

Assignment 3 Variational Autoencoders

Gang Peng # 1002961921

April 9, 2020

1 Implementing the Model [5 points]

```
# Load all the packages, provided functions, in order for the jmd to compile
using Flux
using MLDatasets
using Statistics
using Logging
using Test
using Random
using StatsFuns: log1pexp, logit
using Distributions
Random.seed!(412414);

# log-pdf of x under Factorized or Diagonal Gaussian  $N(x|\mu, \sigma I)$ 
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  $\mu/(1-\mu)$ """
    b = x .* 2 .- 1 # {0,1} -> {-1,1}
    return -log1pexp.(-b .* logit_means)
end

# sample from Diagonal Gaussian  $x \sim N(\mu, \sigma I)$  (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))

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

# 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 = MNISTtestdata(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))

## 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

(a) [1 point] Implement a function log_prior that computes the log of the prior.

```

function log_prior(z)
    factorized_gaussian_log_density(0,0,z)
end

log_prior (generic function with 1 method)

```

(b) [2 points] Implement a function decoder.

```

decoderm = Chain(Dense(Dz, Dh, tanh), Dense(Dh, Ddata)) # The MLP model outside the
function to pass to train function

```

```

function decoder(z)
    m = decoderm(z)
    return m
end

decoder (generic function with 1 method)

```

(c) [1 point] Implement a function log_likelihood

```

function log_likelihood(x,z)
    """ Compute log likelihood log_p(x|z)"""

```

```

    return sum(bernoulli_log_density(decoder(z),x), dims=1) # return likelihood for each
element in batch
end

```

log_likelihood (generic function with 1 method)

(d) [1 point] Implement a function joint logdensity.

```

function joint_log_density(x,z)
    return log_prior(z) .+ log_likelihood(x,z)
end

```

joint_log_density (generic function with 1 method)

2 Amortized Approximate Inference and training [13 points]

(a) [2 points] Write a function encoder.

```

encoderm=Chain(Dense(Ddata, Dh, tanh),Dense(Dh, Dz*2),)#The MLP model outside the
function to pass to train function

```

```

function encoder(x)
    m = unpack_gaussian_params(encoderm(x))
    return m
end

```

encoder (generic function with 1 method)

(b) [1 points] Write a function log_q.

```

function log_q(q_mu, q_logsigma, z)
    return factorized_gaussian_log_density(q_mu, q_logsigma, z)
end

```

log_q (generic function with 1 method)

(c) [5 points] Implement a function elbo.

```

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

```

elbo (generic function with 1 method)

(d) [2 points] Write a loss function called loss that returns the negative elbo estimate over a batch of data.

```

function loss(x)
    return -1 .* elbo(x)#scalar value for the variational loss over elements in the
batch
end

```

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

- (e) [3 points] Write 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=10)
    # 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)
            gs = Flux.gradient(ps) do
                batch_loss = loss(d)
            return batch_loss# compute gradients with respect to variational loss over
batch
        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 frequently
        @info "Test loss at epoch $i: $(loss(batch_x(test_x)[1]))"
    end
    end
    @info "Parameters of encoder and decoder trained!"
end

## Load the Data
train_data, test_data = load.binarized_mnist(10000, 10000)
train_x, train_label = train_data;
test_x, test_label = test_data;
```

Use the following line to train the model, but comment out during compiling of jmd as it takes long time. The result was saved using provided method.

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

Final ELBO on test set:

```
# Load the saved result from training
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"

elbo(test_x)
```

```
-155.66149211739722
```

3 Visualizing Posteriors and Exploring the Model [15 points]

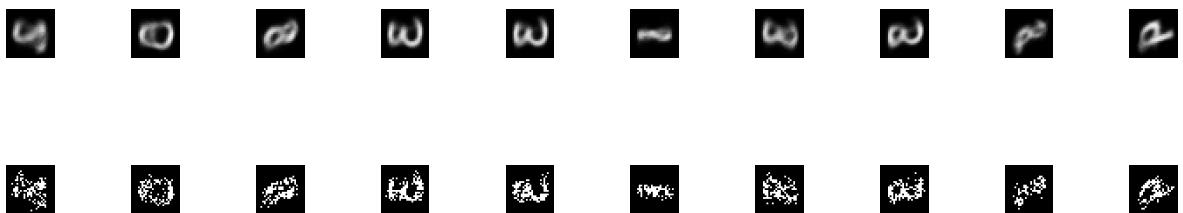
- (a) [5 points] Plot samples from the trained generative model using ancestral sampling.

```
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))

plot_list=Matrix{Union{Nothing, Any}}(nothing, 2, 10)
for i in 1:10
    z = randn(Dz,1)
    logitz = decoder(z)
    means = exp.(logitz)./ (1 .+ exp.(logitz))
    x = sample_bernoulli(means)
    plot_list[1,i] = plot(mnist_img(vec(means)),framestyle=:none)
    plot_list[2,i] = plot(mnist_img(vec(x)),framestyle=:none)
end

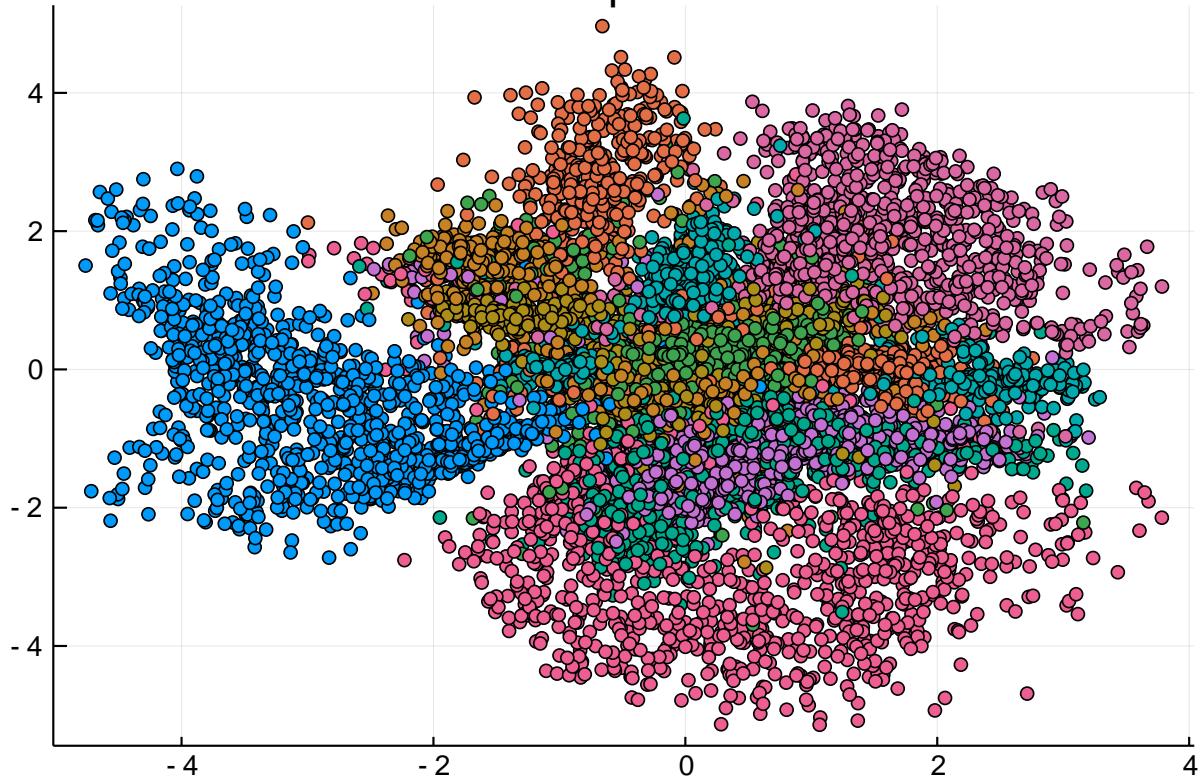
display(plot(plot_list[1,:]...,plot_list[2,:]...,layout=(2,10),size = (800,200)))
```



- (b) [5 points] Produce a scatter plot in the latent space, where each point in the plot represents a different image in the training set.

```
p1=plot()
q_μ= encoder(train_x[:,1])[1]
p1=plot!(q_μ[1,:], q_μ[2,:], seriestype = :scatter, title = "Latent Space Plot",
          color=train_label[1], legend=false)
for i in 2:10000
    d= train_x[:,i]
    q_μ= encoder(d)[1]
    colour=train_label[i]
    plot!(p1, q_μ[1,:], q_μ[2,:], seriestype = :scatter, color=colour,legend=false)
end
display(p1)
```

Laten Space Plot



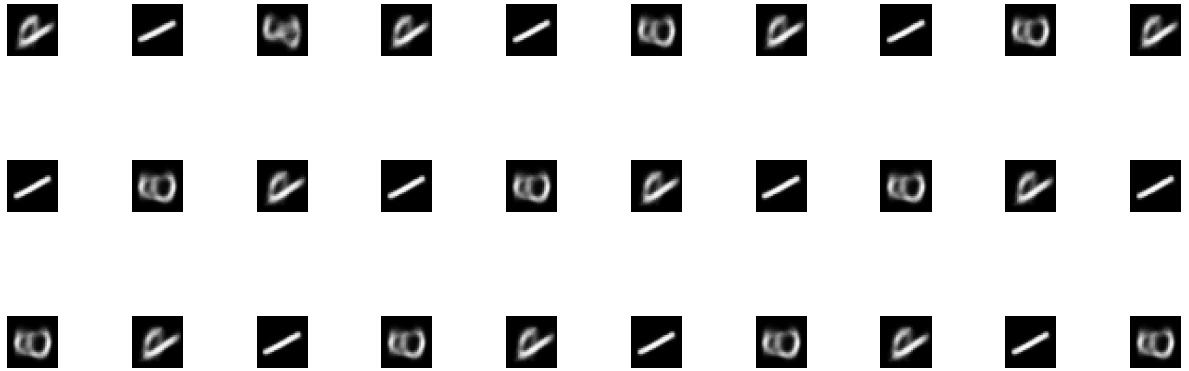
(c) Display generative distribution along linear interpolation

```

function linear_interp(Za, Zb, alpha)
    Za = alpha .* Za .+ (1 .- alpha) .* Zb
    return Za
end

plot_list=Matrix{Union{Nothing, Any}}(nothing, 3, 10)
p2=plot()
for i in 1:3
    x = rand(1:size(train_x)[2])
    y = rand(1:size(train_x)[2])
    while train_label[x] == train_label[y]
        y = rand(1:size(train_x)[2])
    end
    Za = encoder(train_x[:,x])[1]
    Zb = encoder(train_x[:,y])[1]
    for j in 1:10
        alpha = j/10
        Za = linear_interp(Za, Zb, alpha)
        logitz = decoder(Za)
        means=exp.(logitz) ./ (1 .+ exp.(logitz))
        plot_list[i,j] = plot(mnist_img(vec(means)),framestyle=:none)
    end
end
display(plot(plot_list...,layout=(3,10),size = (800,300)))

```



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

(a) [5 points] Write a function that computes $p(z, \text{top half of image } x)$. Note the image plotted by given `mnist_img(x)` function needs rotate to normal direction. So here the reshaped single digit 28×28 data is transposed.

```
# function which returns only the top half of a 28 by 28 array.
function top_half(x)
    x = permutedims(reshape(x,28,28)) # The digits in mnist need rotate
    return x[1:14,:]
end

#function that computes log p(top half of image x/z)
function logp_top(x,z)
    z_top = decoder(z)[1:392,:]
    sum_top = sum(bernoulli_log_density(z_top,x),dims=1)
    z_bottom = decoder(z)[393:784,:]
    kones = ones(1,k)
    kzzeros = zeros(1,k)
    # Integrate over the bottom half for every pixels being 1 and 0
    sum_bottom = sum(log.(exp.(bernoulli_log_density(z_bottom,kones)))
        .+ exp.(bernoulli_log_density(z_bottom,kzzeros))),dims=1)
    return sum_top .+ sum_bottom
end

# Function compute log joint density log p(z, top half of image x)
function log_joint_density(x,z)
    return log_prior(z) .+ logp_top(x, z)
end

log_joint_density (generic function with 1 method)
```

(b) [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).

```
using Zygote

# Function for compute ELBO
function elbosv(params, logp, k)
```

```

z = sample_diag_gaussian(params[1],params[2])
log_q_z = log_q(params[1], params[2], z)
logp = logp(z)
return sum(logp .- log_q_z) ./k
end

# Function for compute loss
function loss(params, x, k)
    logp(z) = log_joint_density(x,z)
    return -1 .* elbosv(params,logp, k)
end

# Function for training to optimize parameters by maximize ELBO
function opti_elbo(init_params, x; num_itrs=200, lr= 1e-2, k = 10)
    params_cur = init_params
    elbo_val = loss(params_cur, x, k)
    for i in 1:num_itrs
        f(param) = loss(param, x, k)
        grad_params = gradient(f, params_cur)[1]
        params_cur = params_cur .- grad_params .* lr
        elbo_val = loss(params_cur, x, k)
        @info "loss: $(elbo_val)"
    end
    return params_cur, elbo_val
end

k = 100
init_params = (randn(Dz,k), randn(Dz,k)) #Initialize variational parameters
x=top_half(train_x[:,2]) # Choose the second digit of mnist data set, which is the
# image of "0"
x = reshape(x, 392,1)
# Traing the data, optimaize parameters by maximize ELBO
trained_params = opti_elbo(init_params, x; num_itrs=200, lr= 1e-2, k = 100)

# Function for plot contour, basically the same from assignment 2
function plotcontour!(f; colour=nothing)
    n = 100
    x = range(-3,stop=4,length=n)
    y = range(-3,stop=4,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

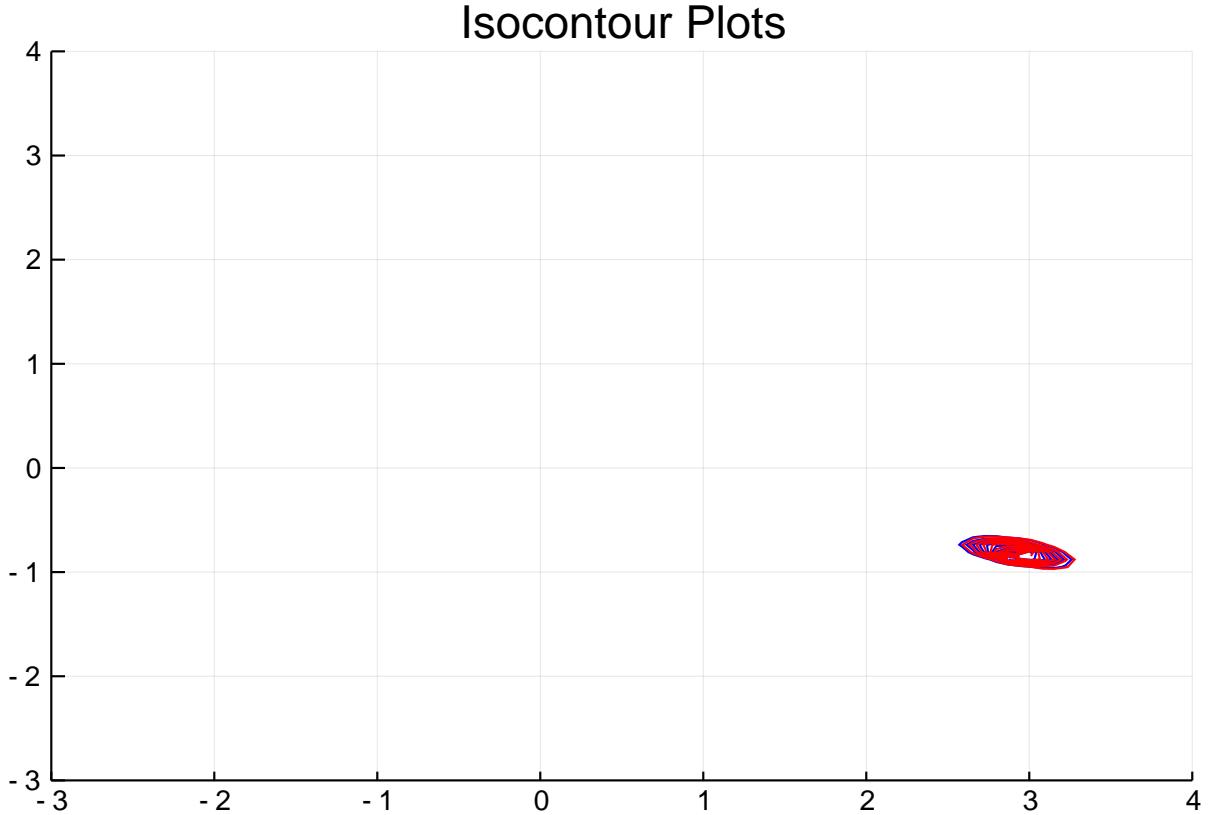
# Plot the contours
params_res = (trained_params[1][1], trained_params[1][2])# Get result parameters from
# training
z = sample_diag_gaussian(params_res[1],params_res[2]) # Sample z by reparameterization
jointq(z)=exp.(log_q(params_res[1],params_res[2],z) .+ log_top(x,z)) # Posterior
q(z/top half of image x)

```

```

jointp(z) = exp.(log_joint_density(x,z)) # Joint p(z, top half of image x),
plot(title="Isocontour Plots")
plotcontour!(jointp ,colour="blue")
display(plotcontour!(jointq, colour="red"))

```



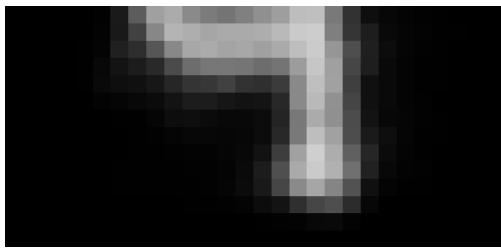
```

# See the image of predicting bottom half
mnist_img_half(x) = ndims(x)==2 ? Gray.(reshape(x,14,28,:)) : Gray.(reshape(x,14,28)) #
for plot half image
s = rand(1:k) # Sample one z from k samples
param = (trained_params[1][1][:,s], trained_params[1][2][:,s]) # Get the sample
parameters in trained parameters
z = sample_diag_gaussian(param[1],param[2]) #sample a z with the parameters
logitz = permutedims(reshape(decoder(z),28,28)) #Use decoder get the logits and
transpose
bottom_half = logitz[15:28,:]
bottom_means=exp.(bottom_half) ./ (1 .+ exp.(bottom_half)) # Transfer from logit
p_sim_bottom=plot(mnist_img_half(vec(bottom_means)),framestyle=:none) #The simulated
bottom half

true_top_means = top_half(train_x[:,2]) # True top half means of first mnist data
p_top1 = plot(mnist_img_half(vec(true_top_means)),framestyle=:none)
p_top2 = plot(mnist_img_half(vec(true_top_means)),framestyle=:none)

true_bottom_means = permutedims(reshape(train_x[:,2],28,28))[15:28,:]
p_bottom = plot(mnist_img_half(vec(true_bottom_means)),framestyle=:none)
#Display concatenated images
display(plot(p_top1, p_top2, p_sim_bottom,p_bottom,layout=(2,2)))

```



(c) True or false.

- (a) Yes
- (b) Yes
- (c) No
- (d) No
- (e) No