```
# import Automatic Differentiation
# You may use Neural Network Framework, but only for building MLPs
# i.e. no fancy probabilistic implementations
import Pkg;Pkg.add("Flux")
```

```
Resolving package versions...
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
```

```
import Pkg;Pkg.add("MLDatasets")
```

```
Resolving package versions...
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
```

```
import Pkg;Pkg.add("BSON")
```

```
Resolving package versions...
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
```

```
import Pkg;Pkg.add("Images")
```

```
Resolving package versions...
  Updating `~/.julia/environments/v1.3/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.3/Manifest.toml`
 [no changes]
```

```
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!
function skillcontour!(f; colour=nothing)
  n = 100
```

```
  x = range(-5,stop=3,length=n)
  y = range(-5,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=fa
lse)
  end
  plot!(p1)
end

function plot_line_equal_skill!()
  plot!(range(-3, 3, length=200), range(-3, 3, length=200), label="Equa
l Skill")
end
# 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),dim
s=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_densit
y(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, test_size)
  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))
function skillcontour!(f; colour=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=fa
lse)
  end
  plot!(p1)
end

function plot_line_equal_skill!()
  plot!(range(-3, 3, length=200), range(-3, 3, length=200), label="Equa
l Skill")
end

### Implementing the model

## Load the Data
batch_size = 10000
train_data, test_data = load_binarized_mnist(batch_size, batch_size)
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,batch_size)
  @test size(train_label) == (batch_size,)
  @test size(test_x) == (784,batch_size)
  @test size(test_label) == (batch_size,)
end
```

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

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

# ## Generative Model
# This will require implementing a simple MLP neural network
# See example_flux_model.jl for inspiration
# Further, you should read the Basics section of the Flux.jl documentat
ion
# https://fluxml.ai/Flux.jl/stable/models/basics/
# that goes over the simple functions you will use.
# You will see that there's nothing magical going on inside these neura
l network libraries
# and when you implemented a neural network in previous assignments you
```

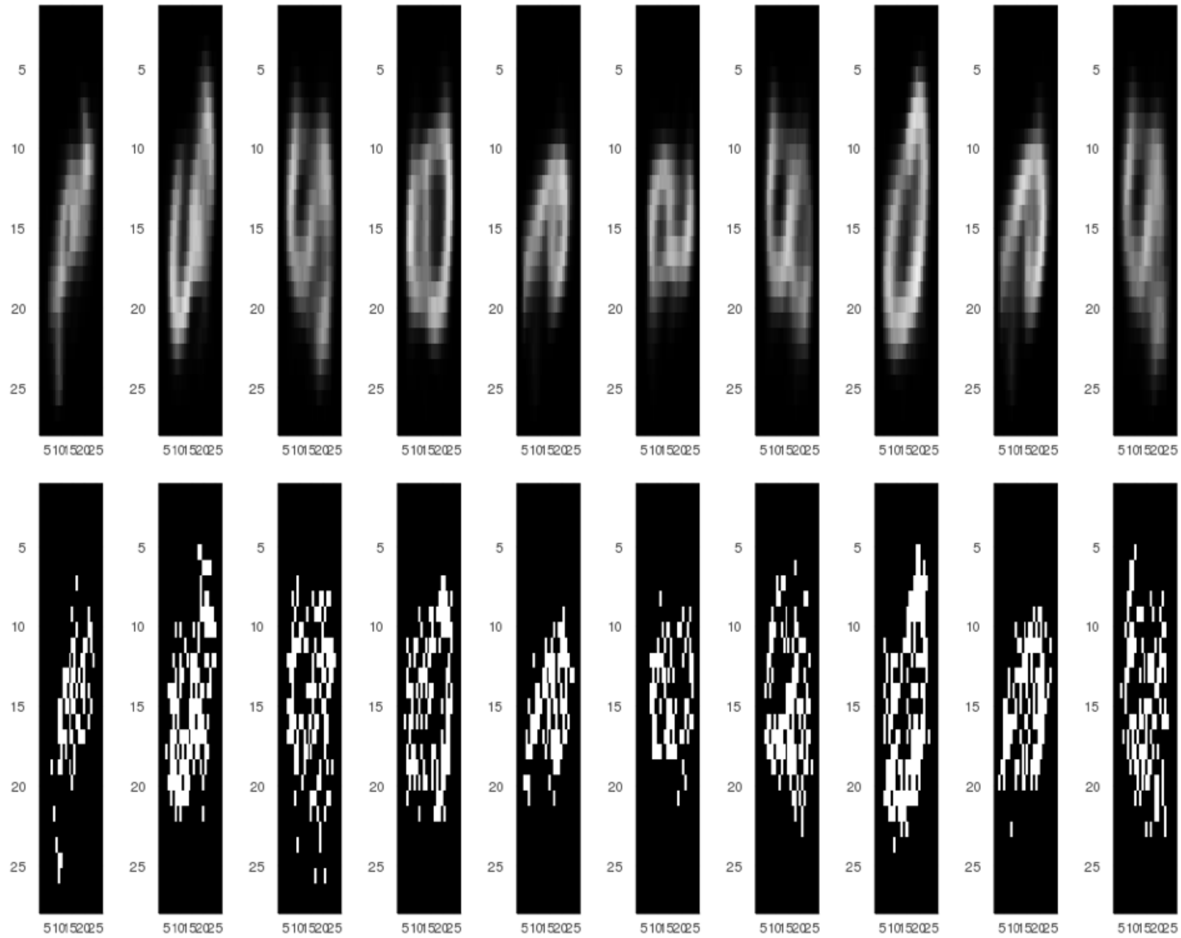
```
 did most of the work.
# If you want more information about how to use the functions from Flux
, you can always reference
# the internal docs for each function by typing `?` into the REPL:
# ? Chain
# ? Dense
#Q1(b)
decoder = Chain(Dense(Dz, Dh, tanh), Dense(Dh, Ddata))
## Model Distributions
#Q1(a)
log_prior(z) =  factorized_gaussian_log_density(0, 0, z)#TODO

#Q1(c)
function log_likelihood(x,z)
  """ Compute log likelihood log_p(x|z)"""
  d = decoder(z)# TODO: parameters decoded from latent z
  return  sum(bernoulli_log_density(d, x), dims = 1) # return likelihoo
d for each element in batch
end

#Q1(d)
joint_log_density(x,z) = log_prior(z) .+ log_likelihood(x,z)#TODO

## Amortized Inference
function unpack_gaussian_params(θ)
  μ, logσ = θ[1:2,:], θ[3:end,:]
  return  μ, logσ
end
#Q2(a)
encoder = Chain(Dense(Ddata,Dh,tanh), Dense(Dh,2*Dz), unpack_gaussian_p
arams)#TODO
# Hint: last "layer" in Chain can be 'unpack_gaussian_params'

#Q2(b)
log_q(q_μ, q_logσ, z) = factorized_gaussian_log_density(q_μ, q_logσ, z)
#TODO: write log likelihood under variational distribution.

#Q2(c)
function elbo(x)
  q_μ, q_logσ = encoder(x)#TODO variational parameters from data
  z = sample_diag_gaussian(q_μ, q_logσ)#TODO: sample from variational d
istribution
  log_joint_ll = joint_log_density(x,z) #TODO: log joint density of z a
nd x under model
  log_q_z = log_q(q_μ, q_logσ, z)#TODO: log likelihood of z under varia
tional distribution
  elbo_estimate = mean(log_joint_ll - log_q_z) #TODO: Scalar value, mea
n variational evidence lower bound over batch
  return elbo_estimate
end

#Q2(d)
```

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

# Training with gradient optimization:
# See example_flux_model.jl for inspiration

#Q2(e)
function train_model_params!(loss, encoder, decoder, train_x, test_x; n
epochs=10)
  # model params
  ps = Flux.params(encoder,decoder)#TODO parameters to update with grad
ient descent
  # ADAM optimizer with default parameters
  opt = ADAM()
  # over batches of the data
  for i in 1:nepochs
    # compute gradients with respect to variational loss over batch
    for d in batch_x(train_x)
      gs = Flux.gradient(ps) do
        batch_loss = loss(d)
        return batch_loss
      end
      #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 f
requently
      @info "Test loss at epoch $i: $(loss(batch_x(test_x)[1]))"
    end
  end
  @info "Parameters of encoder and decoder trained!"
end
## 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"

### 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
```

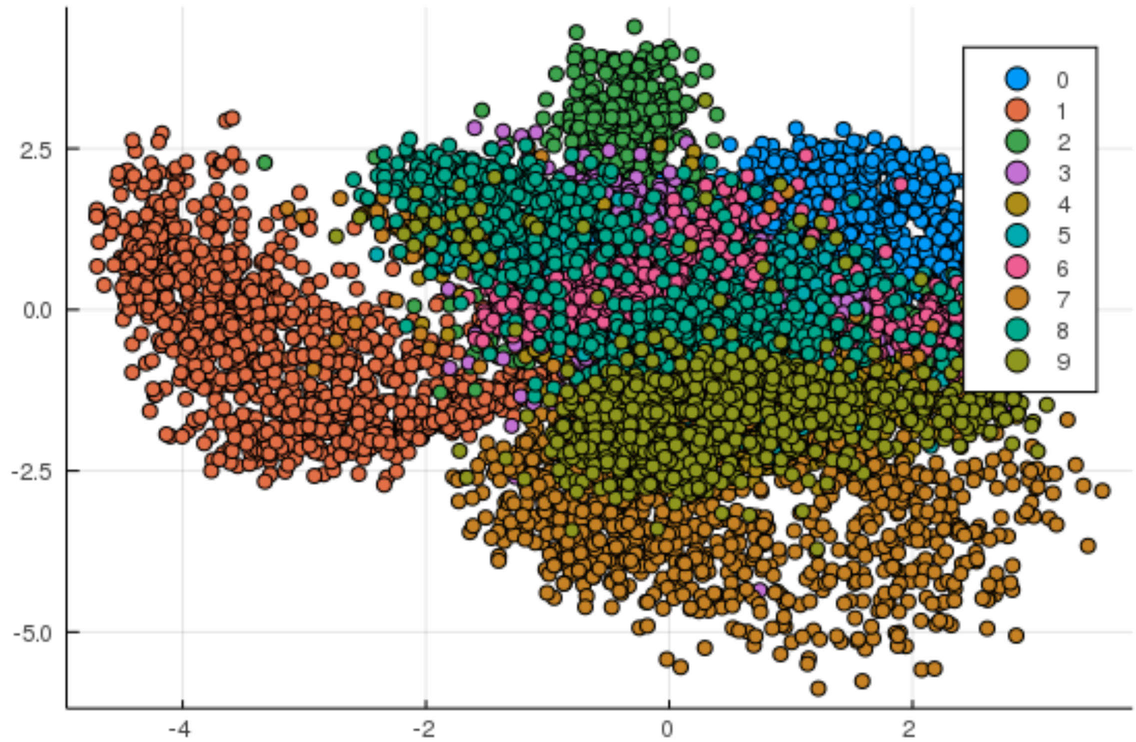
```
@save joinpath(save_dir,"encoder_params.bson") encoder
@save joinpath(save_dir,"decoder_params.bson") decoder
@info "Saved model params in $save_dir"

## Train the model(already done)
#train_model_params!(loss,encoder,decoder,train_x,test_x, nepochs=10)
@show  "final elbo is $(loss(batch_x(test_x)[1]))"
```

```
"final elbo is $(loss((batch_x(test_x))[1]))" = "final elbo is 159.819
04829
745534"
```

```
# 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))

## Example for how to use mnist_img to plot digit from training data
plot(mnist_img((train_x[:,1])'[1,:]))
p = plot(mnist_img(train_x[:,1]))
#Q3(a)
plot_list1 = Any[]
plot_list2 = Any[]
for i in 1:10
  z = randn(2,)
  log_mean = decoder(z)
  ber_mean = exp.(log_mean) ./ (1 .+exp.(log_mean))
  samp = sample_bernoulli(ber_mean)
  push!(plot_list1, plot(mnist_img(ber_mean) ))
  push!(plot_list2, plot(mnist_img(samp) ))
end
plot_list = vcat(plot_list1, plot_list2)
display(plot(plot_list..., layout = grid(2, 10), size = (1000, 800)))
```



```
#Q3(b)
num = 10000
mean_vec = encoder(train_x[:,1:num])[1]
display(scatter(mean_vec[1,:], mean_vec[2,:], group = train_label[1:num
]))
```

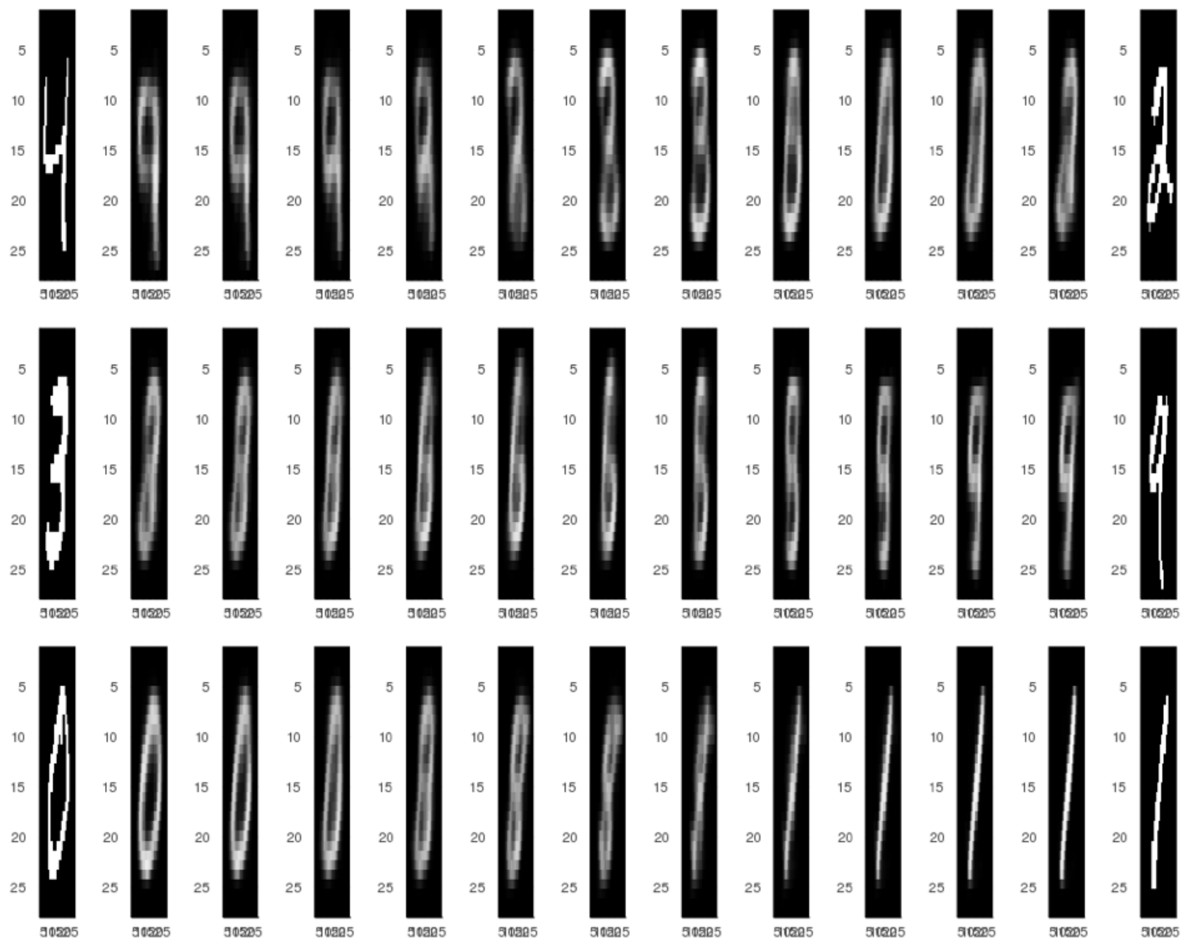



```
#Q3(c)
function interpolate(za,zb,α)
  return α .* za .+ (1-α) .* zb
end
function find_first_occ(x,A)
    for i in 1:length(A)
      if A[i] == x
         return i
      end
    end
    return -1
end
function bern_mean(z)
  logit_mean = decoder(z)
  return exp.(logit_mean) ./ (1 .+ exp.(logit_mean))
end
function plot_latent(z)
  return plot(mnist_img(bern_mean(z))')
end
#plot interpolation graphs for label a and b
function plot_interp(a,b)
  result = Any[]
  indices = [find_first_occ(i,train_label) for i in [a,b]]
  pair_X = train_x[:,indices]
  #display(plot(mnist_img(pair_X[:,1])'))
  push!(result,plot(mnist_img(pair_X[:,1])'))
  means = encoder(pair_X)[1]
  α = 0:0.1:1
  for alpha in α
    z = interpolate(means[:,2], means[:,1],alpha)
```

```
    plot = plot_latent(z)
    #display(plot)
    push!(result, plot)
  end
  #display(plot(mnist_img(pair_X[:,2])'))
  push!(result, plot(mnist_img(pair_X[:,2])'))
  return result
end
final_plot = vcat(plot_interp(4,2), plot_interp(3,9), plot_interp(0,1))
@show "I've included the original binary images as well for comparison.
"
```

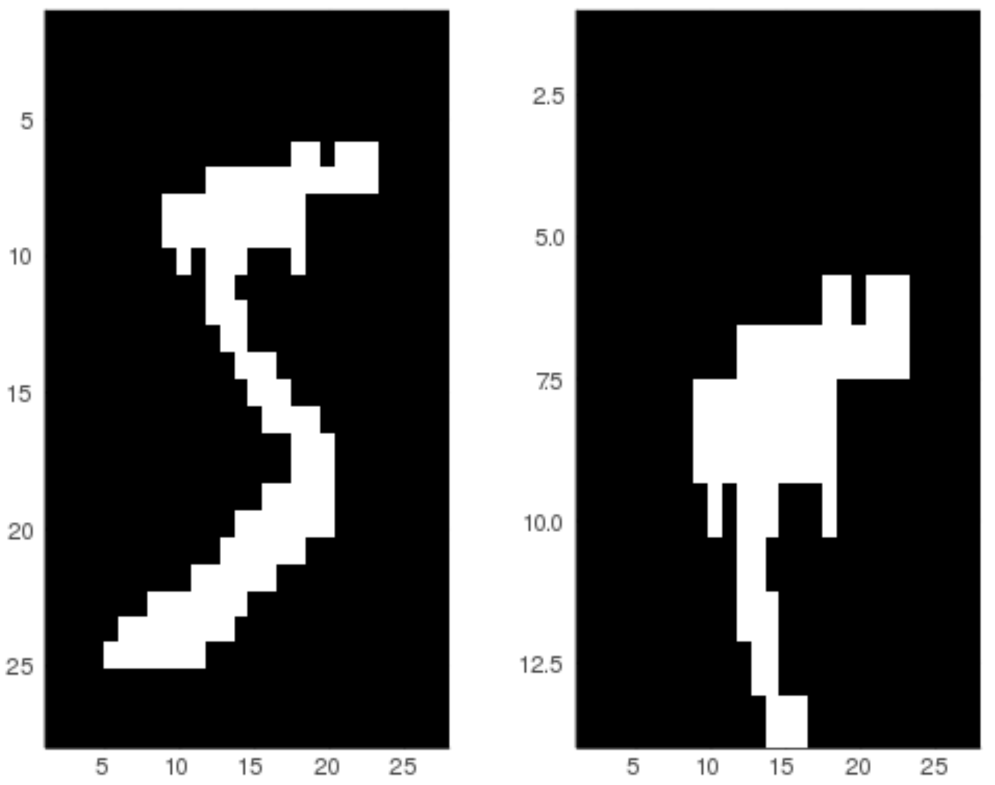
```
"I've included the original binary images as well for comparison." = "
I've
included the original binary images as well for comparison."
```

```
display(plot(final_plot..., layout = grid(3,13), size = (1000,800)))
```



```
#Q4(a)
Dtop = convert(Int,0.5*Ddata)
#note: although this function accepts only 784*1 arrays, I've written a
nother function
#to help plot only half the graph
function tophalf(A)
  return A[1: Dtop, :]
```

```
end
#function to plot half of the graph
mnist_img_half(x) = ndims(x)==2 ? Gray.(reshape(x,28,14,:)) : Gray.(res
hape(x,28,14))
a = plot(mnist_img(train_x[:,1])')
b = plot(mnist_img_half(tophalf(train_x[:,1])[:,1] )')
display(plot([a,b]..., layout = grid(1,2), size = (500,400)))
```



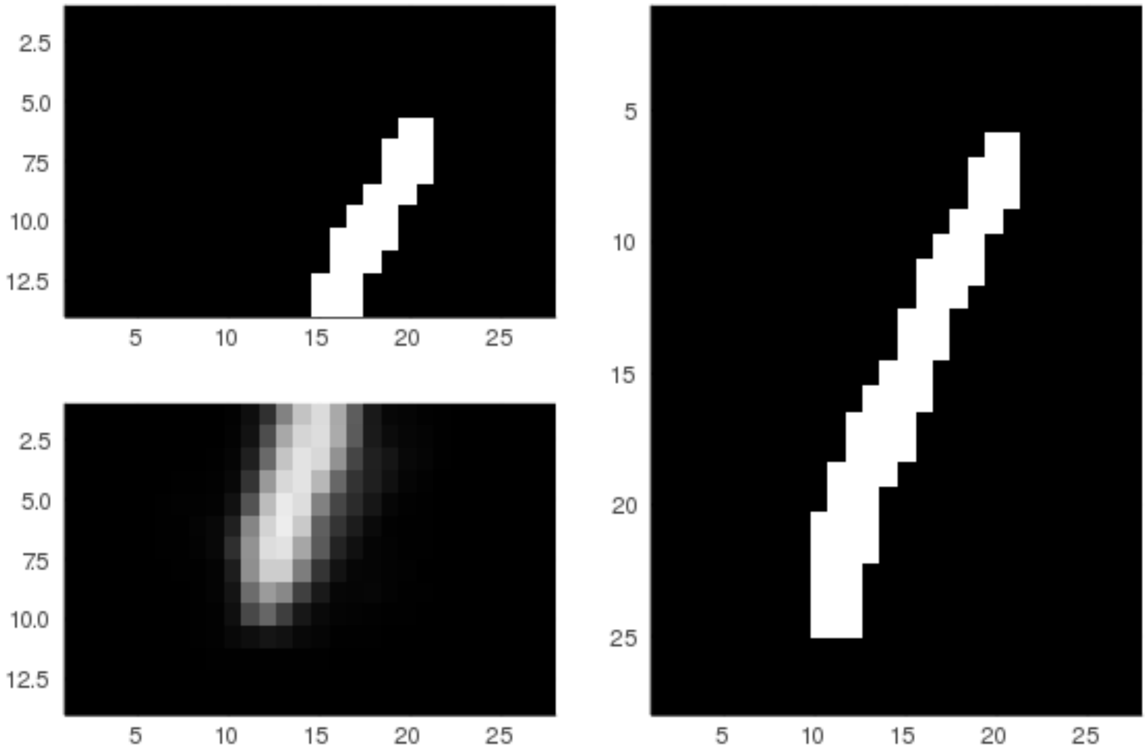
```
## Example, left is original, right is tophalf
#p(xtop | z)
function logp_tophalf(xtop,z)
  d = decoder(z)
  return sum(bernoulli_log_density(d[1:Dtop,:], xtop), dims = 1)
end
function log_joint_tophalf(xtop,zs)
  return log_prior(zs) .+ logp_tophalf(xtop,zs)
end
#Q4(b)
#pick digit = 1
train_dig1_ind = [i for i in 1:num if train_label[i] == 1]
train_label1 = train_label[train_dig1_ind]
train_x1 = train_x[:, train_dig1_ind]
M = (length(train_label1) ÷ 100 )*100
#initialize variational parameters
q_mean, q_logsig = param(rand(Dz,) ) , param(rand(Dz,))
#compute elbo estimate
function neg_elbo_top(xtop, q_mean, q_logsig)
  #q_means = repeat(q_mean, 1, K)
  #q_logsigs = repeat(q_logsig, 1, K)
```

```
  #sum = 0
  #for k in 1:K
    z = sample_diag_gaussian(q_mean, q_logsig)
  #z = sample_diag_gaussian(q_means, q_logsigs)
    log_joint_ll = log_joint_tophalf(xtop, z) #TODO: log joint density of z and x under model
    log_q_z = factorized_gaussian_log_density(q_mean, q_logsig, z)#TODO: log likelihood of z under variational distribution
    elbo_est = mean(log_joint_ll .- log_q_z)
    #sum = sum + elbo_est
  #end
  return -elbo_est
end
# function to train the model
function train_tophalf_para(train_x1, q_mean, q_logsig, loss_function;n = 10)
  # ADAM optimizer with default parameters
  ps = Flux.params([q_mean, q_logsig])
  opt = ADAM()
  # over batches of the data
  x_first = tophalf(batch_x(train_x1[:,1:M])[1])
  for i in 1:n
    for x in batch_x(train_x1[:,1:M])
      xtop = tophalf(x)
      gs = Flux.gradient(() -> loss_function(xtop), ps)
      #gs = Flux.gradient(ps) do
      #  return loss_function(xtop)
      #end
      Flux.Optimise.update!(opt,ps,gs)
    end
    if i%20 == 0 # change 1 to higher number to compute and print less frequently
      @info "Variational Test loss at epoch $i: $(loss_function(x_first))"
    end
  end
end
q_mean, q_logsig = randn(2,), randn(2,)
@info "qmean, qlogsig is $((q_mean, q_logsig))"
loss_top(xtop) = neg_elbo_top(xtop, q_mean, q_logsig)
#training parameters, already done
#train_tophalf_para(train_x1,q_mean,q_logsig, loss_top; n = 1000)
@info "qmean, qlogsig is $((q_mean, q_logsig))"

#load training models
using BSON:@load
cd(@__DIR__)
@info "Changed directory to $(@__DIR__)"
load_dir = "trained_models"
@load joinpath(load_dir,"q_mean.bson") q_mean
@load joinpath(load_dir,"q_logsig.bson") q_logsig
@info "Load model params from $load_dir"
```

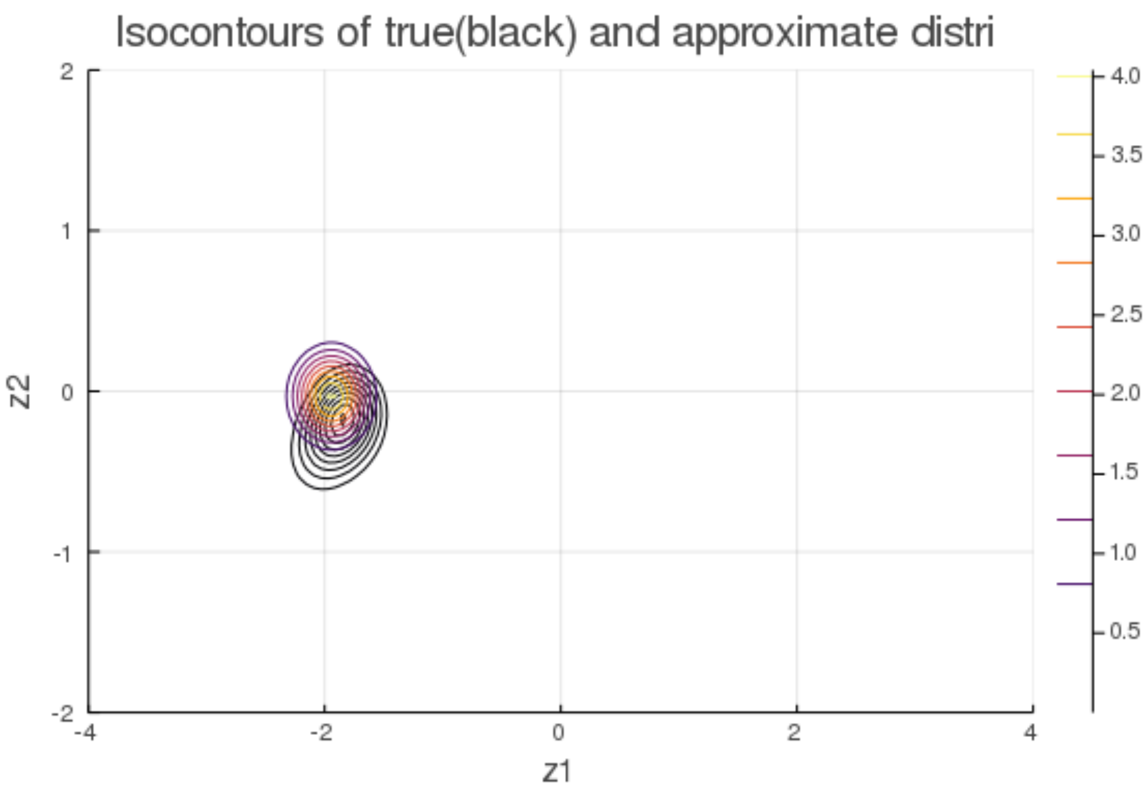
```
@info "Load model params from $load_dir"

### 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,"q_mean.bson") q_mean
@save joinpath(save_dir,"q_logsig.bson") q_logsig
@info "Saved model params in $save_dir"

Xtop = tophalf(batch_x(train_x1[:,1:M])[1])
@show "final test loss is $(loss_top(Xtop))"
```

```
"final test loss is $(loss_top(Xtop))" = "final test loss is 58.144529
91509
796"
```

```
#sample a random z
zs = randn(Dz) .* exp.(q_logsig) .+ q_mean
logit_mean = decoder(zs)
ber_mean = exp.(logit_mean) ./ (exp.(logit_mean) .+ 1)
lower = 393:784
lowerhalf_mean = ber_mean[lower]
#plot the original image and variational one
  plot_lower = plot(mnist_img_half(lowerhalf_mean[:,1] )')
  original = train_x1[:,1]
  plot_original = plot(mnist_img(original)')
  plot_upper = plot(mnist_img_half(original[1:Dtop])')
  plot_appr = plot([plot_upper,plot_lower]..., layout = grid(2,1))
  display(plot([plot_appr, plot_original]..., layout = grid(1,2)))
```



```
# plotting isocontours
#using all data
plot(title="Isocontours of true(black) and approximate distri",
    xlabel = "z1",
    ylabel = "z2",
    xlim = [-4,4],
    ylim = [-2,2]
   )
X = tophalf(batch_x(train_x1[:,1:M])[1])
true_joint(zs) = exp.(mean(log_joint_tophalf(X, zs)))
skillcontour!(true_joint)
appr_inf(zs) = exp.(mean(factorized_gaussian_log_density(q_mean, q_logs
ig,zs )))
display(skillcontour!(appr_inf))
```




```
#Q4(c)
print("My answer is TFFFT.")
```

```
My answer is TFFFT.
```

Published from vae.jl using Weave.jl on 2020-04-18.