Assignment 3

Arina Sitnikova, 1005863585

2020-04-16

1. Implementing the Model [5 points]

a. **[1 point]** Implement a function log_prior that computes the log of the prior over a digit's representation $logp(z)$.

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

b. **[2 points]** Implement a function decoder that, given a latent representation z and a set of neural network parameters θ (again, implicitly in Flux), produces a 784-dimensional mean vector of a product of Bernoulli distributions, one for each pixel in a 28×28 image. Make the decoder architecture a multi-layer perceptron (i.e. a fully-connected neural network) with a single hidden layer with 500 hidden units, and a tanh nonlinearity.

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

c. **[1 point]** Implement a function log_likelihood that, given a latent representation z and a binarized digit x, computes the log-likelihood $logp(x|z)$.

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

d. **[1 point]** Implement a function joint_log_density which combines the log-prior and log-likelihood of the observations to give $logp(z, x)$ for a single image.

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

2. Amortized Approximate Inference and training [13 points]

a. **[2 points]** Write a function encoder that, given an image x (or batch of images) and recognition parameters ϕ, evaluates an MLP to outputs the mean and log-standard deviation of a factorized Gaussian of dimension $D_z = 2$. Make the encoder architecture a multi-layer perceptron (i.e. a fully-connected neural network) with a single hidden layer with 500 hidden units, and a tanh nonlinearity.

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

b. **[1 point]** Write a function log_q that given the parameters of the variational distribution, evaluates the likelihood of z.

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

c. **[5 points]** Implement a function elbo which computes an unbiased estimate of the mean variational evidence lower bound on a batch of images. Use the output of encoder to give the parameters for $q_\phi(z|data)$.

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

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

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

e. **[3 points]** Write a function that initializes and optimizes the encoder and decoder parameters jointly on the training set. Note that this function should optimize with gradients on the elbo estimate over batches of data, not the entire dataset. Train the data for 100 epochs (each epoch involves a loop over every batch). Report the final ELBO on the test set.

```julia
function train_model_params!(loss, encoder, decoder, train_x, test_x; nepochs = 10)
  # model params
  ps = Flux.params(encoder, decoder)
  # 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
      end
      Flux.Optimise.update!(opt,ps,gs)
    end
    if i%nepochs == 0 # prints the final ELBO
      @info "Final ELBO, epoch $i: $(loss(batch_x(test_x)[1]))"
    end
  end
  @info "Parameters of encoder and decoder trained!"
end

train_model_params!(loss,encoder,decoder,train_x,test_x, nepochs = 100)
```

When train_size = 1000:

```
## Final ELBO, epoch 100: 181.08747254857292
```

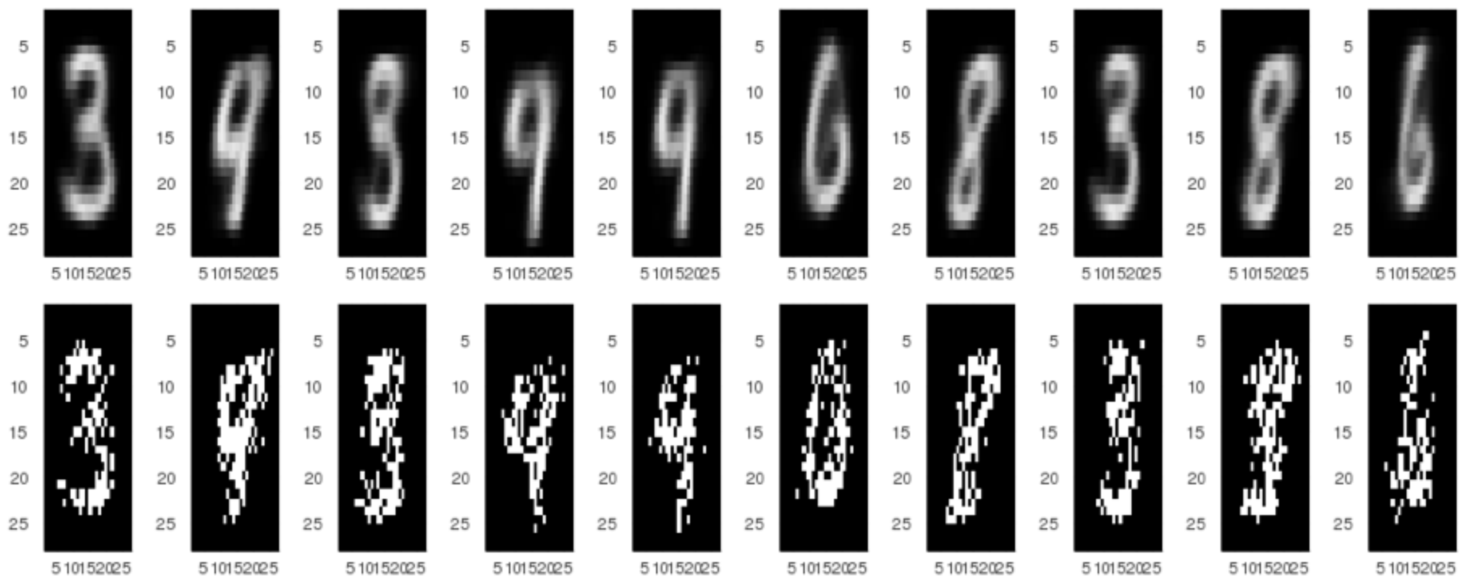
When train_size = 10,000:

```
## Final ELBO, epoch 100: 152.19436985004887
```

3. Visualizing Posteriors and Exploring the Model [15 points]

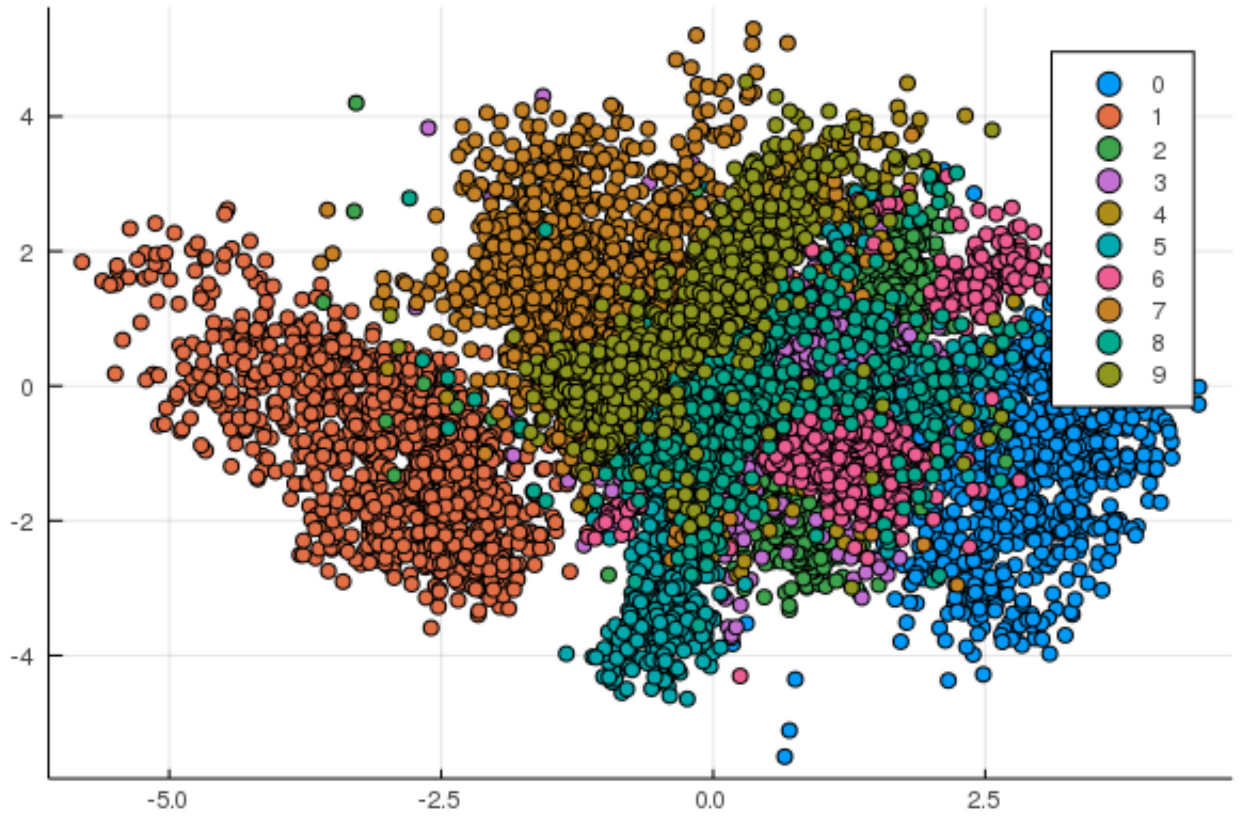
a. **[5 points]** Plot samples from the trained generative model using ancestral sampling. Do this for 10 samples z from the prior.

```
z = randn(2,10)
decode = decoder(z)
ber_mean = exp.(decode) ./ (1 .+ exp.(decode))
sample_ber = sample_bernoulli(ber_mean)

plot_list = Any[]
for i = 1:10
  plt = plot(mnist_img(ber_mean[:,i]))
  push!(plot_list, plt)
end
for i = 1:10
  plt2 = plot(mnist_img(sample_ber[:,i]))
  push!(plot_list, plt2)
end

plot(plot_list..., layout = grid(2, 10), size = (1000, 400))
#savefig("3a.png")
```



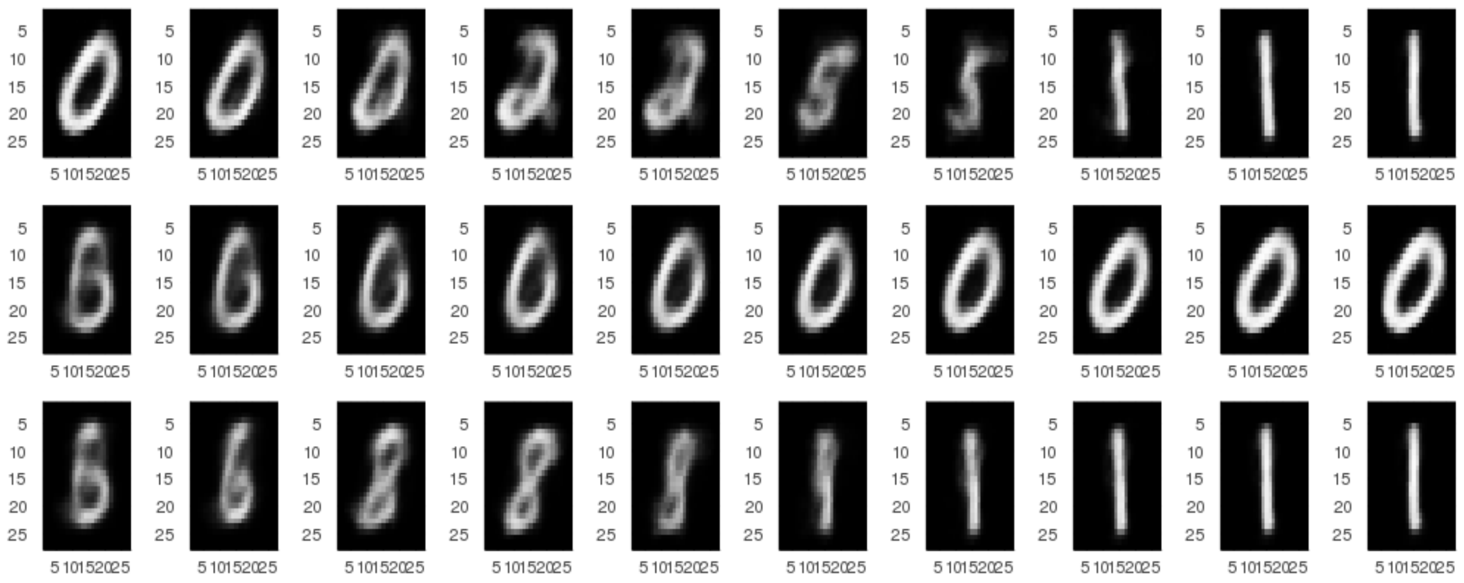
b. **[5 points]** One way to understand the meaning of latent representations is to see which parts of the latent space correspond to which kinds of data. Here we'll produce a scatter plot in the latent space, where each point in the plot represents a different image in the training set.

```
means = encoder(train_x)[1]
scatter(means[1,:], means[2,:], group = train_label)
#savefig("3b.png")
```



c. **[5 points]** Another way to examine a latent variable model with continuous latent variables is to interpolate between the latent representations of two points. Here we will encode 3 pairs of data points with different classes. Then we will linearly interpolate between the mean vectors of their encodings. We will plot the generative distributions along the linear interpolation.

```julia
function interpolation(za, zb, alpha)
  zalpha = alpha*za + (1 - alpha)*zb
  return zalpha
end

z1 = encoder(train_x[:,15])[1] #1
z2 = encoder(train_x[:,120])[1] #0
z3 = encoder(train_x[:,999])[1] #3

plot_list = Any[]
for i = 0.1:0.1:1
  ex = interpolation(z1, z2, i)
  decode = decoder(ex)
  ber_mean = exp.(decode) ./ (1 .+ exp.(decode))
  plt = plot(mnist_img(ber_mean[:,1]))
  push!(plot_list, plt)
end
for i = 0.1:0.1:1
  ex = interpolation(z2, z3, i)
  decode = decoder(ex)
  ber_mean = exp.(decode) ./ (1 .+ exp.(decode))
  plt = plot(mnist_img(ber_mean[:,1]))
  push!(plot_list, plt)
end
for i = 0.1:0.1:1
  ex = interpolation(z1, z3, i)
  decode = decoder(ex)
  ber_mean = exp.(decode) ./ (1 .+ exp.(decode))
  plt = plot(mnist_img(ber_mean[:,1]))
  push!(plot_list, plt)
end

plot(plot_list..., layout = grid(3, 10), size = (1000, 400))
#savefig("3c.png")
```



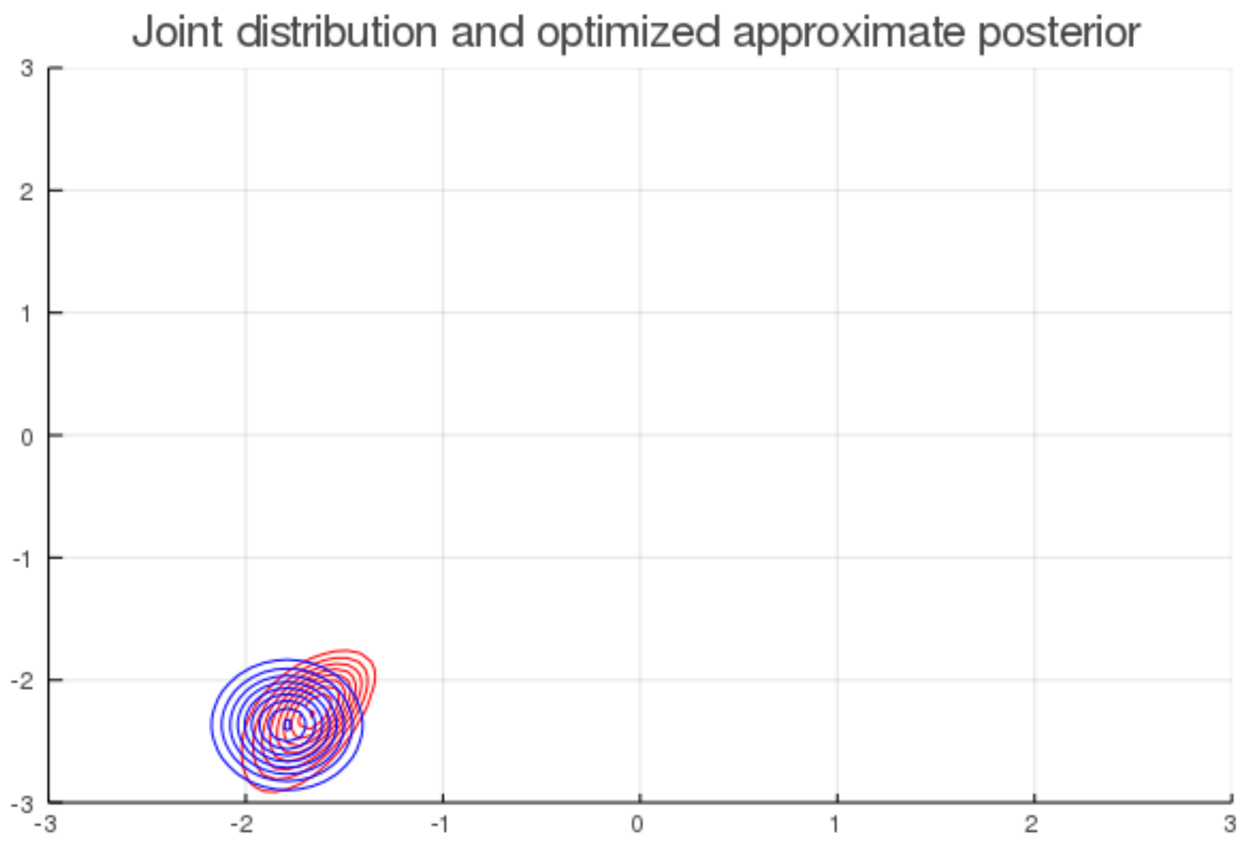
4. Predicting the Bottom of Images given the Top [15 points]

a. **[5 points]** Write a function that computes p(z, top half of image x).

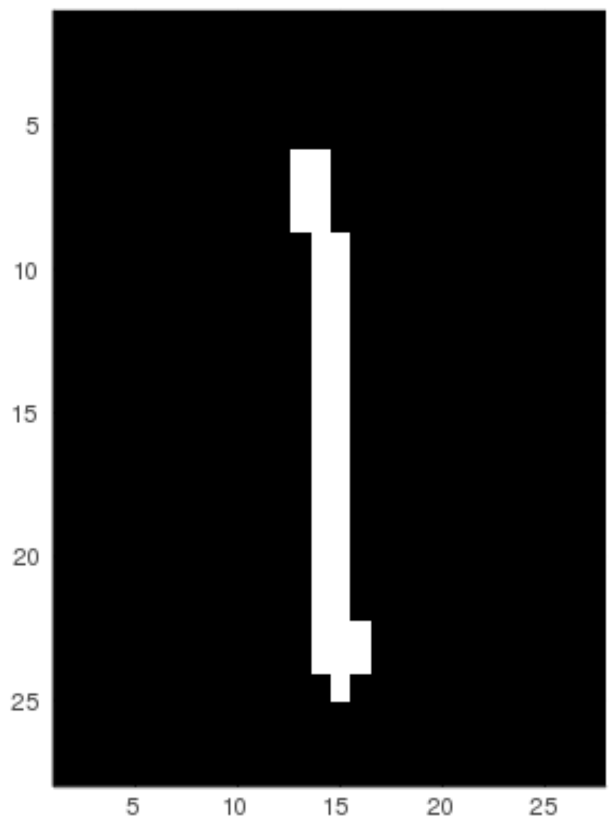
```
function top_half(x)
  #x_reshaped = reshape(x, 28, 28, 1)
  #top = x_reshaped[1:14,:, :]
  #return reshape(top, 392, 1)
  return x[1:392,:]
end

function top_half_ll(x, z)
  decoded = decoder(z)[1:392,:]
  return sum(bernoulli_log_density(decoded, top_half(x)), dims = 1)
end

function top_half_joint(x, zs)
  return log_prior(zs) + top_half_ll(x, zs)
end
```

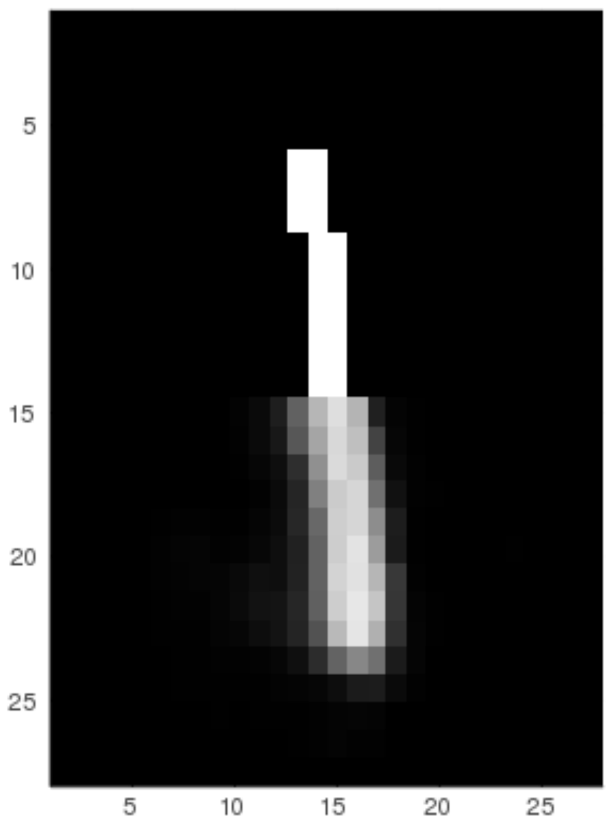
b. **[5 points]** Now, to approximate p(z| top half of image x) in a scalable way, we'll use stochastic variational inference.

```
train_img = train_x[:,15] #easily visible "1"
μ_init, logσ_init = randn(2, 1), randn(2, 1)
params_init = (μ_init, logσ_init)

function top_half_elbo(params, x, num_samples) #just like elbo from 2c
  f_μ, f_logσ = params
  z = sample_diag_gaussian(f_μ, f_logσ)
  joint_ll = top_half_joint(x, z)
  log_q_z = log_q(f_μ, f_logσ, z)
  elbo_estimate = mean(joint_ll - log_q_z)
  return elbo_estimate
end

function top_half_loss(params, x, num_samples)
  return -top_half_elbo(params, x, num_samples)
end

function top_half_optimize(init_params, x;  num_itrs = 100, lr = 1e-2, num_q_samples
= 50) #function from A2
  params_cur = init_params
  for i in 1:num_itrs
    grad_params = gradient(params ->  top_half_loss(params, x, num_q_samples), params
_cur)[1]
    params_cur =  params_cur .- lr .* grad_params
    @info "Current loss $i: $(top_half_loss(params, x, num_q_samples))"
  end
  return params_cur
end

trained = top_half_optimize(params_init, train_img)

using Revise #skillcontour from A2
includet("A2_src.jl")
using .A2funcs: skillcontour!

joint(zs) = exp.(top_half_joint(train_img, zs))
posterior(zs) = exp.(factorized_gaussian_log_density(trained[1], trained[2], zs))
plot()
skillcontour!(joint, colour=:red)
display(skillcontour!(posterior, colour=:blue))
title!("Joint distribution and optimized approximate posterior")
#savefig("4bd")
```



```
z = sample_diag_gaussian(trained[1], trained[2])
decode = decoder(z)
ber_mean = exp.(decode) ./ (1 .+ exp.(decode))
top = top_half(train_img)
bottom = ber_mean[393:end,:]
concat = vcat(top, bottom)
real = plot(mnist_img(train_img))
approx = plot(mnist_img(vec(concat)))
#savefig("4be")
plot(real, approx, layout = (1, 2))
#savefig("4be_twoimg")
```





c. **[5 points]** True or false: Questions about the model and variational inference.

1. Does the distribution over p(bottom half of image $x|z$) factorize over the pixels of the bottom half of image x?

Yes, it does.

2. Does the distribution over p(bottom half of image $x|$top half of image x) factorize over the pixels of the bottom half of image x?

No, it does not.

3. When jointly optimizing the model parameters θ and variational parameters ϕ, if the ELBO increases, has the KL divergence between the approximate posterior $q_\phi(z|x)$ and the true posterior $p_\theta(z|x)$ necessarily gotten smaller?

Yes; maximizing ELBO means minimizing KL divergence.

4. If $p(x) = N(x|\mu, \sigma^2)$, for some $x \in R$, $\mu \in R$, $\sigma \in R^+$, can $p(x) < 0$?

No; a probability density function cannot be negative.

5. If $p(x) = N(x|\mu, \sigma^2)$, for some $x \in R$, $\mu \in R$, $\sigma \in R^+$, can $p(x) > 0$?

Yes; a probability density function can be larger than 1.