Assignment 3

Chien-Che Hung 1004330164, collaborated with Jeremy Shi 1003113043

April 17, 2020

Probability Starter Codes

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

```julia
# 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=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

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();
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 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

a. log_prior

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

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

b. Decoder

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

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

c. log_likelihood

```
function log_likelihood(x,z)
  """ Compute log likelihood log_p(x|z)"""
  θ = decoder(z)
  return bernoulli_log_density(θ, x)
end
```

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

d. joint log density

```
joint_log_density(x,z) = sum(log_likelihood(x, z), dims = 1) + log_prior(z)
```

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

2 Amortized Approximate Inference and Training

a. Encoder

```
function unpack_gaussian_params(θ)
  μ, logσ = θ[1:2,:], θ[3:end,:]
  return  μ, logσ
end
encoder = Chain(Dense(Ddata, Dh, tanh), Dense(Dh, 2*Dz), unpack_gaussian_params)
```

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

b. log_q

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

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

c. elbo

```
function elbo(x)
  q_μ, q_logσ = encoder(x)
  z = sample_diag_gaussian(q_μ, q_logσ)
  joint_ll = joint_log_density(x, z)
  log_q_z = log_q(q_μ, q_logσ, z)
  elbo_estimate =  mean(joint_ll-log_q_z)
  return elbo_estimate
end
```

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

d. loss

```
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)
```

e. train*model*params

```
function train_model_params!(loss, encoder, decoder, train_x, test_x; nepochs=10)
  # model params
  ps = Flux.params(encoder, decoder)#TODO 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)
      gs = Flux.gradient(ps) do # compute gradients with respect to variational loss
over batch
        batch_loss = loss(d)
      end
      #TODO update the paramters 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
```

```
train_model_params! (generic function with 1 method)
```

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

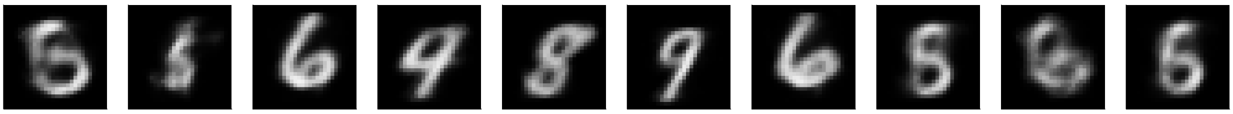
```
using Images
using Plots
# make vector of digits into images, works on batches also
mnist_img(x) = ndims(x)==2 ? Gray.(reshape(x,28,28,:))' : Gray.(reshape(x,28,28))'
```

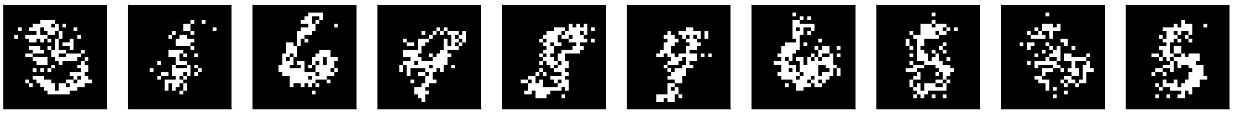
```
mnist_img (generic function with 1 method)
```

3 Visualizing Posterier and Exploring

a. Plot samples from the trained generative model using ancestral sampling:

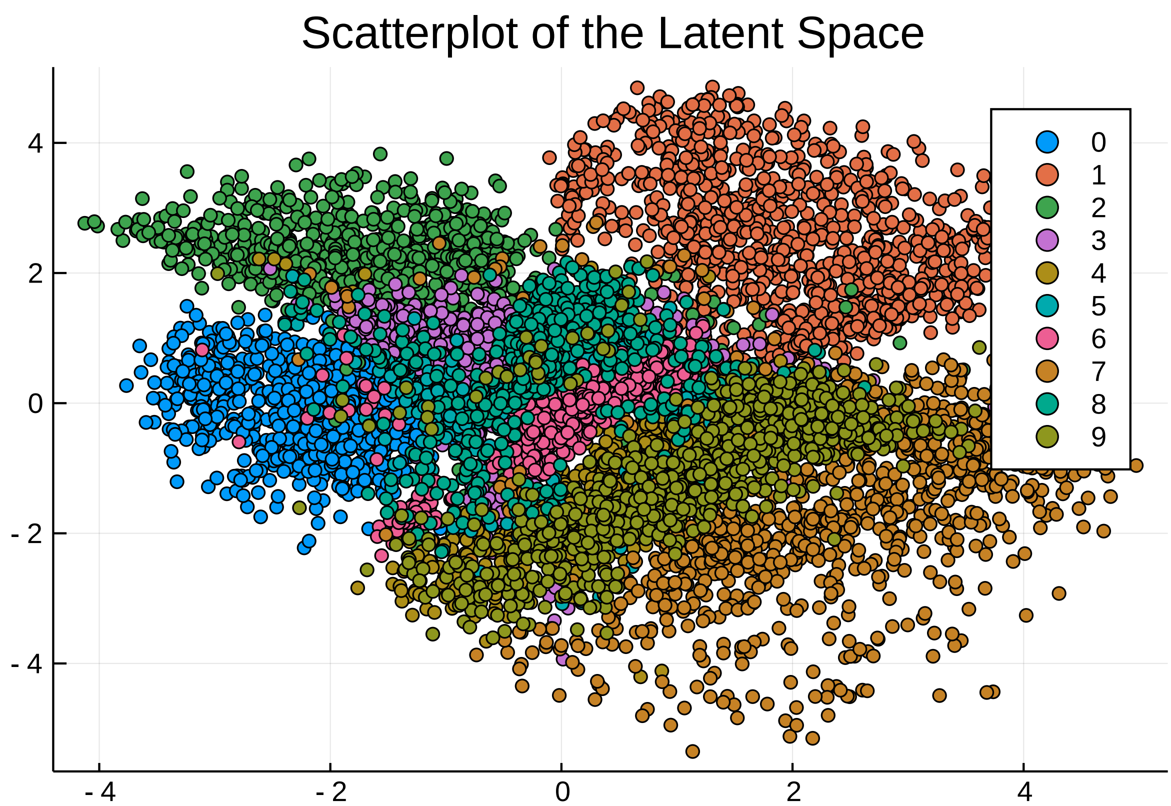
```
z = randn(2, 10)
decoded = decoder(z)
ilogit_decoded = exp.(decoded)./(exp.(decoded).+1)
bernoulli_sample = sample_bernoulli(ilogit_decoded)
plot_list = Any[]
b_1 = plot(mnist_img(ilogit_decoded[:,1]))
b_2 = plot(mnist_img(ilogit_decoded[:,2]))
b_3 = plot(mnist_img(ilogit_decoded[:,3]))
b_4 = plot(mnist_img(ilogit_decoded[:,4]))
b_5 = plot(mnist_img(ilogit_decoded[:,5]))
b_6 = plot(mnist_img(ilogit_decoded[:,6]))
b_7 = plot(mnist_img(ilogit_decoded[:,7]))
b_8 = plot(mnist_img(ilogit_decoded[:,8]))
b_9 = plot(mnist_img(ilogit_decoded[:,9]))
b_10 = plot(mnist_img(ilogit_decoded[:,10]))
c_1 = plot(mnist_img(bernoulli_sample[:,1]))
c_2 = plot(mnist_img(bernoulli_sample[:,2]))
c_3 = plot(mnist_img(bernoulli_sample[:,3]))
c_4 = plot(mnist_img(bernoulli_sample[:,4]))
c_5 = plot(mnist_img(bernoulli_sample[:,5]))
c_6 = plot(mnist_img(bernoulli_sample[:,6]))
c_7 = plot(mnist_img(bernoulli_sample[:,7]))
c_8 = plot(mnist_img(bernoulli_sample[:,8]))
c_9 = plot(mnist_img(bernoulli_sample[:,9]))
c_10 = plot(mnist_img(bernoulli_sample[:,10]))
plot_list = [b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_10, c_1, c_2, c_3, c_4, c_5,
c_6, c_7, c_8, c_9, c_10]
display(plot(plot_list..., layout = grid(2, 10), size = (2000, 1000), axis = nothing))
```





b. Scatter plot of the latent space

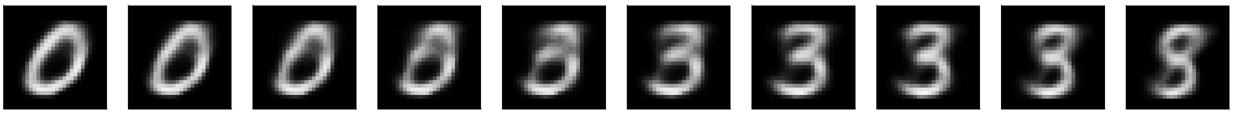
```
q_μ, q_logσ= encoder(train_x)
display(scatter(q_μ[1,:], q_μ[2,:], group= train_label, title = "Scatterplot of the
Latent Space"))
```

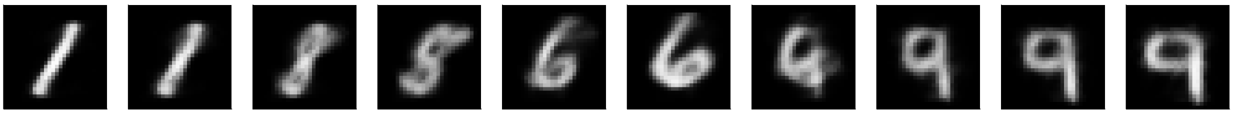


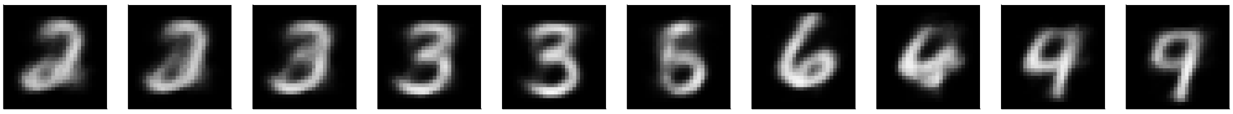
c. Interpolate between the latent representations of two points

```
function interpolate(z_a, z_b, α)
  return z_a = α.*z_a .+ (1 - α).*z_b
end

interpolate (generic function with 1 method)
```

```julia
### First Pair
steps = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]/10
elem_1 = train_x[:,1]
elem_2 = train_x[:,2]
μ1_1, sig1_1 = encoder(elem_1)
μ1_2, sig1_2= encoder(elem_2)

plot_1 = Any[]
for i in steps
  z = interpolate(μ1_1, μ1_2, i)
  decoded = decoder(z)
  μ = vec(exp.(decoded)./(exp.(decoded).+1))
  push!(plot_1, plot(mnist_img(μ)))
end

### Second Pair
steps = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]/10
elem_1 = train_x[:,3]
elem_2 = train_x[:,4]
μ1_1, sig1_1 = encoder(elem_1)
μ1_2, sig1_2= encoder(elem_2)

plot_2 = Any[]
for i in steps
  z = interpolate(μ1_1, μ1_2, i)
  decoded = decoder(z)
  μ = vec(exp.(decoded)./(exp.(decoded).+1))
  push!(plot_2, plot(mnist_img(μ)))
end

### Third Pair
steps = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]/10
elem_1 = train_x[:,5]
elem_2 = train_x[:,6]
μ1_1, sig1_1 = encoder(elem_1)
μ1_2, sig1_2= encoder(elem_2)
plot_3 = Any[]
for i in steps
  z = interpolate(μ1_1, μ1_2, i)
  decoded = decoder(z)
  μ = vec(exp.(decoded)./(exp.(decoded).+1))
  push!(plot_3, plot(mnist_img(μ)))
end
total_plot = vcat(plot_1, plot_2, plot_3)

# Display the Interpolation Results
display(plot(total_plot..., layout = grid(3, 10), size = (2000, 1000), axis = nothing))
```







4 Predicting the bottom of Images given the top

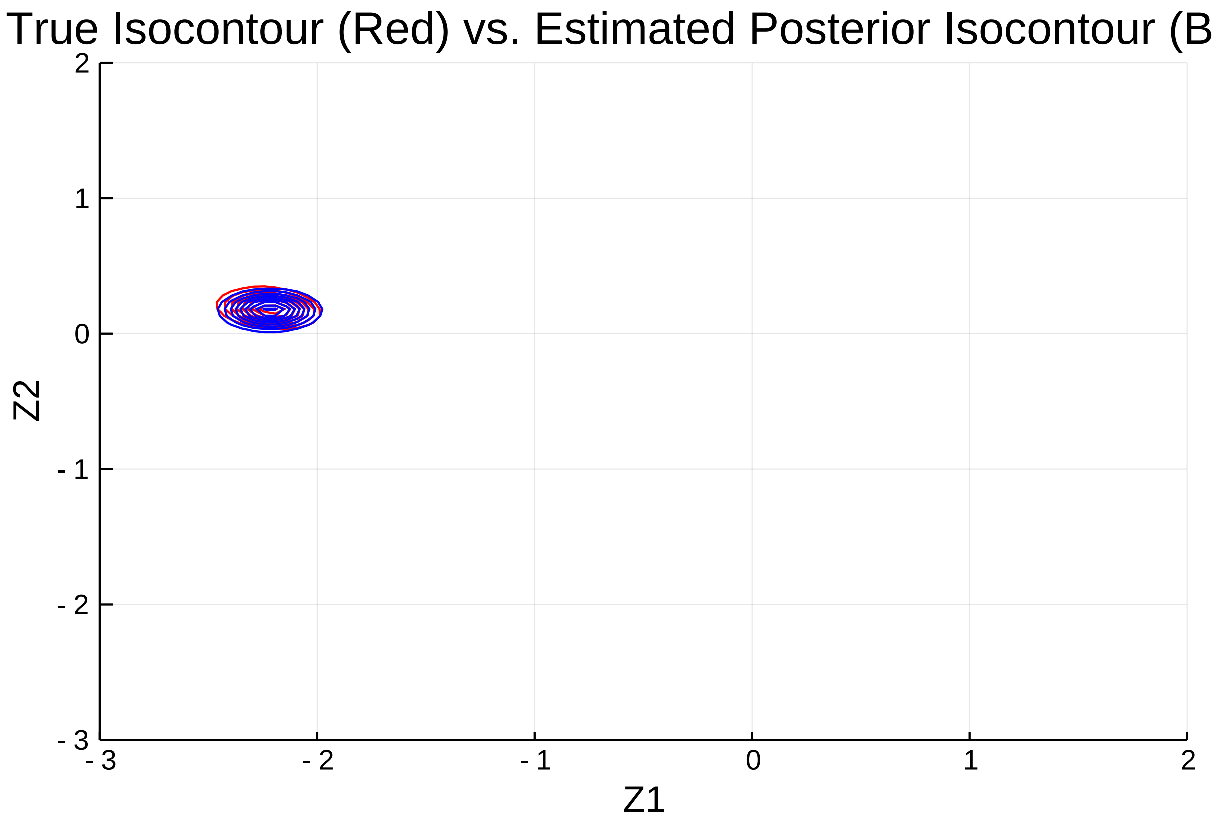
a. P(z, top half of image x)

```
#top half of a 28x28 array
function top_half(x)
  return x[1:392,:]
end

#logp(top half of image x|z)
function llp_top_give_z(x, z)
  θ = top_half(decoder(z))
  likelihoods = sum(bernoulli_log_density(θ, top_half(x)), dims = 1)
  return likelihoods
end

#logp(z, top half of image x)
function log_top_joint(x, z)
  return log_prior(z) + llp_top_give_z(x, z)
end

#top_elbo
function top_elbo(params, x, num_samples)
  q_μ, q_logσ = params
  zs = exp.(q_logσ) .*  randn(length(q_μ), num_samples) .+ q_μ
  joint_ll = log_top_joint(x, zs)
  log_q_z= log_q(q_μ, q_logσ, zs)
  elbo_estimate = mean(joint_ll -log_q_z)
  return elbo_estimate
end
#negative top_elbo
function neg_top_elbo(params,x ;num_sample = 10)
  return -top_elbo(params, x, num_sample)
end
```

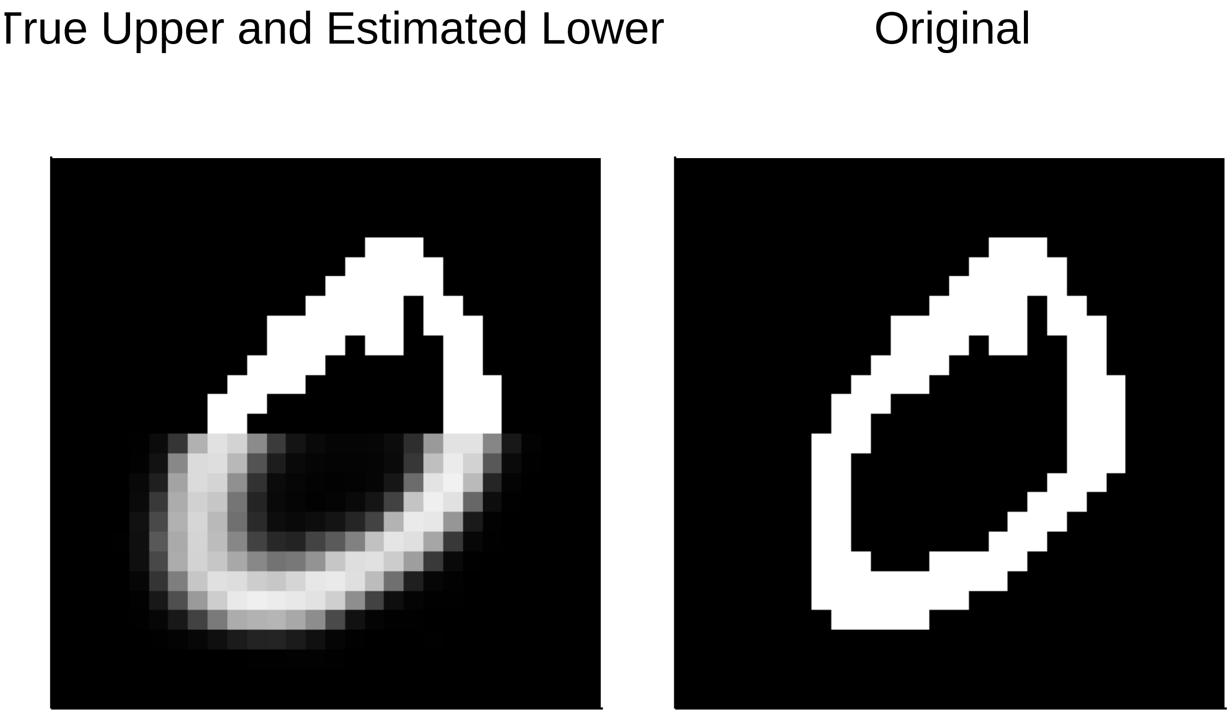
```
neg_top_elbo (generic function with 1 method)
```

b. **Approximate P(z|top half of image x)**

```
#Initialization
choice = 2
init_mu = randn(Dz)
init_log_sigma = randn(Dz)/100
init_params = (init_mu, init_log_sigma)
#training function with elbo function
function optimize_phi(init_param, x; iter = 200, lr = 1e-2, num_sample = 50)
  params_cur = collect(init_param)
  for i in 1:iter
    grad_params = gradient((param)-> neg_top_elbo(param, x, num_sample=num_sample),
params_cur)[1]
    params_cur[1] = params_cur[1] - lr * grad_params[1]
    params_cur[2] = params_cur[2] - lr * grad_params[2]
    elbo = -neg_top_elbo(params_cur, x)
    @info "Elbo at iter $i: $(elbo)"
  end
  return params_cur
end
#Optimize the parameters
optim_param = optimize_phi(init_params, train_x[:,choice])
#Isocontour function from A2
function skillcontour!(f; colour=nothing)
  n = 100
  x = range(-3,stop=2,length=n)
  y = range(-3,stop=2,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
#Isocontour Plot
plot(xlabel = "Z1", ylabel = "Z2", title = "True Isocontour (Red) vs. Estimated
Posterior Isocontour (Blue)")
log_estp(zs) = exp(log_top_joint(train_x[:, choice], zs))
skillcontour!(log_estp, colour =:red)
log_estq(zs) = exp(factorized_gaussian_log_density(optim_param[1], optim_param[2], zs))
skillcontour!(log_estq, colour =:blue)
```




```
#Posterior vs. True
decoded = decoder(optim_param[1])
μ = vec(exp.(decoded)./(exp.(decoded).+1))
orig_upper = train_x[:,choice][1:392]
post_lower = μ[393:end, :1]
post_whole = [orig_upper; post_lower]
post = plot(mnist_img(post_whole), title = "True Upper and Estimated Lower", axis =
nothing)
org = plot(mnist_img(train_x[:, choice]), title = "Original", axis = nothing)
display(plot(post, org))
```



c. **True or False**

1. True
2. False
3. False
4. False
5. True