Assignment 3 - Variational Autoencoders

Kevin Shi, 1002519134

April 15, 2020

```
using Flux
using MLDatasets
using Statistics
using Logging
using Test
using Random
using StatsFuns: log1pexp
Random.seed!(412414);
```

```
#### Probability Stuff
# Make sure you test these against a standard implementation!

# 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
## This is really bernoulli
@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) (hint: use reparameterization trick here)
```

```
sample_diag_gaussian(μ,logσ) = (ϵ = randn(size(μ)); μ .+ exp.(logσ).*ϵ)
# sample from Bernoulli (this can just be supplied by library)
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.
# You may use the utilities from A2, or dataloaders provided by a framework
function load_binarized_mnist(train_size=1000, test_size=1000)
  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
  @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
# if you only want to batch xs
batch_x(x::AbstractArray, batch_size=100) =
first.(batch_data((x,zeros(size(x)[end])),batch_size))


### Implementing the model

## Load the Data
train_data, test_data = load_binarized_mnist(10000, 10000);
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
```

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

784

1 Implementing the Model [5 points]

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

```
log_prior(z) = factorized_gaussian_log_density(0, 0, z)
```

```
log_prior (generic function with 1 method)
```

2. [**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 productof 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 eachdimension $D_{data} \times B$. For numerical stability, instead of outputting the mean $\mu \in [0, 1]$, you should output $\log(\frac{\mu}{1-\mu}) \in R$ called "logit".

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

```
Chain(Dense(2, 500, tanh), Dense(500, 784))
```

3. [**1 point**] Implement a function `log_likelihood` that, given a latent representation z and a binarized digit x, computes the log-likelihood $\log p(x|z)$

```
function log_likelihood(x,z)
  """ Compute log likelihood log_p(x|z)"""
  θ = decoder(z) # TODO: parameters decoded from latent z
  return  sum(bernoulli_log_density(θ, x), dims = 1) # return likelihood for each
element in batch
end
```

```
log_likelihood (generic function with 1 method)
```

4. [**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(x,z) = log_prior(z) + log_likelihood(x,z)
```

```
joint_log_density (generic function with 1 method)
```

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]

```
function unpack_gaussian_params(θ)
  μ, logσ = θ[1:2,:], θ[3:end,:]
  return  μ, logσ
end
```

```
unpack_gaussian_params (generic function with 1 method)
```

1. [**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 = Chain(Dense(Ddata, Dh, tanh), Dense(Dh, 2*Dz), unpack_gaussian_params)
```

```
Chain(Dense(784, 500, tanh), Dense(500, 4), unpack_gaussian_params)
```

2. [**1 points**] Write a function `log_q` that given the parameters of the variational distribution, evaluates the likelihood of z.

```
log_q(q_μ, q_logσ, z) = factorized_gaussian_log_density(q_μ, q_logσ, z) #TODO: write log
likelihood under variational distribution.
```

```
log_q (generic function with 1 method)
```

3. [**5 points**] Implement a function `elbo` which computes an unbiased estimate of the mean variationalevidence lower bound on a batch of images. Use the output of `encoder` to give the parameters for $q_\phi(z|data)$. 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.

```
function elbo(x)
  batch_size = size(x)[2]
  q_μ, q_logσ = encoder(x) #TODO variational parameters from data
  z = sample_diag_gaussian(q_μ,q_logσ)  #TODO: sample from variational distribution
  joint_ll = joint_log_density(x,z) #TODO: joint likelihood of z and x under model
  log_q_z = log_q(q_μ, q_logσ, z) #TODO: likelihood of z under variational distribution
  elbo_estimate = sum(joint_ll - log_q_z)/batch_size #TODO: Scalar value, mean
variational evidence lower bound over batch
  return elbo_estimate
end
```

```
elbo (generic function with 1 method)
```

4. **[2 points]** Write a loss function called `loss` that returns the negative elbo estimate over a batch of data.

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

```
loss (generic function with 1 method)
```

5. **[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 overbatches of data, not the entire dataset. Train the data for 100 epochs (each epoch involves a loop overevery 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

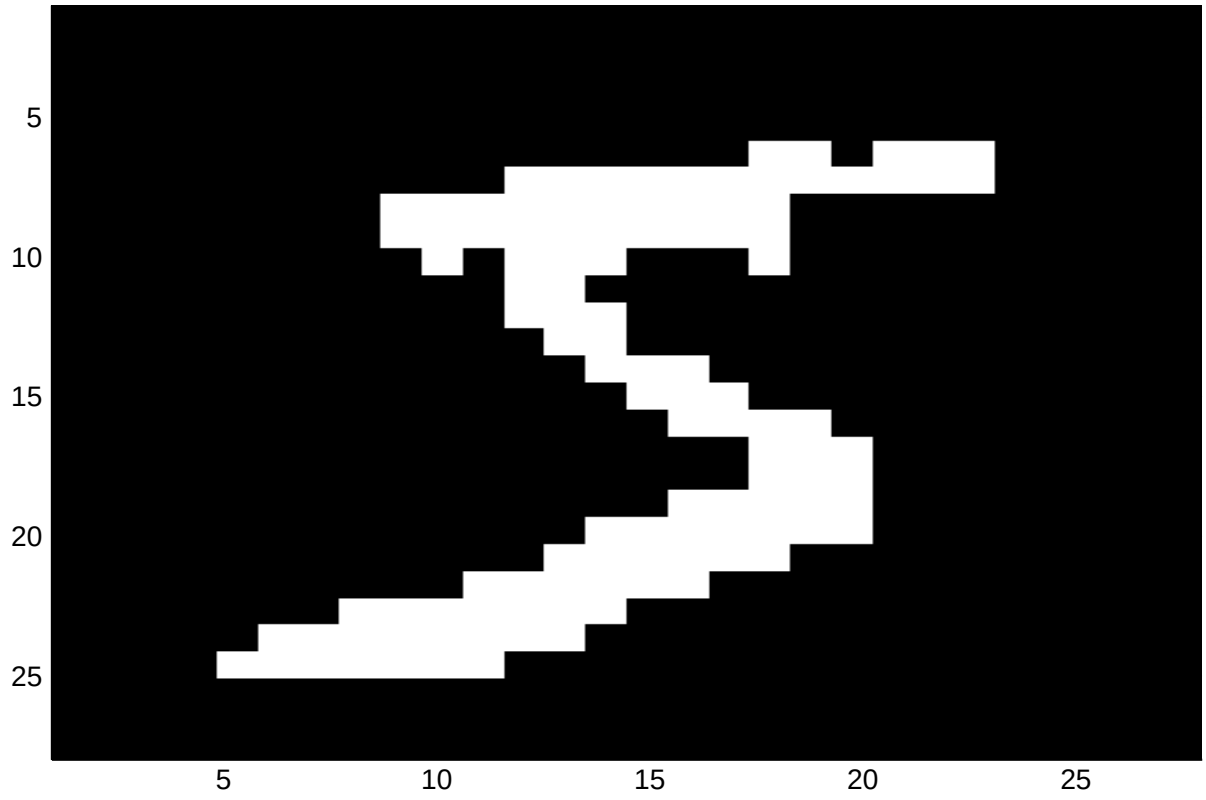
```
# function train_model_params!(loss, encoder, decoder, train_x, test_x; nepochs=10)
#   # model params
#   # ps = Flux.params(decoder) #TODO parameters to update with gradient descent
#   ps = Flux.params(encoder, decoder)
#   # ADAM optimizer with default parameters
#   opt = ADAM()
#   # over batches of the data
#   for i in 1:nepochs
#     for d in batch_x(train_x)
#       gs = Flux.gradient(ps) do
#         batch_loss = loss(d)
#         return batch_loss
#       end
#       Flux.Optimise.update!(opt,ps,gs)#TODO update the paramters with gradients
#     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

## Train the model
#### this is commented out for the report - if u want to see it run, check the
file vae.jl
# 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 "Load model params from $load_dir"
```

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 generativemodel, sampling from it, and interpolating between digits.

```
# Visualization
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))
mnist_img(x) = ndims(x)==2 ? Gray.(permutedims(reshape(x,28,28,:), [2, 1, 3])) :
Gray.(transpose(reshape(x,28,28)))

## Example for how to use mnist_img to plot digit from training data
plot(mnist_img(train_x[:,1]))
```



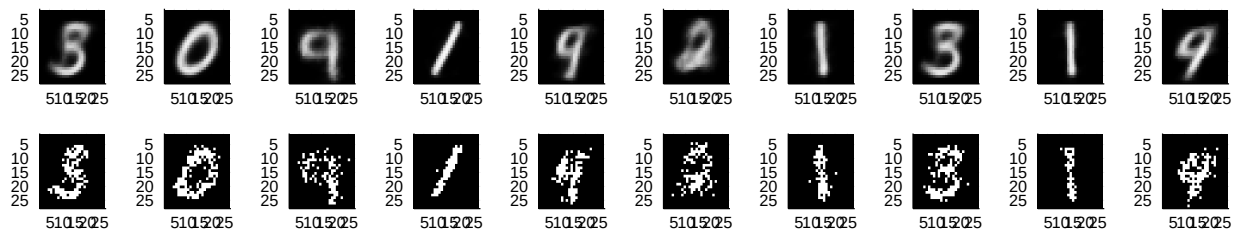
1. **[5 points]** Plot samples from the trained generative model using ancestral sampling

    1. First sample a z from the prior.
    2. Use the generative model to compute the bernoulli means over the pixels of x given z. Plot these means as a greyscale image.
    3. 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 2×10 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.

```
plots = Any[]
# This loop generates the plots of the first row - the bernoulli means
for i in 1:10
  sample = train_x[:, i]
  z = sample_diag_gaussian(encoder(train_x[:,i]) ...)       # sample from prior
  logit_b_means = decoder(z)                                 # logit bernoulli means
  b_means = sigmoid.(logit_b_means) # transform from logit to regular mean
  p = plot(mnist_img(vec(b_means)))
  push!(plots, p)
end

# This loop generates the plots of the second row - the bernoulli samples from the
previous means
for i in 1:10
  sample = train_x[:, i]
  z = sample_diag_gaussian(encoder(train_x[:,i]) ...)
  logit_b_means = decoder(z)
  b_means = sigmoid.(logit_b_means)
  p = plot(mnist_img(vec(sample_bernoulli(b_means))))  # sample from bernoulli with
params
  push!(plots, p)
end

display(plot(plots ..., layout = grid(2, 10), size = (1000, 200)))
```



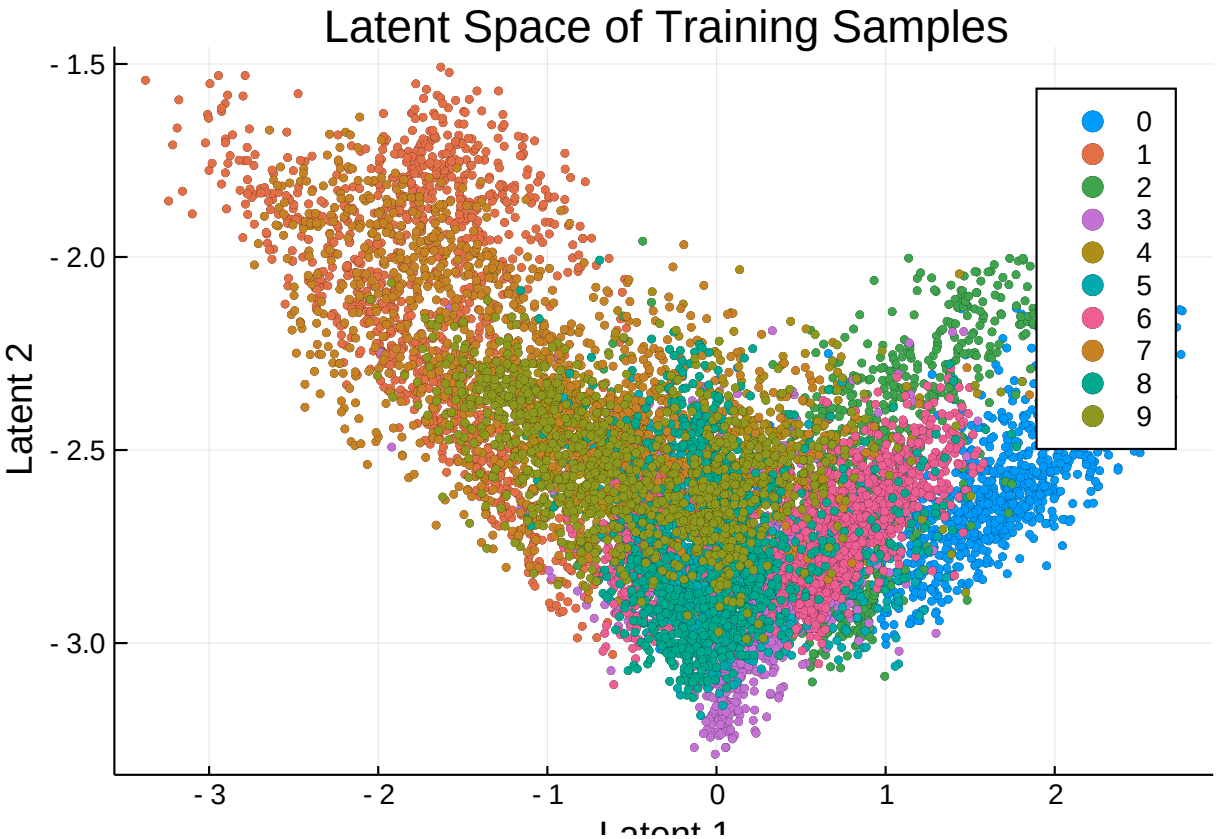
```
savefig(joinpath("plots","3a_ten_bernoulli_means_and_samples"))
```

2. **[5 points]** One way to understand the meaning of latent representations is to see which parts of thelatent 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.

1. Encode each image in the training set.
2. Take the 2D mean vector of each encoding $q_\phi(z|x)$.
3. Plot these mean vectors in the 2D latent space with a scatterplot.
4. 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 classlabels to the model!

```
q_μ, q_logσ = encoder(train_x)
mean_μ = vec(sum(q_μ, dims = 1)/Dz)
mean_logσ = vec(sum(q_logσ, dims = 1)/Dz)
display(scatter(mean_μ, mean_logσ, group = train_label,
        title = "Latent Space of Training Samples",
        xlabel = "Latent 1",
        ylabel = "Latent 2",
        markerstrokewidth = 0.25,
        markersize = 2.5))
```




```
savefig(joinpath("plots","3b_plot_of_encoder_params_vs_labels"))
```

3. **[5 points]** Another way to examine a latent variable model with continuous latent variables is tointerpolate between the latent representations of two points.

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

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

2. Sample 3 pairs of images, each having a different class.
3. Encode the data in each pair, and take the mean vectors
4. Linearly interpolate between these mean vectors
5. At 10 equally-space points along the interpolation, plot the Bernoulli means $p(x|z)$
6. Concatenate these plots into one figure.

```
function lin_interpolate(z_a, z_b, α)
  if !( 0 <= α && α <= 1)
    println("α is not between 0 and 1")
  end
  return α .* z_a .+ (1 - α) .* z_b
end

endpoints_1 = [train_x[:, 1], train_x[:, 2], train_x[:, 15]]
endpoints_2 = [train_x[:, 4], train_x[:, 3], train_x[:, 2]]

plots = Any[]
for i in 1:3
  mean_params_1 = sum.(encoder(endpoints_1[i]))./Dz
  mean_params_2 = sum.(encoder(endpoints_2[i]))./Dz
  for d in reverse(0:9)
    inter_params = lin_interpolate(mean_params_1, mean_params_2, d*float(1/9))
    logit_b_means = decoder(collect(inter_params))                    # logit
bernoulli means
    b_means = sigmoid.(logit_b_means)
    p = plot(mnist_img(vec(b_means)))
    push!(plots, p)
  end
end

display(plot(plots ..., layout = grid(3, 10), size = (1000, 300)))
```

```
savefig(joinpath("plots","3c_interpolated_bernoulli_means"))
```

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

Now we'll use the trained generative model to perform inference for$p(z|$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 imageconditioned 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 posteriorp(zjtop half of image x), we'll use stochastic variational inference.

1. **[5 points]** Write a function that computes $p(z, \text{top half of image x})$

    1. First, write a function which returns only the top half of a 28×28 array. This will be useful forplotting, as well as selecting the correct Bernoulli parameters.
    2. 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.
    3. Combine this likelihood with the prior to get a function that takes an x and an array of zs, andcomputes the log joint density $\log p(z|\text{top half of image x})$ for each z in the array.

```
function get_top_half(img, num_samples)
  # """Takes a 28 x 28 matrix and returns the top half"""
  img = reshape(img, 28, 28, num_samples)
  img = img[1:28, 1:14, 1:num_samples]
  return reshape(img, 392, 1, num_samples)
end

function get_bot_half(img, num_samples)
  # """Takes a 28 x 28 matrix and returns the top half"""
  img = reshape(img, 28, 28, num_samples)
  img = img[1:28, 15:28, 1:num_samples]
  return reshape(img, 392, 1, num_samples)
end

function top_half_log_likelihood(x, z, num_samples)
  θ = decoder(z)
  θ_top = get_top_half(θ, num_samples)
  x_top = get_top_half(x, 1)
  log_dens = sum(bernoulli_log_density.(θ_top, x_top), dims = 1)
  return reshape(log_dens, 1, num_samples)
end

function top_half_joint_log_density(x, zs, num_samples)
  """Computes the joint log density log_p(z, top half of x)"""
  return log_prior(zs) + top_half_log_likelihood(x, zs, num_samples)
end

top_half_joint_log_density (generic function with 1 method)
```

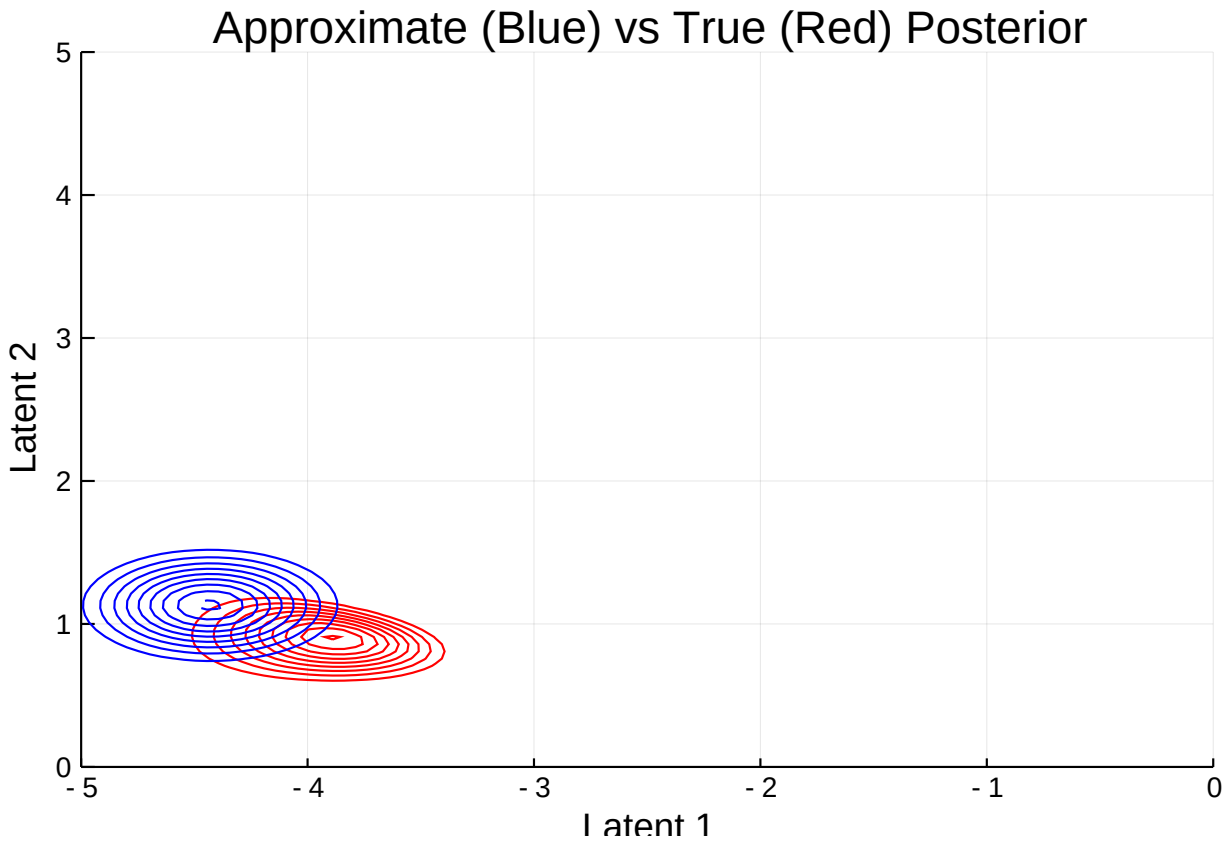
2. **[5 points]** Now, to approximate $p(z|\text{top half of image x})$ in a scalable way, we'll use stochastic variational inference. For a digit of your choosing from the training set (choose one that is modelled well, i.e. the resulting plot looks reasonable):

1. Initialize variational parameters ϕ_μ and $\phi_{\log\sigma}$ for a variational distribution $q(z|\text{top half ofx})$.
2. Write a function that computes estimates the ELBO over K samples $z \sim q(z|\text{top half of x})$. Use $\log p(z)$, $\log p(\text{top half ofx}|z)$, and $\log q(z|\text{top half of x})$.
3. Optimize ϕ_μ and $\phi_{\log\sigma}$ to maximize the ELBO.
4. On a single plot, show the isocontours of the joint distribution $p(z, \text{top half of image x})$, and the optimized approximate posterior $q_\phi(z|\text{top half of image x})$.
5. Finally, take a sample z from your approximate posterior, and feed it to the decoder to findnd the Bernoulli means of $p(\text{bottom half of image x}|z)$. Contatenate this greyscale image to the true topof the image. Plot the original whole image beside it for comparison.

```
function skillcontour!(f; colour=nothing)
  n = 100
  # x = range(-3,stop=3,length=n)
  # y = range(-3,stop=3,length=n)
  x = range(-5,stop=0,length=n)
  y = range(0,stop=5,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

function top_half_elbo(params, logp, num_samples)
  """Computes elbo estimate for top half of the image"""
  samples = exp.(params[2]) .* randn(Dz, num_samples) .+ params[1]
  logp_estimate = logp(samples)
  # logq_estimate = factorized_gaussian_log_density(params[1], params[2], samples)
  logq_estimate = log_q(params[1], params[2], samples)
  return sum(logp_estimate - logq_estimate)
end

function top_half_neg_elbo(params; x = train_x[:, 1], num_samples = 100)
  logp(zs) = top_half_joint_log_density(x, zs, num_samples)
  return -top_half_elbo(params,logp, num_samples)
end

init_mu = randn(2,1)
init_ls = randn(2,1)
init_params = (init_mu, init_ls)
function fit_toy_variational_dist(init_params, toy_evidence; num_itrs=200, lr= 1e-2,
num_q_samples = 10, title = "Fit Toy Variational Dist.")
  params_cur = init_params
  for i in 1:num_itrs
    grad_params = gradient(params->top_half_neg_elbo(params; x = toy_evidence, num_samples
= num_q_samples),params_cur)[1]
    params_cur = params_cur .- grad_params .* lr
    @info top_half_neg_elbo(params_cur; x = toy_evidence, num_samples = num_q_samples)
```

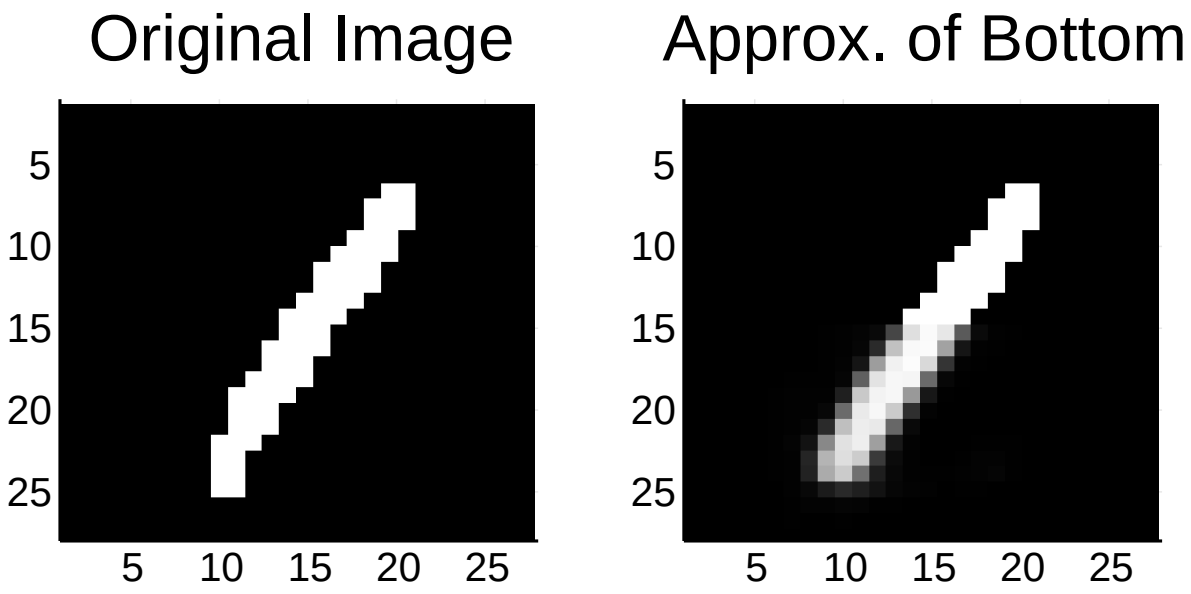
```
  end

  plot(title = title,
      xlabel = "Latent 1",
      ylabel = "Latent 2"
     )
  # joint_posterior(zs) = sigmoid.(top_half_joint_log_density(zs, toy_evidence,
size(zs)[2]))
  # joint_posterior(zs) = sigmoid.(decoder(zs))
  joint_posterior(zs) = sigmoid.(joint_log_density(toy_evidence, zs))
  skillcontour!(joint_posterior;colour=:red)
  iter_gaussian(zs) = exp(factorized_gaussian_log_density(params_cur[1],params_cur[2],zs))
  # iter_bernoulli(zs) = sigmoid.(bernoulli_log_density())
  display(skillcontour!(iter_gaussian;colour=:blue))  # run this line to see the model
train
  println("Final Loss: ", top_half_neg_elbo(params_cur; x = toy_evidence, num_samples =
num_q_samples))
  return params_cur
end

train_image = train_x[:, 24]
latent_params = fit_toy_variational_dist(init_params, train_image, title = "Approximate
(Blue) vs True (Red) Posterior")
```

```
Final Loss: 297.9073161657537
```




```
zs = sample_diag_gaussian(latent_params[1], latent_params[2])
savefig(joinpath("plots","4b_contours_approx_vs_true.pdf"))
logit_b_means = decoder(zs)
b_means = sigmoid.(logit_b_means)

plots = []
top_image = get_top_half(train_image, 1)
```

```
bot_image = get_bot_half(vec(b_means), 1)
cat_image = hcat(reshape(top_image[:,:, 1], 28, 14), reshape(bot_image[:,:,1], 28, 14))
push!(plots, plot(mnist_img(train_image), title = "Original Image"))
push!(plots, plot(mnist_img(vec(cat_image)), title = "Approx. of Bottom"))
display(plot(plots ..., layout = grid(1, 2), size = (400, 200)))
```




```
savefig(joinpath("plots","4b_original_vs_approx_bottom"))
```

3. [**5 points**] True or false: Questions about the model and variational inference.

   There is no need to explain your work in this section.

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

      True

   2. 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?

      False

   3. 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?

      False

   4. If $p(x) = N(x|\mu, \sigma^2)$, for some $x \in R, \mu \in R, \sigma \in R^+$, can $p(x) < 0$?

      False

   5. If $p(x) = N(x|\mu, \sigma^2)$, for some $x \in R, \mu \in R, \sigma \in R^+$, can $p(x) > 1$?

      False