Assignment 3: Variational Autoencoders

STA414/STA2014 and CSC412/CSC2506 Winter 2020
David Duvenaud and Jesse Bettencourt
Janet Wang 1003337980

April 19, 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_gaaussian` 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_x` 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 $\log p(z)$.

```
# log_prior: compute log of the prior over a digit's representation z
log_prior(z) = factorized_gaussian_log_density(0,0,z)
```

(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 multi-layer 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".

```
## Model Dimensionality
Dz, Dh = 2, 500
Ddata = 28^2
## Generative Model
decoder = Chain(Dense(Dz, Dh, tanh), Dense(Dh, Ddata))
```

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

```
### log_likelihood: given a latent representation $z$ and a binarized digit $x$,
# computes the log-likelihood $\log p(x|z)$.
function log_likelihood(x,z)
    """ Compute log likelihood log_p(x|z)"""
    # return likelihood for each element in batch
    p = decoder(z)
    # @info p, x
    return sum(bernoulli_log_density(p,x),dims=1)
end
```

(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.

```
### joint_log_density: combines log-prior and log-likelihood of the observations
# to give $\log p(z, x)$ for a single image.
joint_log_density(x,z) = log_prior(z) .+ log_likelihood(x,z)
```

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.

```
### encoder: evaluates MLP to give mean and log-standard deviation of a
# factorized Gaussian with Dim D_z=2.
# MLP uses a single hidden layer with 500 hidden units and a tanh nonlinearity
encoder = Chain(Dense(Ddata, Dh, tanh), Dense(Dh, Dz*2), unpack_gaussian_params)
```

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

```
### log_q: write log likelihood under variational distribution.
log_q(q_mu, q_logsig, z) = factorized_gaussian_log_density(q_mu, q_logsig, z)
```

(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.

```
### elbo: computing unbiased estimate of the elbo on a batch of images xs
function elbo(x)
    # variational parameters from data
    q_mu, q_logsigma = encoder(x)
    # sample from variational distribution
    z = sample_diag_gaussian(q_mu, q_logsigma)
    # joint likelihood of z and x under model
    joint_ll = joint_log_density(x,z)
    # likelihood of z under variational distribution
    log_q_z = log_q(q_mu, q_logsigma, z)
    # Scalar value, mean variational evidence lower bound over batch
    elbo_estimate = mean(joint_ll - log_q_z, dims=2)
    return elbo_estimate[1]
end
```

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

```
### loss: gives negative elbo estimate over batch of data x
function loss(x)
    # scalar value for the variational loss over elements in the batch
```

```
    return -elbo(x)
end
```

(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.

```
### train_model_params: initializes and optimizes the encoder and decoder params
# jointly on the training set. Optimizes with gradients on the elbo estimate over
# batches of data, not the whole dataset. Trains for 10 epochs by default.
# Training with gradient optimization:
# See example_flux_model.jl for inspiration
function train_model_params!(loss, encoder, decoder, train_x, test_x; nepochs=10)
    # model params: parameters to update with gradient descent
    ps = Flux.params(encoder, decoder)
    # ADAM optimizer with default parameters
    opt = ADAM()
    # over batches of the data
    for i in 1:nepochs
        for d in batch_x(train_x)
            # compute gradients with respect to variational loss over batch
            # first argument is an anonymous function
            gs = Flux.gradient(() -> loss(d), ps)
            # update the parameters with gradients
            Flux.Optimise.update!(opt, ps, gs)
        end
        if i%1 == 0 # change 1 to higher number to compute and print less frequently
            @info "Test loss at epoch $i: $(loss(batch_x(test_x)[1]))"
        end
    end
    @info "Parameters of encoder and decoder trained!"
end
```

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.

```
### 3a: plot samples from the trained generative model using ancestral sampling
num_samples = 10

# sample a z from the prior.
sample_z = randn((2,num_samples))
# compute the bernoulli means over the pixels of $x$ given $z$ using the generative
    model.
bernoulli_means = 1.0 ./ (1.0 .+ exp.(-decoder(sample_z)))
# Sample a binary image $x$ from this product of Bernoullis. Plot this sample as an
    image.
binary_sample = sample_bernoulli(bernoulli_means)

# plots of bernoulli means of p(x|z) for each sample of z
plots_bernoulli = [plot(mnist_img(bernoulli_means[:,i])) for i in 1:num_samples]
# binary images sampled from the bernoulli distribution
plots_binary =  [plot(mnist_img(binary_sample[:,i])) for i in 1:num_samples]
# each column is an independent sample
plots = [ plots_bernoulli; plots_binary ]

#### plot and save 3a
display(plot(
    plots...,
    layout = grid(2,num_samples),
    size = (125*num_samples, 125*2)
))
savefig(joinpath("plots","3a.png"))
```

Final test set loss: 154.92761753905447 Final ELBO on the test set: -154.87946244674814

(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).

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

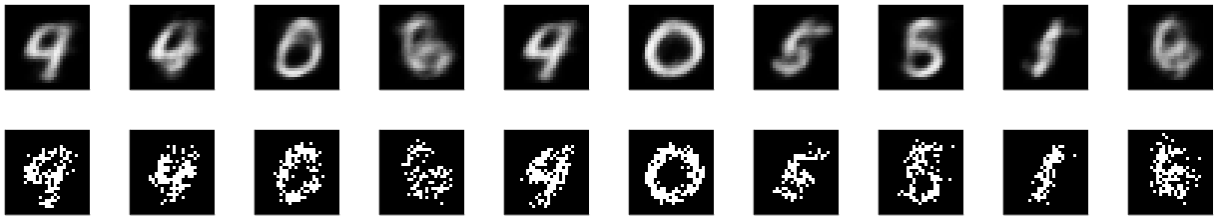

Figure 1: Generative samples from the trained generative model using ancestral sampling. Top: Bernoulli means. Bottom: Binarized sample image.

```
### 3b scatter plot where each point represents an image in the training set
vector_x = [[] for i in 1:10]
vector_y = [[] for i in 1:10]
for i in 1:size(train_x)[2]
    # encode each image in the trianing set
    q_mu, q_logsigma = encoder(train_x[:,i])
    # take 2D mean vector of each encoding
    push!(vector_x[1 + train_label[i]], q_mu[1])
    push!(vector_y[1 + train_label[i]], q_mu[2])
end

#### plot and save 3b
display(plot(
    vector_x,
    vector_y,
    seriestype = :scatter,
    title = "Latent Space Encoding from MNIST Training Set",
    xlabel = "z_1",
    ylabel = "z_2",
    label = ["0" "1" "2" "3" "4" "5" "6" "7" "8" "9"],
    size = (800, 800)
))
savefig(joinpath("plots","A3Q3b.png"))
```

See plot in Figure 2 at the end of the document.

(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.

```
### 3c visualizing generative output along linear interpolations between mean
# vectors of 3 pairs of encoded data points with different classes to examine
```

```
# the latent variable model

# get linear interpolation of 2 points
function linear_interpolation(za, zb, alpha)
    zalpha = alpha .* za + (1 - alpha) .* zb
    return zalpha
end

# sample 3 pairs of images, each with a different class
sample_images = [
    # 5, 0
    (train_x[:,1], train_x[:,2]),
    # 4, 1
    (train_x[:,3], train_x[:,4]),
    # 9, 2
    (train_x[:,5], train_x[:,5]),
]

# encode the data in each pair and take the mean vectors
encoded_sample_images = [
    (encoder(sample_images[i][1]), encoder(sample_images[i][2]))
    for i in 1:3
]

# make plots of the means at 10 equally spaced points
plots = []
for i in 1:3
    encoded_a, encoded_b = encoded_sample_images[i]
    for j in 1:10
            alpha = j/10.0
            meanalpha = linear_interpolation(encoded_a[1], encoded_b[1], alpha)
            # get Bernoulli means
            logit_mean = decoder(meanalpha)
            # plot means
            ber_mean = exp.(logit_mean) ./ (1 .+ exp.(logit_mean))
            push!(plots, plot(mnist_img(ber_mean[:])))
    end
end

#### plot and save
display(plot(
    plots...,
    layout=grid(3,10),
    size =(850, 250),
    # layout=grid(10, 3),
    # size =(500, 1700),
    axis = nothing
))
savefig(joinpath("plots","A3Q3c.png"))
```

See Figure 3 at the end of the document.

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.

```
#### top_half: gets top half of a 28x28 array
function top_half(x)
    half_image = x[1:14*28, :]
    return half_image
end
```

(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.

```
#### log_likelihood_top_half: computes log p(top half of image x| z)
function log_likelihood_top_half(top_half_x, z)
    p = decoder(z)
    half_p = top_half(p)
    # sums top half
    sum(bernoulli_log_density(half_p, top_half_x), dims=1)
end
```

(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.

```
#### joint_log_density_top_half: compute log p(z, top half of image x)
# compute 3 log densities for the elbo: log p(z), log p(x|z), and log q(z).
# log p(z) and log q(z) are both 2-dimensional Gaussians.
joint_log_density_top_half(z, top_half_x) =
    log_prior(z) .+ log_likelihood_top_half(top_half_x, z)
```

(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 $q(z|\text{top half of } x)$.

```
phi_mu = randn(2, 10)
phi_logsigma = randn(2, 10)
```

(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 $\log q(z|\text{top half of } x)$.

```
#### elbo_svi: gets elbo over K samples $z ~ q(z|top half of x)$
function elbo_svi(params, logp)
```

```
    # needs to be (2, k)
    phi_mu, p_logsigma = params
    # @info "elbo svi: $(size(phi_mu)) $(size(p_logsigma))"
    # needs to be (2, k)
    z = sample_diag_gaussian(phi_mu, p_logsigma)
    # @info "elbo svi samples z: $(size(z))"
    joint_ll = logp(z)
    # @info "joint_ll: $(size(joint_ll))"
    # likelihood of z under variational distribution
    log_q_z = log_q(phi_mu, p_logsigma, z)
    # return sum(logp_estimate - logq_estimate) / k
    elbo_estimate = mean(joint_ll - log_q_z, dims=2)
    # Scalar value, mean variational evidence lower bound over batch
    # @info "elbo_estimate: $(elbo_estimate[1])"
    return elbo_estimate[1]
end

#### neg_elbo_svi: Takes parameters for q, evidence as an array of game outcomes,
# and returns the -elbo estimate with k many samples from q
function neg_elbo_svi(params; x = train_x)
    # print(size(x))
    half_x = top_half(x)
    function logp(z)
        return joint_log_density_top_half(z, half_x)
    end
    return -elbo_svi(params, logp)
end
```

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

```
#### contours
function contours!(f; colour=nothing)
    n = 100
    x = range(-1,stop=1,length=n)
    y = range(0,stop=1,length=n)
    z_grid = Iterators.product(x,y) # meshgrid for contour
    z_grid = reshape.(collect.(z_grid),:,1) # add single batch dim
    z = f.(z_grid)
    z = getindex.(z,1)'
    max_z = maximum(z)
    levels = [.99, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2] .* max_z
    if colour==nothing
    p1 = contour!(x, y, z, fill=false, levels=levels)
    else
    p1 = contour!(x, y, z, fill=false, c=colour,levels=levels,colorbar=false)
    end
    plot!(p1)
end

### train_model_params_svi: optimize phi_mu and phi_logsigma
function train_model_params_svi!(init_params, data; num_itrs=200, lr= 1e-2)
    @info "Training model params using SVI"
    params_cur = init_params
    for i in 1:num_itrs
        # gradients of variational objective with respect to parameters
        grad_params = gradient(
            params -> neg_elbo_svi(params, x = data), params_cur
        )[1]
        # update paramters with lr-sized step in descending gradient
```

```
            params_cur = (
                params_cur[1] - lr * grad_params[1],
                params_cur[2] - lr * grad_params[2]
            )
            # report the current elbo during training
            if i%10 == 0 # change 1 to higher number to compute and print less
                frequently
                @info "Loss at iter $i: $(neg_elbo_svi(params_cur, x = data))"

                # plot true posterior in red and variational in blue
                # hint: call 'display' on final plot to make it display during training
                plot(
                title="Train model params using SVI: iteration $i");
                # plot likelihood contours for target posterior
                contours!(zs -> exp.(joint_log_density_top_half(zs, data)),
                    colour=:red)
                # display(skillcontour!(..., colour=:blue)) plot likelihood contours
                    for variational posterior
                display(
                contours!(
                    zs -> factorized_gaussian_log_density(params_cur[1],
                        params_cur[2], zs),
                    colour=:blue
                )
                )
            end
        end
        return params_cur
    end
```

(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 $q_\phi(z|\text{top half of image x})$.

```
### set up training model
function set_up_svi(num_samples, x = train_x)
    train = x[:, 1:num_samples]
    half_train = top_half(train)
    phi_mu = randn(2, num_samples)
    phi_logsigma = randn(2, num_samples)
    return half_train, phi_mu, phi_logsigma
end

#### train the model
half_train, phi_mu, phi_logsigma = set_up_svi(10000)
params_svi = train_model_params_svi!((phi_mu, phi_logsigma), half_train;
    num_itrs=200, lr= 1e-2)
```

See Figure 4 at end of document

(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.

```
# TODO
```

(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: False

(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: True

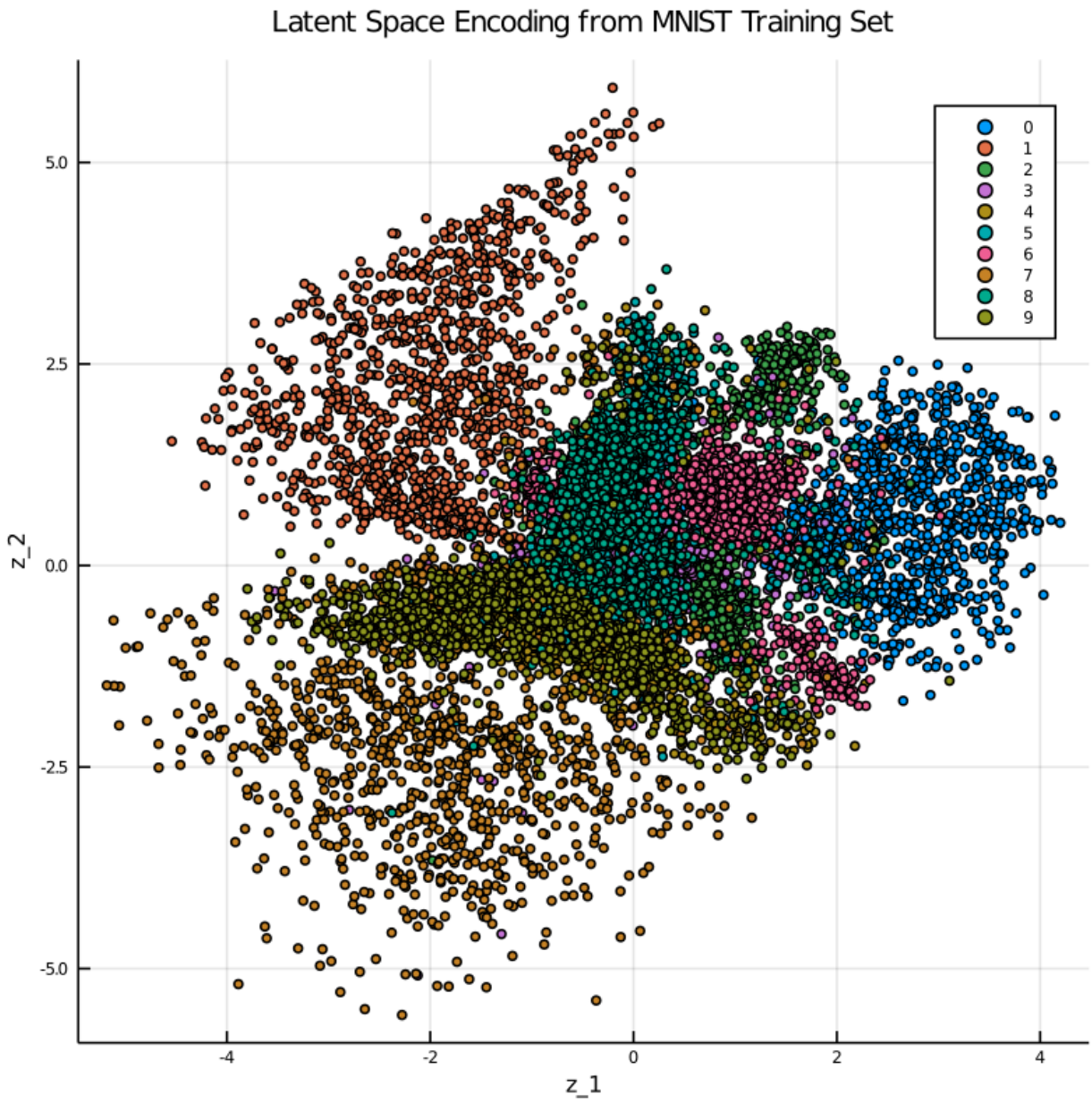

Figure 2: Latent space encoding from MNIST training set

Figure 3: Visualizing generative output along linear interpolations between mean vectors of 3 pairs of encoded data points with different classes

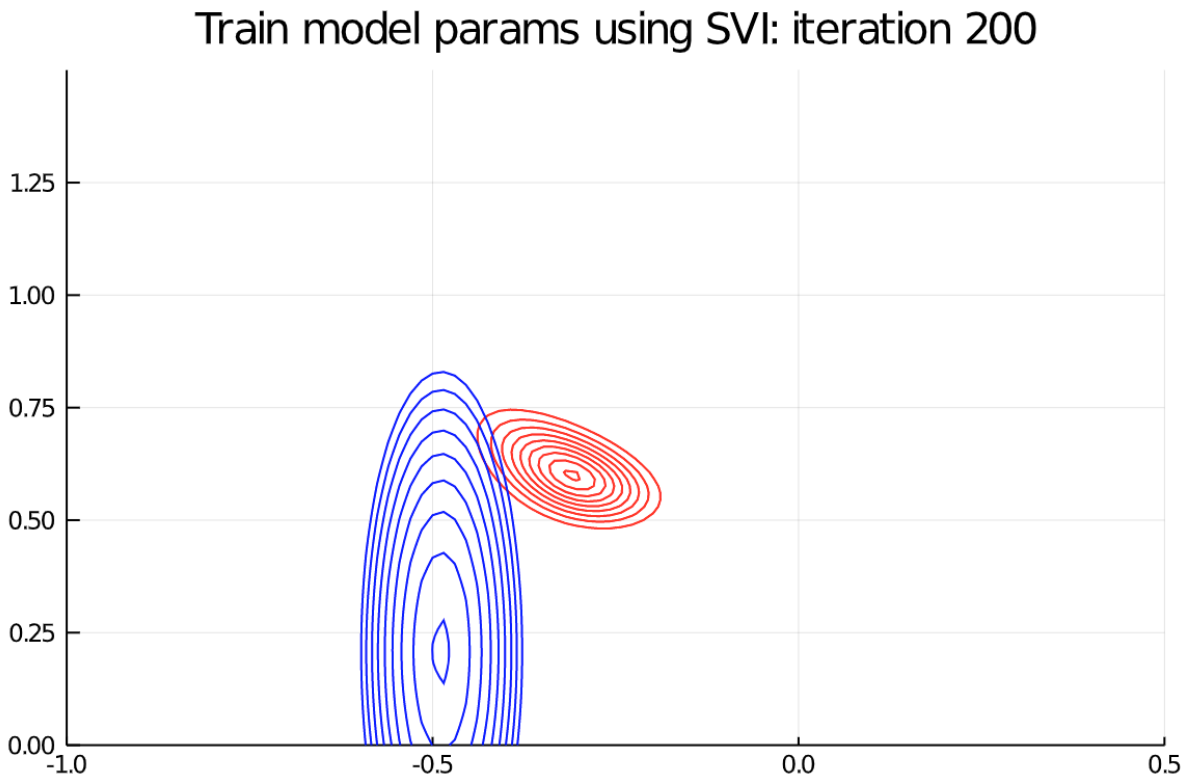

Figure 4: Isocontours of joint distribution (red) and optimized approximate posterior (blue)