Variational Autoencoders on MNIST digits

Amr Mohamed

January 25, 2022

1 Background

In this article, I reproduce the results in Kingma and Welling (2013) implementing a Variational Autoencoder (Auto-Encoding Variational Bayes) to reconstruct MNIST digits.

The article is mathematically and code heavy. The highlights are as such:

1. In Section 4, I plot MNIST digit characterization in the autoencoders latent space illustrating that the trained model is able to group (cluster) MNIST images according to their class (label)

2. In Section 4, we use this latent space representation to see what 'interpolating' between numbers looks like. For example, we generate numbers between 0 and 5, and get something like 0 -> 6 -> 3 -> 5.

3. In Section 5, we reconstruct a number image given only its top half; these are the most exciting results.

Tools. I use `Flux.jl` for automatic differentation.

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 the number.

1.1 Model Definition

Prior. The prior over each digit's latent representation is a multivariate standard normal distribution. The dimension of the latent space D_z to 2. The two-dimensional latent space is employed to enable visualization.

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 $f_\theta(z)$:

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

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

Starter functions

```julia
using Flux
using MLDatasets
using Statistics
using Logging
using Test
using Random
using Statistics: mean
using Distributions: pdf, Normal
using Zygote: gradient
using StatsFuns: log1pexp
Random.seed!(412414);

#### Probability Stuff
# log-pdf of x under Factorized or Diagonal Gaussian N(x|μ,σI)
function factorized_gaussian_log_density(mu, logsig,xs)
  """
  mu and logsig either same size as x in batch or same as whole batch
  returns a 1 x batchsize array of likelihoods
  """
  σ = exp.(logsig)
  return sum((-1/2)*log.(2π*σ.^2) .+ -1/2 * ((xs .- mu).^2)./(σ.^2),dims=1)
end;

# log-pdf of x under Bernoulli
function bernoulli_log_density(logit_means,x)
  """Numerically stable log_likelihood under bernoulli by accepting μ/(1-μ)"""
  b = x .* 2 .- 1 # {0,1} -> {-1,1}
  return - log1pexp.(-b .* logit_means)
end;

@testset "test stable bernoulli" begin
  using Distributions
  x = rand(10,100) .> 0.5
  μ = rand(10)
  logit_μ = log.(μ./(1 .- μ))
  @test logpdf.(Bernoulli.(μ),x) ≈ bernoulli_log_density(logit_μ,x)
  # over i.i.d. batch
  @test sum(logpdf.(Bernoulli.(μ),x),dims=1) ≈
sum(bernoulli_log_density(logit_μ,x),dims=1)
end;

Test Summary:          | Pass  Total
test stable bernoulli |    2      2

# sample from Diagonal Gaussian x~N(μ,σI)
sample_diag_gaussian(μ,logσ) = (ϵ = randn(size(μ)); μ .+ exp.(logσ).*ϵ)
# sample from Bernoulli
sample_bernoulli(θ) = rand.(Bernoulli.(θ))

# Load MNIST data, binarise it, split into train and test sets (10000 each)
# and partition train into mini-batches of M=100.
function load_binarized_mnist(train_size=10000, test_size=10000)
  train_x, train_label = MNIST.traindata(1:train_size);
  test_x, test_label = MNIST.testdata(1:test_size);
  @info "Loaded MNIST digits with dimensionality $(size(train_x))"
  train_x = reshape(train_x, 28*28,:)
  test_x = reshape(test_x, 28*28,:)
  @info "Reshaped MNIST digits to vectors, dimensionality $(size(train_x))"
  train_x = train_x .> 0.5; #binarize
  test_x = test_x .> 0.5; #binarize
```

```julia
  @info "Binarized the pixels"
  return (train_x, train_label), (test_x, test_label)
end;

function batch_data((x,label)::Tuple, batch_size=100)
  """
  Shuffle both data and image and put into batches
  """
  N = size(x)[end] # number of examples in set
  rand_idx = shuffle(1:N) # randomly shuffle batch elements
  batch_idx = Iterators.partition(rand_idx,batch_size) # split into batches
  batch_x = [x[:,i] for i in batch_idx]
  batch_label = [label[i] for i in batch_idx]
  return zip(batch_x, batch_label)
end
# batch xs
batch_x(x::AbstractArray, batch_size=100) =
first.(batch_data((x,zeros(size(x)[end])),batch_size));
```

2 Implementing the Model

```julia
## Load the Data
train_data, test_data = load_binarized_mnist();
train_x, train_label = train_data;
test_x, test_label = test_data;

## Test the dimensions of loaded data
@testset "correct dimensions" begin
@test size(train_x) == (784,10000)
@test size(train_label) == (10000,)
@test size(test_x) == (784,10000)
@test size(test_label) == (10000,)
end;
```

```
Test Summary:       | Pass  Total
correct dimensions |    4      4
```

```julia
## Model Dimensionality
# #### Set up model (using Bernoulli decoder for Binarized MNIST)
# Set latent dimensionality=2 and number of hidden units=500.
Dz, Dh = 2, 500;
Ddata = 28^2;
```

After having loaded the data above, here I implement a function `log_prior` that computes the log of the prior over a digit's representation $\log p(z)$.

```julia
log_prior(z) = factorized_gaussian_log_density(zeros(Dz), zeros(Dz), z);
```

Next, I implement a function `decoder` that, given a latent representation z and a set of neural network parameters θ, produces a 784-dimensional mean vector of a product of Bernoulli distributions, one for each pixel in a 28×28 image. The decoder architecture is 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 $\mu \in [0, 1]$, I output $\log\left(\frac{\mu}{1-\mu}\right) \in \mathbb{R}$ (i.e. the "logit").

I am coding the decoder so that it outputs the means. Later, I will code the bits needed to hand over the logits to any Bernoulli distribution handling functions. It is a personal preference

```
decoder = Chain(Dense(Dz, Dh, tanh), Dense(Dh, Ddata, σ));
```

Next, I implement a function `log_likelihood` that, 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)"""
  θ =  decoder(z) # parameters decoded from latent z
  return sum(bernoulli_log_density(log.(θ ./ (1 .- θ)),x),dims=1) # return likelihood
for each element in batch
end;
```

Next, I 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(x,z) = log_likelihood(x,z) + log_prior(z);
```

All of the functions in this section will 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.

3 Amortized Approximate Inference and training

```
function unpack_gaussian_params(θ)
  μ, logσ = θ[1:Dz,:], θ[Dz+1:end,:]
  return  μ, logσ
end;
```

Here, I 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 $D_z = 2$. The encoder architecture is a multi-layer perceptron (i.e. a fully-connected neural network) with a single hidden layer with 500 hidden units, and a `tanh` nonlinearity. Again, the function is evaluated in parallel on a batch of images, using the same parameters ϕ for each image.

```
encoder = Chain(Dense(Ddata, Dh, tanh), Dense(Dh, 2*Dz));
```

Next, I implement a function `log_q` that given the parameters of the variational distribution, evaluates the log likelihood of z.

```
log_q(q_μ, q_logσ, z) = factorized_gaussian_log_density(q_μ, q_logσ, z);
```

Next, I 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`, a batch of B images, $D_x \times B$.
- `encoder_params`, the parameters ϕ of the encoder (recognition network).
- `decoder_params`, the parameters θ of the decoder (likelihood).

This function returns a single scalar.

```
function elbo(x)
  q_μ, q_logσ = unpack_gaussian_params(encoder(x)) # variational parameters from data
  z = sample_diag_gaussian(q_μ,q_logσ) # sample from variational distribution
  joint_ll = joint_log_density(x,z) # joint likelihood of z and x under model
  log_q_z = log_q(q_μ, q_logσ, z) # likelihood of z under variational distribution
  elbo_estimate = mean(joint_ll .- log_q_z) # Scalar value, mean variational evidence
lower bound over batch
  return elbo_estimate
end;
```

Next, I write the loss function `loss` which returns the negative elbo estimate over a batch of data.

```
function loss(x)
  return -elbo(x) # scalar value for the variational loss over elements in the batch
end;
```

Next, I implement a function that initializes and optimizes the encoder and decoder parameters jointly on the training set.

```
# Training with gradient optimization:
function train_model_params!(loss, encoder, decoder, train_x, test_x; nepochs=100)
  # model params
  ps = Flux.params(encoder, decoder) # parameters to update with gradient descent
  # 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
      gs = Flux.gradient(ps) do
        return loss(d)
      end
      # update the paramters with gradients
      Flux.Optimise.update!(opt,ps,gs)
    end
    if i%10 == 0
      @info "Test loss at epoch $i: $(loss(batch_x(test_x)[1]))"
    end
  end
  @info "Parameters of encoder and decoder trained!"
end;

# Train the model
train_model_params!(loss,encoder,decoder,train_x,test_x, nepochs=100)

# Save the trained model
using BSON:@save
cd(@__DIR__)
@info "Changed directory to $(@__DIR__)"
save_dir = "trained_models"
if !(isdir(save_dir))
  mkdir(save_dir)
  @info "Created save directory $save_dir"
end
@save joinpath(save_dir,"encoder_params.bson") encoder
@save joinpath(save_dir,"decoder_params.bson") decoder
@info "Saved model params in $save_dir"

# Load the trained model!
```

```
# using BSON:@load
# cd(@__DIR__)
# @info "Changed directory to $(@__DIR__)"
# load_dir = "trained_models"
# @load joinpath(load_dir,"encoder_params.bson") encoder
# @load joinpath(load_dir,"decoder_params.bson") decoder
# @info "Successfully loaded model params from $load_dir"

print("Final nELBO loss: $(loss(batch_x(test_x)[1]))")
```

```
Final nELBO loss: 161.83902675755343
```

4 Visualizing Posteriors and Exploring the Model

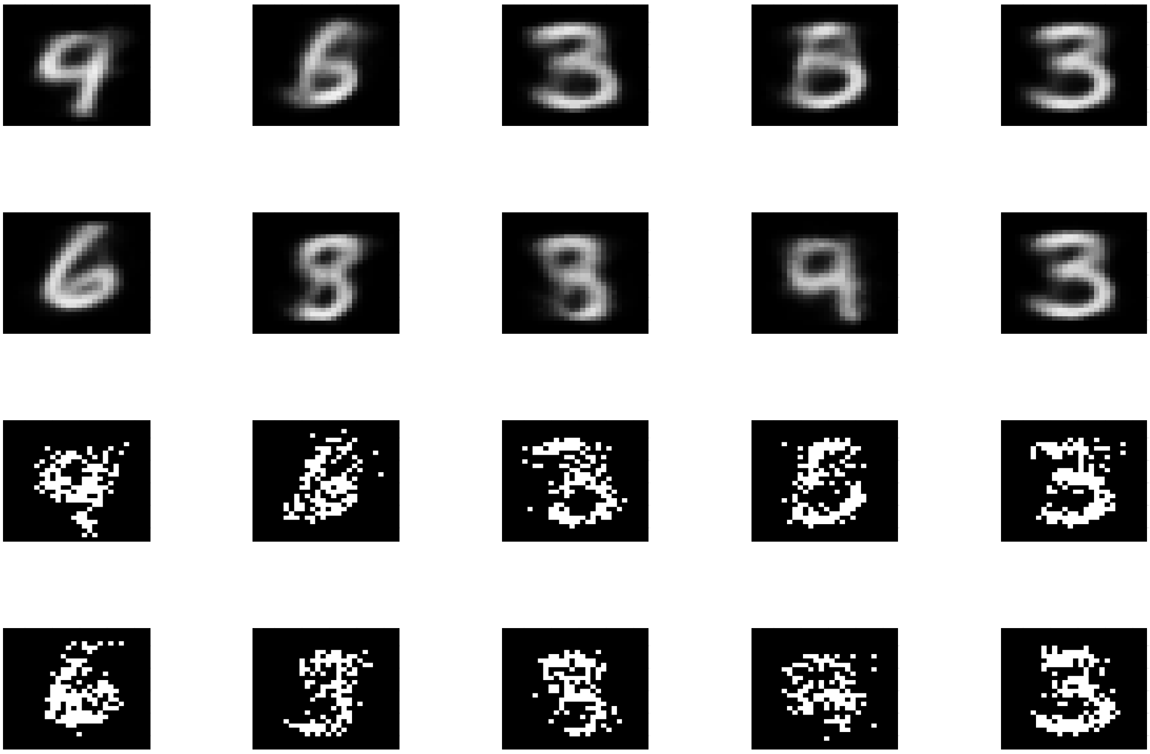
In this section, I investigate the model by visualizing the distribution over data given by the generative model, sampling from it, and interpolating between digits.

Helper functions

```
using Images
using Plots
# make vector of digits into images, works on batches also
mnist_img(x) = ndims(x)==2 ? Gray.(reshape(x,28,28,:)') : Gray.(reshape(x,28,28)');

## helper plotting functions
function skillcontour!(f; colour=nothing, label=nothing)
  n = 100
  x = range(-3,stop=3,length=n)
  y = range(-3,stop=3,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,label=label)
  end
  plot!(p1,legend=true)
end;
```

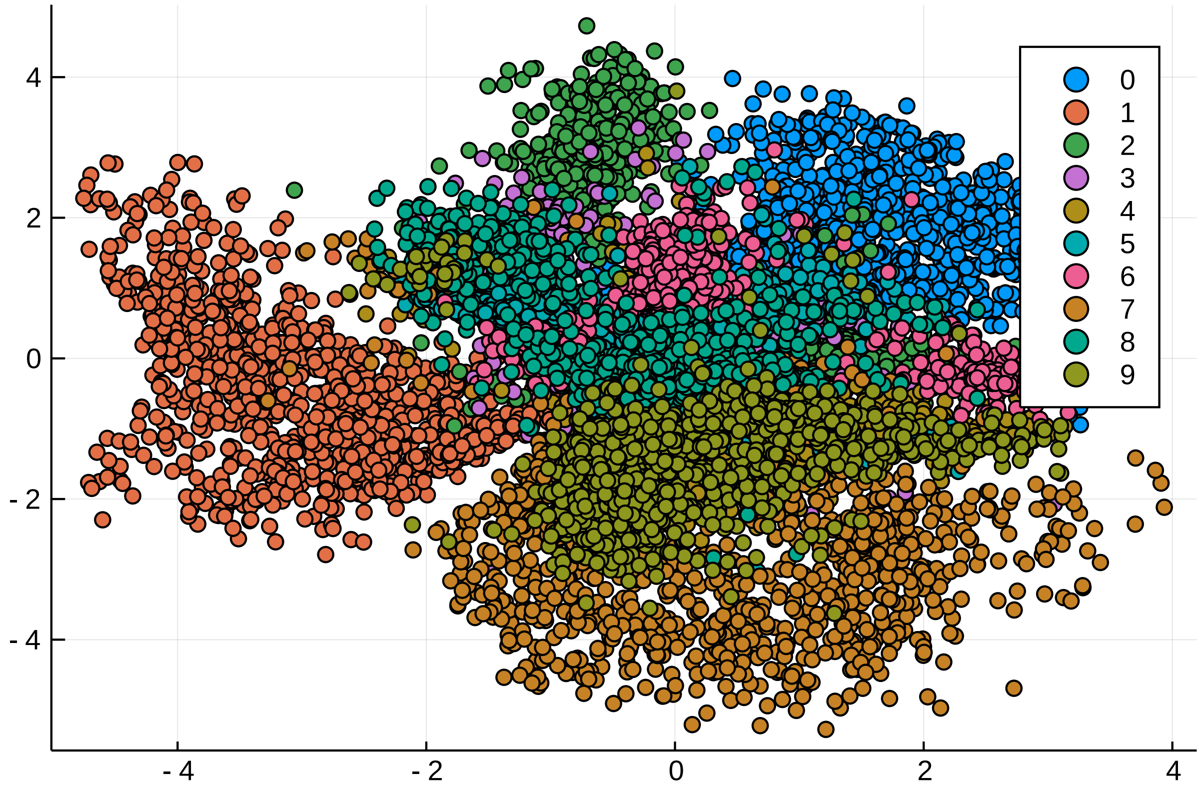
Here, I plot samples from the trained generative model using ancestral sampling:

```
img_array = Any[]
bin_array = Any[]
for i in 1:10
  z = sample_diag_gaussian(zeros(Dz), zeros(Dz))
  gray_img = decoder(z)
  push!(img_array, plot(mnist_img(gray_img), showaxis=false))
  bin_img = sample_bernoulli(gray_img)
  push!(bin_array, plot(mnist_img(bin_img), showaxis=false))
end
img_array = vcat(img_array, bin_array)
display(plot(img_array...));
```



Below is a scatter plot in the latent space, where each point in the plot represents a different image in the training set. Notice that the latent space groups images of different classes, even though we never provided class labels to the model!

```
μ = unpack_gaussian_params(encoder(train_x))[1]
scatter(μ[:,train_label .== 0][1,:], μ[:,train_label .== 0][2,:], label="0")
for i in 1:8
  scatter!(μ[:,train_label .== i][1,:], μ[:,train_label .== i][2,:], label=i)
end;
display(scatter!(μ[:,train_label .== 9][1,:], μ[:,train_label .== 9][2,:], label="9"))
```



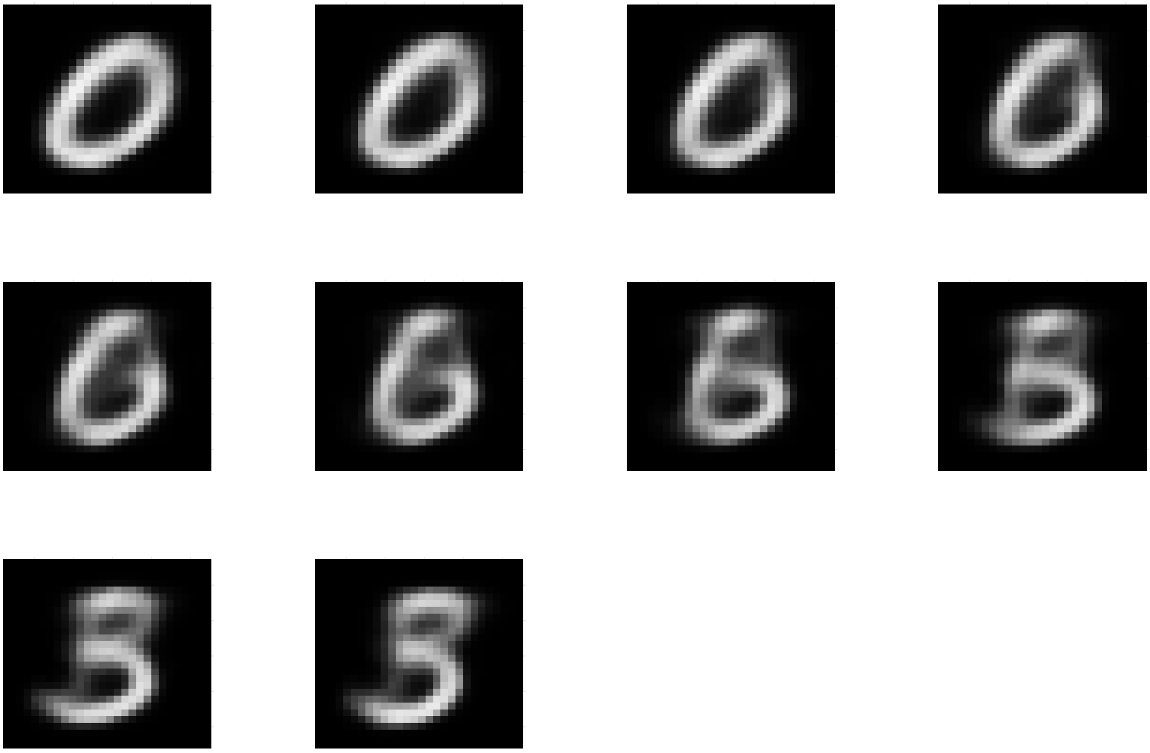
Another way to examine a latent variable model with continuous latent variables is to interpolate between the latent representations of two points.

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

```
zα(za, zb, α) = α.*za + (1-α).*zb;

function lvm_interp(xa, xb)
  za = unpack_gaussian_params(encoder(xa))[1]
  zb = unpack_gaussian_params(encoder(xb))[1]

  plot_array = Any[]
  for α in range(0,stop=1,length=10)
    push!(plot_array, plot(mnist_img(reshape(decoder(zα(za, zb, α)), (Ddata,))),
showaxis=false))
  end
  display(plot(plot_array...))
end;
```

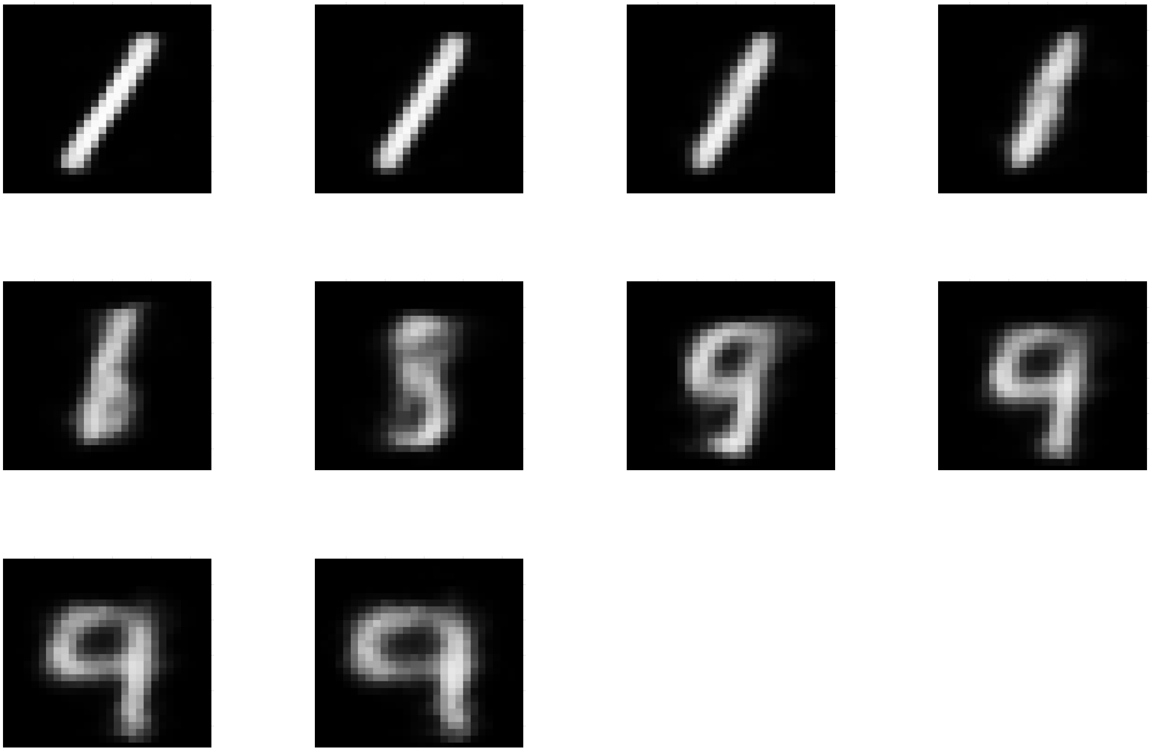
From 0 to 5

```
lvm_interp(train_x[:,1],train_x[:,2]); #0 to 5
```



From 1 to 4

```
lvm_interp(train_x[:,3],train_x[:,4]); #1 to 4
```



From 2 to 9

```
lvm_interp(train_x[:,5],train_x[:,6]); #2 to 9
```

5 Predicting the Bottom of Images given the Top

Now I will use the trained generative model to perform inference for $p(z|\text{top half of image x})$. To illustrate this, I approximately infer the distribution over the pixels in the bottom half an image conditioned on the top half of the image:

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

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

```
top_half(x) = x[1:convert(Int, Ddata/2), :]
bottom_half(x) = x[convert(Int, Ddata/2 + 1):end, :]

function likelihood_top_half(x,z)
    """ Compute the likelihood over the top half of the image.
    x is the top half """
    θ =  top_half(decoder(z))
    return sum(bernoulli_log_density(log.(θ ./ (1 .- θ)),x), dims=1)
end;

joint_log_density_top_half(x,z) = likelihood_top_half(x,z) + log_prior(z);
```

To approximate $p(z|\text{top half of image x})$ in a scalable way, I will use stochastic variational inference. First, I initialize variational parameters ϕ_μ and $\phi_{\log\sigma}$ for a variational distribution $q(z|\text{top half of x})$. Next, I implement a function that computes estimates the ELBO over K samples $z \sim q(z|\text{top half of x})$ using $\log p(z)$, $\log p(\text{top half of x}|z)$, and $\log q(z|\text{top half of x})$.

Next, I optimize ϕ_μ and $\phi_{\log\sigma}$ to maximize the ELBO.

Finally, I take a sample z from the approximate posterior, and feed it to the decoder to find the Bernoulli means of p(bottom half of image x$|z$). I contatenate this greyscale image to the true top of the image, and plot the original whole image beside it for comparison.

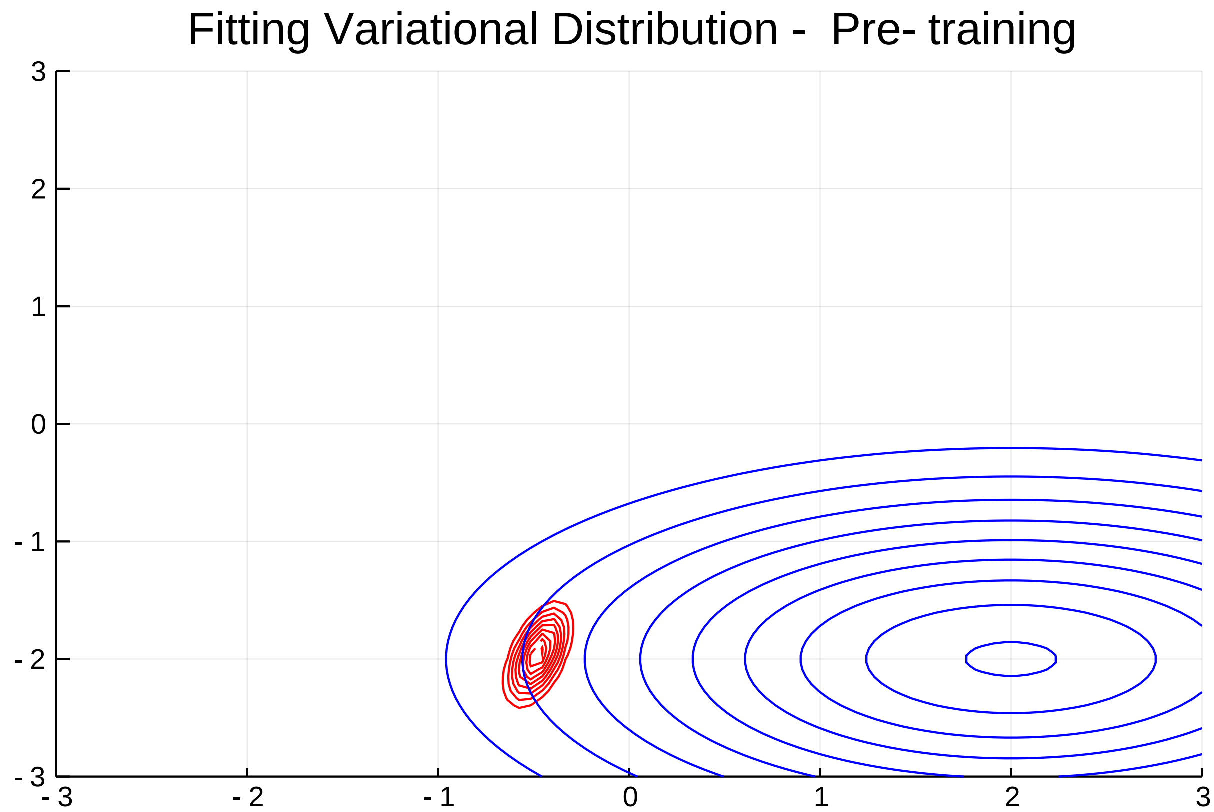
Selected digit image:

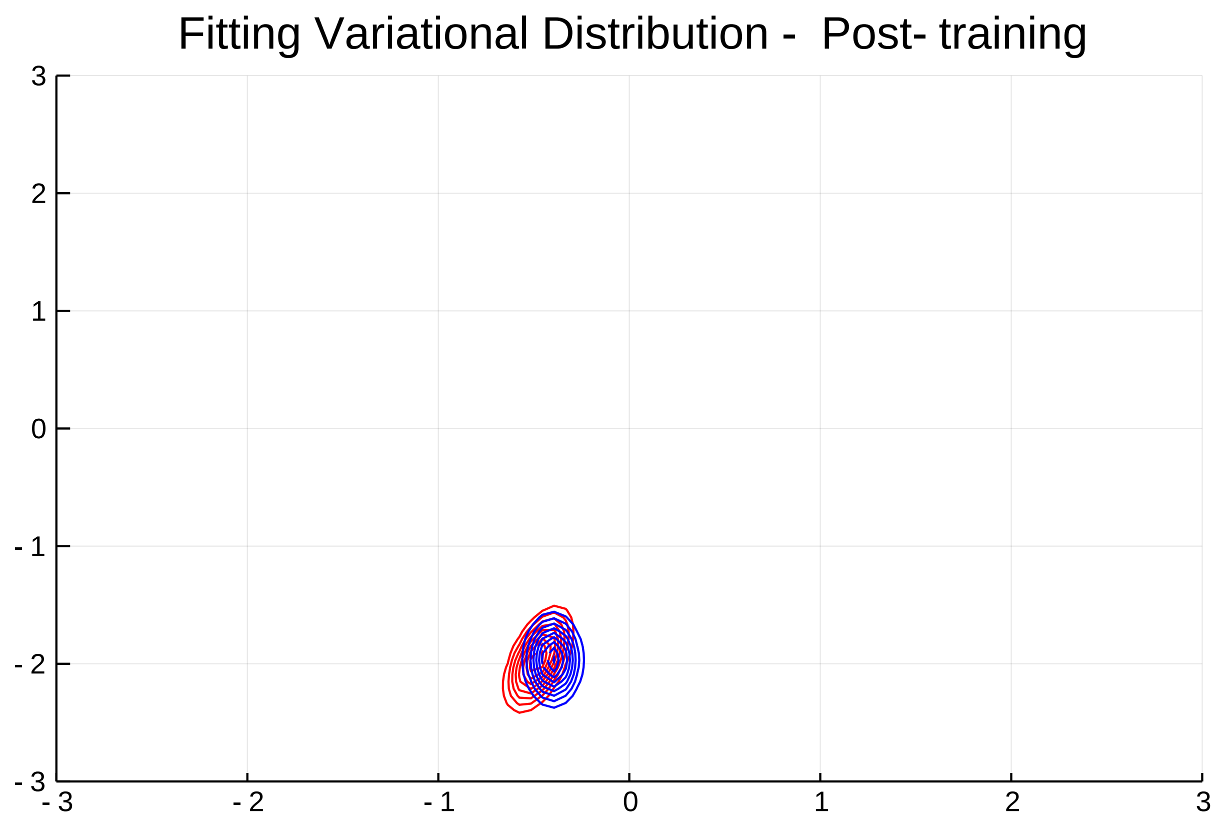
```
## Chosen digit
p1 = plot(mnist_img( train_x[:,train_label .== 9][:,9]), showaxis=false, title="whole
image");

μ = [2., -2.];
ls = [0.5, 0.];
init_params = (μ, ls);

function top_half_elbo(params, logp, num_samples)
  samples = params[1] .+ exp.(params[2]) .* randn(size(params[1])[1], num_samples)
  logp_estimate = logp(samples)
  logq_estimate = factorized_gaussian_log_density(params[1], params[2], samples)
  return mean(logp_estimate .- logq_estimate)
end;

function top_half_loss(params, x, num_samples = 10)
  logp(zs) = joint_log_density_top_half(x,zs)
  return -top_half_elbo(params, logp, num_samples)
end;

function fit_variational_dist(init_params, x; num_itrs=200, lr= 1e-2, num_q_samples = 10)
  # Pre-training plot
  p(zs) = exp(joint_log_density_top_half(x, zs))
  q(zs) = exp(factorized_gaussian_log_density(init_params[1], init_params[2], zs))
  plot(title="Fitting Variational Distribution - Pre-training")
  skillcontour!(p, colour=:red, label="model")
  display(skillcontour!(q, colour=:blue, label="approximate posterior"))

  params_cur = init_params
  for i in 1:num_itrs
    # gradients of variational objective with respect to parameters
    grad_params = gradient((params_cur) -> top_half_loss(params_cur, x, num_q_samples),
params_cur)[1]
    params_cur = (params_cur[1] - lr*grad_params[1], params_cur[2] - lr*grad_params[2])
    if i%10 == 0 @info "loss at iteration $i: $(top_half_loss(params_cur, x,
num_q_samples))" end
  end
  # print final elbo loss
  println("Final loss: $(top_half_loss(params_cur, x, num_q_samples))")
  # Post-training plot
  plot(title="Fitting Variational Distribution - Post-training")
  skillcontour!(p, colour=:red)
  q_f(zs) = exp(factorized_gaussian_log_density(params_cur[1], params_cur[2], zs))
  display(skillcontour!(q_f, colour=:blue))
  return params_cur
end;
```

The blue contour is the approximate posterior. The red contour is the model joint distribution

```
params = fit_variational_dist(init_params, top_half(train_x[:,train_label .== 9][:,9]))
```

```
Final loss: 47.73735035417486
([-0.39944102213924193, -1.9654583010365163], [-2.445817084498054, -1.48584
62781436024])
```






```
zs = params[1] .+ exp.(params[2]) .* randn(size(params[1])[1], 1)
θ = decoder(zs)
```

```
x = vcat(top_half(train_x[:,train_label .== 9][:,9]), bottom_half(θ))
p2 = plot(mnist_img(reshape(x, (Ddata,))), showaxis=false, title="reconstructed image")
display(plot(p1, p2))
```