Assignment 3: Variational Autoencoders

- Student Name: Tianyi Liu
- Student #: 1005820827
- Collaborators: Lazar Atanackovic, Phil Fradkin

Background

In this assignment we will implement and investigate a Variational Autoencoder as introduced by Kingma and Welling in **Auto-Encoding Variational Bayes**.

Data: Binarized MNIST

In this assignment we will consider an MNIST dataset of 28×28 pixel images where each pixel is **either on or off**.

The binary variable $x_i \in \{0, 1\}$ indicates whether the i-th pixel is off or on.

Additionally, we also have a digit label $y \in \{0, \ldots, 9\}$. Note that we will not use these labels for our generative model. We will, however, use them for our investigation to assist with visualization.

Tools

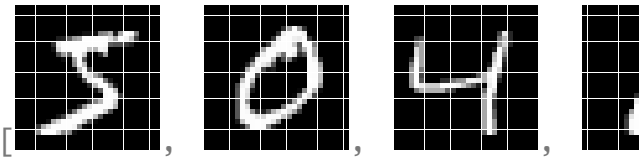
In previous assignments you were required to implement a simple neural network and gradient descent manually. In this assignment you are permitted to use a machine learning library for convenience functions such as optimizers, neural network layers, initialization, dataloaders.

However, you **may not use any probabilistic modelling elements** implemented in these frameworks. You cannot use `Distributions.jl` or any similar software. In particular, sampling from and evaluating probability densities under distributions must be written explicitly by code written by you or provided in starter code.

```
using Flux
```

```
train_digits =
```

```
Matrix{ColorTypes.Gray{FixedPointNumbers.N0f8}}[
```



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

```
greyscale_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
```

```
# 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
```

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

```
BS = 200
```

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

```
batches =
```

```
Flux.Data.DataLoader{BitMatrix, Random._GLOBAL_RNG}(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
```

```
# 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))
```

Model Definition

Each element in the data $x \in D$ is a vector of 784 pixels. Each pixel x_d is either on, $x_d = 1$ or off $x_d = 0$.

Each element corresponds to a handwritten digit $\{0, \ldots, 9\}$. Note that we do not observe these labels, we are *not* training a supervised classifier.

We will introduce a latent variable $z \in \mathbb{R}^2$ to represent the digit. The dimensionality of this latent space is chosen so we can easily visualize the learned features. A larger dimensionality would allow a more powerful model.

- **Prior**: The prior over a digit's latent representation is a multivariate standard normal distribution. $p(z) = \mathcal{N}(z \mid \mathbf{0}, \mathbf{1})$
- **Likelihood**: Given a latent representation z we model the distribution over all 784 pixels as the product of independent Bernoulli distributions parametrized by the output of the "decoder" neural network $f_\theta(z)$.

$$p_\theta(x \mid z) = \prod_{d=1}^{784} \mathrm{Ber}(x_d \mid f_\theta(z)_d)$$

Model Parameters

Learning the model will involve optimizing the parameters θ of the "decoder" neural network, f_θ.

You may also use library provided layers such as `Dense` **as described in the documentation**.

Note that, like many neural network libraries, Flux avoids explicitly providing parameters as function arguments, i.e. `neural_net(z)` instead of `neural_net(z, params)`.

You can access the model parameters `params(neural_net)` for taking gradients `gradient(()->loss(data), params(neural_net))` and updating the parameters with an **Optimiser**.

However, if all this is too fancy feel free to continue using your implementations of simple neural networks and gradient descent from previous assignments.

Numerical Stability

The Bernoulli distribution $\mathrm{Ber}(x \mid \mu)$ where $\mu \in [0, 1]$ is difficult to optimize for a few reasons.

We prefer unconstrained parameters for gradient optimization. This suggests we might want to transform our parameters into an unconstrained domain, e.g. by parameterizing the `log` parameter.

We also should consider the behaviour of the gradients with respect to our parameter, even under the transformation to unconstrained domain. For instance a poor transformation might encourage optimization into regions where gradient magnitude vanishes. This is often called "saturation".

For this reasons we should use a numerically stable transformation of the Bernoulli parameters. 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)"""
    """
        in: x, logit_means: dim × batch_size
        out: dim × batch_size
    """
    # Naive
    # return x .* logit_means - log1pexp.(logit_means)
    # Stable
    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.

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

```
function fully_factorizerd_gaussian(z, μ, logσ)
    σ = exp.(logσ)
    return sum(log.(1 ./ (2 * π * σ .^ 2) .^ 0.5) .+ (-0.5 * ((z .- μ) ./ σ) .^ 2),
dims=1) # 1 × BS
end
```

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

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

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

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

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

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

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

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

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

```
    joint_log_density(x,z) = log_likelihood(x,z) + log_prior(z) # 1 × BS
```

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)
```

```
log_q(z, q_μ, q_logσ) = fully_factorizerd_gaussian(z, q_μ, q_logσ) # 1 × BS
```

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

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

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

```
function wrap_to_gaussian_par(out)
    return out[1:Dz, :], out[Dz+1:2*Dz, :]
end
```

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

```
function elbo(x)
    #TODO variational parameters from data
    q_μ, q_logσ = wrap_to_gaussian_par(encoder(x))
    #TODO: sample from variational distribution
    z = q_μ .+ exp.(q_logσ) .* randn(size(q_μ))
    #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] # scalar
    return elbo_estimate
end
```

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

```
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.

```
using BSON: @save, @load # To save model
```

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

```
function train!(enc, dec, data; nepochs=100, force_retrain=false)
    params = Flux.params(enc, dec)
    opt = ADAM()
    losses_train = []

    if isfile("params.bson") & (~force_retrain)
        @info "Loading pre-trained weights from params.bson\n"
        Core.eval(Main, :(import Zygote))
        @load "params.bson" params
        Flux.loadparams!((encoder, decoder), params)
    else
        @info "Training model from scratch\n"
        steps = length(data)
        tic_base = time()
        for epoch in 1:nepochs
            # CHANGE FOR LOGGING!
            # for batch in data
            for (step, batch) in enumerate(data)
                tic = time()

                loss_train = loss(batch)
                push!(losses_train, loss_train)
                # compute gradient wrt loss
                dparams = Flux.gradient(() -> loss(batch), params)
                # update parameters
                Flux.Optimise.update!(opt, params, dparams)

                toc = time()
                eta = convert(Int, round((toc - tic_base) / ((epoch - 1) * steps +
    step) * ((nepochs - epoch) * steps + steps - step), digits=0))
                @info "iter: $epoch / $nepochs\tstep: $step /
$steps\tloss:$loss_train\ttime: $(round(toc - tic, digits=4)) s ETA: $eta s\n"

            end
            # Optional: log loss using @info "Epoch $epoch: loss:..."
            # Optional: visualize training progress with plot of loss
            # Optional: save trained parameters to avoid retraining later
        end
        toc_base = time()
        @info "Total training time: $(round(toc_base - tic_base, digits=4)) s"
        params = Flux.params(enc, dec)
        @save "params.bson" params
        # return nothing, this mutates the parameters of enc and dec!
    end

end
```

```
train!(encoder, decoder, batches, nepochs=20, force_retrain=false)
```

Visualizing the Model Learned Representation

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

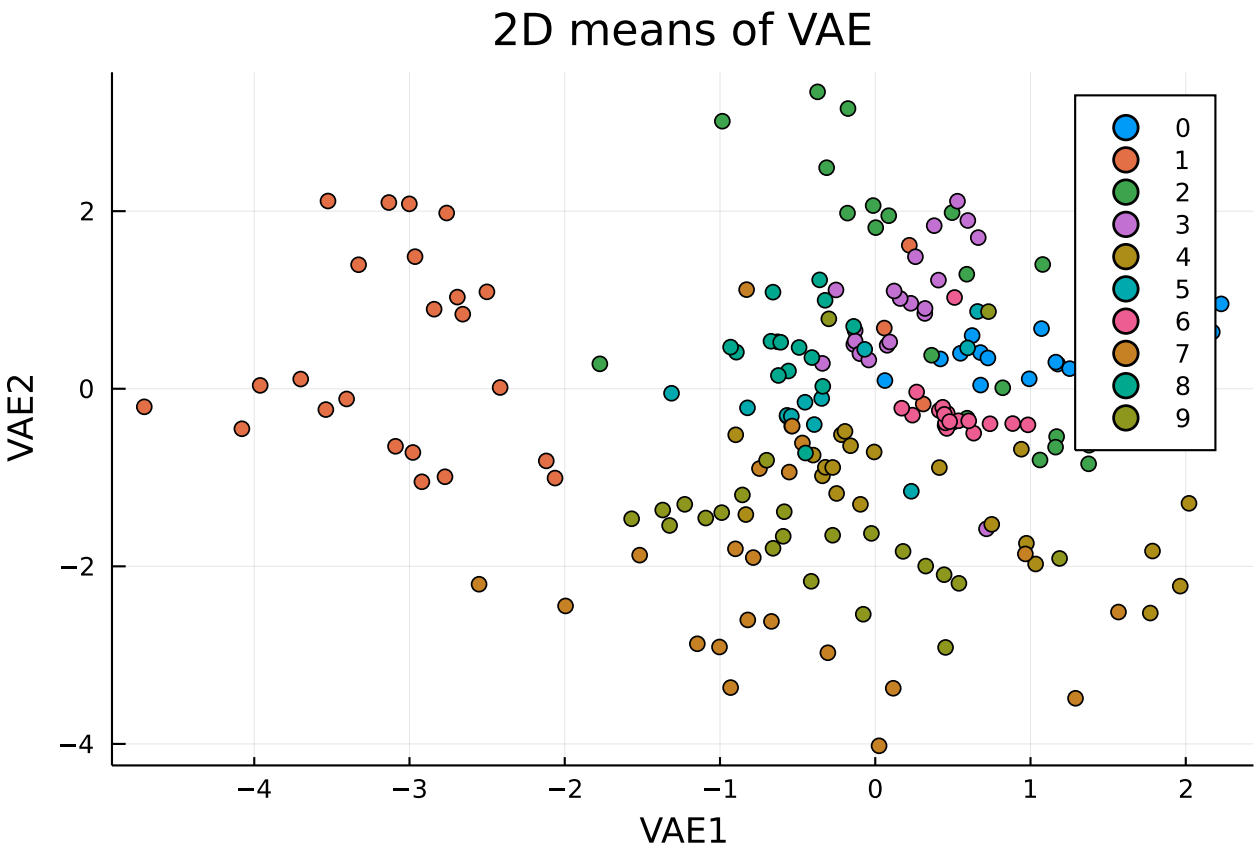
learned by the model.

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 ##
```

1. Latent Distribution of Batch

1. Use `encoder` to produce a batch of latent parameters $\mu, \log \sigma$
2. Take the 2D mean vector μ for each latent parameter in the batch.
3. Plot these mene vectors in the 2D latent space with a scatterplot
4. Colour each point according to its "digit class label" 0 to 9.
5. Display a single colourful scatterplot



```
begin
    μ, logσ = wrap_to_gaussian_par(encoder(first(batches)))
    label = Flux.Data.MNIST.labels(:train)[1:BS]
    plot()
    for i in 0:9
        plot!(μ[1,label.==i],μ[2,label.==i],seriestype = :scatter,label=i)
    end
    title!("2D means of VAE")
    xlabel!("VAE1")
    ylabel!("VAE2")
end
```

2. Visualizing Generative Model of Data

1. Sample 10 z from the prior $p(z)$.
2. Use the model to decode each z to the distribution logit-means over x.
3. Transform the logit-means to the Bernoulli means μ. (this does not need to be efficient)
4. For each z, visualize the μ as a 28×28 greyscale images.
5. For each z, sample 3 examples from the Bernoulli likelihood $x \sim \text{Bern}(x \mid \mu(z))$.
6. Display all plots in a single 10 x 4 grid. Each row corresponding to a sample z. Do not include any axis labels.

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

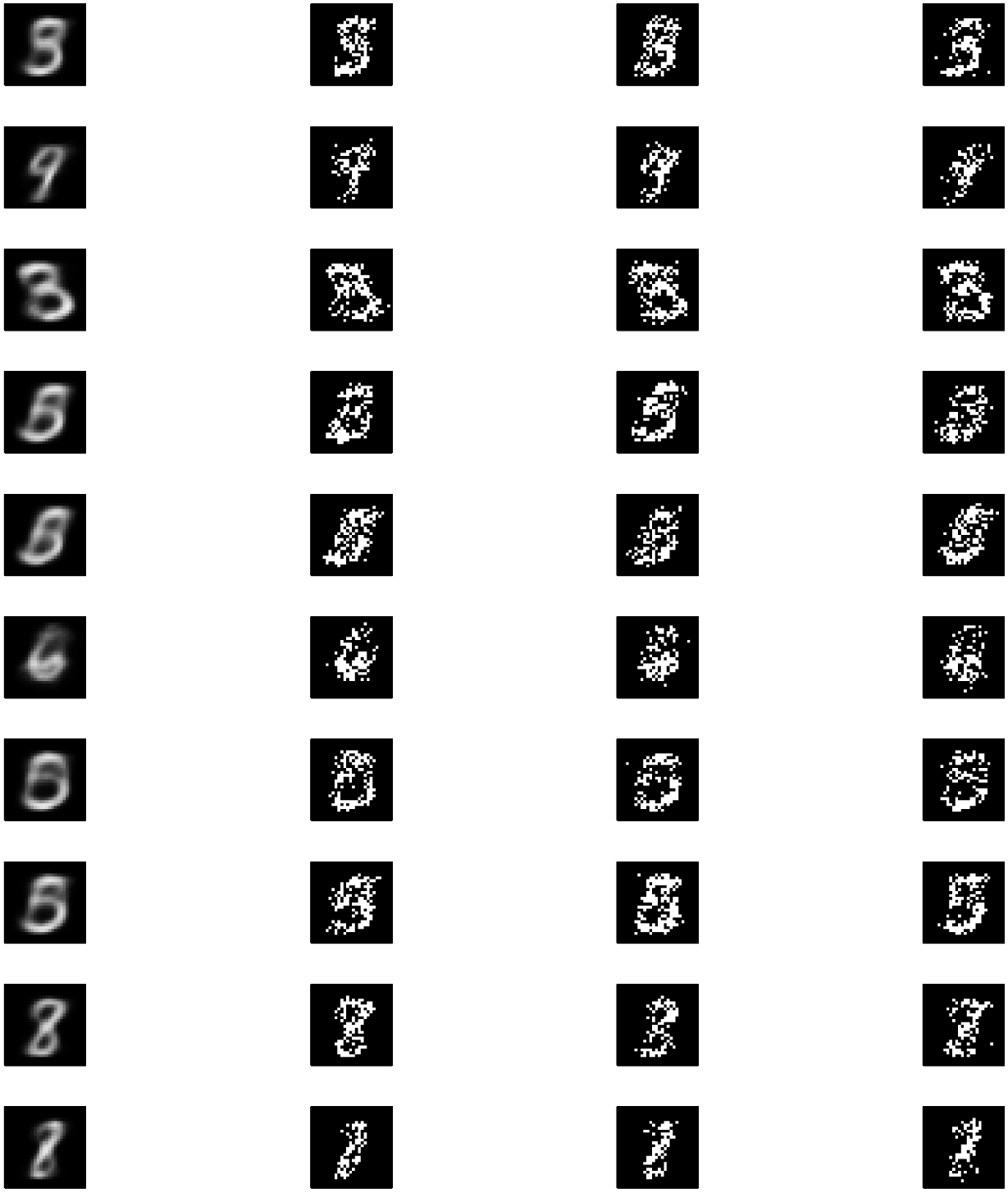
```
manual_bernoulli(μ) = convert(Int64, rand() < μ)
```

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

```
wrap_to_image(array) = Gray.(reshape(array, (28, 28, size(array)[2])))
```

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

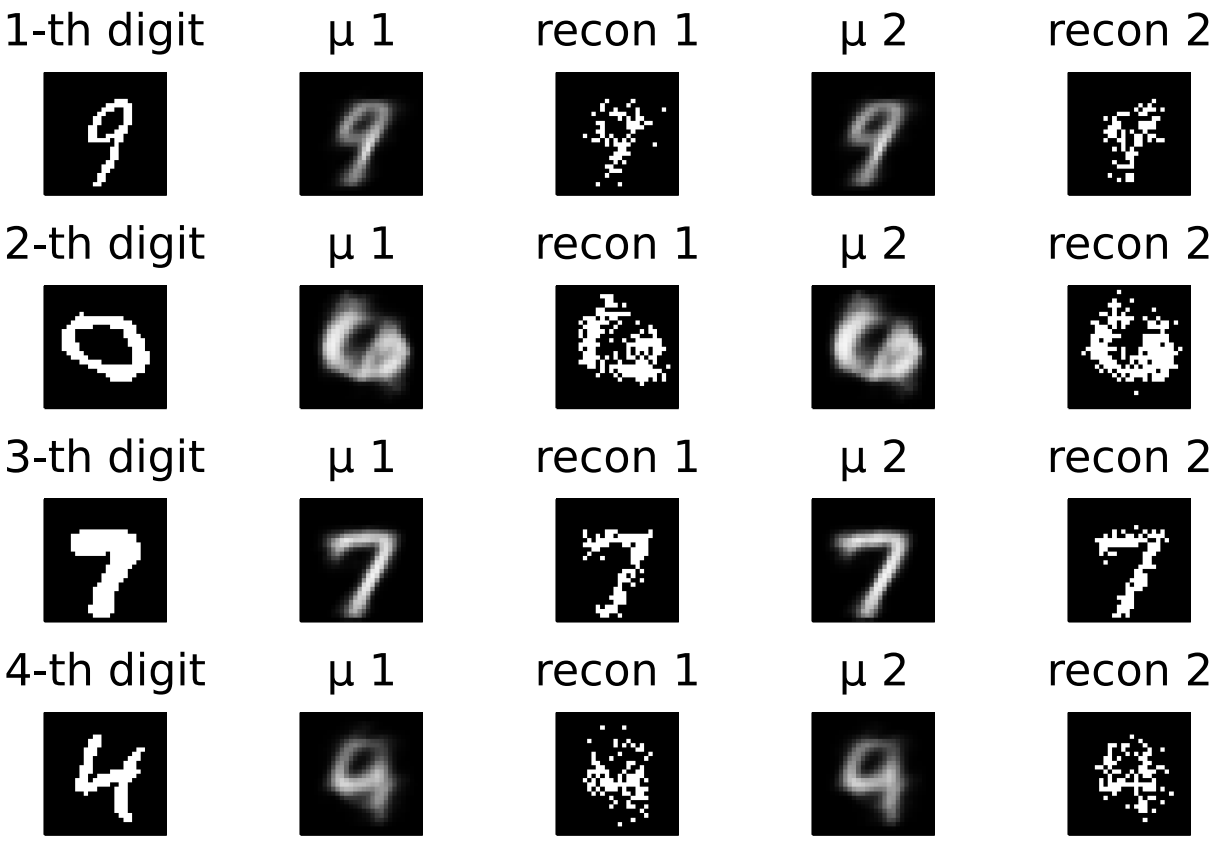
```
wrap_to_μ(logit_means) = 2 .- exp.(log1pexp.(-log1pexp.(logit_means)))
```



```
begin
    z = reshape(randn(20),(2,10))
    log_mean = decoder(z)
    μs = wrap_to_μ(log_mean)
    sample1 = wrap_to_image(manual_bernoulli.(μs))
    sample2 = wrap_to_image(manual_bernoulli.(μs))
    sample3 = wrap_to_image(manual_bernoulli.(μs))
    imgs = wrap_to_image(μs)
    imshow = []
    for i in 1:10
        push!(imshow, plot(imgs[:,:,i], axis=nothing))
        push!(imshow, plot(sample1[:,:,i], axis=nothing))
        push!(imshow, plot(sample2[:,:,i], axis=nothing))
        push!(imshow, plot(sample3[:,:,i], axis=nothing))
    end
    display(plot(imshow..., layout=((10,4)), size=(700,700)))
    current();
end
```

3. Visualizing Regenerative Model and Reconstruction

1. Sample 4 digits from the data $x \sim \mathcal{D}$
2. Encode each digit to a latent distribution $q_\phi(z)$
3. For each latent distribution, sample 2 representations $z \sim q_\phi$
4. Decode each z and transform to the Bernoulli means μ
5. For each μ, sample 1 "reconstruction" $\hat{x} \sim \mathrm{Bern}(x \mid \mu)$
6. For each digit x display (28x28) greyscale images of $x, \mu, \hat{x}$




```
begin
    sample_digits = binarized_MNIST[:,rand(1:size(binarized_MNIST)[2], 4)]
    μ3, logσ3 = wrap_to_gaussian_par(encoder(sample_digits))
    σ3 = exp.(logσ3)
    samples_latent = μ3 .+ σ3 .* randn(2, 4)
    # 1st: (1,5) 2nd: (2,6) 3rd: (3,7) 4th: (4,8)
    samples_latent = hcat(samples_latent, μ3 .+ σ3 .* randn(2, 4))
    log_mean3 = decoder(samples_latent)
    μs3 = wrap_to_μ(log_mean3)
    recon = manual_bernoulli.(μs3)
    img_x = wrap_to_image(sample_digits)
    img_μs3 = wrap_to_image(μs3)
    img_recon = wrap_to_image(recon)
    imshow3 = []
    for i in 1:4
        push!(imshow3, plot(img_x[:,:,i],title="$i-th digit", axis=nothing))
        push!(imshow3, plot(img_μs3[:,:,i], title="μ 1", axis=nothing))
        push!(imshow3, plot(img_recon[:,:,i], title="recon 1", axis=nothing))
        push!(imshow3, plot(img_μs3[:,:,i + 4], title="μ 2", axis=nothing))
        push!(imshow3, plot(img_recon[:,:,i + 4], title="recon 2", axis=nothing))
    end
    display(plot(imshow3..., layout=((4,5))))
    current();
end
```

4. Latent Interpolation Along Lattice

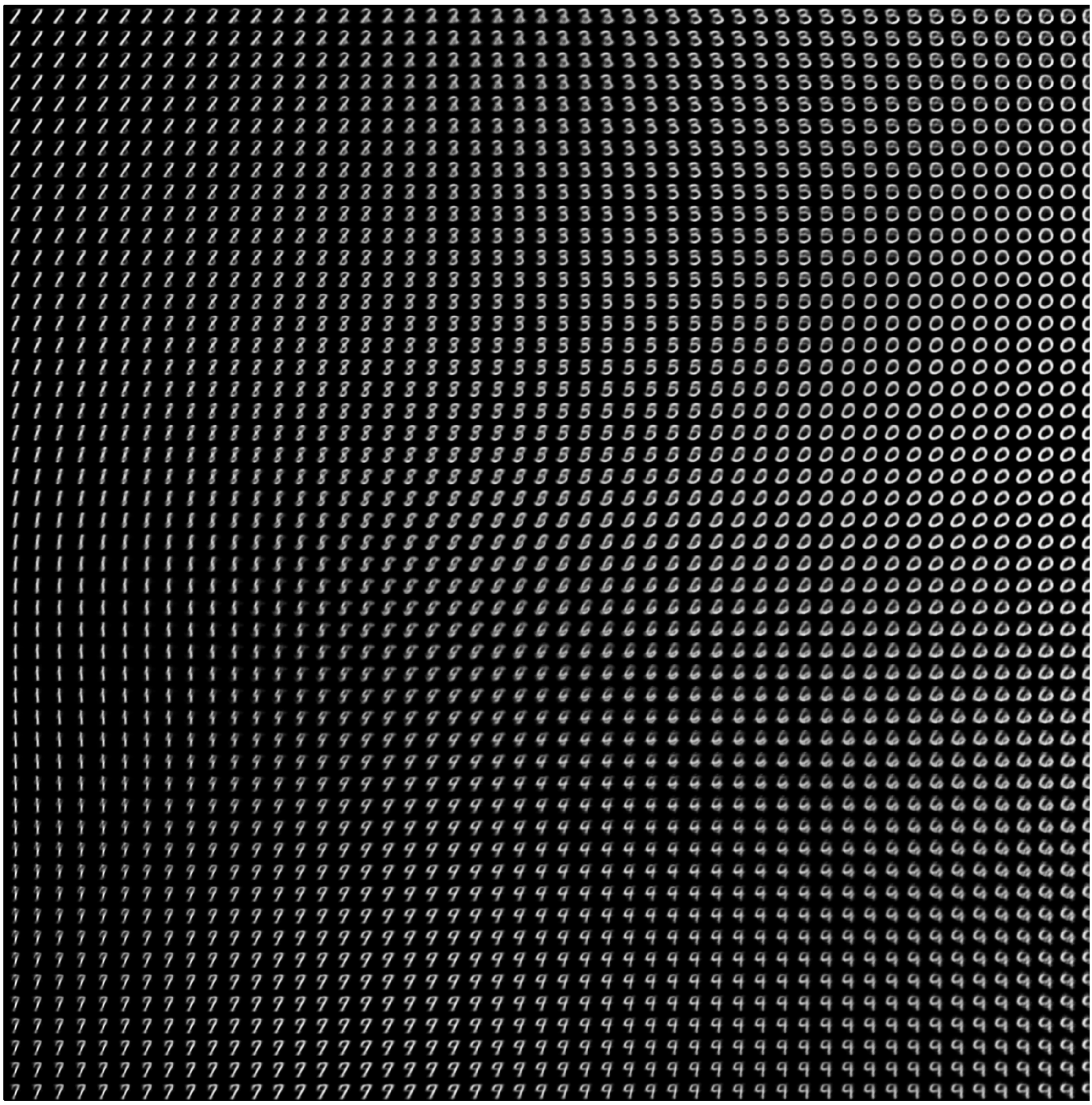
1. Produce a 50×50 "lattice" or collection of cartesian coordinates $z = (z_x, z_y) \in \mathbb{R}^2$.
2. For each z, decode and transform to a 28x28 greyscale image of the Bernoulli means μ

3. Each point in the `50x50` latent lattice corresponds now to a `28x28` greyscale image. Concatenate all these images appropriately.
4. Display a single `1400x1400` pixel greyscale image corresponding to the learned latent space.

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

```
function manifold_cat(μz_img)
    img_whole = nothing
    for row in 1:50
        row_cur = nothing
        for column in 1:50
            img_cur = μz_img[:,:,(row-1) * 50 + column]
            if row_cur == nothing
                row_cur = img_cur
            else
                row_cur = hcat(row_cur, img_cur)
            end
        end
        if img_whole == nothing
            img_whole = row_cur
        else
            img_whole = vcat(img_whole, row_cur)
        end
    end
    return img_whole
end
```

manifold



```
begin
    vmax = 2
    vmin = -vmax
    res = 2 * vmax / (50 - 1)

    z_lattice = reshape([[i; j] for i in vmin:res:vmax, j in vmax:-res:vmin], 2500)
    z_input = reshape([(z_lattice...)...], (2,2500)) # 2 × 2500
    log_μz = decoder(z_input)
    μz = wrap_to_μ(log_μz)
    img_μz = wrap_to_image(μz)
    img_manifold = manifold_cat(img_μz)
    plot(img_manifold, title="manifold", axis=nothing, size=(700,700))
end
```

The manifold is displayed in a standard Cartesian coordinates, i.e., the top-left corner corresponds to (-2, 2), the bottom-right corner corrsponds to (2, -2).