Assignment 3

Raj Patel

April 16, 2020

Learning Outcomes: 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).

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

1 Implementing the model

1.1 Predefined functions:

```
# log-pdf of x under Factorized or Diagonal Gaussian
function factorized_gaussian_log_density(mu,logsig,xs)
  sig = exp.(logsig)
  return sum((-1/2)*log.(2*pi*sig.^2) .+ -1/2 * ((xs .- mu).^2)./(sig.^2),dims = 1)
end
```

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

```
# log-pdf of x under Bernoulli
function bernoulli_log_density(logit_means,x)

  b = x .* 2 .- 1
  return - log1pexp.(-b .* logit_means)

end
```

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

```
@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)
  @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
Test.DefaultTestSet("test stable bernoulli", Any[], 2, false)

# sample from Diagonal Gaussian (using 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))

sample_bernoulli (generic function with 1 method)

# 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
  @info "Binarized the pixels"
  return (train_x, train_label), (test_x, test_label)
end

load_binarized_mnist (generic function with 3 methods)

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))

batch_x (generic function with 2 methods)

# Implementing the model

# 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
```

```
## 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
```

1.2 Question 1 Part a:

Implement a function log_prior that computes the log of the prior over a digit's representation log p(z).

```
function log_prior(z)

  # Input: z - latent representation of a single image
  # Output: normal prior distribution. (scalar: if batch size is 1 and vector of
batch size if not 1)
  l_prior = factorized_gaussian_log_density(0,0,z)
  return l_prior

end
```

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

1.3 Question 1 Part b:

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 $D_{data} \times B$. For numerical stability, instead of outputting the mean μ in [0, 1], we output logit.

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

function decoder(z)

  # Input: z- Latent representation of a single image (2*batch_size)
  # Output: image representation (784 * batch_size)
```

```
  return decoder_net(z)

end
```

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

1.4 Question 1 Part c:

Implement a function log-likelihood that, given a latent representation z and a binarized digit x, computes the log-likelihood log p(x given z).

```
function log_likelihood(x,z)

  # Input 1: z - latent representation of a single image
  # Input 2: x - a binarized digit x
  # Output: scalar ll if batch size is 1 and vector of batch size if not 1
  logit_mu = decoder_net(z)
  sum(bernoulli_log_density.(logit_mu,x), dims = 1)

end
```

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

1.5 Question 1 Part d:

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.

```
function joint_log_density(x,z)

  # Input: z - latent representation of a single image, x - a binarized digit x
  # Output: scalar log density
  log_prior(z) .+ log_likelihood(x,z)

end
```

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

2 Amortized Approximate Inference and training

2.1 Question 2 Part a:

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 Dz = 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.

```
function unpack_gaussian_params(θ)

  μ, logσ = θ[1:2,:], θ[3:end,:]
  return  μ, logσ
```

```
end

encoder_net = Chain(Dense(Ddata,Dh,tanh), Dense(Dh,2*Dz))

function encoder(x)

  # Input: x - batch of image (784*batch_size)
  # Output: z - latent representation of a digit
  return unpack_gaussian_params(encoder_net(x))

end

encoder (generic function with 1 method)
```

2.2 Question 2 Part b:

Write a function log q that given the parameters of the variational distribution, evaluates the likelihood of z.

```
function log_q(q_μ, q_logσ, z)

  # Input: parameters which would be mean and log std and both would be of
size(batch_size)
  # Output: Outputs the log likelihood of observing the z given the parameters.
  return factorized_gaussian_log_density(q_μ,q_logσ,z)

end

log_q (generic function with 1 method)
```

2.3 Question 2 Part c:

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_ϕ(z given data). This estimator takes the following arguments: • x, an batch of B images, $Dx \times B$ • encoder params, the parameters ϕ of the encoder (recognition network). • decoder params, the parameters θ of the decoder (likelihood).

This function should return a single scalar.

```
function elbo(x)

  q_μ, q_logσ  = unpack_gaussian_params(encoder_net(x)) # variational parameters from
data
  z  = q_μ  .+ (exp.(q_logσ) .* randn(size(q_μ))) # 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

elbo (generic function with 1 method)
```

2.4 Question 2 Part d:

Write a loss function called loss that 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
```

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

2.5 Question 2 Part e:

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.

```
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 and update
the paramters with gradients
       gs = Flux.gradient(()-> loss(d),ps)
       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!"
  return loss(batch_x(test_x)[1])
end
```

```
## Train the model
final_test_elbo = train_model_params!(loss,encoder_net,decoder_net,train_x,test_x,
nepochs=100)
print("The final ELBO on test set is $final_test_elbo")
```

```
The final ELBO on test set is 159.02207324379992
```

```
### 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

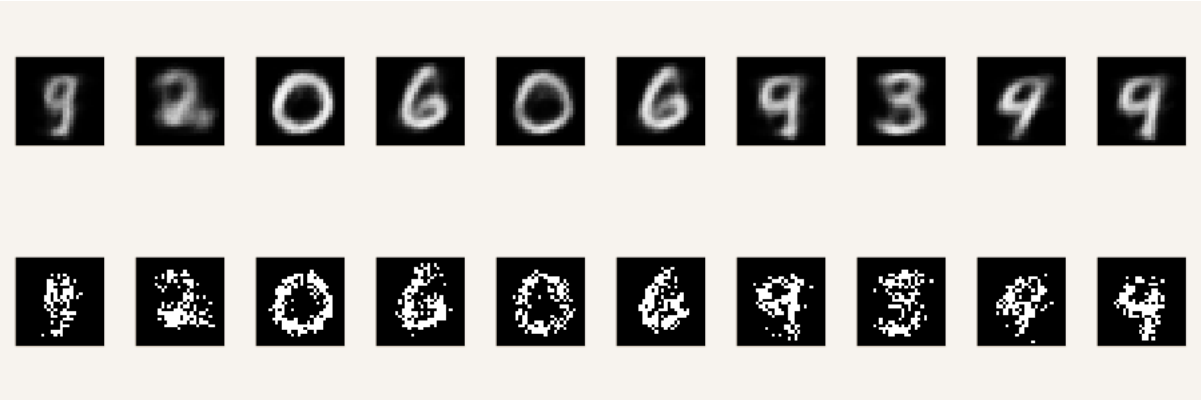
3.1 Question 3 Part a:

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.

```
mnist_img(x) = ndims(x)==2 ? Gray.(reshape(x,28,28,:)) : Gray.(reshape(x,28,28))
mnist_img_alt(x) = ndims(x)==2 ? Gray.(permutedims(reshape(x,28,28,:), [2, 1, 3])) :
Gray.(transpose(reshape(x,28,28)))
mnist_img_alt_2(x) = ndims(x)==2 ? Gray.(permutedims(reshape(x,28,28,:), [2, 1,
3])[1:14,:,:]) : Gray.(transpose(reshape(x,28,28))[1:14,:])
```

```
#Sampling z from my distribution:
d = Normal()

bernoulli_plots = repeat([plot()],10)
sample_bernoulli_plots = repeat([plot()],10)
for i = 1:10

    # Sample from prior
    sample_prior_z = rand(d,2)

    # Run the trained decoder to get the vector using which we can generate the
image
    logit_means = decoder_net(sample_prior_z)

    # Output of decoder is logit of mean. So, we get the mean vector
    bernoulli_means = exp.(logit_means) ./(1 .+ exp.(logit_means))

    # Sample 1 number from bernoulli with success equal to the respective element
from mean vector for all 784 elements
    samples_bernoulli = sample_bernoulli(bernoulli_means)

    # plot the mean vector
    bernoulli_plots[i] = plot(mnist_img_alt(bernoulli_means[:,1]),xticks = false,yticks =
false)
```

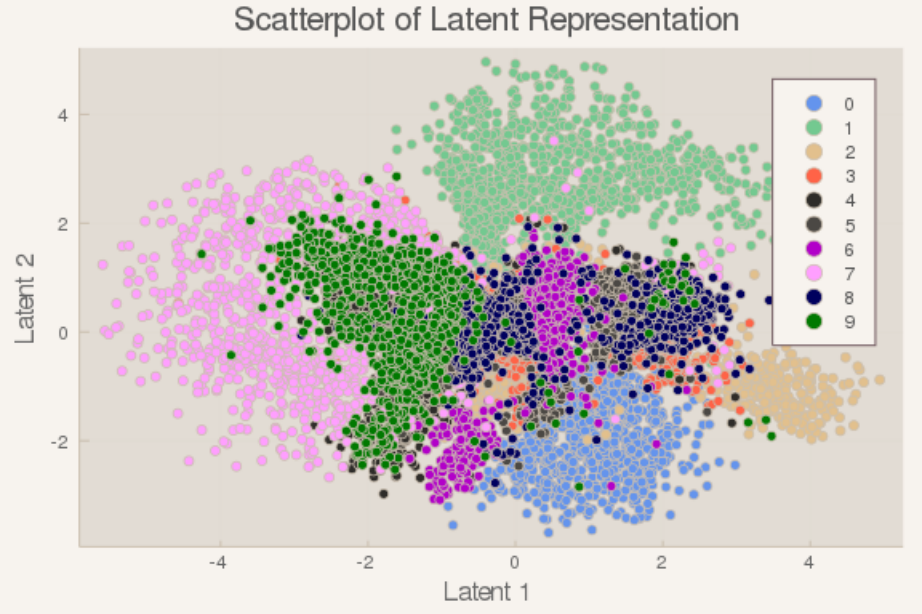
```
    # plot the sample of binary image from the product of bernoulli
    sample_bernoulli_plots[i] = plot(mnist_img_alt(samples_bernoulli[:,1]),xticks =
false,yticks = false)

end

# get all the elements into an array
full_bernoulli_plots = hcat(bernoulli_plots,sample_bernoulli_plots)

# Plot all 20 images
plot(full_bernoulli_plots...,layout =(2,10),size = (900, 300))
```



3.2 Question 3 Part b:

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(z given x) with ϕ parameters. (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).

```
# Encoding each image in the training set
encoded_img = unpack_gaussian_params(encoder_net(train_x))

# Encoded mean vectors
mean_enc_img = encoded_img[1]

gr() # We will continue onward using the GR backend

plot(reuse = false)

using PlotThemes
using Plots
theme(:sand)

# Plotting mean vectors in 2D latent space with a scatterplot
plot(mean_enc_img[1,:], mean_enc_img[2,:], seriestype = :scatter, title = "Scatterplot of
Latent Representation", group=train_label,markersize = 4,xlab = "latent 1",ylab =
"latent 2")
```



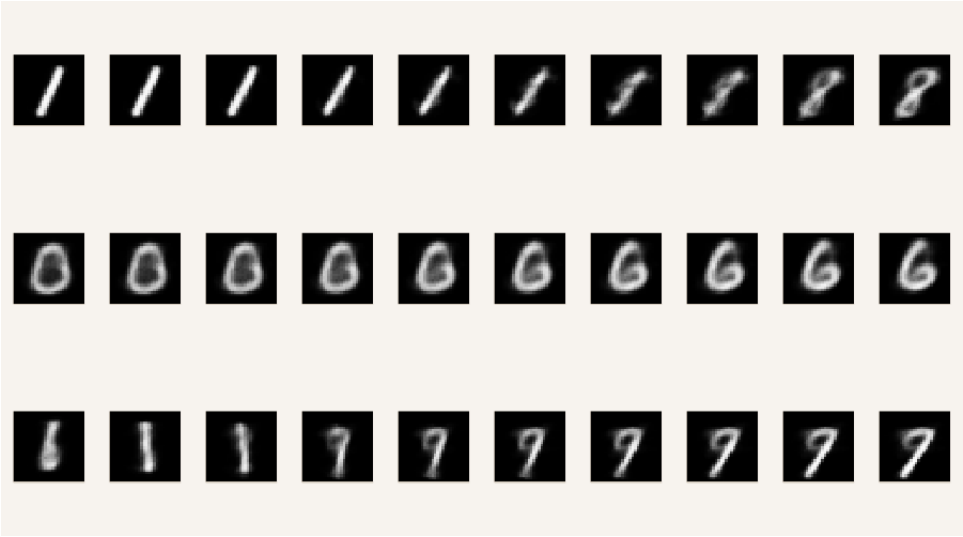
3.3 Question 3 Part c:

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 za and zb, and a value α 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 given z_α) (f) Concatenate these plots into one figure.

```
# Part a: Creating the linear interpolating function

function linear_interpol(z_a,z_b,alpha)
  return (alpha * z_a) + ((1 - alpha) * z_b)
end


# Part b: Creating 3 pairs with different class

global pair = ()
global index = ()
length(pair)
while length(pair) < 3
  ind1 = rand(1:1000)
  ind2 = rand(1:1000)
  if train_label[ind1] != train_label[ind2]
    global pair = pair...,(train_x[:,ind1],train_x[:,ind2])
    global index = index...,(ind1,ind2)
  end
end


# Part c: Encode data in each pair and take mean vectors
```

```
global encode_mu = ()
for i in 1:3
  mu_1 = unpack_gaussian_params(encoder_net(pair[i][1]))[1]
  mu_2 = unpack_gaussian_params(encoder_net(pair[i][2]))[1]

  global encode_mu = encode_mu...,(mu_1,mu_2)
end

# Part d,e,f: Linearly interpolate between the mean vectors

plot(reuse = false)
bernoulli_plots = repeat([plot()],30)
singlify(x::Float64) = Float32(x)
alpha_ten_space = [0:1/9:1;]
alpha_ten_space = singlify.(alpha_ten_space)

for i =1:10

    # Linear interpolating for first element
    linear_interpol_1 =
linear_interpol(encode_mu[1][1][1,:],encode_mu[1][2][1,:],alpha_ten_space[11-i])

    # Linear interpolating for second element
    linear_interpol_2 =
linear_interpol(encode_mu[1][1][2,:],encode_mu[1][2][2,:],alpha_ten_space[11-i])

    # Combining the vectors
    linear_interpol_combine = vec([linear_interpol_1 linear_interpol_2]')

    # Finding the logit means using decoder
    logit_means = decoder_net(linear_interpol_combine)

    # Finding the means from logit means
    bernoulli_means = exp.(logit_means) ./(1 .+ exp.(logit_means))

    # Plotting the bernoulli means
    bernoulli_plots[i] = plot(mnist_img_alt(bernoulli_means[:,1]),xticks = false,yticks =
false)

end

for j =11:20

    # Linear interpolating for first element
    linear_interpol_1 =
linear_interpol(encode_mu[2][1][1,:],encode_mu[2][2][1,:],alpha_ten_space[21-j])

    # Linear interpolating for second element
    linear_interpol_2 =
linear_interpol(encode_mu[2][1][2,:],encode_mu[2][2][2,:],alpha_ten_space[21-j])

    # Combining the vectors
    linear_interpol_combine = vec([linear_interpol_1 linear_interpol_2]')

    # Finding the logit means using decoder
    logit_means = decoder_net(linear_interpol_combine)

    # Finding the means from logit means
    bernoulli_means = exp.(logit_means) ./(1 .+ exp.(logit_means))
```

```
    # Plotting the bernoulli means
    bernoulli_plots[j] = plot(mnist_img_alt(bernoulli_means[:,1]),xticks = false,yticks =
false)

end

for k =21:30

    # Linear interpolating for first element
    linear_interpol_1 =
linear_interpol(encode_mu[3][1][1,:],encode_mu[3][2][1,:],alpha_ten_space[31-k])

    # Linear interpolating for second element
    linear_interpol_2 =
linear_interpol(encode_mu[3][1][2,:],encode_mu[3][2][2,:],alpha_ten_space[31-k])

    # Combining the vectors
    linear_interpol_combine = vec([linear_interpol_1 linear_interpol_2]')

    # Finding the logit means using decoder
    logit_means = decoder_net(linear_interpol_combine)

    # Finding the means from logit means
    bernoulli_means = exp.(logit_means) ./(1 .+ exp.(logit_means))

    # Plotting the bernoulli means
    bernoulli_plots[k] = plot(mnist_img_alt(bernoulli_means[:,1]),xticks = false,yticks =
false)

end

# Plotting all bernoulli means together in a single plot
plot(bernoulli_plots...,layout =(3,10),size = (900, 500))
```



4 Predicting the Bottom of Images given the Top

Now we'll use the trained generative model to perform inference for p(z given 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.

To approximate the posterior p(z given top half of image x), we'll use stochastic variational inference

4.1 Question 4 Part a:

Write a function that computes p(z, top half of image x) (a) First, write a function which returns only the top half of a 28 × 28 array. This will be useful for plotting, as well as selecting the correct Bernoulli parameters. (b) Write a function that computes log p(top half of image x given z). (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, top half of image x) for each z in the array.

```
# (a) returning just the top of the image
mnist_img_alt_2(x) = ndims(x)==2 ? Gray.(permutedims(reshape(x,28,28,:), [2, 1,
3])[1:14,:,:]) : Gray.(transpose(reshape(x,28,28))[1:14,:])

# Returning a vector corresponding to the top half of the image
function top_array(z)
  return z[1:14,:]
end

# (Optional) Function which plots only top half of a particular image
function top_img(x)
  # applying the function that only returns top of the image
  return plot(mnist_img_alt_2(x))
end



# (b) Class conditional function

function log_likelihood_top(x,z)
  # Input 1: z - latent representation of a single image
  # Input 2: x - a binarized digit x
  # Output: scalar ll if batch size is 1 and vector of batch size if not 1

  # Decoding the latent representation of the image
  logit_mu = decoder_net(z)

  # Taking the top half of the image from the decoded vector
  logit_mu_top = logit_mu[1:392,:]

  # log-likelihood logp(top half of x given z) using top half of image vector and
top half of decoded vector
  sum(bernoulli_log_density.(logit_mu_top,x[1:392,:]), dims = 1)
end

# (c) log joint density using prior and likelihood

function joint_log_density_top(x,z)
  # Input: z - latent representation of top of a single image, x - a binarized
digit x
  # Output: scalar log density
```

```
  log_prior(z) .+ log_likelihood_top(x,z)

end

joint_log_density_top (generic function with 1 method)
```

4.2 Question 4 Part b:

Now, to approximate p(z given top half of image x) in a scalable way, we'll use stochastic variational inference. For a digit of our choosing from the training set: (a) Initialize variational parameters ϕ_μ and $\phi_{log\sigma}$ for a variational distribution q(z given top half of x). (b) Write a function that computes estimates the ELBO over K samples z $\sim$ q(z given top half of x). Use log p(z), log p(top half of x given z), and log q(z given 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, top half of image x), and the optimized approximate posterior q_ϕ(z given top half of image x). (e) Finally, take a sample z from approximate posterior, and feed it to the decoder to find the Bernoulli means of p(bottom half of image x given z). Contatenate this greyscale image to the true top of the image. Plot the original whole image beside it for comparison.

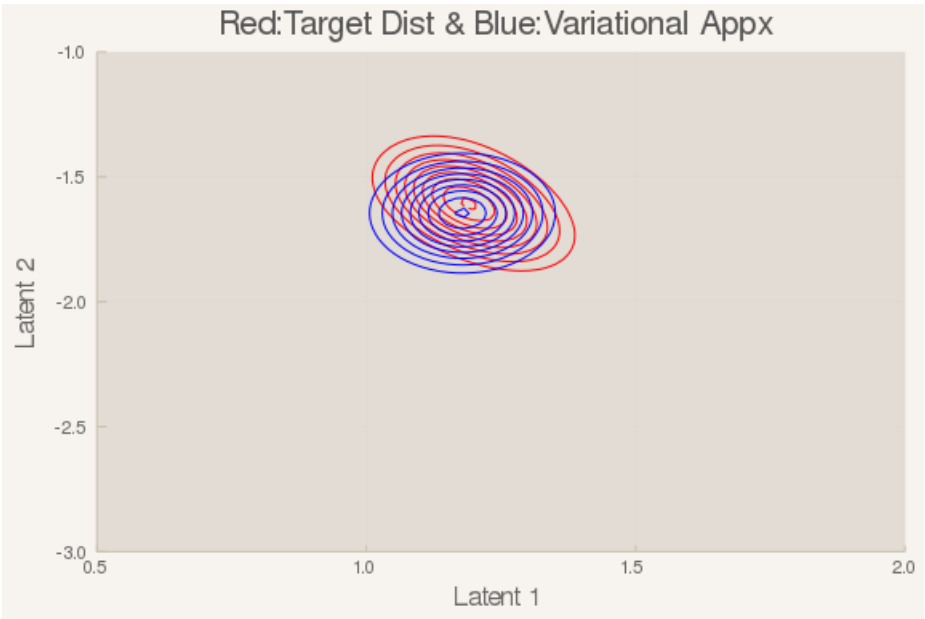
```
# Part a: function for initializing parameters

function init_parameters(n)
    d =Normal()
    phi_mu = rand(d,n)
    phi_log_sigma = rand(d,n)
    return phi_mu,phi_log_sigma
end

# Part b: function for generating K samples

function sample_K(mu,log_sig,k)
    sample_return = sample_diag_gaussian(mu,log_sig)
    for i =2:k
        sample_return = hcat(sample_return,sample_diag_gaussian(mu,log_sig))
    end
    return sample_return
end

# Initializing parameters

mu_init,logsig_init = init_parameters(2)

# ELBO Function

function ELBO(x,params,num_samples)

  # Z samples
  z = sample_K(params[1],params[2],num_samples)

  # joint likelihood of z and x
  joint_ll = joint_log_density_top(x,z)

  # likelihood of z under variational distribution
  log_q_z= log_q(params[1], params[2], z)
```

```
  # mean variational evidence lower bound
  return mean(joint_ll-log_q_z)

end

function neg_ELBO(params,x,num_samples)

    # returning negative ELBO
    return -ELBO(x,params,num_samples)

end

# Extra: Plotting Contours

function skillcontour!(f; colour=nothing)
  n = 100
  x = range(0,stop=2,length=n)
  y = range(-3,stop=-1,length=n)
  z_grid = Iterators.product(x,y)
  z_grid = reshape.(collect.(z_grid),:,1)
  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

#  Part c: Optimizing the parameters to maximize the ELBO

function fit_variational_dist(init_params, x; num_itrs=200, lr= 1e-2, num_q_samples = 10)
  params_cur = init_params
  for i in 1:num_itrs

    # gradients of variational objective with respect to parameters
    grad_params = gradient(params -> neg_ELBO(params,x,num_q_samples), params_cur)[1]

    # update paramters with lr-sized step in descending gradient
    params_cur = params_cur .- lr.*grad_params

    # report the current elbbo during training
    @info "loss: $(neg_ELBO(params_cur,x,num_q_samples))"

  end

  # Samples for plotting isocontours
  zs = params_cur[1].+ exp.(params_cur[2]).*randn(size(params_cur[1])[1],num_q_samples)

  # Variational posterior
  variational_posterior(zs) = exp.(factorized_gaussian_log_density(params_cur[1],
params_cur[2],zs))

  # Target posterior
  target_posterior(zs) = exp.(joint_log_density_top(x,zs))
```

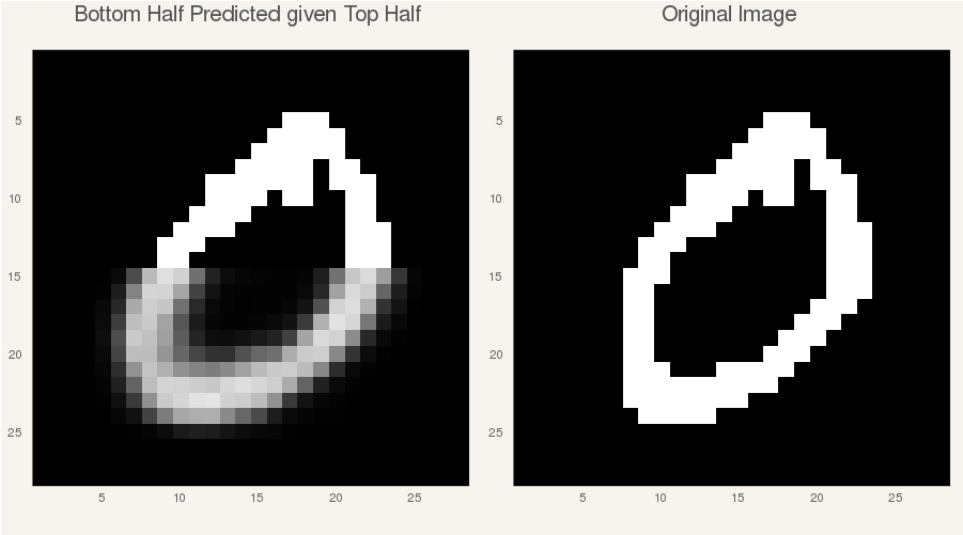
```
  # Plotting
  plot(reuse = false)
  plot()
  display(skillcontour!(target_posterior,colour=:red))
  display(skillcontour!(variational_posterior, colour=:blue))

  # final loss
  final_loss = neg_ELBO(params_cur,x,num_q_samples)

  # Returning optimized parameters and final loss
  return [params_cur, final_loss]

end


# Declaring the initial parameters
initial_parameters = (mu_init,logsig_init)

# Training image to be used

# Here, we will use the first image with label 0 as 0 is easy to model
train_image = train_x[:,train_label.== 0][:,1]

# Running the model to train the parameters
trained_info = fit_variational_dist(initial_parameters,train_image,num_itrs = 1000)

# Final loss:
final_loss = trained_info[2]
print("Final loss with trained parameters is $final_loss")
```

```
Final loss with trained parameters is 59.678097764005926
```

```
# Part d: Isocontours of the joint distribution p(z, top half of image x), and the
optimized approximate posterior
display(plot(title = "Blue:Variational Appx & Red: Target Dist",xlab = "latent 1",ylab =
"latent 2"))
```



```
# Part e: take a sample z from your approximate posterior, and feed it to the
decoder to find the Bernoulli means of p(bottom half of image x given z).
Contatenate this greyscale image to the true top of the image. Plot the original
whole image beside it for comparison.

# Using optimized parameters to get a sample from approximate posterior
mu_plot = trained_info[1][1]
logsigma_plot =  trained_info[1][2]
sample_z = sample_diag_gaussian(mu_plot, logsigma_plot)

# Feeding it to the decoder to get logits
bern_logit = decoder_net(sample_z)

# transform logits to means
bern_means = exp.(bern_logit) ./(1 .+ exp.(bern_logit))

# Taking the vector corresponding to the bottom of the image
bottom_vec = bern_means[393:784]

# For the top, taking the true image
top_vec = train_x[:,train_label.== 0][:,1][1:392]

# Combining the two
total_img = vcat(top_vec, bottom_vec)

#  Plotting this with the complete true image
bottom_inf_plots = repeat([plot()],2)
bottom_inf_plots[1] = plot(mnist_img_alt(total_img), title = "Bottom Half Predicted given
Top Half")
bottom_inf_plots[2] = plot(mnist_img_alt(train_x[:,2]), title = "Original Image")
display(plot(bottom_inf_plots...,layout =(1,2),size = (900, 500)))
```



4.3 Question 4 Part c:

1. Does the distribution over p(bottom half of the image x given z) factorize over the pixels of bottom half of the image?

   Answer: Yes

2. Does the distribution over p(bottom half of the image x given top half of image x) factorize over the pixels of bottom half of the image?

   Answer: No

3. When jointly optimizing the model parameters θ and variational parameters ϕ, if the ELBO increases, has the KL Divergence between the approximate posterior $q_\phi(z \mid x)$ and the true posterior $p_\theta(z \mid x)$ necessarily gotten smaller?

   Answer: Yes

4. If p(x) $\sim$ N(x $\mid$ μ,σ^2), for some x $\in \mathbb{R}$,$\mu \in \mathbb{R}$, σ in $\mathbb{R}^+$, can p(x) < 0?

   Answer: No

5. If p(x) $\sim$ N(x $\mid$ μ,σ^2), for some x $\in \mathbb{R}$,$\mu \in \mathbb{R}$, σ in $\mathbb{R}^+$, can p(x) > 1?

   Answer: Yes