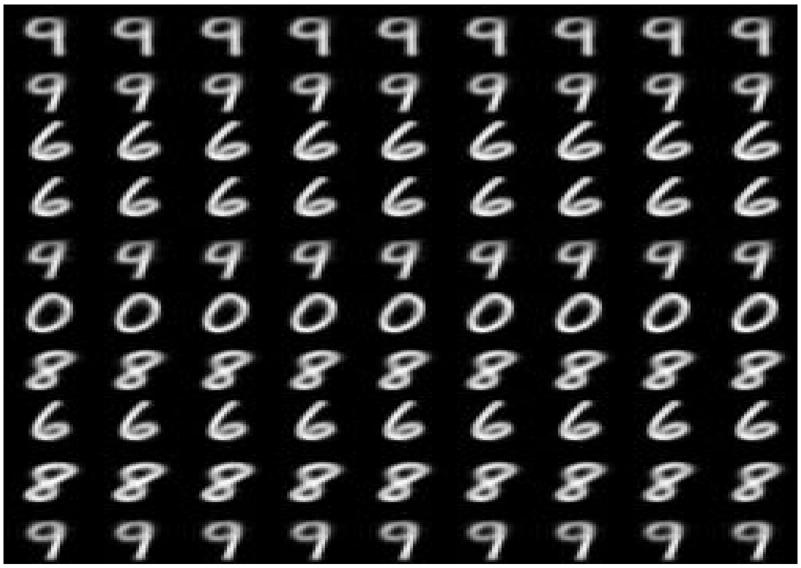
CS498_HW11

Mostafa Elkabir, Sooraj Kumar
Elkabir2, Sskumar5

Part 1. Show grid of 10 x 9 for a single MNIST digit- decoded images.



Part 2. Show a grid of 10 x 9 for different digits - decoded images.

Code Snippet:

Below is the lines of code for the most important parts of the assignment. Containing decoder, encoder parts, and building the model.

```python
def train(model, mnist_dataset, learning_rate=0.0005, batch_size=16, num_steps=100000):#5000):
    """Implements the training loop of mini-batch gradient descent.

    Performs mini-batch gradient descent with the indicated batch_size and
    learning_rate. (**Do not modify this function.**)

    Args:
        model(VariationalAutoencoder): Initialized VAE model.
        mnist_dataset: Mnist dataset.
        learning_rate(float): Learning rate.
        batch_size(int): Batch size used for training.
        num_steps(int): Number of steps to run the update ops.
    """
    for step in range(0, num_steps):
        batch_x, _ = mnist_dataset.train.next_batch(batch_size)
        model.session.run(
                model.update_op_tensor,
                feed_dict={model.x_placeholder: batch_x,
                           model.learning_rate_placeholder: learning_rate}
                )

def enc_lin_interpolate_dec(model, train_mnist):
    """
    This function encodes sampled images to the latent space after training. After this is done, we find the linear inter-
    -polation between each pair (seven of them). We then decode each of these linear interpolates and plot them in a 10x9
    grid.
    :param: model: A trained Variational Autoencoder model that can decode and encode
    :param: train_mnist:
    :return:
    """

    one_hot_mnist = train_mnist.images
    normal_mnist = read_data_sets('MNIST_data')
    normal_mnist_train_labels = normal_mnist.train.labels

    # First, we pick 10 pairs of different digits
    digits = np.random.randint(low=0, high=10, size=20).reshape(10, 2)
    images = np.empty((10*28, 9*28))

    for pair_idx in range(10):
        pair = digits[pair_idx, :]

        # We need two images that correspond to the random digit pair that we extracted
        indx1 = np.where(normal_mnist_train_labels == pair[0])
        indx2 = np.where(normal_mnist_train_labels == pair[1])
        pair_1_image, pair_2_image = normal_mnist.train.images[indx1[0][0], :][:, np.newaxis].T, normal_mnist.train.images[indx2[0][0], :][:, np.newaxis].T

        l1 = model.session.run(model.z, feed_dict={model.x_placeholder:pair_1_image})
        l2 = model.session.run(model.z, feed_dict={model.x_placeholder:pair_2_image})

        l1_interp, l2_interp = np.linspace(start=l1[0, 0], stop=l2[0, 0], num=9), np.linspace(start=l1[0, 1], stop=l2[0, 1], num=9)
        for i in range(9):
```

```python
            # else:
            #     images[pair_idx*28:(pair_idx + 1)*28, i*28:(i + 1)*28] = pair_2_image[0, :].reshape(28, 28)
    plt.imsave('inter_images_same_2.png', images, cmap="gray")
    return images

def main(_):
    """High level pipeline.

    This script performs the training for VAEs.
    """
    # Get dataset.
    mnist_dataset = read_data_sets('MNIST_data', one_hot=True)

    # Build model.
    model = VariationalAutoencoder()

    # Start training
    train(model, mnist_dataset)

    images = enc_lin_interpolate_dec(model, mnist_dataset.train)
    print(np.shape(images))
    iamges_same =enc_lin_interpolate_dec_same(model, mnist_dataset.train)

    pass

    #    Plot out latent space, for +/- 3 std.
    std = 1
    x_z = np.linspace(-3*std, 3*std, 20)
    y_z = np.linspace(-3*std, 3*std, 20)

    out = np.empty((28*20, 28*20))
    for x_idx, x in enumerate(x_z):
        for y_idx, y in enumerate(y_z):
            z_mu = np.array([[y, x]])
            img = model.generate_samples(z_mu)
            out[x_idx*28:(x_idx+1)*28,
                y_idx*28:(y_idx+1)*28] = img[0].reshape(28, 28)
    plt.imsave('latent_space_vae.png', out, cmap="gray")

if __name__ == "__main__":
    tf.app.run()
```

```python
            z_mean(tf.Tensor): Tensor of dimension (None, _nlatent)
            z_log_var(tf.Tensor): Tensor of dimension (None, _nlatent)

        Returns:
            latent_loss(tf.Tensor): A scalar Tensor of dimension ()
                containing the latent loss.
        """
        latent_loss = 0.5 *(tf.reduce_sum((1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)), axis=1))
        # vars = tf.exp(z_log_var)
        # latent_loss = tf.reduce_sum(tf.reduce_sum(0.5 * tf.reduce_sum(vars, axis=1) + (tf.reduce_sum(-z_mean*-z_mean, axis=1))
        #                                       - self._nlatent + tf.log(1/tf.reduce_prod(vars, axis=1))))
        return latent_loss

    def _reconstruction_loss(self, f, x_gt):
        """Constructs the reconstruction loss, assuming Gaussian distribution.

        Args:
            f(tf.Tensor): Predicted score for each example, dimension (None,
                _ndims).
            x_gt(tf.Tensor): Ground truth for each example, dimension (None,
                _ndims).
        Returns:
            recon_loss(tf.Tensor): A scalar Tensor for dimension ()
                containing the reconstruction loss.
        """
        # recon_loss = tf.nn.l2_loss(f - x_gt)
        recon_loss = tf.reduce_sum(tf.square(f-x_gt), axis=1)
        return recon_loss

    def loss(self, f, x_gt, z_mean, z_var):
        """Computes the total loss.

        Computes the sum of latent and reconstruction loss.

        Args:
            f (tf.Tensor): Decoded image for each example, dimension (None,
                _ndims).
            x_gt (tf.Tensor): Ground truth for each example, dimension (None,
                _ndims)
            z_mean (tf.Tensor): The latent mean,
                tensor of dimension (None, _nlatent)
            z_log_var (tf.Tensor): The latent log variance,
                tensor of dimension (None, _nlatent)

        Returns:
            total_loss: Tensor for dimension (). Sum of
                latent_loss and reconstruction loss.
        """
        self._reconstruction_tensor = self._reconstruction_loss(f, x_gt)
        self._latent_tensor = self._latent_loss(z_mean, z_var)
        total_loss = self._reconstruction_loss(f, x_gt) - self._latent_loss(z_mean, z_var)
        total_loss = tf.reduce_mean(total_loss)
        return total_loss
```

```
        Returns:
            total_loss: Tensor for dimension (). Sum of
                latent_loss and reconstruction loss.
        """
        self._reconstruction_tensor = self._reconstruction_loss(f, x_gt)
        self._latent_tensor = self._latent_loss(z_mean, z_var)
        total_loss = self._reconstruction_loss(f, x_gt) - self._latent_loss(z_mean, z_var)
        total_loss = tf.reduce_mean(total_loss)
        return total_loss

    def update_op(self, loss, learning_rate):
        """Creates the update optimizer.

        Use tf.train.AdamOptimizer to obtain the update op.

        Args:
            loss(tf.Tensor): Tensor of shape () containing the loss function.
            learning_rate(tf.Tensor): Tensor of shape (). Learning rate for
                gradient descent.
        Returns:
            train_op(tf.Operation): Update opt tensorflow operation.
        """
        train_op = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(loss)
        return train_op

    def generate_samples(self, z_np):
        """Generates random samples from the provided z_np.

        Args:
            z_np(numpy.ndarray): Numpy array of dimension
                (batch_size, _nlatent).

        Returns:
            out(numpy.ndarray): The sampled images (numpy.ndarray) of
                dimension (batch_size, _ndims).
        """
        out = self.session.run(self.outputs_tensor, feed_dict={self.z: z_np})
        return out
```

```
        z = z_mean + tf.sqrt(tf.exp(z_log_var)) epsilon
        return z

    def _encoder(self, x):
        """Encoder block of the network.

        Builds a two layer network of fully connected layers, with 100 nodes,
        then 50 nodes, and outputs two branches each with _nlatent nodes
        representing z_mean and z_log_var. Network illustrated below:

                             |-> _nlatent (z_mean)
        Input --> 100 --> 50 -
                             |-> _nlatent (z_log_var)

        Use activation of tf.nn.softplus for hidden layers.

        Args:
            x (tf.Tensor): The input tensor of dimension (None, _ndims).
        Returns:
            z_mean(tf.Tensor): The latent mean, tensor of dimension
                (None, _nlatent).
            z_log_var(tf.Tensor): The latent log variance, tensor of dimension
                (None, _nlatent).
        """

        dense_1 = fully_connected(x, num_outputs=100, activation_fn=tf.nn.softplus)
        dense_2 = fully_connected(dense_1, num_outputs=50, activation_fn=tf.nn.softplus)
        z_mean = fully_connected(dense_2, num_outputs=self._nlatent, activation_fn=None)
        z_log_var = fully_connected(dense_2, num_outputs=self._nlatent, activation_fn=None)
        return z_mean, z_log_var

    def _decoder(self, z):
        """From a sampled z, decode back into image.

        Builds a three layer network of fully connected layers,
        with 50, 100, _ndims nodes.

        z (_nlatent) --> 50 --> 100 --> _ndims.

        Use activation of tf.nn.softplus for hidden layers.

        Args:
            z(tf.Tensor): z from _sample_z of dimension (None, _nlatent).
        Returns:
            f(tf.Tensor): Decoded features, tensor of dimension (None, _ndims).
        """

        dense_1 = fully_connected(inputs=z, num_outputs=50, activation_fn=tf.nn.softplus)
        dense_2 = fully_connected(inputs=dense_1, num_outputs=100, activation_fn=tf.nn.softplus)
        f = fully_connected(inputs=dense_2, num_outputs=self._ndims, activation_fn=tf.nn.sigmoid)
        return f

    def _latent_loss(self, z_mean, z_log_var):
        """Constructs the latent loss.
```