print(kl_loss, z_mu, z_logvar)
    #print(recon_loss, kl_loss)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #print(kl_loss)
    vae_loss = tf.reduce_mean(recon_loss + kl_loss)
    return vae_loss, tf.reduce_mean(recon_loss)
```

Optimizing our loss

```
Q = q_phi()
P = p_theta()
solver = tf.keras.optimizers.Adam(learning_rate)
mnist = MNIST(batch_size=batch_size, shuffle=True)
```

Visualize generated samples before training

```
z_gen = tf.random.normal(shape=[num_to_show, Z_DIM])
x_gen = P(z_gen)
imgs_numpy = tf.nn.sigmoid(x_gen).numpy()
show_images(imgs_numpy)
plt.show()
```



Training a VAE!

If everything works, your batch average reconstruction loss should drop below 95.

```
iter_count = 0
show_every = 400
for epoch in range(num_epochs):
  for (x_i, _) in mnist:
    with tf.GradientTape() as tape:
      z_concat = Q(preprocess_img(x_i))
      z_mu, z_logvar = tf.split(z_concat, num_or_size_splits=2, axis=1)
      z_i = sample_z(z_mu, z_logvar)

      x_logit = P(z_i)
      loss, recon_loss = vae_loss(x_i, x_logit, z_mu, z_logvar)

      grads = tape.gradient(loss,
                [Q.trainable_variables, P.trainable_variables])

      solver.apply_gradients(zip([*grads[0],*grads[1]],
                [*Q.trainable_variables, *P.trainable_variables]))

      if (iter_count % show_every == 0):
        print('Epoch: {}, Iter: {}, Loss: {:.4}, Recon: {:.4}'.format(
            epoch, iter_count, loss, recon_loss))
        #imgs_numpy = tf.nn.sigmoid(x_logit).numpy()
        #show_images(imgs_numpy[0:16])
        #plt.show()
      iter_count += 1
```

```
Epoch: 0, Iter: 0, Loss: 546.7, Recon: 544.4
Epoch: 0, Iter: 400, Loss: 143.0, Recon: 133.9
Epoch: 1, Iter: 800, Loss: 146.4, Recon: 136.4
Epoch: 2, Iter: 1200, Loss: 129.1, Recon: 118.9
Epoch: 2, Iter: 1600, Loss: 127.4, Recon: 117.2
Epoch: 3, Iter: 2000, Loss: 136.3, Recon: 125.4
Epoch: 4, Iter: 2400, Loss: 123.0, Recon: 112.2
Epoch: 4, Iter: 2800, Loss: 122.5, Recon: 111.7
Epoch: 5, Iter: 3200, Loss: 131.5, Recon: 120.2
Epoch: 6, Iter: 3600, Loss: 120.8, Recon: 109.6
Epoch: 6, Iter: 4000, Loss: 120.3, Recon: 109.4
Epoch: 7, Iter: 4400, Loss: 127.7, Recon: 116.2
Epoch: 8, Iter: 4800, Loss: 117.1, Recon: 105.7
Epoch: 8, Iter: 5200, Loss: 118.6, Recon: 107.3
Epoch: 9, Iter: 5600, Loss: 124.6, Recon: 113.0
Epoch: 10, Iter: 6000, Loss: 115.2, Recon: 103.4
Epoch: 10, Iter: 6400, Loss: 116.0, Recon: 104.6
Epoch: 11, Iter: 6800, Loss: 123.6, Recon: 111.9
Epoch: 12, Iter: 7200, Loss: 114.6, Recon: 102.7
Epoch: 12, Iter: 7600, Loss: 114.9, Recon: 103.3
Epoch: 13, Iter: 8000, Loss: 120.5, Recon: 108.5
Epoch: 14, Iter: 8400, Loss: 113.0, Recon: 101.0
Epoch: 14, Iter: 8800, Loss: 114.3, Recon: 102.6
Epoch: 15, Iter: 9200, Loss: 119.7, Recon: 107.7
Epoch: 16, Iter: 9600, Loss: 112.2, Recon: 100.2
Epoch: 16, Iter: 10000, Loss: 113.0, Recon: 101.1
Epoch: 17, Iter: 10400, Loss: 119.1, Recon: 107.1
Epoch: 18, Iter: 10800, Loss: 110.1, Recon: 98.11
Epoch: 18, Iter: 11200, Loss: 112.6, Recon: 100.7
Epoch: 19, Iter: 11600, Loss: 117.2, Recon: 105.0
Epoch: 20, Iter: 12000, Loss: 110.3, Recon: 98.27
Epoch: 20, Iter: 12400, Loss: 111.5, Recon: 99.46
Epoch: 21, Iter: 12800, Loss: 116.9, Recon: 104.6
Epoch: 22, Iter: 13200, Loss: 110.1, Recon: 97.97
Epoch: 22, Iter: 13600, Loss: 111.6, Recon: 99.47
Epoch: 23, Iter: 14000, Loss: 116.0, Recon: 103.7
Epoch: 24, Iter: 14400, Loss: 109.2, Recon: 96.86
Epoch: 24, Iter: 14800, Loss: 110.4, Recon: 98.39
Epoch: 25, Iter: 15200, Loss: 115.6, Recon: 103.2
Epoch: 26, Iter: 15600, Loss: 109.0, Recon: 96.77
Epoch: 26, Iter: 16000, Loss: 109.9, Recon: 97.56
Epoch: 27, Iter: 16400, Loss: 114.8, Recon: 102.4
Epoch: 28, Iter: 16800, Loss: 108.7, Recon: 96.25
Epoch: 28, Iter: 17200, Loss: 109.8, Recon: 97.72
Epoch: 29, Iter: 17600, Loss: 114.0, Recon: 101.6
Epoch: 30, Iter: 18000, Loss: 109.1, Recon: 96.74
Epoch: 30, Iter: 18400, Loss: 109.2, Recon: 96.76
Epoch: 31, Iter: 18800, Loss: 113.6, Recon: 101.0
Epoch: 32, Iter: 19200, Loss: 108.4, Recon: 95.97
Epoch: 32, Iter: 19600, Loss: 109.1, Recon: 96.91
Epoch: 33, Iter: 20000, Loss: 112.9, Recon: 100.2
Epoch: 34, Iter: 20400, Loss: 107.8, Recon: 95.33
Epoch: 34, Iter: 20800, Loss: 109.0, Recon: 96.74
Epoch: 35, Iter: 21200, Loss: 112.0, Recon: 99.29
Epoch: 36, Iter: 21600, Loss: 107.4, Recon: 94.9
Epoch: 36, Iter: 22000, Loss: 108.5, Recon: 96.07
Epoch: 37, Iter: 22400, Loss: 113.3, Recon: 100.6
Epoch: 38, Iter: 22800, Loss: 107.2, Recon: 94.72
```

```
Epoch: 38, Iter: 23200, Loss: 109.2, Recon: 96.74
Epoch: 39, Iter: 23600, Loss: 113.0, Recon: 100.3
Epoch: 40, Iter: 24000, Loss: 107.7, Recon: 95.02
Epoch: 40, Iter: 24400, Loss: 108.0, Recon: 95.67
Epoch: 41, Iter: 24800, Loss: 112.1, Recon: 99.38
Epoch: 42, Iter: 25200, Loss: 106.8, Recon: 94.33
Epoch: 42, Iter: 25600, Loss: 107.8, Recon: 95.49
Epoch: 43, Iter: 26000, Loss: 112.9, Recon: 100.1
Epoch: 44, Iter: 26400, Loss: 106.8, Recon: 94.21
Epoch: 44, Iter: 26800, Loss: 108.4, Recon: 95.91
Epoch: 45, Iter: 27200, Loss: 111.7, Recon: 98.76
Epoch: 46, Iter: 27600, Loss: 106.6, Recon: 94.01
Epoch: 46, Iter: 28000, Loss: 107.6, Recon: 95.2
Epoch: 47, Iter: 28400, Loss: 112.3, Recon: 99.28
Epoch: 48, Iter: 28800, Loss: 106.7, Recon: 94.18
Epoch: 48, Iter: 29200, Loss: 107.7, Recon: 95.3
Epoch: 49, Iter: 29600, Loss: 111.2, Recon: 98.14
Epoch: 50, Iter: 30000, Loss: 107.7, Recon: 95.09
Epoch: 50, Iter: 30400, Loss: 106.7, Recon: 94.34
Epoch: 51, Iter: 30800, Loss: 111.4, Recon: 98.47
Epoch: 52, Iter: 31200, Loss: 106.4, Recon: 93.87
Epoch: 52, Iter: 31600, Loss: 107.5, Recon: 95.04
Epoch: 53, Iter: 32000, Loss: 111.2, Recon: 98.25
Epoch: 54, Iter: 32400, Loss: 105.9, Recon: 93.23
Epoch: 54, Iter: 32800, Loss: 107.0, Recon: 94.57
Epoch: 55, Iter: 33200, Loss: 111.4, Recon: 98.49
Epoch: 56, Iter: 33600, Loss: 106.5, Recon: 93.89
Epoch: 56, Iter: 34000, Loss: 106.9, Recon: 94.37
Epoch: 57, Iter: 34400, Loss: 110.3, Recon: 97.25
Epoch: 58, Iter: 34800, Loss: 106.1, Recon: 93.51
Epoch: 58, Iter: 35200, Loss: 107.3, Recon: 94.81
Epoch: 59, Iter: 35600, Loss: 110.0, Recon: 97.03
Epoch: 60, Iter: 36000, Loss: 106.2, Recon: 93.64
Epoch: 60, Iter: 36400, Loss: 106.5, Recon: 93.87
Epoch: 61, Iter: 36800, Loss: 111.4, Recon: 98.36
Epoch: 62, Iter: 37200, Loss: 105.5, Recon: 92.96
Epoch: 62, Iter: 37600, Loss: 107.1, Recon: 94.58
Epoch: 63, Iter: 38000, Loss: 110.2, Recon: 97.3
Epoch: 64, Iter: 38400, Loss: 106.8, Recon: 94.11
Epoch: 64, Iter: 38800, Loss: 106.6, Recon: 94.08
Epoch: 65, Iter: 39200, Loss: 109.6, Recon: 96.68
Epoch: 66, Iter: 39600, Loss: 104.9, Recon: 92.21
Epoch: 66, Iter: 40000, Loss: 106.2, Recon: 93.76
Epoch: 67, Iter: 40400, Loss: 109.1, Recon: 96.09
Epoch: 68, Iter: 40800, Loss: 105.8, Recon: 93.19
Epoch: 68, Iter: 41200, Loss: 106.8, Recon: 94.1
Epoch: 69, Iter: 41600, Loss: 109.7, Recon: 96.83
Epoch: 70, Iter: 42000, Loss: 106.1, Recon: 93.46
Epoch: 70, Iter: 42400, Loss: 106.4, Recon: 93.77
Epoch: 71, Iter: 42800, Loss: 109.4, Recon: 96.42
Epoch: 72, Iter: 43200, Loss: 105.7, Recon: 92.98
Epoch: 72, Iter: 43600, Loss: 106.7, Recon: 94.1
Epoch: 73, Iter: 44000, Loss: 109.7, Recon: 96.59
Epoch: 74, Iter: 44400, Loss: 104.5, Recon: 91.67
Epoch: 74, Iter: 44800, Loss: 106.6, Recon: 93.86
Epoch: 75, Iter: 45200, Loss: 110.4, Recon: 97.36
Epoch: 76, Iter: 45600, Loss: 104.4, Recon: 91.66
Epoch: 76, Iter: 46000, Loss: 106.0, Recon: 93.37
```

```
Epoch: 77, Iter: 46400, Loss: 109.3, Recon: 96.33
Epoch: 78, Iter: 46800, Loss: 104.6, Recon: 91.86
Epoch: 78, Iter: 47200, Loss: 106.4, Recon: 93.68
Epoch: 79, Iter: 47600, Loss: 109.3, Recon: 96.31
Epoch: 80, Iter: 48000, Loss: 104.8, Recon: 92.17
Epoch: 80, Iter: 48400, Loss: 105.9, Recon: 93.37
Epoch: 81, Iter: 48800, Loss: 109.6, Recon: 96.44
Epoch: 82, Iter: 49200, Loss: 104.6, Recon: 91.97
Epoch: 82, Iter: 49600, Loss: 106.1, Recon: 93.5
Epoch: 83, Iter: 50000, Loss: 108.5, Recon: 95.31
Epoch: 84, Iter: 50400, Loss: 104.5, Recon: 91.88
Epoch: 84, Iter: 50800, Loss: 105.5, Recon: 92.79
Epoch: 85, Iter: 51200, Loss: 108.1, Recon: 94.98
Epoch: 86, Iter: 51600, Loss: 105.0, Recon: 92.19
Epoch: 86, Iter: 52000, Loss: 105.7, Recon: 93.06
Epoch: 87, Iter: 52400, Loss: 109.0, Recon: 95.72
Epoch: 88, Iter: 52800, Loss: 105.3, Recon: 92.65
Epoch: 88, Iter: 53200, Loss: 105.5, Recon: 92.92
```

Visualize generated samples after training

```
Epoch: 90, Iter: 54400, Loss: 105.6, Recon: 93.04
```

```
z_gen = tf.random.normal(shape=[num_to_show, Z_DIM])
x_gen = P(z_gen)
imgs_numpy = tf.nn.sigmoid(x_gen).numpy()
show_images(imgs_numpy)
plt.show()
```

PART II. Compute the inception score for your trained VAE m

In this part, we will quantitavely measure how good your VAE model is.

Train a classifier

We first need to train a classifier.

```
batch_size = 128
num_classes = 10
epochs = 20

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(512, activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))

model.summary()

model.compile(loss='categorical_crossentropy',
              optimizer=tf.keras.optimizers.RMSprop(),
              metrics=['accuracy'])

history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

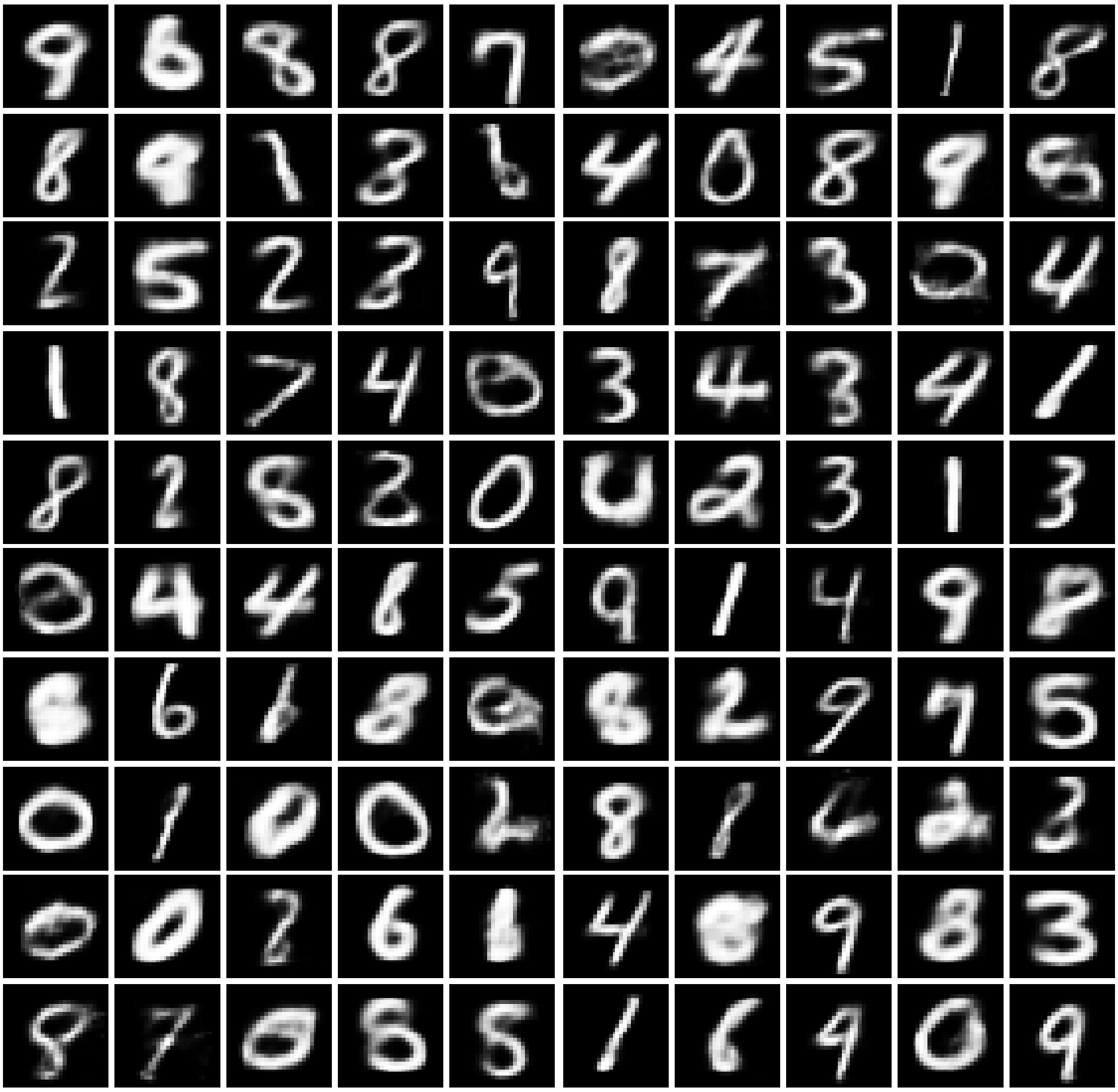
```
60000 train samples
10000 test samples
Model: "sequential_16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_64 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_65 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_66 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
469/469 [==============================] - 2s 4ms/step - loss: 0.2448 - accuracy: 0.9259 - va
Epoch 2/20
469/469 [==============================] - 2s 4ms/step - loss: 0.1008 - accuracy: 0.9695 - va
Epoch 3/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0748 - accuracy: 0.9774 - va
Epoch 4/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0595 - accuracy: 0.9814 - va
Epoch 5/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0513 - accuracy: 0.9848 - va
Epoch 6/20
469/469 [==============================] - 2s 3ms/step - loss: 0.0442 - accuracy: 0.9866 - va
Epoch 7/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0375 - accuracy: 0.9888 - va
Epoch 8/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0341 - accuracy: 0.9900 - va
Epoch 9/20
469/469 [==============================] - 2s 3ms/step - loss: 0.0298 - accuracy: 0.9914 - va
Epoch 10/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0278 - accuracy: 0.9915 - va
Epoch 11/20
469/469 [==============================] - 2s 3ms/step - loss: 0.0277 - accuracy: 0.9923 - va
Epoch 12/20
469/469 [==============================] - 2s 3ms/step - loss: 0.0242 - accuracy: 0.9928 - va
Epoch 13/20
469/469 [==============================] - 2s 3ms/step - loss: 0.0246 - accuracy: 0.9931 - va
Epoch 14/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0217 - accuracy: 0.9939 - va
Epoch 15/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0219 - accuracy: 0.9937 - va
Epoch 16/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0183 - accuracy: 0.9945 - va
Epoch 17/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0193 - accuracy: 0.9946 - va
Epoch 18/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0165 - accuracy: 0.9953 - va
Epoch 19/20
469/469 [==============================] - 2s 4ms/step - loss: 0.0189 - accuracy: 0.9949 - va
```

```
Epoch 20/20
469/469 [==============================] - 2s 3ms/step - loss: 0.0179 - accuracy: 0.9958 - va
Test loss: 0.14113247394561768
```

Verify the trained classifier on the generated samples

Generate samples and visually inspect if the predicted labels on the samples match the actual dig

```
z_gen = tf.random.normal(shape=[num_samples, Z_DIM])
x_gen = P(z_gen)
imgs_numpy = tf.nn.sigmoid(x_gen[:num_to_show]).numpy()
show_images(imgs_numpy)
plt.show()
```



```
np.argmax(model.predict(tf.nn.sigmoid(x_gen[:20])), axis=-1)
```

```
array([9, 6, 8, 8, 7, 0, 4, 5, 1, 8, 8, 8, 1, 3, 1, 4, 0, 8, 8, 9])
```

Implement the inception score

Implement Equation 1 in the reference [3]. Replace expectation in the equation with empirical aver exponentiation at the end. You should get Inception score of at least 9.0.

```
kld_obj = tf.keras.losses.KLDivergence()
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

score = None
image = tf.nn.sigmoid(x_gen)
predicted_y = model.predict(image) #p(y|x)
true_y  =  np.ones ((num_samples, 1)) * np.mean(predicted_y, axis = 0)

score = np.exp(kld_obj(predicted_y, true_y))

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
print('Inception score: {:.4}'.format(score))
```

```
Inception score: 9.229
```

Plot the histogram of predicted labels

Let's additionally inspect the class diversity of the generated samples.

```
plt.hist(np.argmax(model.predict(tf.nn.sigmoid(x_gen)), axis=-1),
         bins=np.arange(11)-0.5, rwidth=0.8, density=True)
plt.xticks(range(10))
plt.show()
```