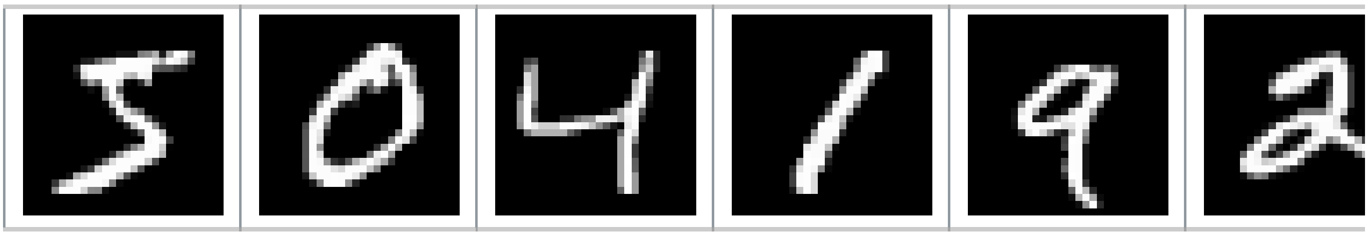
```julia
using Flux
```

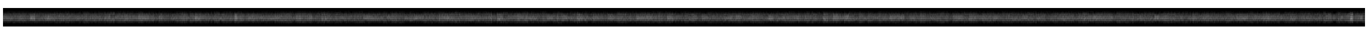
train_digits =



(a vector displayed as a row to save space)

```julia
# load the original greyscale digits
train_digits = Flux.Data.MNIST.images(:train)
```

greyscale_MNIST =



```julia
# convert from tuple of (28,28) digits to vector (784,N)
greyscale_MNIST = hcat(float.(reshape.(train_digits,:))...)
```

binarized_MNIST =

```
784×60000 BitMatrix:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 ⋮              ⋮              ⋮           ⋱           ⋮              ⋮
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
```

```julia
# binarize digits
binarized_MNIST = greyscale_MNIST .> 0.5
```

BS = 200

```julia
# partition the data into batches of size BS
BS = 200
```

batches =

```
DataLoader(784×60000 BitMatrix:
           0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           ⋮              ⋮              ⋮           ⋱           ⋮              ⋮
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
           0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
```

```julia
# batch the data into minibatches of size BS
```

```
batches = Flux.Data.DataLoader(binarized_MNIST, batchsize=BS)
```

```
(784, 200)
```

```
# confirm dimensions are as expected (D,BS)
size(first(batches))
```

Bernoulli Log Density

The Bernoulli distribution $\text{Ber}(x \mid \mu)$ where $\mu \in [0,1]$ is difficult to optimize for a few reasons. One solution is to parameterize the "logit-means": $y = \log(\frac{\mu}{1-\mu})$.

We can exploit further numerical stability, e.g. in computing $\log(1 + exp(x))$, using library provided functions `log1pexp`

```
using StatsFuns: log1pexp #log(1 + exp(x))
```

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

```
# Numerically stable bernoulli density, why do we do this?
function bernoulli_log_density(x, logit_means)
  """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
```

Model Implementation

- `log_prior` that computes the log-density of a latent representation under the prior distribution.
- `decoder` that takes a latent representation z and produces a 784-dimensional vector y. This will be a simple neural network with the following architecture: a fully connected layer with 500 hidden units and `tanh` non-linearity, a fully connected output layer with 784-dimensions. The output will be unconstrained, no activation function.
- `log_likelihood` that given an array binary pixels x and the output from the decoder, y corresponding to "logit-means" of the pixel Bernoullis $y = log(\frac{\mu}{1-\mu})$ compute the **log-likelihood** under our model.
- `joint_log_density` that uses the `log_prior` and `log_likelihood` and gives the log-density of their joint distribution under our model $\log p_\theta(x, z)$.

Note that these functions should accept a batch of digits and representations, an array with elements concatenated along the last dimension.

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

```
function factorized_gaussian_log_density(samples, μ, logσ)
    σ = exp.(logσ)
    return sum(-0.5*((samples.-μ)./σ).^2 .- log.(σ.*sqrt(2π)),dims=1)
end
```

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

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

```
(2, 500, 784)
```

```
Dz, Dh, Ddata = 2, 500, 28^2
```

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

```
decoder = Chain(Dense(Dz, Dh, tanh), Dense(Dh, Ddata)) # You can use Flux's Chain and Dense here
```

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

```
function log_likelihood(x,z)
  """ Compute log likelihood log_p(x|z)"""
    # use numerically stable bernoulli
    return sum(bernoulli_log_density(x, decoder(z)),dims=1)
end
```

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

```
joint_log_density(x,z) = log_prior(z) .+ log_likelihood(x,z)
```

Amortized Approximate Inference with Learned Variational Distribution

Now that we have set up a model, we would like to learn the model parameters θ. Notice that the only indication for *how* our model should represent digits in $z \in \mathbb{R}^2$ is that they should look like our prior $\mathcal{N}(0,1)$.

How should our model learn to represent digits by 2D latent codes? We want to maximize the likelihood of the data under our model $p_\theta(x) = \int p_\theta(x,z)dz = \int p_\theta(x \mid z)p(z)dz$.

We have learned a few techniques to approximate these integrals, such as sampling via MCMC. Also, 2D is a low enough latent dimension, we could numerically integrate, e.g. with a quadrature.

Instead, we will use variational inference and find an approximation $q_\phi(z) \approx p_\theta(z \mid x)$. This approximation will allow us to efficiently estimate our objective, the data likelihood under our model. Further, we will be able to use this estimate to update our model parameters via gradient optimization.

Following the motivating paper, we will define our variational distribution as q_ϕ also using a neural network. The variational parameters, ϕ are the weights and biases of this "encoder" network.

This encoder network q_ϕ will take an element of the data x and give a variational distribution over latent representations. In our case we will assume this output variational distribution is a fully-factorized Gaussian. So our network should output the $(\mu, \log \sigma)$.

To train our model parameters θ we will need also train variational parameters ϕ. We can do both of these optimization tasks at once, propagating gradients of the loss to update both sets of parameters.

The loss, in this case, no longer being the data likelihood, but the Evidence Lower BOund (ELBO).

1. Implement `log_q` that accepts a representation z and parameters $\mu, \log \sigma$ and computes the logdensity under our variational family of fully factorized guassians.
2. Implement `encoder` that accepts input in data domain x and outputs parameters to a fully-factorized guassian $\mu, \log \sigma$. This will be a neural network with fully-connected architecture, a

single hidden layer with 500 units and `tanh` nonlinearity and fully-connected output layer to the parameter space.

3. Implement `elbo` which computes an unbiased estimate of the Evidence Lower BOund (using simple monte carlo and the variational distribution). This function should take the model p_θ, the variational model q_ϕ, and a batch of inputs x and return a single scalar averaging the ELBO estimates over the entire batch.
4. Implement simple loss function `loss` that we can use to optimize the parameters θ and ϕ with `gradient`. We want to maximize the lower bound, with gradient descent. (This is already implemented)

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

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

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

```julia
function unpack_guassian_params(output)
    μ, logσ = output[1:2,:], output[3:4,:]
    return μ, logσ
end
```

```
encoder = Chain(Dense(784, 500, tanh), Dense(500, 4), unpack_guassian_params)
```

```julia
encoder = Chain(Dense(Ddata, Dh, tanh), Dense(Dh, Dz*2), unpack_guassian_params)
```

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

```julia
sample_from_var_dist(μ, logσ) = (randn(size(μ)) .* exp.(logσ) .+ μ)
```

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

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

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

```julia
function loss(x)
  return -elbo(x)
end
```

Optimize the model and amortized variational parameters

If the above are implemented correctly, stable numerically, and differentiable automatically then we can train both the `encoder` and `decoder` networks with graident optimzation.

We can compute `gradient`s of our `loss` with respect to the `encoder` and `decoder` parameters `theta` and `phi`.

We can use a `Flux.Optimise` provided optimizer such as `ADAM` or our own implementation of gradient descent to `update!` the model and variational parameters.

Use the training data to learn the model and variational networks.

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

```
function train!(enc, dec, data; nepochs=100)
    params = Flux.params(enc, dec)
    opt = ADAM()
    @info "Begin training in 2D latent space"
    for epoch in 1:nepochs
        b_loss = 0
        for batch in data
            # compute gradient wrt loss
            grads = Flux.gradient(params) do
                b_loss = loss(batch)
                return b_loss
            end
            # update parameters
            Flux.Optimise.update!(opt, params, grads)
        end
        # Optional: log loss using @info "Epoch $epoch: loss:..."
        @info "Epoch $epoch: loss:$b_loss"
        # Optional: visualize training progress with plot of loss
    end
    @info "Training in 2D is done"
    # return nothing, this mutates the parameters of enc and dec!
end
```

```
train!(encoder, decoder, batches, nepochs=5)
```

```
using BSON: @save
```

```
begin
    @save "encoder.bson" encoder
    @save "decoder.bson" decoder
end
```

```
using BSON
```

```
Dict(:decoder ⇒ Chain(Dense(2, 500, tanh), Dense(500, 784)))
```

```
begin
    BSON.load("encoder.bson", @__MODULE__)
    BSON.load("decoder.bson", @__MODULE__)
end
```

```
(2×200 Matrix{Float32}:                                                              , 2×200 M
  -0.858582  0.00238673   1.66498  -4.57047  …  -1.82859   0.609256  0.0182087    -2.076
   0.608285  1.3631      -1.28454   0.668784     0.646646  0.100764  0.646799     -2.117
```

```
q_μ, q_logσ = encoder(first(batches))
```

Visualizing the Model Learned Representation

We will use the model and variational networks to visualize the latent representations of our data learned by the model.

We will use a variatety of qualitative techniques to get a sense for our model by generating distributions over our data, sampling from them, and interpolating in the latent space.

```
using Plots ##
```

```
using Images
```

calculate_bernoulli_mean (generic function with 1 method)

```
function calculate_bernoulli_mean(logit_means)
    return exp.(logit_means) ./ (1 .+ exp.(logit_means))
end
```

Larger Latent Space

Experimented a 3D latent space and make visualization

create_enc_dec (generic function with 1 method)

```
function create_enc_dec(Dz, unpack_method)
    encoder = Chain(Dense(Ddata, Dh, tanh), Dense(Dh, Dz*2), unpack_method)
    decoder = Chain(Dense(Dz, Dh, tanh), Dense(Dh, Ddata))
    return encoder, decoder
end
```

Dz_3d = 3

```
Dz_3d = 3
```

unpack_guassian_params_3d (generic function with 1 method)

```
function unpack_guassian_params_3d(output)
    μ, logσ = output[1:3,:], output[4:6,:]
    return μ, logσ
end
```

(Chain(Dense(784, 500, tanh), Dense(500, 6), unpack_guassian_params_3d), Chain(Dense(3,

```
encoder_3d, decoder_3d = create_enc_dec(Dz_3d, unpack_guassian_params_3d)
```

log_likelihood_larger (generic function with 1 method)

```
function log_likelihood_larger(x,z)
  """ Compute log likelihood log_p(x|z)"""
    return sum(bernoulli_log_density(x, decoder_3d(z)),dims=1)
end
```

sample_from_var_dist_3d (generic function with 1 method)

```
sample_from_var_dist_3d(μ, logσ) = (randn(size(μ)) .* exp.(logσ) .+ μ)
```

joint_log_density_3d (generic function with 1 method)

```
joint_log_density_3d(x,z) = log_prior(z) .+ log_likelihood_larger(x,z)
```

elbo_3d (generic function with 1 method)

```
function elbo_3d(x)
    q_μ, q_logσ = encoder_3d(x)
    z = sample_from_var_dist_3d(q_μ, q_logσ)
    joint_ll = joint_log_density_3d(x,z)
    log_q_z = log_q(z, q_μ, q_logσ)
    elbo_estimate = sum(joint_ll - log_q_z)/size(x)[2]
    return elbo_estimate, q_logσ
end
```

loss_3d (generic function with 1 method)

```
function loss_3d(x)
  elbo_estimate, logσ = elbo_3d(x)
  return -elbo_estimate, logσ
end
```

```
# logσs = Any[]
```

```
train_3d! (generic function with 1 method)
```

```
function train_3d!(enc, dec, data; nepochs=100)
    params = Flux.params(enc, dec)
    opt = ADAM()
    @info "Begin training in 3D latent space"
    for epoch in 1:nepochs
        b_loss = 0
        logσ = Any[]
        for batch in data
            grads = Flux.gradient(params) do
                b_loss, logσ = loss_3d(batch)
                # push!(logσs, logσ)
                # if epoch == nepochs
                #   var = (exp.(logσ)) .^ 2
                #   vs = size(var)
                # end
                return b_loss
            end

            Flux.Optimise.update!(opt, params, grads)
        end
        var = (exp.(logσ)) .^ 2
        # var_size = size(var)
        # @info "vs: $var_size"
        scatter(var[1,:], var[2,:], var[3,:])
        # Optional: log loss using @info "Epoch $epoch: loss:..."
        @info "Epoch $epoch: loss:$b_loss"
        # Optional: visualize training progress with plot of loss
        # Optional: save trained parameters to avoid retraining later
    end
    @info "Training in 3D is done"
    # return nothing, this mutates the parameters of enc and dec!
    # return logσs
end
```

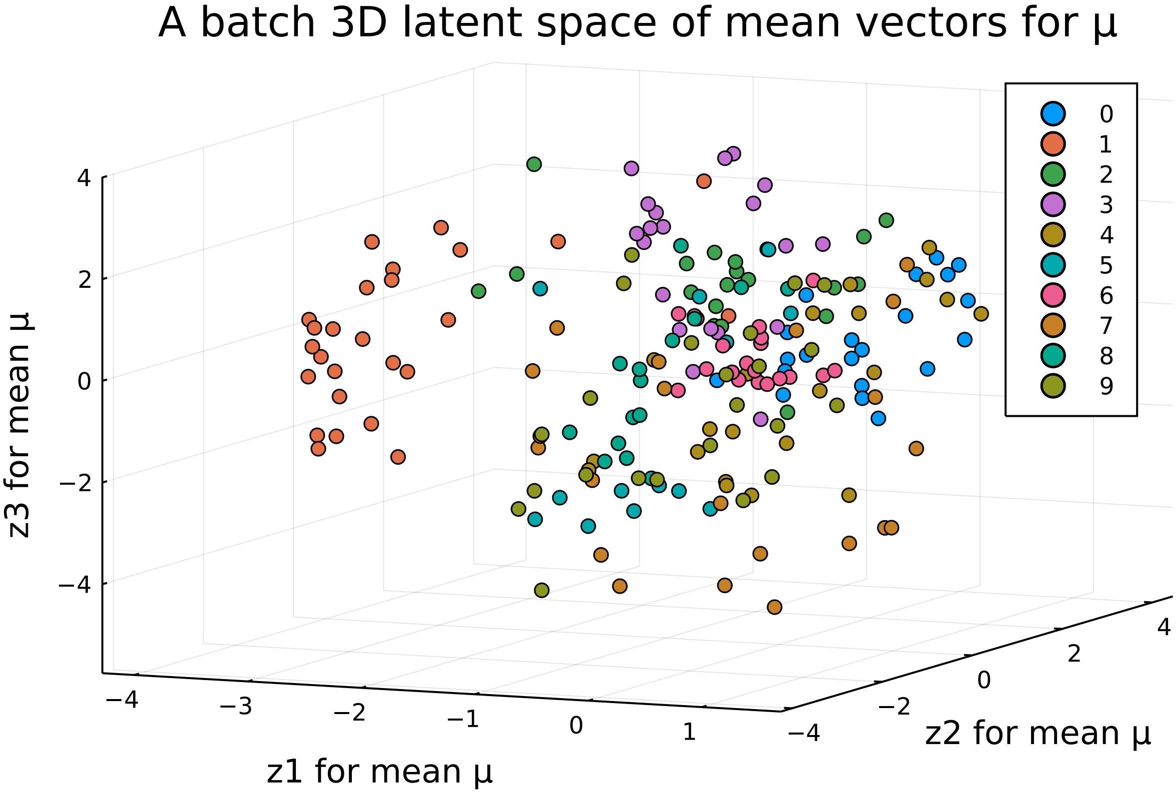
```
train_3d!(encoder_3d, decoder_3d, batches, nepochs=5)
```

```
(3×200 Matrix{Float32}:                                                              , 3×200
  -0.0739408   0.943786  0.187911  -2.48795  …  -0.646299  -1.39841   -0.0249961   -2.2
  -0.990187   -0.55024   3.25409   -3.99575     -1.94905    0.237793  -1.2267      -2.1
   0.442513   -0.398217  0.211053  -0.910293    -1.36731    3.3748     0.824378    -1.8
```

```
q_μ_3d, q_logσ_3d = encoder_3d(first(batches))
```

```
[5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9,    more , 2, 5
```

```
# Training set labels
begin
    train_labels = Flux.Data.MNIST.labels(:train)
    label_batches = Flux.Data.DataLoader(train_labels, batchsize=BS)
    labels = first(label_batches)
end
```




```
scatter(q_μ_3d[1,:], q_μ_3d[2,:], q_μ_3d[3,:], group=labels, title="A batch 3D latent
space of mean vectors for μ", xlabel="z1 for mean μ", ylabel="z2 for mean μ",
zlabel="z3 for mean μ")
```

Comparison with baselines

draw_image (generic function with 1 method)

```
# Helper function for drawing the MNIST digit in 28*28 shape
function draw_image(x)
    dim = ndims(x)
    if dim == 2
        x_2d = reshape(x, 28, 28, :)
        return Gray.(x_2d)
    else
        x_3d = reshape(x, 28, 28)
        return Gray.(x_3d)
    end
end
```

visualize_samples (generic function with 1 method)

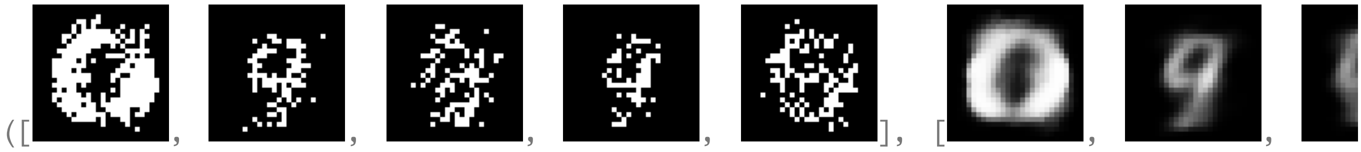
```
function visualize_samples(decoder, dim)
    plots1 = Any[]
    plots = Any[]
    for i in 1:5
        # 1. Sample five 2D/3D zs from the prior p(z)
        z = randn(dim,)
        # 2. decode each z to get logit-means
        logit_means = decoder(z)
        # 3. Transfer logit-means to Bernoulli means μ
        bern_mean = calculate_bernoulli_mean(logit_means)[1:784]
        push!(plots, draw_image(bern_mean))
        # 5. Sample 1 example from Bernoulli
        samples1 = rand(Float64, size(bern_mean)) .< bern_mean
        push!(plots1, draw_image(samples1))
    end
    return plots1, plots
end
```

plot_mnist_image (generic function with 1 method)

```
function plot_mnist_image(plots, plots1)
    # 6. Display all plots in a single 10 x 4 grid
    p = plot(layout = (5,1), size=(500,800))
```

```
    for i in 1:5
        heatmap!(cat(plots[i], plots1[i], dims=2), subplot=i)
    end
    plot(p)
end
```

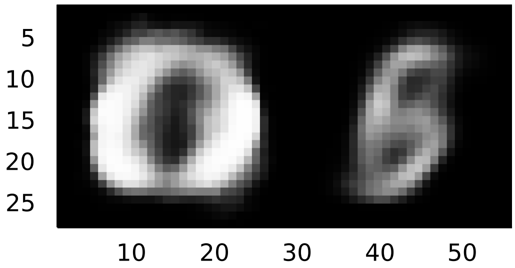
Baseline (2D latent space)

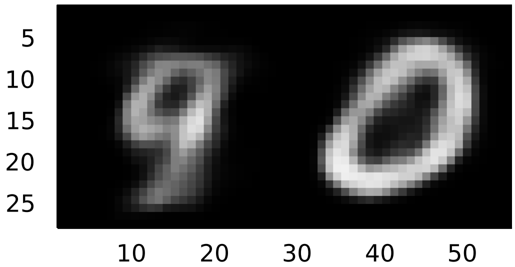


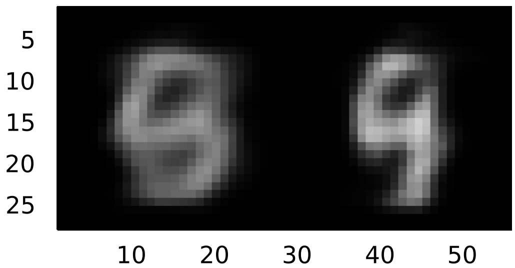
```
plots1_2D, plots_2D = visualize_samples(decoder, 2)
```

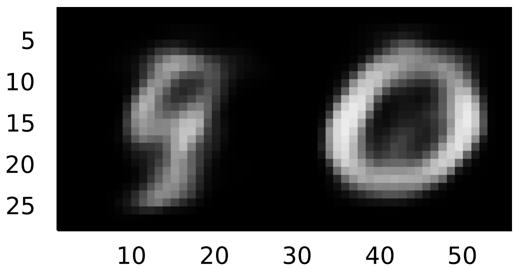
Model with larger dimension (3D latent space)

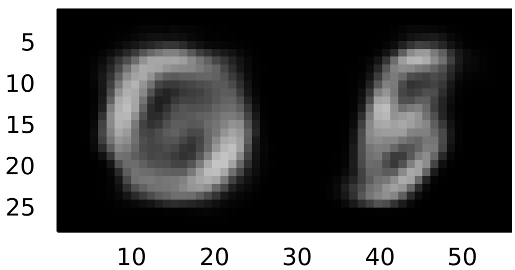
```
plots1_3D, plots_3D = visualize_samples(decoder_3d, 3)
```



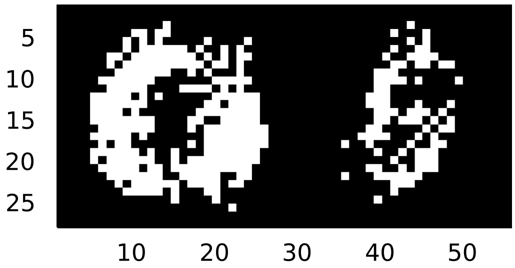


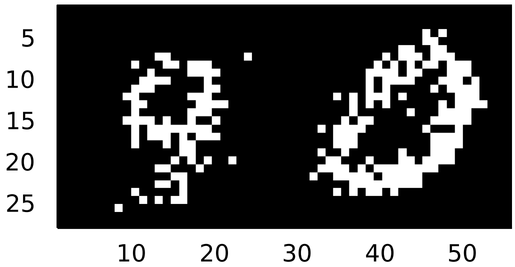


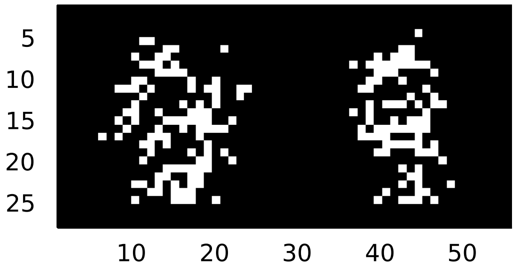


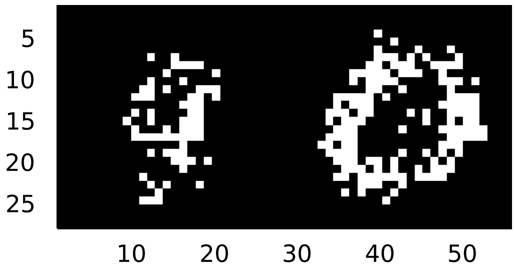


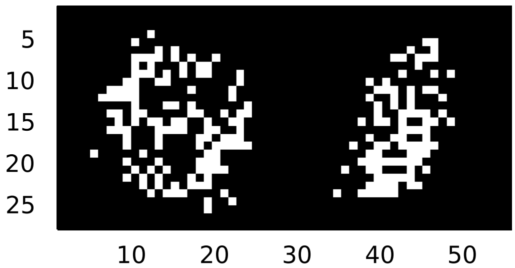

```
plot_mnist_image(plots_2D, plots_3D)
```











```
plot_mnist_image(plots1_2D, plots1_3D)
```

Variance respond as the dimensionality of latent space increases

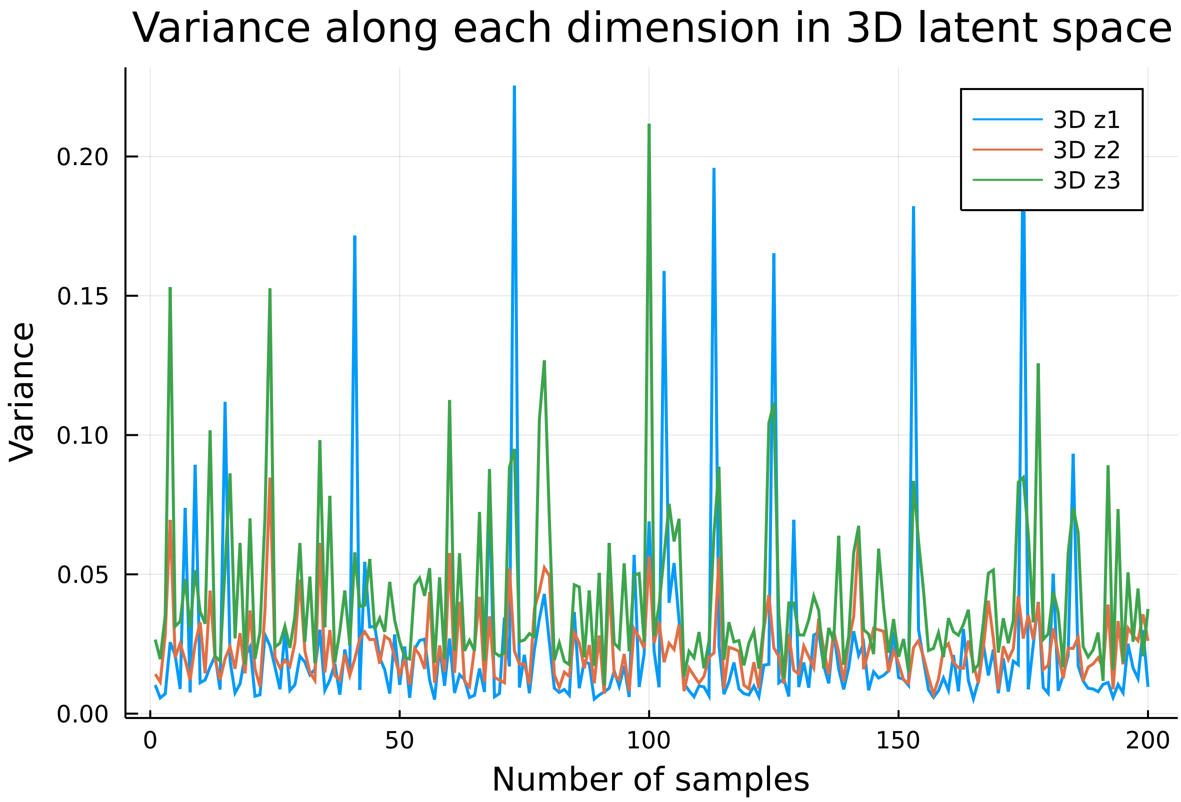
2D latent space v.s. 3D latent space

```
q_var_3d =
3×200 Matrix{Float32}:
 0.0102591  0.00566477  0.00709673  0.0256235  …  0.0125992  0.0299519  0.00961012
 0.0141764  0.0113781   0.0282756   0.0695169     0.0261679  0.0356457  0.0261307
 0.026577   0.0195623   0.0351642   0.153108      0.0449009  0.020621   0.0375741
```

```
q_var_3d = (exp.(q_logσ_3d)).^2
```

```
q_var_2d =
2×200 Matrix{Float32}:
 0.015712   0.00800832  0.00578431  0.119965   …  0.0194409  0.00770532  0.0132927
 0.0144806  0.00882775  0.0235562   0.0331087     0.0193194  0.011105    0.0155383
```

```
q_var_2d = (exp.(q_logσ)).^2
```

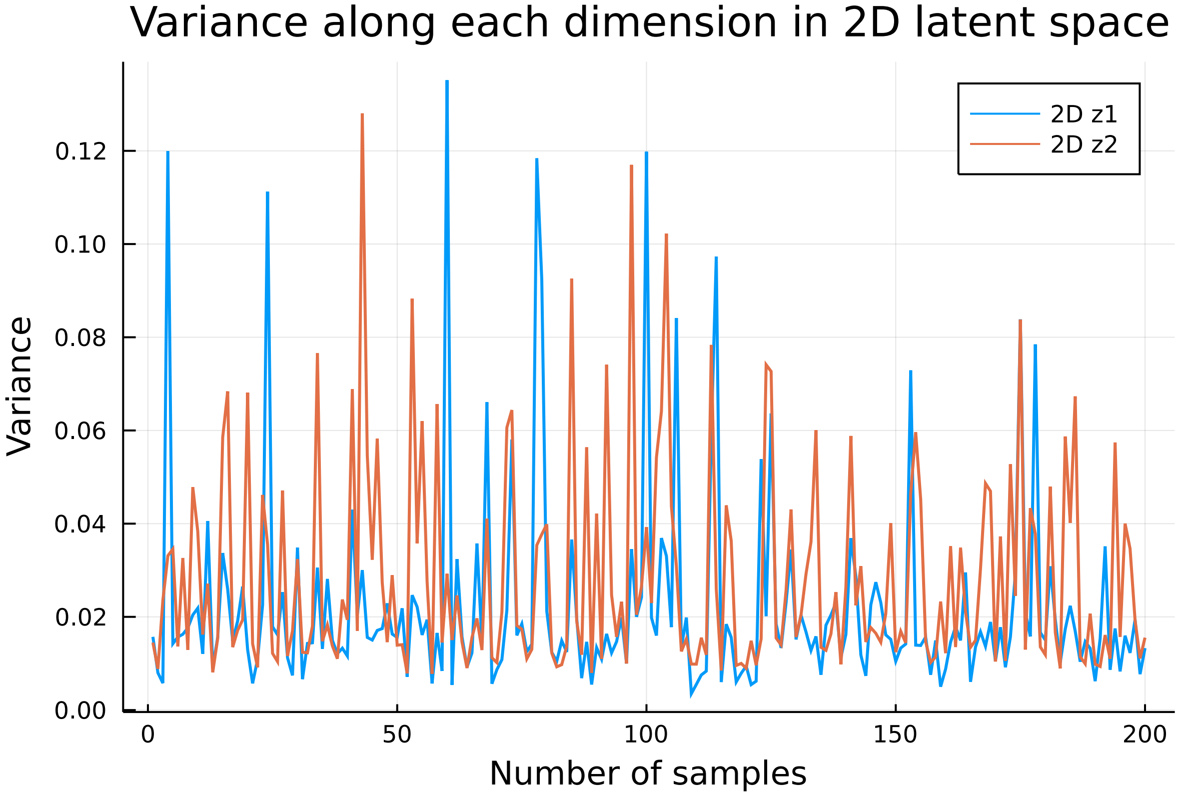



```
begin
    plot(q_var_3d[1,:], label="3D z1", lw = 1.5)
    plot!(q_var_3d[2,:], label="3D z2", lw = 1.5)
    plot!(q_var_3d[3,:], label="3D z3", lw = 1.5)
    xlabel!("Number of samples")
    ylabel!("Variance")
    title!("Variance along each dimension in 3D latent space")
end
```

```
using Statistics
```

```
3×1 Matrix{Float32}:
 0.02445558
 0.023458395
 0.041988656
```

```
mean(q_var_3d, dims=2)
```




```
begin
    plot(q_var_2d[1,:], label="2D z1", lw = 1.5)
    plot!(q_var_2d[2,:], label="2D z2", lw = 1.5)
    xlabel!("Number of samples")
    ylabel!("Variance")
    title!("Variance along each dimension in 2D latent space")
end
```

```
2×1 Matrix{Float32}:
 0.022270106
 0.02870878
```

```
mean(q_var_2d, dims=2)
```

As the dimension of latent space increases, the variance along each dimension tends to increase.

Condition on MNIST Digit Supervision

Horizontally concate labels to data

```
Chain(Dense(13, 500, tanh), Dense(500, 794))
```

```
begin
    encoder_cond = Chain(Dense(Ddata+10, Dh, tanh), Dense(Dh, (Dz_3d+10)*2),
unpack_guassian_params_3d)
    decoder_cond = Chain(Dense(Dz_3d+10, Dh, tanh), Dense(Dh, Ddata+10))
end
```

Change labels to one-hot encoding vectors

```
using Flux: onehotbatch
```

```
onehot_labels =
10×60000 Flux.OneHotArray{10,2,Vector{UInt32}}:
 0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
0  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  (
 0  0  0  1  0  0  1  0  1  0  0  0  0  0  1  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0  1  0  0  0  0  0  (
 0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  1  0  0  1  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  :
 0  0  0  0  0  0  0  1  0  0  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0
```

```
0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0  0  0  1  0  0  1  0  0  0  0  0  0
 0  0  1  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0
1  0  1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  1  0  1  0  0  0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  1
0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  1  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0
 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0  0  0  0  0  0  0
0  1  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0
```

```julia
onehot_labels = onehotbatch(train_labels, [:0, :1, :2, :3, :4, :5, :6, :7, :8, :9])
```

```
batches_cond =
  DataLoader(794×60000 BitMatrix:
             0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             ⋮              ⋮              ⋮              ⋱           ⋮              ⋮
             0  0  1  0  0  0  0  0  0  1  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             1  0  0  0  0  0  0  0  0  0  0  1  0  0     0  0  0  0  0  0  1  0  0  0  1
             0  0  0  0  0  0  0  0  0  0  0  0  0  1  …  0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  1  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  1  0  0  0  0  0  1  0  0
             0  0  0  0  1  0  0  0  0  0  0  0  0  0     0  0  0  1  0  1  0  0  0  0  0
```

```julia
batches_cond = Flux.Data.DataLoader(cat(binarized_MNIST, onehot_labels, dims=1),
batchsize=BS)
```

log_likelihood_cond (generic function with 1 method)

```julia
function log_likelihood_cond(x,z)
  """ Compute log likelihood log_p(x|z,c)"""
    # Let's just label all samples to be digit 0
    digit_0_zeroes = zeros((9,200))
    digit_0_ones = ones((1,200))
    digit_0 = cat(digit_0_ones, digit_0_zeroes, dims=1)
    cond_z = cat(z, digit_0, dims=1)
    return sum(bernoulli_log_density(x, decoder_cond(cond_z)),dims=1)
end
```

joint_log_density_cond (generic function with 1 method)

```julia
joint_log_density_cond(x,z) = log_prior(z) .+ log_likelihood_cond(x,z)
```

elbo_3d_cond (generic function with 1 method)

```julia
function elbo_3d_cond(x)
    q_μ, q_logσ = encoder_cond(x)
    z = sample_from_var_dist_3d(q_μ, q_logσ)
    joint_ll = joint_log_density_cond(x,z)
    log_q_z = log_q(z, q_μ, q_logσ)
    elbo_estimate = sum(joint_ll - log_q_z)/size(x)[2]
    return elbo_estimate
end
```

loss_3d_cond (generic function with 1 method)

```julia
function loss_3d_cond(x)
    return -elbo_3d_cond(x)
end
```

train_cond! (generic function with 1 method)

```julia
function train_cond!(enc, dec, data; nepochs=100)
```

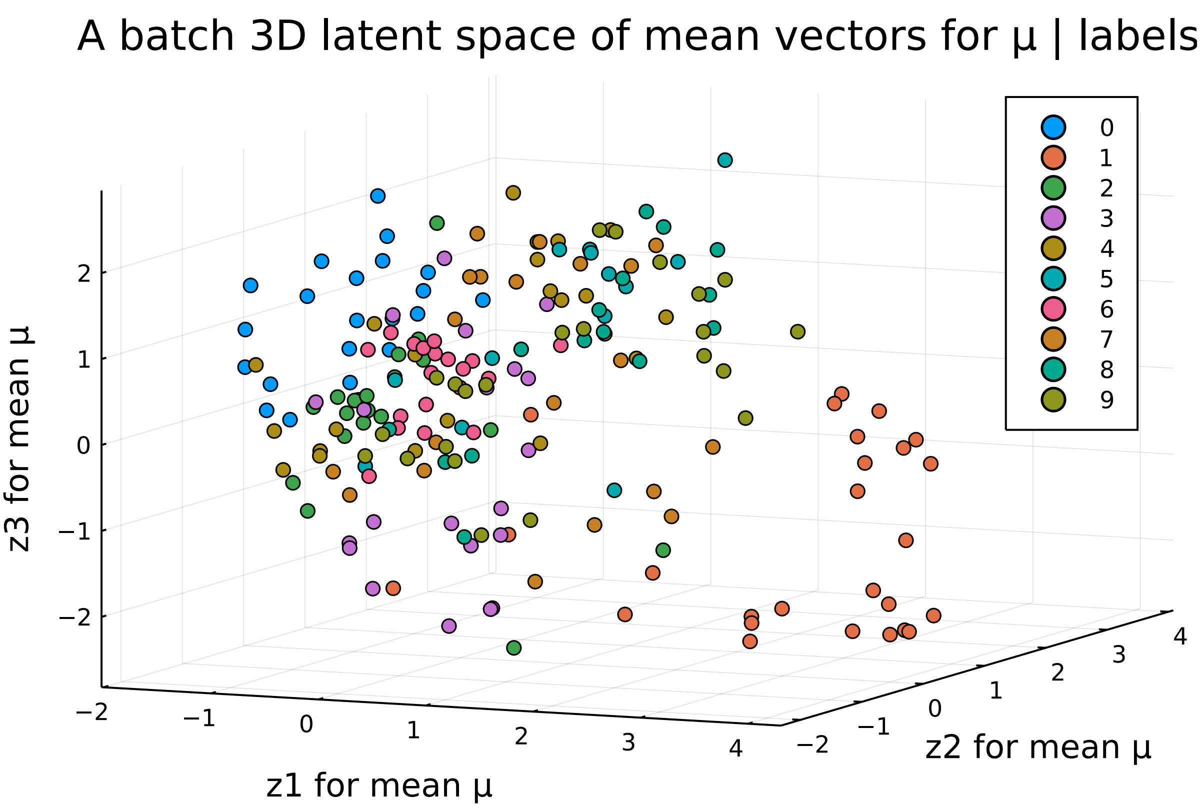
```
    params = Flux.params(enc, dec)
    opt = ADAM()
    @info "Begin training in 3D latent space with given labels"
    for epoch in 1:nepochs
        b_loss = 0
        for batch in data
            grads = Flux.gradient(params) do
                b_loss = loss_3d_cond(batch)
                return b_loss
            end
            Flux.Optimise.update!(opt, params, grads)
        end
        @info "Epoch $epoch: loss:$b_loss"
    end
    @info "Training in 3D latent space(labels) is done"
end
```

```
train_cond!(encoder_cond, decoder_cond, batches_cond, nepochs=3)
```

```
(3×200 Matrix{Float32}:                                                              ,  3×200 Ma
   0.396789  -0.416867  -1.52017   3.08969   …  1.92532    0.421166  -0.323954    -1.7920
  -0.178393  -0.540125   0.37284   0.805868     0.686274  -0.925214  -0.442824    -1.8285
   0.732969   1.88375   -0.598648  0.300632     1.39632   -2.22553    0.781096    -1.4836
```

```
q_μ_cond, q_logσ_cond = encoder_cond(first(batches_cond))
```

Visualize latent representation

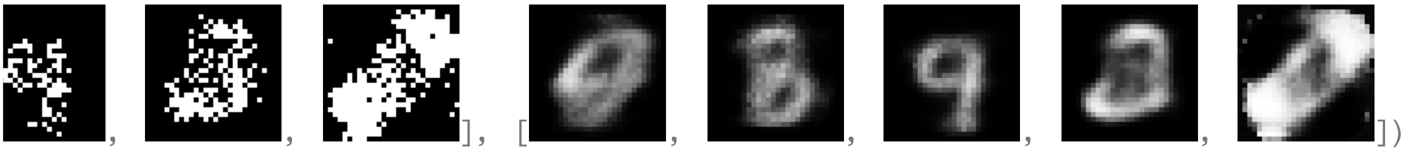



```
scatter(q_μ_cond[1,:], q_μ_cond[2,:], q_μ_cond[3,:], group=labels, title="A batch 3D
latent space of mean vectors for μ | labels", xlabel="z1 for mean μ", ylabel="z2 for
mean μ", zlabel="z3 for mean μ")
```

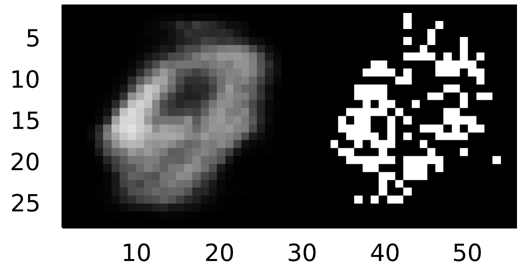
```
draw_image_cond (generic function with 1 method)
```

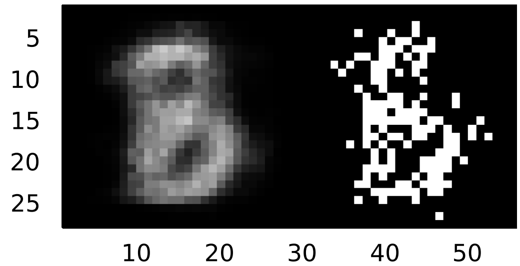
```
# Helper function for drawing the MNIST digit in 28*28 shape
function draw_image_cond(x)
    dim = ndims(x)
    if dim == 2
        x_2d = reshape(x, 28, 28, :)
        return Gray.(x_2d)
    else
        x_3d = reshape(x, 28, 28)
        return Gray.(x_3d)
    end
```

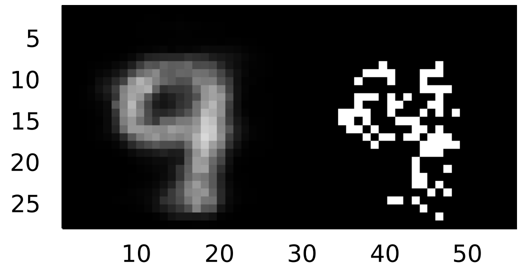
```
end
```

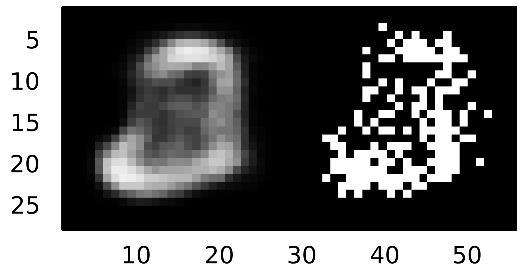


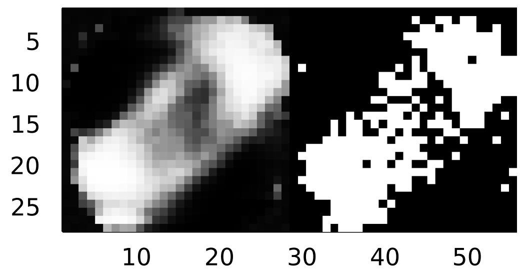
```
plots1_cond, plots_cond = visualize_samples(decoder_cond, 13)
```












```
plot_mnist_image(plots_cond, plots1_cond)
```

Semi-supervised learning

```
10×30000 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
begin
    onehot_semi = Matrix(onehot_labels)
    index = rand(1:60000,30000)
    onehot_semi[:,index] = zeros(10, 30000)
end
```

```
batches_semi =
 DataLoader(794×60000 BitMatrix:
          0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
          ⋮              ⋮              ⋮           ⋱        ⋮              ⋮
          0  0  0  0  0  0  0  0  0  1  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
          1  0  0  0  0  0  0  0  0  0  0  1  0  0     0  0  0  0  0  0  0  0  0  0  1
          0  0  0  0  0  0  0  0  0  0  0  0  0  1  …  0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
          0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  1  0  0  0  0  0  0  0  0
          0  0  0  0  1  0  0  0  0  0  0  0  0  0     0  0  0  0  0  1  0  0  0  0  0
```

```
batches_semi = Flux.Data.DataLoader(cat(binarized_MNIST, onehot_semi, dims=1),
batchsize=BS)
```

```
Chain(Dense(13, 500, tanh), Dense(500, 794))
```

```
begin
    encoder_semi = Chain(Dense(Ddata+10, Dh, tanh), Dense(Dh, (Dz_3d+10)*2),
unpack_guassian_params_3d)
    decoder_semi = Chain(Dense(Dz_3d+10, Dh, tanh), Dense(Dh, Ddata+10))
end
```

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

```
function log_likelihood_semi(x,z)
  """ Compute log likelihood log_p(x|z,c)"""
    # Let's just label all samples to be digit 0
    digit_0_zeroes = zeros((9,200))
    digit_0_ones = ones((1,200))
    digit_0 = cat(digit_0_ones, digit_0_zeroes, dims=1)
    cond_z = cat(z, digit_0, dims=1)
    return sum(bernoulli_log_density(x, decoder_semi(cond_z)),dims=1)
end
```

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

```
joint_log_density_semi(x,z) = log_prior(z) .+ log_likelihood_semi(x,z)
```

```
elbo_3d_semi (generic function with 1 method)
```

```
function elbo_3d_semi(x)
    q_μ, q_logσ = encoder_semi(x)
    z = sample_from_var_dist_3d(q_μ, q_logσ)
    joint_ll = joint_log_density_semi(x,z)
    log_q_z = log_q(z, q_μ, q_logσ)
    elbo_estimate = sum(joint_ll - log_q_z)/size(x)[2]
    return elbo_estimate
end
```

```
loss_3d_semi (generic function with 1 method)
```

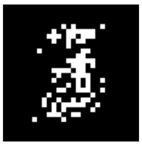
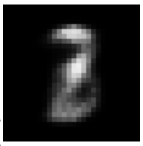
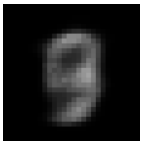
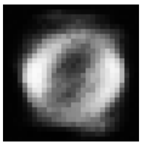
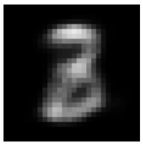
```
function loss_3d_semi(x)
    return -elbo_3d_semi(x)
```

```
end
```

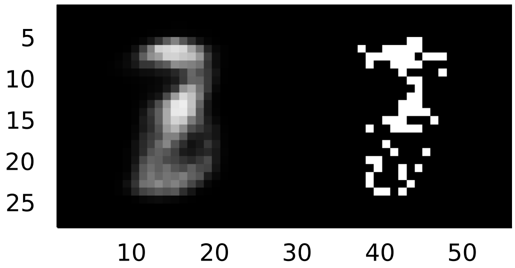
```
train_semi! (generic function with 1 method)
```

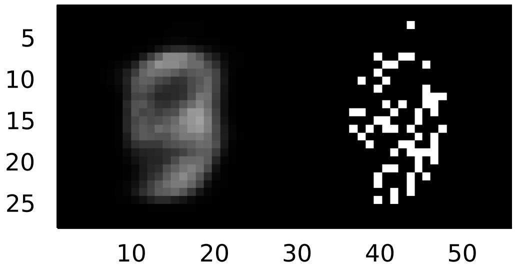
```
function train_semi!(enc, dec, data; nepochs=100)
    params = Flux.params(enc, dec)
    opt = ADAM()
    @info "Begin training in 3D latent space with given semi labels"
    for epoch in 1:nepochs
        b_loss = 0
        for batch in data
            grads = Flux.gradient(params) do
                b_loss = loss_3d_semi(batch)
                return b_loss
            end
            Flux.Optimise.update!(opt, params, grads)
        end
        @info "Epoch $epoch: loss:$b_loss"
    end
    @info "Training in 3D latent space(semi labels) is done"
end
```

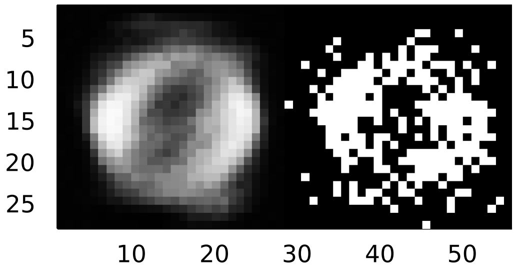
```
train_semi!(encoder_semi, decoder_semi, batches_semi, nepochs=3)
```

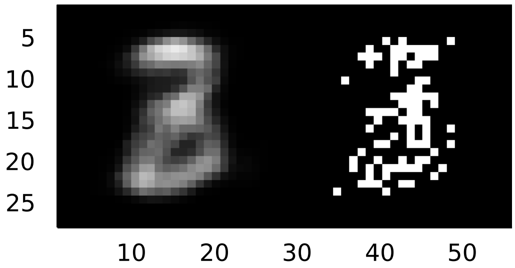
, , ], [, , , , ])

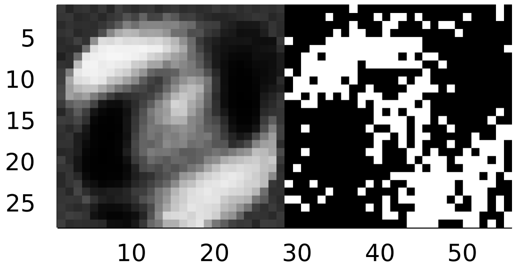
```
plots1_semi, plots_semi = visualize_samples(decoder_semi, 13)
```












```
plot_mnist_image(plots_semi, plots1_semi)
```

Optimizing Different Divergences

```
(Chain(Dense(784, 500, tanh), Dense(500, 4), unpack_guassian_params),  Chain(Dense(2, 500
```

```
encoder_js, decoder_js = create_enc_dec(2, unpack_guassian_params)
```

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

```
function log_likelihood_js(x,z)
  """ Compute log likelihood log_p(x|z)"""
    return sum(bernoulli_log_density(x, decoder_js(z)),dims=1)
end
```

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

```
joint_log_density_js(x,z) = log_prior(z) .+ log_likelihood_js(x,z)
```

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

```julia
function elbo_js(x)
    q_μ, q_logσ = encoder_js(x)
    z = sample_from_var_dist(q_μ, q_logσ)
    joint_ll = joint_log_density_js(x,z)
    log_q_z = log_q(z, q_μ, q_logσ)

    p = exp.(joint_ll)
    q = exp.(log_q_z)
    m = 0.5*log.(p+q)

    pm = sum(joint_ll - m)/size(x)[2]
    qm = sum(log_q_z - m)/size(x)[2]

    elbo_estimate = 0.5*pm + 0.5*qm
    return elbo_estimate
end
```

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

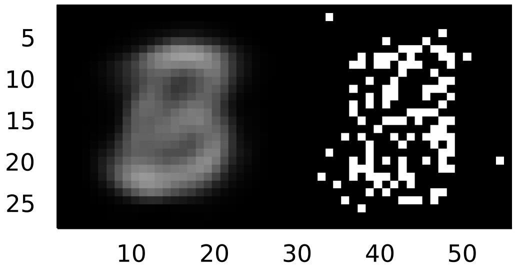
```julia
function loss_js(x)
    return -elbo_js(x)
end
```

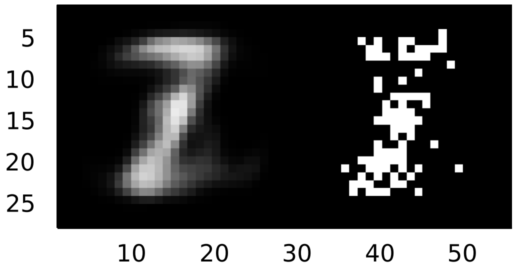
```
train_js! (generic function with 1 method)
```

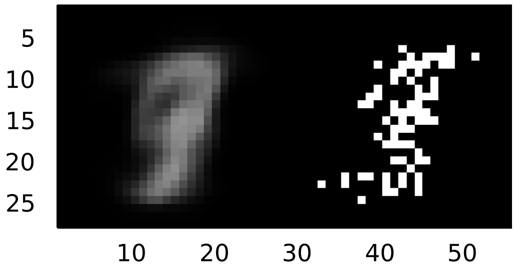
```julia
function train_js!(enc, dec, data; nepochs=100)
    params = Flux.params(enc, dec)
    opt = ADAM()
    @info "Begin training in 2D latent space using JS Divergence"
    for epoch in 1:nepochs
        b_loss = 0
        for batch in data
            # compute gradient wrt loss
            grads = Flux.gradient(params) do
                b_loss = loss_js(batch)
                return b_loss
            end
            # update parameters
            Flux.Optimise.update!(opt, params, grads)
        end
        @info "Epoch $epoch: loss:$b_loss"
    end
    @info "Training in 2D using JS Divergence is done"
end
```

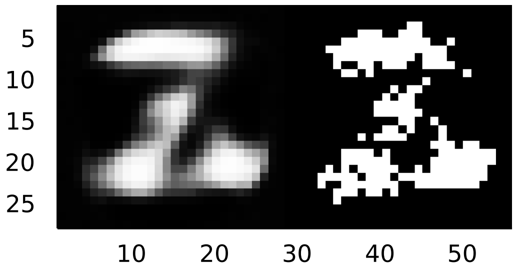
```julia
train_js!(encoder_js, decoder_js, batches, nepochs=3)
```

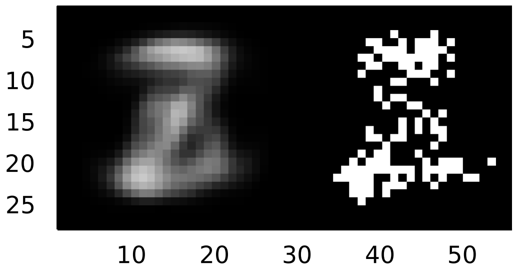
```julia
plots1_js, plots_js = visualize_samples(decoder_js, 2)
```












```julia
plot_mnist_image(plots_js, plots1_js)
```

More Expressive Likelihood Model

Use beta likelihood model with $\alpha = 2, \beta = 2$ on float MNIST

```julia
using Distributions
```

```julia
using SpecialFunctions
```

```
float_MNIST =
784×60000 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
⋮                        ⋮              ⋱                 ⋮
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```
float_MNIST = convert(Array{Float64}, greyscale_MNIST)
```

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

```
function beta_log_density(x, logit_means, α, β)
    b = x .* 2 .- 1
    B = gamma(α)*gamma(β) / gamma(α+β)
    return - log1pexp.(-b .* logit_means / B)
end
```

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

```
function log_likelihood_beta(x, z, α, β)
  """ Compute log likelihood log_p(x|z)"""
    return sum(beta_log_density(x, decoder_beta(z), α, β),dims=1)
end
```

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

```
joint_log_density_beta(x,z, α, β) = log_prior(z) .+ log_likelihood_beta(x,z,α, β)
```

```
(Chain(Dense(784, 500, tanh), Dense(500, 4), unpack_guassian_params),  Chain(Dense(2, 500
```

```
encoder_beta, decoder_beta = create_enc_dec(2, unpack_guassian_params)
```

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

```
function elbo_beta(x)
    q_μ, q_logσ = encoder_beta(x)
    z = sample_from_var_dist(q_μ, q_logσ)
    joint_ll = joint_log_density_beta(x,z, 2, 2)
    log_q_z = log_q(z, q_μ, q_logσ)
    elbo_estimate = mean(joint_ll - log_q_z)
    return elbo_estimate
end
```

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

```
function loss_beta(x)
    return -elbo_beta(x)
end
```

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

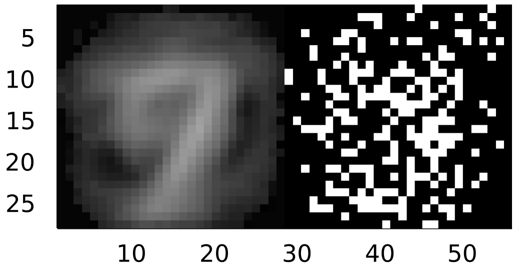
```
function train_beta!(enc, dec, data; nepochs=100)
    params = Flux.params(enc, dec)
    opt = ADAM()
    @info "Begin training in 2D latent space using Beta likelihood on float MNIST"
    for epoch in 1:nepochs
        b_loss = 0
        for batch in data
            # compute gradient wrt loss
            grads = Flux.gradient(params) do
                b_loss = loss_beta(batch)
                return b_loss
            end
            # update parameters
            Flux.Optimise.update!(opt, params, grads)
        end
        @info "Epoch $epoch: loss:$b_loss"
    end
    @info "Training in 2D using Beta likelihood is done"
end
```

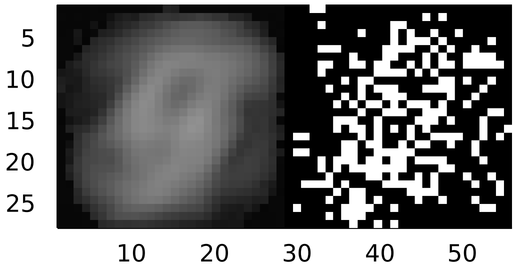
```
float_batches =
  DataLoader(784×60000 Matrix{Float64}:
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 ⋮                        ⋮              ⋱                 ⋮
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

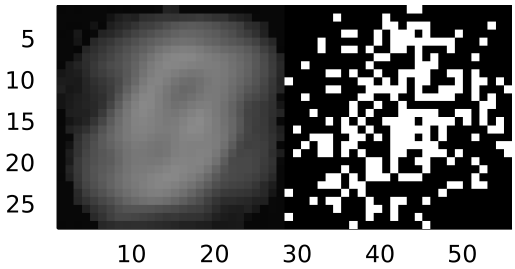
```
float_batches = Flux.Data.DataLoader(float_MNIST, batchsize=BS)
```

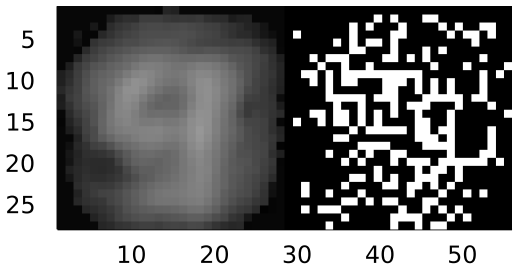
```
train_beta!(encoder_beta, decoder_beta, float_batches, nepochs=5)
```

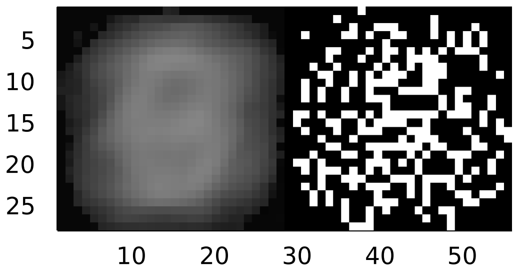
```
plots1_beta, plots_beta = visualize_samples(decoder_beta, 2)
```












```
plot_mnist_image(plots_beta, plots1_beta)
```

Inference

Use the baseline model to infer the bottom of a digit given the top

```
Dhalf = 392
```

```
Dhalf = Int(28*28/2)
```

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

```
# Helper function for drawing only top half of the MNIST digit in 28*28 shape
function draw_top_half_image(x)
    x = reshape(x, 28, 28, :)
    bot_x = x[1:14,:,:]
    return reshape(bot_x, (14,28))
end
```

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

```julia
# Helper function for drawing only top half of the MNIST digit in 28*28 shape
function draw_bot_half_image(x)
    x = reshape(x, 28, 28, :)
    bot_x = x[15:28,:,:]
    return reshape(bot_x, (14,28))
end
```

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

```julia
# Calculate log likelihood log_p(top|z)
function log_p_top_z(top, z)
    x̂_half = decoder(z)[1:Dhalf,:]
    return sum(bernoulli_log_density(top,x̂_half),dims=1)
end
```

Log joint density $p(z, top)$

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

```julia
# Calculate log_p(top, z)
joint_log_density_top(top,z) = log_prior(z) .+ log_p_top_z(top,z)
```

Stochastic variational inference $p(z|top)$

```
q_μ_top = 2×1 Matrix{Float64}:
           0.6707227541019384
          -0.12684732107979857
```

```julia
q_μ_top = randn(Dz,1)
```

```
q_logσ_top = 2×1 Matrix{Float64}:
              -1.6022092621845037
               0.7220838252581145
```

```julia
q_logσ_top = randn(Dz,1)
```

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

```julia
function elbo_top(top, q_μ_top, q_logσ_top)
    z = sample_from_var_dist(q_μ_top, q_logσ_top)
    joint_ll = joint_log_density_top(top,z)
    log_q_z = log_q(z, q_μ_top, q_logσ_top)
    elbo_estimate = mean(joint_ll - log_q_z)
    return -elbo_estimate
end
```

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

```julia
function loss_top(top, q_μ_top, q_logσ_top)
    return -elbo_top(top, q_μ_top, q_logσ_top)
end
```

```
n = 60000
```

```julia
n = size(train_labels)[1]
```

```
indices_0 =
 [2,  22,  35,  38,  52,  57,  64,  69,  70,  76,  82,  89,  96,  109,  115,  119,  120,  122,
```

```julia
# Construct a dataset consists of digit 0
indices_0 = [i for i in 1:n if train_labels[i]==0]
```

```
digit_0 =
784×5923 BitMatrix:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
⋮              ⋮              ⋮              ⋱  ⋮              ⋮              ⋮
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
digit_0 = binarized_MNIST[:,indices_0]
```

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

```
function train_top!(q_μ, q_logσ, data, loss_func; nepochs=100)
    params = Flux.params(q_μ, q_logσ)
    opt = ADAM()
    @info "Begin training to optimize q_μ and q_logσ"
    for epoch in 1:nepochs
        b_loss = 0
        grads = Flux.gradient(params) do
            b_loss = loss_func(data[1:Dhalf])
            return b_loss
        end
        Flux.Optimise.update!(opt, params, grads)
        @info "Epoch $epoch: loss:$b_loss"
    end
    @info "Optimizing q_μ and q_logσ is done"
end
```

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

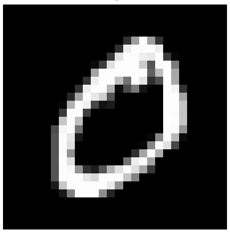
```
loss_tophalf(top) = loss_top(top, q_μ_top, q_logσ_top)
```

```
(784, 5923)
```

```
size(digit_0)
```

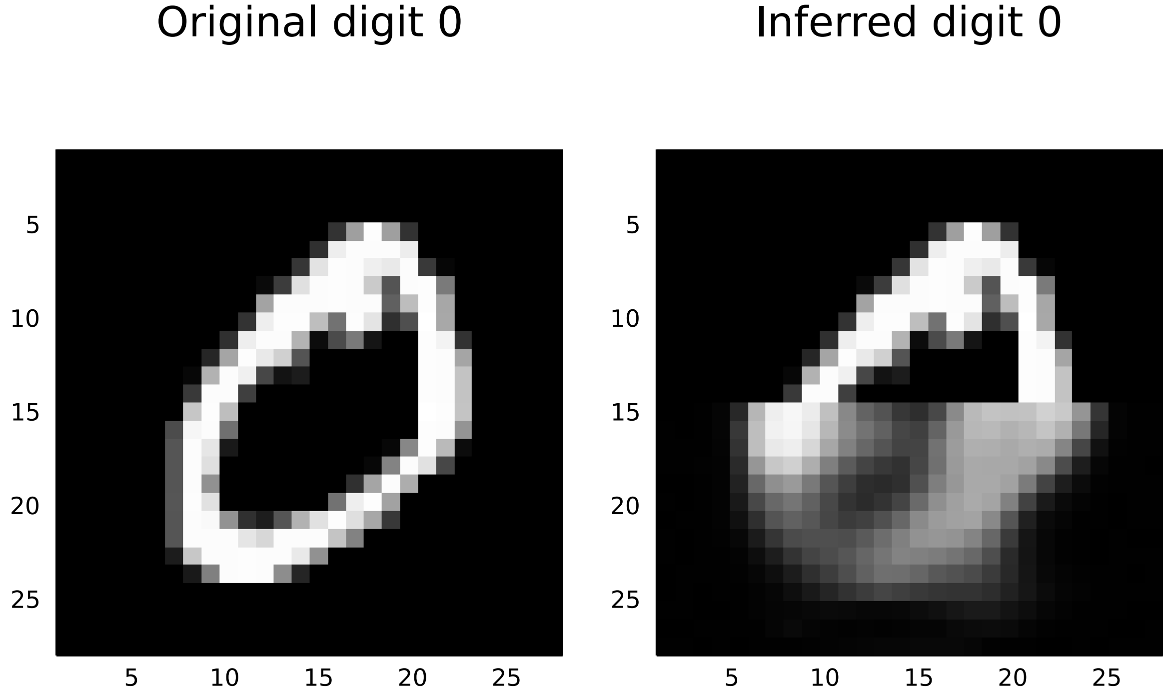
Take digit 0 and infer the bottom part given the top part

test_img =



```
test_img = train_digits[2]
```

```
train_top!(q_μ_top, q_logσ_top, digit_0[:,2], loss_tophalf, nepochs=2)
```

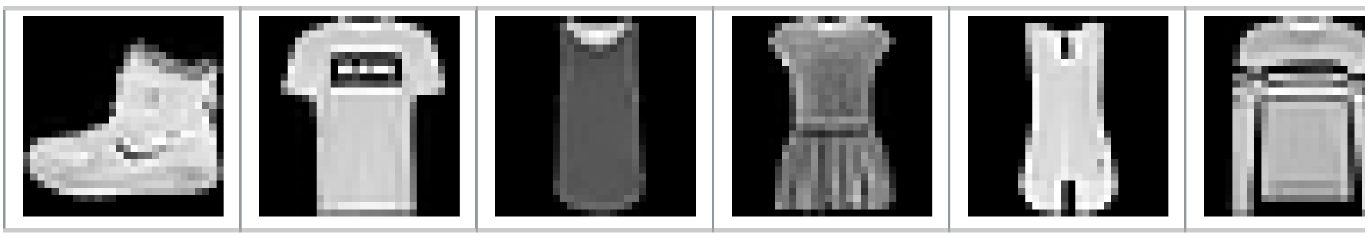



```
begin
    p_top = plot(layout = (1,2))

    # Take a sample z from the approximate posterior
    z_top = sample_from_var_dist(q_μ_top, q_logσ_top)
    # Feed z to decoder
    logits_mean_top = decoder(z_top)
    # Convert to bernoulli mean
    bern_mean_top = calculate_bernoulli_mean(logits_mean_top)
    bot_part = draw_bot_half_image(bern_mean_top)
    top_part = draw_top_half_image(test_img)
    cat_img = cat(top_part, bot_part, dims=1)

    # Plot original and inferred results
    plot!(test_img, title="Original digit 0", subplot=1)
    plot!(draw_image(vec(cat_img)), title= "Inferred digit 0", subplot=2)

    plot(p_top)
end
```

More interesting data

Train the VAE model on Fashion MNIST dataset

`train_fashion =`



(a vector displayed as a row to save space)

```
train_fashion = Flux.Data.FashionMNIST.images(:train)
```

`greyscale_fashion =`

```
greyscale_fashion = hcat(float.(reshape.(train_fashion,:))...)
```

```
binarized_fashion =
784×60000 BitMatrix:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 ⋮              ⋮              ⋮              ⋱           ⋮              ⋮
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
```

```julia
binarized_fashion = greyscale_fashion .> 0.5
```

```
fashion_batches =
 DataLoader(784×60000 BitMatrix:
             0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             ⋮              ⋮              ⋮              ⋱           ⋮              ⋮
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
             0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0
```

```julia
fashion_batches = Flux.Data.DataLoader(binarized_fashion, batchsize=BS)
```

```
 (Chain(Dense(784, 500, tanh), Dense(500, 6), unpack_guassian_params_3d),  Chain(Dense(3,
```

```julia
encoder_fashion, decoder_fashion = create_enc_dec(Dz_3d, unpack_guassian_params_3d)
```

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

```julia
function log_likelihood_fashion(x,z)
    return sum(bernoulli_log_density(x, decoder_fashion(z)),dims=1)
end
```

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

```julia
joint_log_density_fashion(x,z) = log_prior(z) .+ log_likelihood_fashion(x,z)
```

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

```julia
function elbo_fashion(x)
    q_μ, q_logσ = encoder_fashion(x)
    z = sample_from_var_dist_3d(q_μ, q_logσ)
    joint_ll = joint_log_density_fashion(x,z)
    log_q_z = log_q(z, q_μ, q_logσ)
    elbo_estimate = mean(joint_ll - log_q_z)
    return elbo_estimate
end
```

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

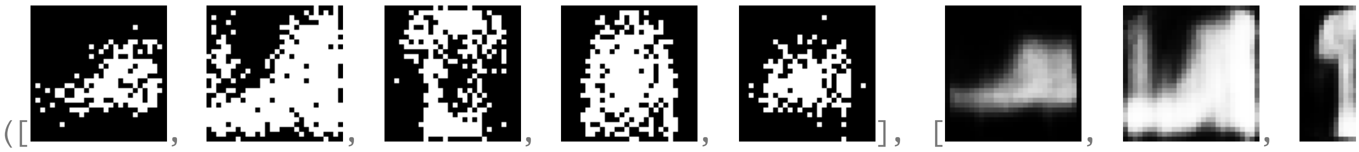
```julia
function loss_fashion(x)
    return -elbo_fashion(x)
end
```

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

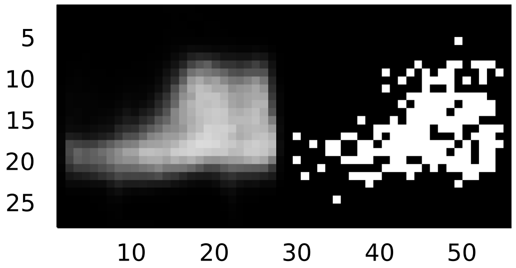
```julia
function train_fashion!(enc, dec, data; nepochs=100)
```

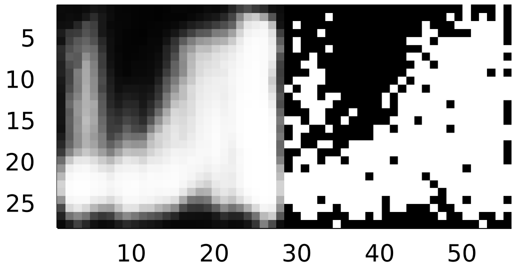
```
    params = Flux.params(enc, dec)
    opt = ADAM()
    @info "Begin training on FashionMNIST in 3D latent space"
    for epoch in 1:nepochs
        b_loss = 0
        for batch in data
            grads = Flux.gradient(params) do
                b_loss = loss_fashion(batch)
                return b_loss
            end
            Flux.Optimise.update!(opt, params, grads)
        end
        @info "Epoch $epoch: loss:$b_loss"
    end
    @info "Training on FashionMNIST in 3D latent space is done"
end
```

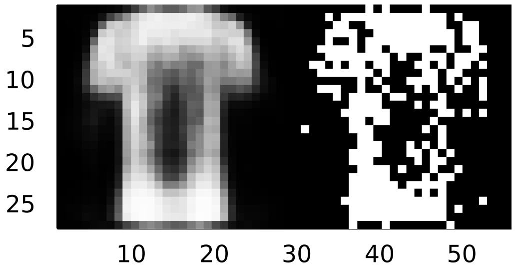
```
train_fashion!(encoder_fashion, decoder_fashion, fashion_batches, nepochs=5)
```

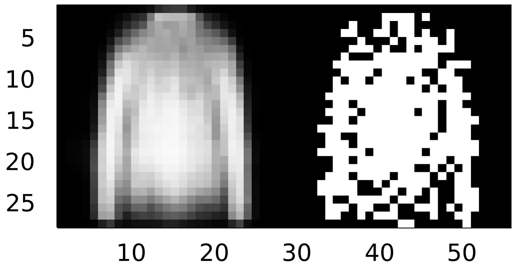


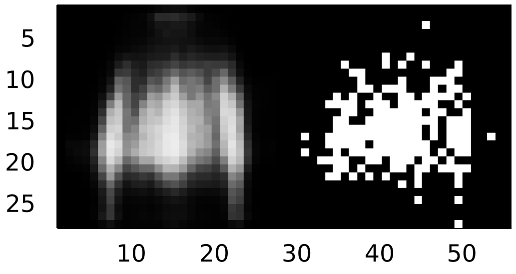
```
plots1_fashion, plots_fashion = visualize_samples(decoder_fashion, 3)
```











```
plot_mnist_image(plots_fashion, plots1_fashion)
```