Assignment 3: Variational Autoencoders

STA414 Winter 2020
Instructors: David Duvenaud and Jesse Bettencourt
Yufeng Tao
Student Number: 1006630246

April 15, 2020

0.1 Background

In this assignment, we will implement and investigate the Variational Autoencoder on binarized MNIST digits, as introduced by the paper Auto-Encoding Variational Bayes by Kingma and Welling (2013). Before starting, we recommend reading this paper.

Data. Each datapoint in the MNIST dataset is a 28x28 grayscale image (i.e. pixels are values between 0 and 1) of a handwritten digit in $\{0 \dots 9\}$, and a label indicating which number. MNIST is the 'fruit fly' of machine learning – a simple standard problem useful for comparing the properties of different algorithms.

Use the first 10000 samples for training, and the second 10000 for testing. Hint: Also build a dataset of only 100 training samples to use when debugging, to make loading and training faster.

Tools. As before, you can (and should) use automatic differentiation provided by your package of choice. Whereas in previous assignments you implemented neural network layers and stochastic gradient descent manually, in this assignment feel free to use those provided by a machine learning framework. In Julia, these will be provided by `Flux.jl`. You can also freely copy and adapt the Python autograd starter code provided. If you do, you should probably remove batch normalization.

However, you **may not use any probabilistic modelling elements** from these frameworks. In particular, sampling from and evaluating densities under distributions must be written by you or provided by the starter code.

0.2 Model Definition

Prior. The prior over each digit's latent representation is a multivariate standard normal distribution. For all questions, we'll set the dimension of the latent space D_z to 2. A larger latent dimension would provide a more powerful model, but for this assignment we'll use a two-dimensional latent space to make visualization and debugging easier..

Likelihood. Given the latent representation z for an image, the distribution over all 784 pixels in the image is given by a product of independent Bernoullis, whose means are given by the output of a neural network $f_\theta(z)$:

$$p(x|z,\theta) = \prod_{d=1}^{784} \text{Ber}(x_d|f_\theta(z)_d)$$

The neural network f_θ is called the decoder, and its parameters θ will be optimized to fit the data.

1 Implementing the Model [5 points]

For your convenience we have provided the following functions:

- `factorized_gaussian_log_density` that accepts the mean and **log** standard deviations for a product of independent gaussian distribtuions and computes the likelihood under them. This function will produce the log-likelihood for each batch element $1 \times B$
- `bernoulli_log_density` that accepts the logits of a bernoulli distribution over D-dimensional data and returns $D \times B$ log-likelihoods.
- `sample_diag_gaussian` that accepts above parameters for a factorized Gaussian distribution and samples with the reparameterization trick.
- `sample_bernoulli` that accepts above parameters for a Bernoulli distribution and samples from it.
- `load_binarized_mnist` that loads and binarizes the MNIST dataset.
- `batch_data` and `batch_x` that splits the data, and just the images, into batches.

Further, in the file `example_flux_model.jl` we demonstrate how to specify neural network layers with Flux library. Note that Flux provides convenience functions that allow us to take gradients of functions with respect to parameters that **are not passed around explicitly**. Other AD frameworks, of if you prefer to implement your own network layers, recycling code from previous assignments, you may need to explicilty provide the network parameters to the functions below.

(a) **[1 point]** Implement a function `log_prior` that computes the log of the prior over a digit's representation $\log p(z)$.

```
def diag_gaussian_log_density(x, mu, log_std):
    return np.sum(norm.logpdf(x, mu, np.exp(log_std)), axis=-1)

def log_prior(z):
    return diag_gaussian_log_density(z, 0, 0)
```

(b) **[2 points]** Implement a function `decoder` that, given a latent representation z and a set of neural network parameters θ (again, implicitly in Flux), produces a 784-dimensional mean vector of a product of Bernoulli distributions, one for each pixel in a 28×28 image. Make the decoder architecture a multilayer perceptron (i.e. a fully-connected neural network) with a single hidden layer with 500 hidden units, and a `tanh` nonlinearity. Its input will be a batch two-dimensional latent vectors (zs in $D_z \times B$) and its output will be a 784-dimensional vector representing the logits of the Bernoulli means for each dimension $D_{\text{data}} \times B$. For numerical stability, instead of outputting the mean $\mu \in [0, 1]$, you should output $\log\left(\frac{\mu}{1-\mu}\right) \in \mathbb{R}$ called "logit".

```
def init_net_params(scale, layer_sizes, rs=npr.RandomState(0)):
    """Build a (weights, biases) tuples for all layers."""
    return [(scale * rs.randn(m, n),   # weight matrix
             scale * rs.randn(n))      # bias vector
            for m, n in zip(layer_sizes[:-1], layer_sizes[1:])]

def neural_net_predict(params, inputs):
    """Params is a list of (weights, bias) tuples.
       inputs is an (N x D) matrix.
       Applies batch normalization to every layer but the last."""
    for W, b in params:
        outputs = np.dot(inputs, W) + b  # linear transformation
        inputs = np.tanh(outputs)           # nonlinear transformation
```

```
    return outputs

# MLP with a single hidden layer with 500 hidden units, and a tanh nonlinearity.
# Model hyper-parameters
latent_dim = 2
data_dim = 784  # How many pixels in each image (28x28).
gen_layer_sizes = [latent_dim, 500, data_dim]

def decoder(z, gen_params):
    # gen_params is a global variable
    return neural_net_predict(gen_params, z)
```

(c) **[1 point]** Implement a function `log_likelihood` that, given a latent representation z and a binarized digit x, computes the log-likelihood $\log p(x|z)$.

```
def bernoulli_log_density(b, unnormalized_logprob):
    # returns log Ber(b | mu)
    # unnormalized_logprob is log(mu / (1 - mu)
    # b must be 0 or 1
    s = b * 2 - 1
    return -np.logaddexp(0., -s * unnormalized_logprob)

def log_likelihood(x, z, gen_params):
    """ Compute log likelihood log_p(x|z)"""
    bernoulli_means = decoder(z, gen_params) # parameters decoded from latent z
    likelihoods = bernoulli_log_density(x, bernoulli_means)
    return np.sum(likelihoods, axis=-1)  # Sum across pixels
    # likelihood for each element in batch
```

(d) **[1 point]** Implement a function `joint_log_density` which combines the log-prior and log-likelihood of the observations to give $\log p(z, x)$ for a single image.

```
def joint_log_density(x,z, gen_params):
    return log_prior(z) + log_likelihood(x, z, gen_params)
```

All of the functions in this section must be able to be evaluated in parallel, vectorized and non-mutating, on a batch of B latent vectors and images, using the same parameters θ for each image. In particular, you can not use a for loop over the batch elements.

2 Amortized Approximate Inference and training [13 points]

(a) **[2 points]** Write a function `encoder` that, given an image x (or batch of images) and recognition parameters ϕ, evaluates an MLP to outputs the mean and log-standard deviation of a factorized Gaussian of dimension $D_z = 2$. Make the encoder architecture a multi-layer perceptron (i.e. a fully-connected neural network) with a single hidden layer with 500 hidden units, and a `tanh` nonlinearity. This function must be able to be evaluated in parallel on a batch of images, using the same parameters ϕ for each image.

```
rec_layer_sizes = [data_dim, 500, latent_dim * 2]

def unpack_gaussian_params(params):
    # Params of a diagonal Gaussian.
    D = np.shape(params)[-1] // 2
    mean, log_std = params[:, :D], params[:, D:]
    return mean, log_std

def nn_predict_gaussian(params, inputs):
    # Returns means and diagonal variances
    return unpack_gaussian_params(neural_net_predict(params, inputs))

def encoder(rec_params, x):
    return nn_predict_gaussian(rec_params, x)
```

(b) **[1 points]** Write a function `log_q` that given the parameters of the variational distribution, evaluates the likelihood of z.

```
# q_mu, q_logstd = encoder(x)
def log_q(q_mu, q_logstd, z):
    return diag_gaussian_log_density(z, q_mu, q_logstd)
```

(c) **[5 points]** Implement a function `elbo` which computes an unbiased estimate of the mean variational evidence lower bound on a batch of images. Use the output of `encoder` to give the parameters for $q_\phi(z|\text{data})$. This estimator takes the following arguments:

- `x`, an batch of B images, $D_x \times B$.
- `encoder_params`, the parameters ϕ of the encoder (recognition network). Note: these are not required with Flux as parameters are implicit.
- `decoder_params`, the parameters θ of the decoder (likelihood). Note: these are not required with Flux as parameters are implicit.

This function should return a single scalar. Hint: You will need to use the reparamterization trick when sampling `zs`. You can use any form of the ELBO estimator you prefer. (i.e., if you want you can write the KL divergence between q and the prior in closed form since they are both Gaussians). You only need to sample a single z for each image in the batch.

```
def sample_diag_gaussian(mean, log_std, rs):
    return rs.randn(*mean.shape) * np.exp(log_std) + mean

def elbo(gen_params, rec_params, data, rs, batch_size):
    # We use a simple Monte Carlo estimate of the KL
    # divergence from the prior.
    q_means, q_log_stds = encoder(rec_params, data)
    # sample z from variational distribution
    latents = sample_diag_gaussian(q_means, q_log_stds, rs)
    # joint log likelihood of z and x under model
```

```
    joint_ll = joint_log_density(data,latents, gen_params)
    # log likelihood of z under variational distribution
    log_q_z = log_q(q_means, q_log_stds, latents)
    # data_dim = 784  # How many pixels in each image (28x28).
    elbo_estimate = np.sum(joint_ll - log_q_z) / batch_size # scalar
    return elbo_estimate
```

(d) **[2 points]** Write a loss function called `loss` that returns the negative elbo estimate over a batch of data.

```
def loss(gen_params, rec_params, data, rs, batch_size):
    return -elbo(gen_params, rec_params, data, rs, batch_size)
```

(e) **[3 points]** Write a function that initializes and optimizes the encoder and decoder parameters jointly on the training set. Note that this function should optimize with gradients on the elbo estimate over batches of data, not the entire dataset. Train the data for 100 epochs (each epoch involves a loop over every batch). Report the final ELBO on the test set. Tip: Save your trained weights here (e.g. with `BSON.jl`, see starter code, or by pickling in Python) so that you don't need to train them again.

```
# Training parameters
param_scale = 0.01
batch_size = 100
num_epochs = 100
step_size = 0.001

## Load the Data
train_x, train_labels, test_x, test_labels = load_binarized_mnist(train_size=10000,
test_size=10000)

init_gen_params = init_net_params(param_scale, gen_layer_sizes)
init_rec_params = init_net_params(param_scale, rec_layer_sizes)
combined_init_params = (init_gen_params, init_rec_params)

num_batches = int(np.ceil(len(train_x) / batch_size))

def batch_indices(iter):
    idx = iter % num_batches
    return slice(idx * batch_size, (idx+1) * batch_size)

def generate_from_prior(gen_params, num_samples, noise_dim, rs):
    latents = rs.randn(num_samples, noise_dim)
    return sigmoid(neural_net_predict(gen_params, latents))

def train_model_params():
    # Define training objective
    seed = npr.RandomState(0)
    # loss in writeup
    def objective(combined_params, iter):
        data_idx = batch_indices(iter)
        gen_params, rec_params = combined_params
        return -elbo(gen_params, rec_params, train_x[data_idx], seed, batch_size)

    # Get gradients of objective using autograd.
    objective_grad = grad(objective)
```

```
    print("     Epoch     |    Objective       |    Test ELBO  ")
    def print_perf(combined_params, iter, grad):
        if iter % 10 == 0:
            gen_params, rec_params = combined_params
            bound = np.mean(objective(combined_params, iter))
            message = "{:15}|{:20}|".format(iter//num_batches, bound)
            if iter % 100 == 0:
                test_bound = -elbo(gen_params, rec_params, test_x[batch_indices(iter)],
                seed, batch_size) #/data_dim
                message += "{:20}".format(test_bound)
            print(message)

            fake_data = generate_from_prior(gen_params, 20, latent_dim, seed)
            save_images(fake_data, 'vae_samples.png', vmin=0, vmax=1)

    # The optimizers provided can optimize lists, tuples, or dicts of parameters.
    optimized_params = adam(objective_grad, combined_init_params, step_size=step_size,
                            num_iters=num_epochs * num_batches, callback=print_perf)

    return optimized_params

optimized_params = train_model_params()
     Epoch     |    Objective       |    Test ELBO
              0|   543.3049204322726|   543.3050622376885
              0|  500.39551010167037|
              0|   312.4831908264722|
              0|    244.376098423773|
              0|  232.13371781925093|
              0|   225.2021478615809|
              0|   231.7358877609071|
              0|  214.00916166407217|
              0|  229.19652493485685|
              0|  208.35518145570444|
              1|   203.5989340144995|  203.69616463986316
              1|  196.91858715963477|
              1|   208.1780420609205|
              1|   204.4994256452567|
                        ......
             98|   164.4836598698129|
             98|  150.50435819446227|
             98|  165.74104285100773|
             98|  158.37738805033962|
             99|   149.1948434381552|  157.09402361232995
             99|  152.61224173789142|
             99|   157.6699604011918|
             99|  164.77850201981772|
             99|  161.21271295868394|
             99|  165.71715868383188|
             99|  164.50132547047352|
             99|  150.37172978470917|
             99|  167.49041250678954|
             99|  159.31055787683405|

# Data persistence
```

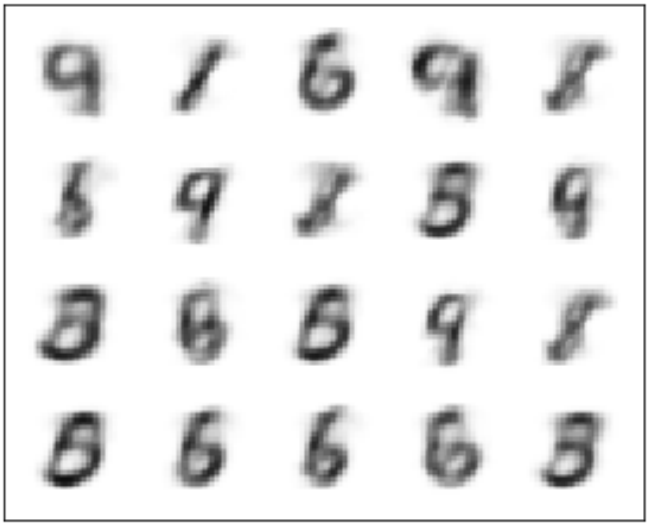

Figure 1: Q2(e) VAE samples

```
gen_params, rec_params = optimized_params
from google.colab import files
import pickle

def local_persist(fname, results):
  fname = fname + '.txt'
  pickle.dump(results, open(fname, 'wb'))
  files.download(fname)

local_persist('trained_rec_params', rec_params)
local_persist('trained_gen_params', gen_params)

from io import BytesIO

# function to restore from local storage
def load_from_local():
  loaded = {}
  uploaded = files.upload()
  for name in uploaded.keys():
      data = uploaded[name]
      loaded[name] = pickle.load(BytesIO(data))
  return loaded

loaded = load_from_local()
rec_params = loaded['trained_rec_params.txt']
gen_params = loaded['trained_gen_params.txt']
```

3 Visualizing Posteriors and Exploring the Model [15 points]

In this section we will investigate our model by visualizing the distribution over data given by the generative model, sampling from it, and interpolating between digits.

(a) **[5 points]** Plot samples from the trained generative model using ancestral sampling:

(a) First sample a z from the prior.

(b) Use the generative model to compute the bernoulli means over the pixels of x given z. Plot these means as a greyscale image.

(c) Sample a binary image x from this product of Bernoullis. Plot this sample as an image.

Do this for 10 samples z from the prior.

Concatenate all your plots into one 2x10 figure where each image in the first row shows the Bernoulli means of $p(x|z)$ for a separate sample of z, and each image in the the second row is a bianry image, sampled from the distribution above it. Make each column an independent sample.

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import random

seed = npr.RandomState(0)
q_means, q_log_stds = encoder(rec_params, train_x)

AX = gridspec.GridSpec(2,10)
AX.update(wspace = 0.0, hspace = 0.0)

for i in range(0,10):
    # sample z from variational distribution
    z = sample_diag_gaussian(q_means, q_log_stds, seed)
    # logits of the Bernoulli means
    logits = decoder(z, gen_params)

    ber_mean = 1/(1 + np.exp(-logits))

    index = random.randrange(0, len(logits))
    plt.subplot(AX[0,i])
    plt.imshow(ber_mean[index].reshape(28,28), cmap='gray')

    # Sample a binary image x from this product of Bernoullis.
    images = sample_bernoulli(ber_mean)
    plt.subplot(AX[1,i])
    plt.imshow(images[index].reshape(28,28), cmap='gray')

plt.show()
```

(b) **[5 points]** One way to understand the meaning of latent representations is to see which parts of the latent space correspond to which kinds of data. Here we'll produce a scatter plot in the latent space, where each point in the plot represents a different image in the training set.

(a) Encode each image in the training set.

(b) Take the 2D mean vector of each encoding $q_\phi(z|x)$.

(c) Plot these mean vectors in the 2D latent space with a scatterplot.

(d) Colour each point according to the class label (0 to 9).

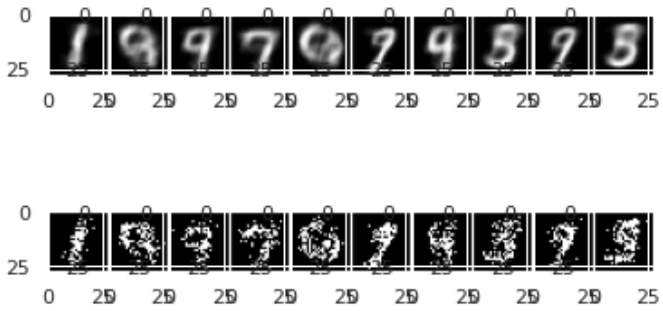

Figure 2: Q3(a) Bernoulli mean vs. Sampled binary

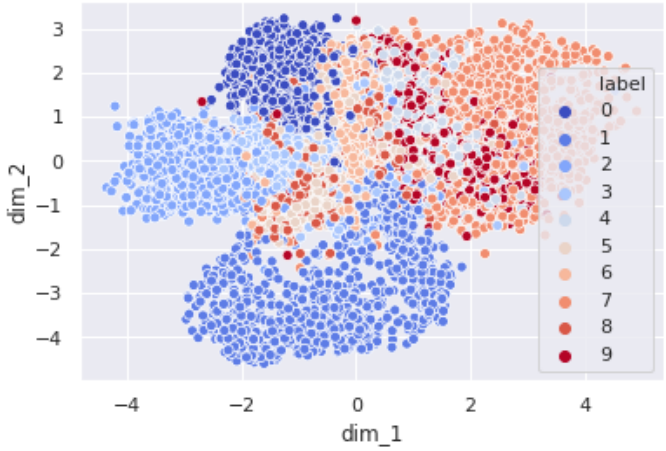

Figure 3: Q3(b) Latent space

Hopefully our latent space will group images of different classes, even though we never provided class labels to the model!

```
import seaborn as sns; sns.set()
import pandas as pd
# labels that were not one-hot encoded
_, train_labels_cat, _, test_labels_cat = mnist()
seed = npr.RandomState(0)
q_means, q_log_stds = encoder(rec_params, train_x)
mean_arr = np.array(q_means) # dim = 10000 * 2
df = pd.DataFrame()
df['dim_1'] = pd.Series(mean_arr[:,0])
df['dim_2'] = pd.Series(mean_arr[:,1])
df['label'] = pd.Series(train_labels_cat)
ax = sns.scatterplot(x="dim_1", y="dim_2", hue="label",
                     data=df, legend="full", palette="coolwarm")
```

(c) **[5 points]** Another way to examine a latent variable model with continuous latent variables is to interpolate between the latent representations of two points.

Here we will encode 3 pairs of data points with different classes. Then we will linearly interpolate between the mean vectors of their encodings. We will plot the generative distributions along the linear interpolation.

(a) First, write a function which takes two points z_a and z_b, and a value $\alpha \in [0, 1]$, and outputs the linear interpolation $z_\alpha = \alpha z_a + (1 - \alpha) z_b$.

(b) Sample 3 pairs of images, each having a different class.

(c) Encode the data in each pair, and take the mean vectors

(d) Linearly interpolate between these mean vectors

(e) At 10 equally-space points along the interpolation, plot the Bernoulli means $p(x|z_\alpha)$

(f) Concatenate these plots into one figure.

```
random.seed(0)
def linear_interpolate(za, zb, alpha):
    res = alpha * np.array(za) + (1-alpha) * np.array(zb)
    return res

def sample_images(num_pairs): # of different classes
    pair_ids = []
    for i in range(num_pairs):
        id1 = random.randrange(0, len(train_x))
        id2 = random.randrange(0, len(train_x))
        while (train_labels_cat[id1] == train_labels_cat[id2]):
            id1 = random.randrange(0, len(train_x))
        pair_ids.extend([id1,id2])
    return pair_ids

pair_ids = sample_images(3)
q_means, q_log_stds = encoder(rec_params, train_x)
q_means_list, q_log_stds_list = [],[]
# [mu_1a, mu_1b, mu_2a, ..., mu_3b]
for i in range(6):
    q_means_list.append(q_means[pair_ids[i]])
    q_log_stds_list.append(q_log_stds[pair_ids[i]])

AX = gridspec.GridSpec(3,10)
AX.update(wspace = 0.0, hspace = 0.0)
alpha = np.linspace(0, 1, num=10)
seed = npr.RandomState(0)

for j in range(0,3):
    # sample z from variational distribution
    za = sample_diag_gaussian(q_means_list[2*j], q_log_stds_list[2*j], seed)
    zb = sample_diag_gaussian(q_means_list[2*j+1], q_log_stds_list[2*j+1], seed)

    for i in range(0,10):
        z_alpha = linear_interpolate(za, zb, alpha[i])
        # logits of the Bernoulli means
        logits = decoder(z_alpha, gen_params)

        ber_mean = 1/(1 + np.exp(-logits))

        plt.subplot(AX[j,i])
        plt.imshow(ber_mean.reshape(28,28), cmap='gray')

plt.show()
```

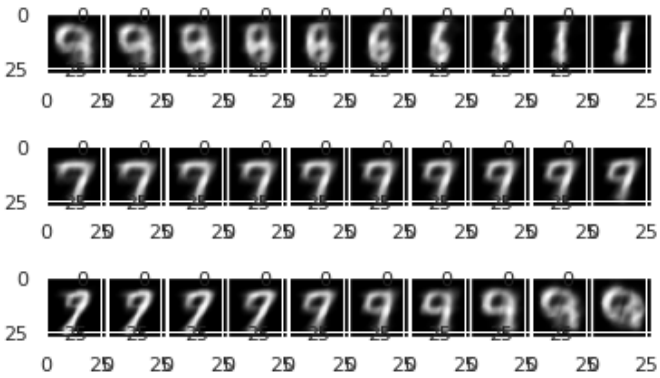

Figure 4: Q3(c) Linear interpolation

4 Predicting the Bottom of Images given the Top [15 points]

Now we'll use the trained generative model to perform inference for $p(z|\text{top half of image x})$. Unfortunately, we can't re-use our recognition network, since it can only input entire images. However, we can still do approximate inference without the encoder.

To illustrate this, we'll approximately infer the distribution over the pixels in the bottom half an image conditioned on the top half of the image:

$$p(\text{bottom half of image x}|\text{top half of image x}) = \int p(\text{bottom half of image x}|z)p(z|\text{top half of image x})dz$$

To approximate the posterior $p(z|\text{top half of image x})$, we'll use stochastic variational inference.

(a) **[5 points]** Write a function that computes $p(z, \text{top half of image x})$

(a) First, write a function which returns only the top half of a 28x28 array. This will be useful for plotting, as well as selecting the correct Bernoulli parameters.

(b) Write a function that computes $\log p(\text{top half of image x}|z)$. Hint: Given z, the likelihood factorizes, and all the unobserved dimensions of x are leaf nodes, so can be integrated out exactly.

(c) Combine this likelihood with the prior to get a function that takes an x and an array of zs, and computes the log joint density $\log p(z, \text{top half of image x})$ for each z in the array.

```
def top_half(arr, h = 28, w = 28):
    # input: 2d array (h,w)
    # lower half zero padding
    # output: 2d array (h,w)
    upper = arr[0:int(h/2),:]
    lower = np.zeros([h-int(h/2),w])
    return np.concatenate((upper,lower), axis = 0)

def log_p_top_half_given_z(image, z, gen_params):
    # image --> single image
    # z --> single z with shape (2,)
    top_half_x = top_half(image.reshape(28,28)).reshape(784,)
    return log_likelihood(top_half_x, z, gen_params) # scalar

# log p(z, top half of image x)
def joint_log_density_top_half(x, zs, gen_params):
    # zs: an array of (num_samples,2)
```

```
    # output: (num_samples,)
    return np.array([log_prior(zs[i]) + log_p_top_half_given_z(x, zs[i], gen_params)
    for i in range(len(zs))])
```

(b) **[5 points]** Now, to approximate $p(z|\text{top half of image x})$ in a scalable way, we'll use stochastic variational inference. For a digit of your choosing from the training set (choose one that is modelled well, i.e. the resulting plot looks reasonable):

(a) Initialize variational parameters ϕ_μ and $\phi_{\log\sigma}$ for a variational distribution $q(z|\text{top half of } x)$.

(b) Write a function that computes estimates the ELBO over K samples $z \sim q(z|\text{top half of } x)$. Use $\log p(z)$, $\log p(\text{top half of } x|z)$, and $\log q(z|\text{top half of } x)$.

(c) Optimize ϕ_μ and $\phi_{\log\sigma}$ to maximize the ELBO.

(d) On a single plot, show the isocontours of the joint distribution $p(z, \text{top half of image x})$, and the optimized approximate posterior $q_\phi(z|\text{top half of image x})$.

(e) Finally, take a sample z from your approximate posterior, and feed it to the decoder to find the Bernoulli means of $p(\text{bottom half of image x}|z)$. Contatenate this greyscale image to the true top of the image. Plot the original whole image beside it for comparison.

```
# Choose a well-modelled image
print(train_labels_cat[316])
temp = train_x[316].reshape(28,28)
plt.imshow(temp)
image_x = train_x[316]

# Set up plotting code
def plot_isocontours(ax, func, xlimits=[-0.5, 0.5], ylimits=[0.5, 1.5], numticks=101):
    x = np.linspace(*xlimits, num=numticks)
    y = np.linspace(*ylimits, num=numticks)
    X, Y = np.meshgrid(x, y)
    zs = func(np.concatenate([np.atleast_2d(X.ravel()), np.atleast_2d(Y.ravel())]).T)
    Z = zs.reshape(X.shape)
    plt.contour(X, Y, Z)
    ax.set_yticks([])
    ax.set_xticks([])

def plot_2d_fun(f):
    fig = plt.figure(figsize=(8,8), facecolor='white')
    ax = fig.add_subplot(111, frameon=False)
    plot_isocontours(ax, f)
    plt.plot([3, -3], [3, -3], 'b-')
    plt.show(block=True)
    plt.draw()

# svi
def elbo_svi(init_params, logp, num_samples, rs): # single image
    (q_means, q_log_stds) = init_params
    # mean_mat.shape = (num_samples,2)
    mean_mat = np.array([[q_means[0], q_means[1]] for i in range(num_samples)])
    log_std_mat = np.array([[q_log_stds[0], q_log_stds[1]] for i in range(num_samples)])
    # sample z from variational distribution
    latents = seed.randn(num_samples,*q_means.shape) * np.exp(log_std_mat) + mean_mat

    # joint log likelihood of z and x under model
```

```
    joint_ll = logp(latents) # (num_samples,)
    # log likelihood of z under variational distribution
    log_q_z = log_q(mean_mat, log_std_mat, latents) # (num_samples,)
    elbo_estimate = np.sum(joint_ll - log_q_z) / num_samples # scalar
    return elbo_estimate

# Conveinence function for taking gradients
def neg_elbo_svi(params, data = image_x, gen_params = gen_params, rs = seed,
num_samples = 10000):
    # TODO: Write a function that takes parameters for q,
    # evidence as an array of game outcomes,
    # and returns the -elbo estimate with num_samples many samples from q
    def logp(latents):
        return joint_log_density_top_half(data, latents, gen_params)
    return -elbo_svi(params, logp, num_samples, rs)

def train_model_params_svi(data = image_x, gen_params = gen_params):
    # Define training objective
    def objective(params, t): # to be minimized
        return neg_elbo_svi(params)

    objective_grad = grad(objective)

    # Set up figure.
    fig = plt.figure(figsize=(8,8), facecolor='white')
    ax = fig.add_subplot(111, frameon=False)
    plt.ion()
    plt.show(block=False)

    def callback(params, t, g):
        print("Iteration {} negative elbo {}".format(t, objective(params,t)))

        plt.cla()
        def target_distribution(zs):
            return np.exp(joint_log_density_top_half(data, zs, gen_params))
        plot_isocontours(ax, target_distribution)

        (mean, log_std) = params
        def variational_contour(z):
            return np.exp(log_q(mean, log_std, z))
        plot_isocontours(ax, variational_contour)
        plt.draw()
        plt.pause(1.0/30.0)

    print("Optimizing variational parameters...")
    seed = npr.RandomState(0)
    init_mu = seed.randn(2) # Initial mu: (2,)
    init_ls = seed.randn(2) # Initial log_sigma: (2,)
    init_params = (init_mu, init_ls)

    # The optimizers provided can optimize lists, tuples, or dicts of parameters.
    optimized_params = adam(objective_grad, init_params, step_size=0.01,
    num_iters=1000, callback=callback)
```

Figure 5: Q4(b) SVI contour

```
    return optimized_params

optimized_params_svi = train_model_params_svi(data = image_x, gen_params = gen_params)
(mean_svi, log_std_svi) = optimized_params_svi

Iteration 100 negative elbo 153.48670501154268

# the isocontours of the joint distribution p(z; top half of image x)
# and the optimized approximate posterior q(z|top half of image x)
fig = plt.figure(figsize=(8,8), facecolor='white')
ax = fig.add_subplot(111, frameon=False)
plt.ion()
plt.show(block=False)
def target_distribution(zs):
    return np.exp(joint_log_density_top_half(image_x, zs, gen_params))
plot_isocontours(ax, target_distribution)

def variational_contour(z):
    return np.exp(log_q(mean_svi, log_std_svi, z))
plot_isocontours(ax, variational_contour)
plt.title('p(z; top half of image x) and q(z|top half of image x)')
plt.xlabel('Latent 1')
plt.ylabel('Latent 2')
plt.draw()

# a sample z from the trained approximate posterior
seed = npr.RandomState(0)
z = sample_diag_gaussian(mean_svi, log_std_svi, seed)
# decoder gives Bernoulli means of p(bottom half of image x|z)
logit = decoder(z, gen_params)
# sigmoid is the inverse of logit
ber_mean = 1/(1 + np.exp(-logit))
# Contatenate this greyscale image to the true top of the image
plt.subplot(1, 2, 1)
plt.imshow(concatenated.reshape(28,28), cmap='gray')
```

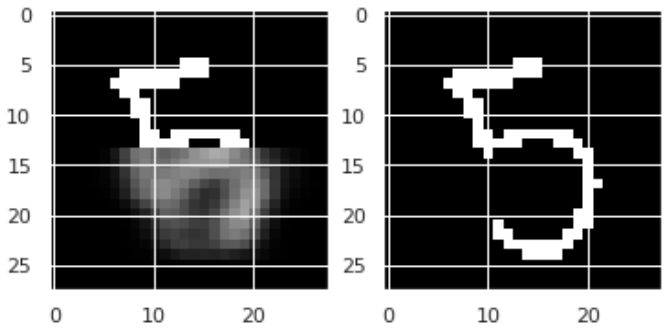

Figure 6: Q4(b) Predicted bottom given top

```
plt.subplot(1, 2, 2)
plt.imshow(image_x.reshape(28,28), cmap='gray')
```

(c) **[5 points]** True or false: Questions about the model and variational inference.
There is no need to explain your work in this section.

(a) Does the distribution over $p(\text{bottom half of image } x|z)$ factorize over the pixels of the bottom half of image x?
Answer: Yes.

(b) Does the distribution over $p(\text{bottom half of image } x|\text{top half of image } x)$ factorize over the pixels of the bottom half of image x?
Answer: No.

(c) When jointly optimizing the model parameters θ and variational parameters ϕ, if the ELBO increases, has the KL divergence between the approximate posterior $q_\phi(z|x)$ and the true posterior $p_\theta(z|x)$ necessarily gotten smaller?
Answer: No.

(d) If $p(x) = \mathcal{N}(x|\mu, \sigma^2)$, for some $x \in \mathbb{R}, \mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$, can $p(x) < 0$?
Answer: No.

(e) If $p(x) = \mathcal{N}(x|\mu, \sigma^2)$, for some $x \in \mathbb{R}, \mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$, can $p(x) > 1$?
Answer: Yes.