Programming Assignment 5

Varun Nagaraj

May 5th, 2019

1 Problem 1.a

Derive the restricted Boltzmann machine algorithm that you will implement, and explain your derivation. Implement the training and inference algorithms for RBM. Train RBMs with 20, 100 and 500 hidden nodes to generate MNIST images using the training data set. Generate MNIST images from the ones in the testing data set that have 20%, 50% and 80% pixels missing/removed. You are free to choose whether you want to use binary nodes or floating point nodes, but the derivation has to match the implementation.

Boltzmann Machines are a particular form of log-linear Markov Random Field i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions, we consider that some of the variables are hidden. By having more hidden variables, we can increase the modeling capacity of the Boltzmann Machine. Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections.
The energy function E(v,h) of an RBM is defined as:

$$E(v,h) = -b^{'}v - c^{'}h - h^{'}Wv \tag{1}$$

Where,
W - Weights connecting the visible and hidden units;
b and c - offsets of the visible and hidden layers respectively;

This translates directly to the following free energy formula:

$$F(v) = -b^{'}v - \Sigma_i log \Sigma_{h_i} e^{h_i(c_i + W_i v)} \tag{2}$$

Because of the specific structure of RBMs, visible and hidden units are conditionally independent given one-another. Using this property, we can write:

$$p(h|v) = \Pi_i p(h_i|v) \tag{3}$$

$$p(v|h) = \Pi_j p(v_j|h) \tag{4}$$

A probabilistic version of the usual neuron activation function can be obtained by combining equation 1 with the below equation, where Z is called the partition function:

$$P(x) = \Sigma_h P(x,h) = \Sigma_h \frac{e^{-E(x,h)}}{Z} \tag{5}$$

Probabilistic version of neuron activation function using binary units where v_j and $h_i \ \epsilon \ 0,1$ is as shown below:

$$P(h_i|v) = sigm(c_i + W_i v) \tag{6}$$

$$P(v_j|h) = sigm(b_j + W_j^{'} h) \tag{7}$$

The free energy of an RBM with binary units further simplifies to:

$$F(v) = -b^{'}v - \Sigma_i log(1 + e^{(c_i + W_i v)}) \tag{8}$$

Samples used to estimate the negative phase gradient are referred to as negative particles, which are denoted as N. The gradient can then be written as:

$$-\frac{\partial logp(x)}{\partial \theta} \approx \frac{\partial F(x)}{\partial \theta} - \frac{1}{|N|} \Sigma_{x' \epsilon N} \frac{\partial F(x)}{\partial \theta} \tag{9}$$

Combining equations 8 and 9, we obtain the following log-likelihood gradients for an RBM with binary units as shown below:

$$\frac{\partial logp(v)}{\partial W_{ij}} = E_v[p(h_i|v) * v_j] - v_j^i * sigm(W_i * v^i + c_j) \tag{10}$$

$$\frac{\partial logp(v)}{\partial c_i} = E_v[p(h_i|v)] - sigm(W_i * v^i) \tag{11}$$

$$\frac{\partial logp(v)}{\partial b_j} = E_v[p(v_j|h) * v_j] - v_j^i \tag{12}$$

2 Problem 1.b

Derive the variational autoencoder algorithm that you will implement, and explain your derivation. Implement the training and inference algorithms for VAE. Train VAE with 2, 8 and 16 code units to encode MNIST images using the training data set. The neural network will be 784 input -> 256 hidden -> 2/8/16 code -> 256 hidden -> 784 output. Then use the 2 code -> 256 hidden -> 784 output part of the trained network with 2 code units to generate images by varying each code unit from -3 to 3. You are free to choose the other parameters.

The idea of VAE is to identify P(z) given P(z|x). This can be done with the help of the KL divergence metric. Below is the proof which shows the encoder net, decoder net and the latent variable used in the auto-encoder.

Let's say we want to infer P(z|x) given Q(z|x). The KL divergence metric can be formulated as follows:

$$D_{KL}[Q(z|X)||P(z|X)] = \Sigma_z Q(z|X) log \frac{Q(z|X)}{P(z|X)} \tag{13}$$

$$D_{KL}[Q(z|X)||P(z|X)] = E[log \frac{Q(z|X)}{P(z|X)}] \tag{14}$$

$$D_{KL}[Q(z|X)||P(z|X)] = E[logQ(z|X) - logP(z|X)] \tag{15}$$

By Bayes rule, we can transform thee above equation to the one shown below

$$D_{KL}[Q(z|X)||P(z|X)] = E[logQ(z|X) - log \frac{P(X|z)P(z)}{P(X)}] \tag{16}$$

$$D_{KL}[Q(z|X)||P(z|X)] = E[logQ(z|X) - (logP(X|z) + logP(z) - logP(X))] \tag{17}$$

$$D_{KL}[Q(z|X)||P(z|X)] = E[logQ(z|X) - logP(X|z) - logP(z) + logP(X)] \tag{18}$$

$$D_{KL}[Q(z|X)||P(z|X)] - logP(X) = E[logQ(z|X) - logP(X|z) - logP(z)] \tag{19}$$

$$logP(X) - D_{KL}[Q(z|X)||P(z|X)] = E[log(P(X|z)] - E[logQ(X|z) - logP(z)] \tag{20}$$

$$logP(X) - D_{KL}[Q(z|X)||P(z|X)] = E[log(P(X|z)] - D_{KL}[Q(z|X)||P(z)] \tag{21}$$

Thus from equation 17 we understand than this is a structure of an Auto-encoder. In this the term Q(z|X) can be treated as the encoder net, z is the encoded representation and P(z|X) is the decoder net.

3 Code for part 1 derivation

```
# Created by Varun at 17/04/19
from __future__ import print_function
import numpy as np
import os
import random
import gzip, struct
import tensorflow as tf

def weight(shape, name='weights'):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.1), name=name)

def bias(shape, name='biases'):
    return tf.Variable(tf.constant(0.1, shape=shape), name=name)

class RBM:
    i = 0 # fliping index for computing pseudo likelihood

    def __init__(self, n_visible=784, n_hidden=500, k=30, momentum=False):
```

```
        self.n_visible = n_visible
        self.n_hidden = n_hidden
        self.k = k

        self.lr = tf.placeholder(tf.float32)
        if momentum:
            self.momentum = tf.placeholder(tf.float32)
        else:
            self.momentum = 0.0
        self.w = weight([n_visible, n_hidden], 'w')
        self.hb = bias([n_hidden], 'hb')
        self.vb = bias([n_visible], 'vb')

        self.w_v = tf.Variable(tf.zeros([n_visible, n_hidden]), dtype=tf.floa
        self.hb_v = tf.Variable(tf.zeros([n_hidden]), dtype=tf.float32)
        self.vb_v = tf.Variable(tf.zeros([n_visible]), dtype=tf.float32)

    def propup(self, visible):
        pre_sigmoid_activation = tf.matmul(visible, self.w) + self.hb
        return tf.nn.sigmoid(pre_sigmoid_activation)

    def propdown(self, hidden):
        pre_sigmoid_activation = tf.matmul(hidden, tf.transpose(self.w)) + se
        return tf.nn.sigmoid(pre_sigmoid_activation)

    def sample_h_given_v(self, v_sample):
        h_props = self.propup(v_sample)
        h_sample = tf.nn.relu(tf.sign(h_props - tf.random_uniform(tf.shape(h_
        return h_sample

    def sample_v_given_h(self, h_sample):
        v_props = self.propdown(h_sample)
        v_sample = tf.nn.relu(tf.sign(v_props - tf.random_uniform(tf.shape(v_
        return v_sample

    def CD_k(self, visibles):
        # k steps gibbs sampling
        v_samples = visibles
        h_samples = self.sample_h_given_v(v_samples)
        for i in range(self.k):
            v_samples = self.sample_v_given_h(h_samples)
            h_samples = self.sample_h_given_v(v_samples)

        h0_props = self.propup(visibles)
        w_positive_grad = tf.matmul(tf.transpose(visibles), h0_props)
```

```
        w_negative_grad = tf.matmul(tf.transpose(v_samples), h_samples)
        w_grad = (w_positive_grad - w_negative_grad) / tf.to_float(tf.shape(v
        hb_grad = tf.reduce_mean(h0_props - h_samples, 0)
        vb_grad = tf.reduce_mean(visibles - v_samples, 0)
        return w_grad, hb_grad, vb_grad

    def learn(self, visibles):
        w_grad, hb_grad, vb_grad = self.CD_k(visibles)
        # compute new velocities
        new_w_v = self.momentum * self.w_v + self.lr * w_grad
        new_hb_v = self.momentum * self.hb_v + self.lr * hb_grad
        new_vb_v = self.momentum * self.vb_v + self.lr * vb_grad
        # update parameters
        update_w = tf.assign(self.w, self.w + new_w_v)
        update_hb = tf.assign(self.hb, self.hb + new_hb_v)
        update_vb = tf.assign(self.vb, self.vb + new_vb_v)
        # update velocities
        update_w_v = tf.assign(self.w_v, new_w_v)
        update_hb_v = tf.assign(self.hb_v, new_hb_v)
        update_vb_v = tf.assign(self.vb_v, new_vb_v)

        return [update_w, update_hb, update_vb, update_w_v, update_hb_v, upda

    def sampler(self, visibles, steps=5000):
        v_samples = visibles
        for step in range(steps):
            v_samples = self.sample_v_given_h(self.sample_h_given_v(v_samples
        return v_samples

    def free_energy(self, visibles):
        first_term = tf.matmul(visibles, tf.reshape(self.vb, [tf.shape(self.v
        second_term = tf.reduce_sum(tf.log(1 + tf.exp(self.hb + tf.matmul(vis
        return - first_term - second_term

    def pseudo_likelihood(self, visibles):
        x = tf.round(visibles)
        x_fe = self.free_energy(x)
        split0, split1, split2 = tf.split(x, [self.i, 1, tf.shape(x)[1] - sel
        xi = tf.concat([split0, 1 - split1, split2], 1)
        self.i = (self.i + 1) % self.n_visible
        xi_fe = self.free_energy(xi)
        return tf.reduce_mean(self.n_visible * tf.log(tf.nn.sigmoid(xi_fe - x

class DataSet:
    batch_index = 0
```

```python

    def __init__(self, data_dir, batch_size = None, one_hot = False, seed = 0
        self.data_dir = data_dir
        X, Y = self.read()
        shape = X.shape
        X = X.reshape([shape[0], shape[1] * shape[2]])
        self.X = X.astype(np.float)/255
        self.size = self.X.shape[0]
        if batch_size == None:
            self.batch_size = self.size
        else:
            self.batch_size = batch_size
        # abandom last few samples
        self.batch_num = int(self.size / self.batch_size)
        # shuffle samples
        np.random.seed(seed)
        np.random.shuffle(self.X)
        np.random.seed(seed)
        np.random.shuffle(Y)
        self.one_hot = one_hot
        if one_hot:
            y_vec = np.zeros((len(Y), 10), dtype=np.float)
            for i, label in enumerate(Y):
                y_vec[i, Y[i]] = 1.0
            self.Y = y_vec
        else:
            self.Y = Y

    def read(self):
        with gzip.open(self.data_dir['Y']) as flbl:
            magic, num = struct.unpack(">II", flbl.read(8))
            label = np.fromstring(flbl.read(), dtype=np.int8)
        with gzip.open(self.data_dir['X'], 'rb') as fimg:
            magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
            image = np.fromstring(fimg.read(), dtype=np.uint8).reshape(len(la
        return image, label

    def next_batch(self):
        start = self.batch_index * self.batch_size
        end = (self.batch_index + 1) * self.batch_size
        self.batch_index = (self.batch_index + 1) % self.batch_num
        if self.one_hot:
            return self.X[start:end, :], self.Y[start:end, :]
        else:
            return self.X[start:end, :], self.Y[start:end]
```

```

    def sample_batch(self):
        index = random.randrange(self.batch_num)
        start = index * self.batch_size
        end = (index + 1) * self.batch_size
        if self.one_hot:
            return self.X[start:end, :], self.Y[start:end, :]
        else:
            return self.X[start:end, :], self.Y[start:end]

    def random_removals(self, percentage):
        index = random.randrange(self.batch_num)
        start = index * self.batch_size
        end = (index + 1) * self.batch_size
        for i in self.X:
            pixels = int(percentage*len(self.X[0]))
            while(pixels>0):
                rand = random.randint(0, 63)
                i[rand] = 0
                pixels -=1
                if pixels<=0:
                    break
        return self.X[start:end, :]

import scipy.misc
def save_images(images, size, path):
    img = (images + 1.0) / 2.0
    h, w = img.shape[1], img.shape[2]
    merge_img = np.zeros((h * size[0], w * size[1]))
    for idx, image in enumerate(images):
        i = idx % size[1]
        j = idx // size[1]
        merge_img[j*h:j*h+h, i*w:i*w+w] = image
    return scipy.misc.imsave(path, merge_img)


def train(train_data, test_data, epoches, percentage):
    logs_dir = './logs'
    samples_dir = './samples'

    x = tf.placeholder(tf.float32, shape=[None, 784])
    noise_x = test_data.random_removals(percentage)
    hidden = 500
    rbm = RBM(n_hidden=hidden)
    step = rbm.learn(x)
```

```python
    sampler = rbm.sampler(x)
    pl = rbm.pseudo_likelihood(x)

    saver = tf.train.Saver()

    with tf.Session() as sess:
        init = tf.global_variables_initializer()
        sess.run(init)
        mean_cost = []
        epoch = 1
        for i in range(epoches * train_data.batch_num):
            # draw samples
            if i % 500 == 0:
                samples = sess.run(sampler, feed_dict = {x: noise_x})
                samples = samples.reshape([train_data.batch_size, 28, 28])
                save_images(samples, [8, 8], os.path.join(samples_dir, 'itera
                print('Saved samples.')
            batch_x, _ = train_data.next_batch()
            sess.run(step, feed_dict = {x: batch_x, rbm.lr: 0.1})
            cost = sess.run(pl, feed_dict = {x: batch_x})
            mean_cost.append(cost)
            # save model
            if i is not 0 and train_data.batch_index is 0:
                checkpoint_path = os.path.join(logs_dir, 'model.ckpt')
                saver.save(sess, checkpoint_path, global_step = epoch + 1)
                print('Saved Model.')
            # print pseudo likelihood
            if i is not 0 and train_data.batch_index is 0:
                print('Epoch %d Cost %g' % (epoch, np.mean(mean_cost)))
                mean_cost = []
                epoch += 1
        print('Test')
        samples = sess.run(sampler, feed_dict = {x: noise_x})
        samples = samples.reshape([train_data.batch_size, 28, 28])
        save_images(samples, [8, 8], os.path.join(samples_dir, 'test{}_{}.png
        print('Saved samples.')

train_dir = {
    'X': './train-images-idx3-ubyte.gz',
    'Y': './train-labels-idx1-ubyte.gz'
}

test_dir = {
    'X': './t10k-images-idx3-ubyte.gz',
    'Y': './t10k-labels-idx1-ubyte.gz'
```

```
}
train_data = DataSet(data_dir=train_dir, batch_size=64, one_hot=True)
test_data = DataSet(data_dir=test_dir, batch_size=64)
train(train_data, test_data,3, 0.8)
```

4 Code for derivation 2

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import os

import matplotlib.pyplot as plt
import numpy as np
from keras import backend as K
from keras.datasets import mnist
from keras.layers import Lambda, Input, Dense
from keras.losses import mse, binary_crossentropy
from keras.models import Model
from keras.utils import plot_model


def sampling(args):
    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
    dim = K.int_shape(z_mean)[1]
    # by default, random_normal has mean=0 and std=1.0
    epsilon = K.random_normal(shape=(batch, dim))
    return z_mean + K.exp(0.5 * z_log_var) * epsilon


def plot_results(models,
                 data,
                 batch_size=128,
                 model_name="vae_mnist"):
    encoder, decoder = models
    x_test, y_test = data
    os.makedirs(model_name)

    filename = os.path.join(model_name, "vae_mean.png")
    z_mean, _, _ = encoder.predict(x_test,
                                   batch_size=batch_size)
    plt.figure(figsize=(12, 10))
```

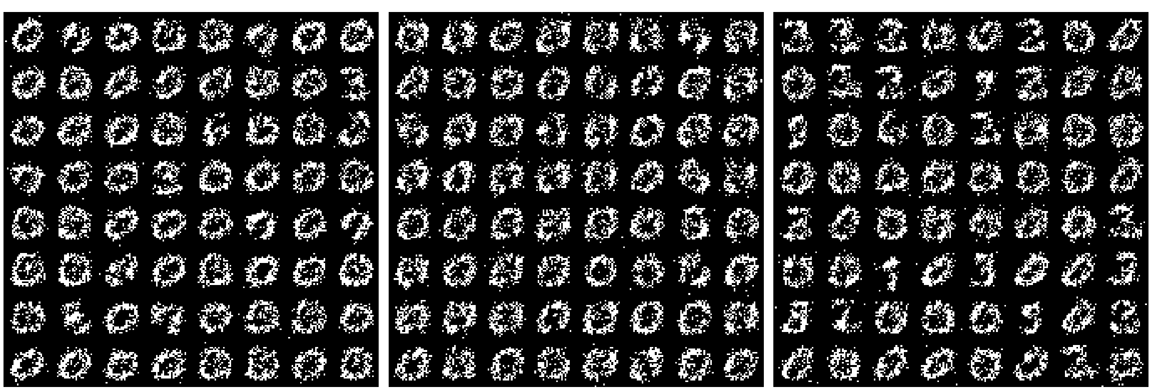
```
    plt.scatter(z_mean[:, 0], z_mean[:, 1], c=y_test)
    plt.colorbar()
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.savefig(filename)
    plt.show()

    filename = os.path.join(model_name, "digits_over_latent.png")
    n = 30
    digit_size = 28
    figure = np.zeros((digit_size * n, digit_size * n))
    grid_x = np.linspace(-3, 3, n)
    grid_y = np.linspace(-3, 3, n)[::-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(digit_size, digit_size)
            figure[i * digit_size: (i + 1) * digit_size,
            j * digit_size: (j + 1) * digit_size] = digit

    plt.figure(figsize=(10, 10))
    start_range = digit_size // 2
    end_range = n * digit_size + start_range + 1
    pixel_range = np.arange(start_range, end_range, digit_size)
    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[0]")
    plt.ylabel("z[1]")
    plt.imshow(figure, cmap='Greys_r')
    plt.savefig(filename)
    plt.show()


(x_train, y_train), (x_test, y_test) = mnist.load_data()
image_size = x_train.shape[1]
original_dim = image_size * image_size
x_train = np.reshape(x_train, [-1, original_dim])
x_test = np.reshape(x_test, [-1, original_dim])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
input_shape = (original_dim,)
```

```
intermediate_dim = 256
batch_size = 128
latent_dim = 2
epochs = 50

inputs = Input(shape=input_shape, name='encoder_input')
x = Dense(intermediate_dim, activation='relu')(inputs)
z_mean = Dense(latent_dim, name='z_mean')(x)
z_log_var = Dense(latent_dim, name='z_log_var')(x)

z = Lambda(sampling, output_shape=(latent_dim,), name='z')([z_mean, z_log_var
encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
encoder.summary()
plot_model(encoder, to_file='vae_mlp_encoder.png', show_shapes=True)
latent_inputs = Input(shape=(latent_dim,), name='z_sampling')
x = Dense(intermediate_dim, activation='relu')(latent_inputs)
outputs = Dense(original_dim, activation='sigmoid')(x)
decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()
plot_model(decoder, to_file='vae_mlp_decoder.png', show_shapes=True)
outputs = decoder(encoder(inputs)[2])
vae = Model(inputs, outputs, name='vae_mlp')

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    help_ = "Load h5 model trained weights"
    parser.add_argument("-w", "--weights", help=help_)
    help_ = "Use mse loss instead of binary cross entropy (default)"
    parser.add_argument("-m",
                        "--mse",
                        help=help_, action='store_true')
    args = parser.parse_args()
    models = (encoder, decoder)
    data = (x_test, y_test)
    if args.mse:
        reconstruction_loss = mse(inputs, outputs)
    else:
        reconstruction_loss = binary_crossentropy(inputs,
                                                  outputs)

    reconstruction_loss *= original_dim
    kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
    kl_loss = K.sum(kl_loss, axis=-1)
    kl_loss *= -0.5
    vae_loss = K.mean(reconstruction_loss + kl_loss)
```

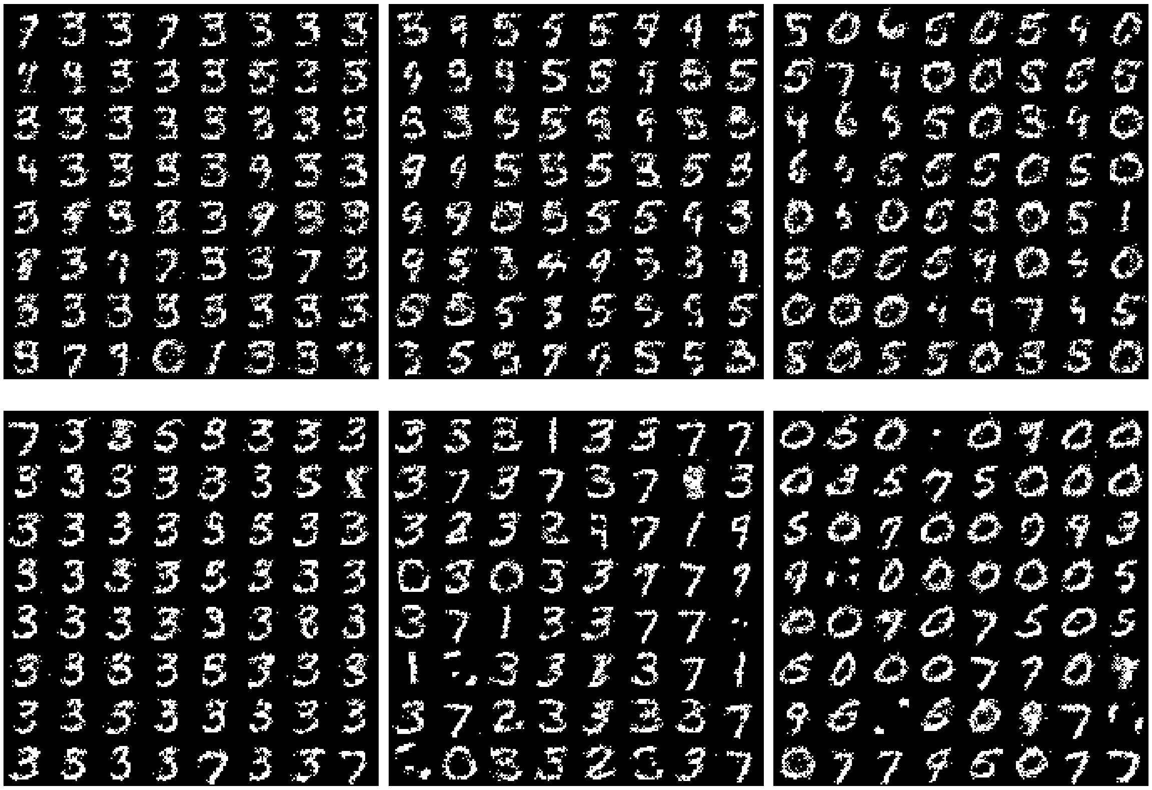
```
    vae.add_loss(vae_loss)
    vae.compile(optimizer='adam')
    vae.summary()
    plot_model(vae,
               to_file='vae_mlp.png',
               show_shapes=True)

    if args.weights:
        vae.load_weights(args.weights)
    else:
        vae.fit(x_train,
                epochs=epochs,
                batch_size=batch_size,
                validation_data=(x_test, None))
        vae.save_weights('vae_mlp_mnist.h5')

    plot_results(models,
                 data,
                 batch_size=batch_size,
                 model_name="vae_mlp")
```

5 Code Output images

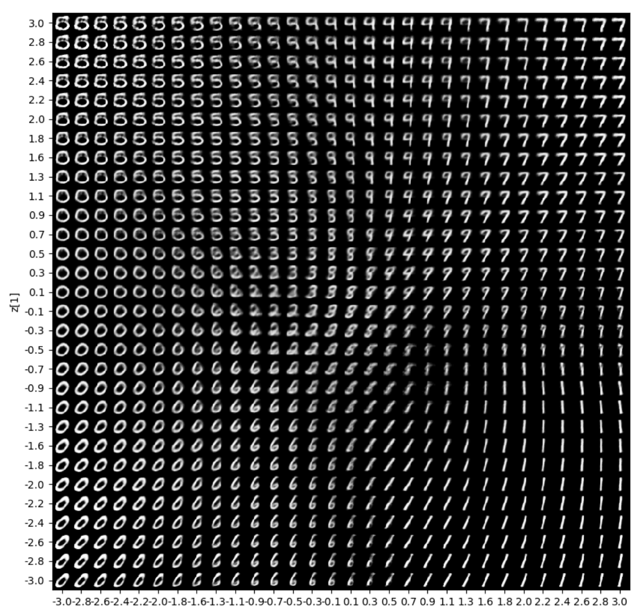
5.1 Part-1

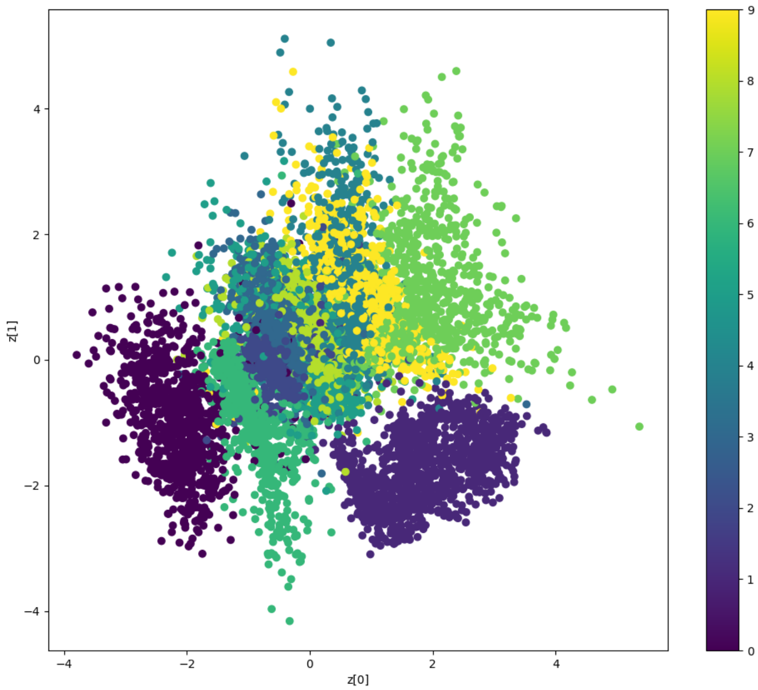


5.1.1 The first row is with 20 hidden nodes, second is with 100 and third is with 500 hidden nodes. The first column shows the output when 20 percent of the pixels are removed, similarly column 2- 50 percent and column 3-80 percent



5.2 Part-2





6 References

https://jaan.io/what-is-variational-autoencoder-vae-tutorial/
https://medium.com/datatype/restricted-boltzmann-machine-a-complete-analysis-part-1-introduction-model-formulation-1a4404873b3
https://towardsdatascience.com/teaching-a-variational-autoencoder-vae-to-draw-mnist-characters-978675c95776
https://github.com/FelixMohr/Deep-learning-with-Python/blob/master/VAE.ipynb
https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/
https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols#Relation_operators
https://medium.com/datadriveninvestor/image-processing-for-mnist-using-keras-f9a1021f6ef0
https://github.com/keras-team/keras/blob/master/examples/variational_autoencoder.py