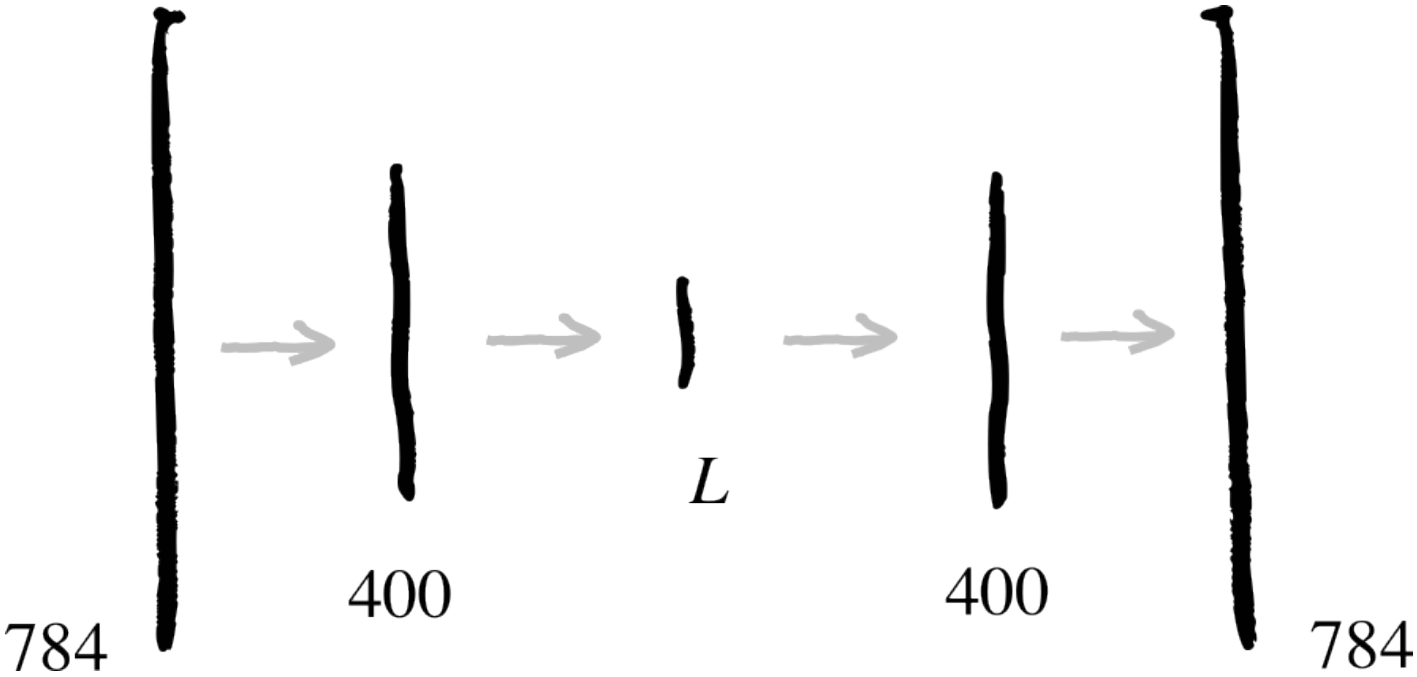
Если вы видите что-то необычное, просто сообщите мне.

Day 9: Roaming The Latent Space

The Secret Space

> “ latent - present and capable of emerging or developing but not now visible, obvious, active, or symptomatic —Webster's Dictionary

Autoencoders



A simple autoencoder with latent dimension L. A simple autoencoder with latent dimension L.

How do we implement an autoencoder? Let us take a multilayer network from the image above: 784→400→L→400→784. Where the latent space dimension L is some smaller number such as 20 or maybe even 2. The L-sized latent vector z is often called a bottleneck. The left part from the bottleck is called an encoder qϕ and the part on the right, a decoder pθ. Where ϕ and θ are trainable parameters of encoder and decoder, respectively.

The encoder takes an input (like an image) and generates a compact representation, typically a vector. It is also not a mistake to call it a compressed representation. The decoder takes this compact representation and creates an output as close as possible to the original input. Hence the name, autoencoder. Of course, some information is lost due to the dimensionality reduction. Therefore, the goal of a autoencoder is to find the most relevant features to preserve as much information about the input object as possible.

```
data AE = AE
  { -- Encoder parameters
    l1 :: Linear,
    l2 :: Linear,
    -- Decoder parameters
    l3 :: Linear,
    l4 :: Linear
  }
  deriving (Generic, Show, Parameterized)
```

Then, the whole autoencoder network is

```
ae :: AE -> Tensor -> (Tensor, Tensor)
ae AE {..} =
  linear l1
  -- Encoder
  ~> relu
  ~> linear l2
  ~> relu
  -- Decoder
  ~> linear l3
  ~> relu
  ~> linear l4
  ~> sigmoid
```

We can also specify the exact dimensions

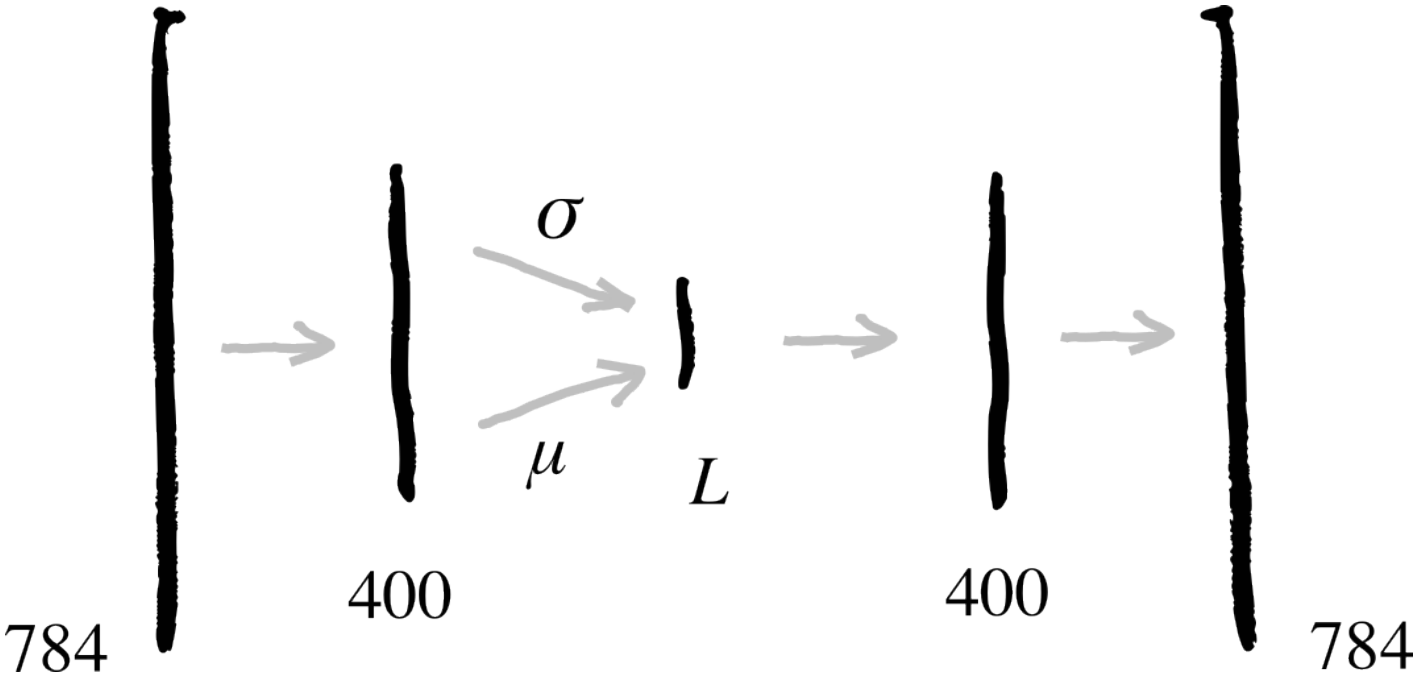
```
aeConfig =
  AESpec
    (LinearSpec 784 400)
    (LinearSpec 400 latent_size)
    (LinearSpec latent_size 400)
```

```
    (LinearSpec 400 784)
```

Of course, a smaller L (latent_size), the more information is lost. Therefore, depending on the application we may want to change this value. As a rule of thumb, L contains the smallest number of neurons to enforce the compression. In this article we set L=2 so that we can simply reveal our latent space in two dimensions.

Variational autoencoder

The principal difference of variational autoencoders (VAE) from normal autoencoders is in the bottleneck. Instead of a compressed input, it estimates a distribution. In practice, VAE estimates the mean μ and the standard deviation σ -- normal distribution parameters. By sampling from that distribution, a new unseen before object can be generated. Like a new font or a new piece of cloth. Or a face. Or a melody. Isn't that nice?



Variational autoencoder. Arrows signify fully-connected layers and vertical bars are data vectors.

```
data VAESpec = VAESpec
  {
    -- Encoder trainable parameters (phi)
    fc1 :: LinearSpec,
    fcMu :: LinearSpec,
    fcSigma :: LinearSpec,
```

```
    -- Decoder trainable parameters (theta)
    fc5 :: LinearSpec,
    fc6 :: LinearSpec
  }
  deriving (Show, Eq)

myConfig =
  VAESpec
    (LinearSpec 784 400)
    (LinearSpec 400 latent_size)
    (LinearSpec 400 latent_size)
    (LinearSpec latent_size 400)
    (LinearSpec 400 784)

data VAE = VAE
  { l1 :: Linear,
    lMu :: Linear,
    lSigma :: Linear,
    l4 :: Linear,
    l5 :: Linear
  }
  deriving (Generic, Show, Parameterized)
```

It can be useful to have separate encode qϕ(z|x) and decode pθ(x|z) functions.

```
encode :: VAE -> Tensor -> (Tensor, Tensor)
encode VAE {..} x0 =
  let enc_ =
        linear l1
         ~> relu

      x1 = enc_ x0
      mu = linear lMu x1
      logSigma = linear lSigma x1
  in (mu, logSigma)

decode :: VAE -> Tensor -> Tensor
decode VAE {..} =
  linear l4
```

```
  ~> relu
  ~> linear l5
  ~> sigmoid
```

Then, the complete variational autoencoder will be

```
vaeForward :: VAE -> Bool -> Tensor -> IO (Tensor, Tensor, Tensor)
vaeForward net@(VAE {..}) _ x0 = do
  let (mu, logSigma) = encode net x0
      sigma = exp (0.5 * logSigma)

  eps <- toLocalModel' <$> randnLikeIO sigma

  let z = (eps `mul` sigma) `add` mu
      reconstruction = decode net z

  return (reconstruction, mu, logSigma)
```

Pay a special attention to the reparametrization: $$\varepsilon \sim \mathcal{N}(0, 1)$$ $$z = \varepsilon \odot \sigma + \mu$$ Where z is our latent vector, noise ε is sampled from the normal distribution (randnLikeIO1), and ⊙ is elementwise product. Thanks to this trick, we can backpropagate through a stochastic layer. See this excellent post for more details. There are two differences between variational and ordinary autoencoder training: (1) the reparametrization and (2) the loss function, which we cover below.

VAE Loss Function

The loss function consists of two parts:

$$\text{loss} = \mathbb{L}(\mathbf{x}, \hat{\mathbf{x}}) + \mathrm{D}_{\mathrm{KL}}\left(\mathrm{q}_{\phi}(\mathrm{z}) || \mathrm{p}_{\theta}(\mathrm{z})\right).$$

L(x,^x) is the reconstruction loss. It decreases when the decoded output ^x is closer to the original input x. This is basically the loss of an ordinary autoencoder. For instance, it can be binary cross-entropy loss or L2 loss.

And the second term DKL(·) is the Kullback-Leibner divergence. It tells how much the distribution pθ(z) is different from the distribution qϕ(z). Or even more informative is to say that KL divergence

tells ho

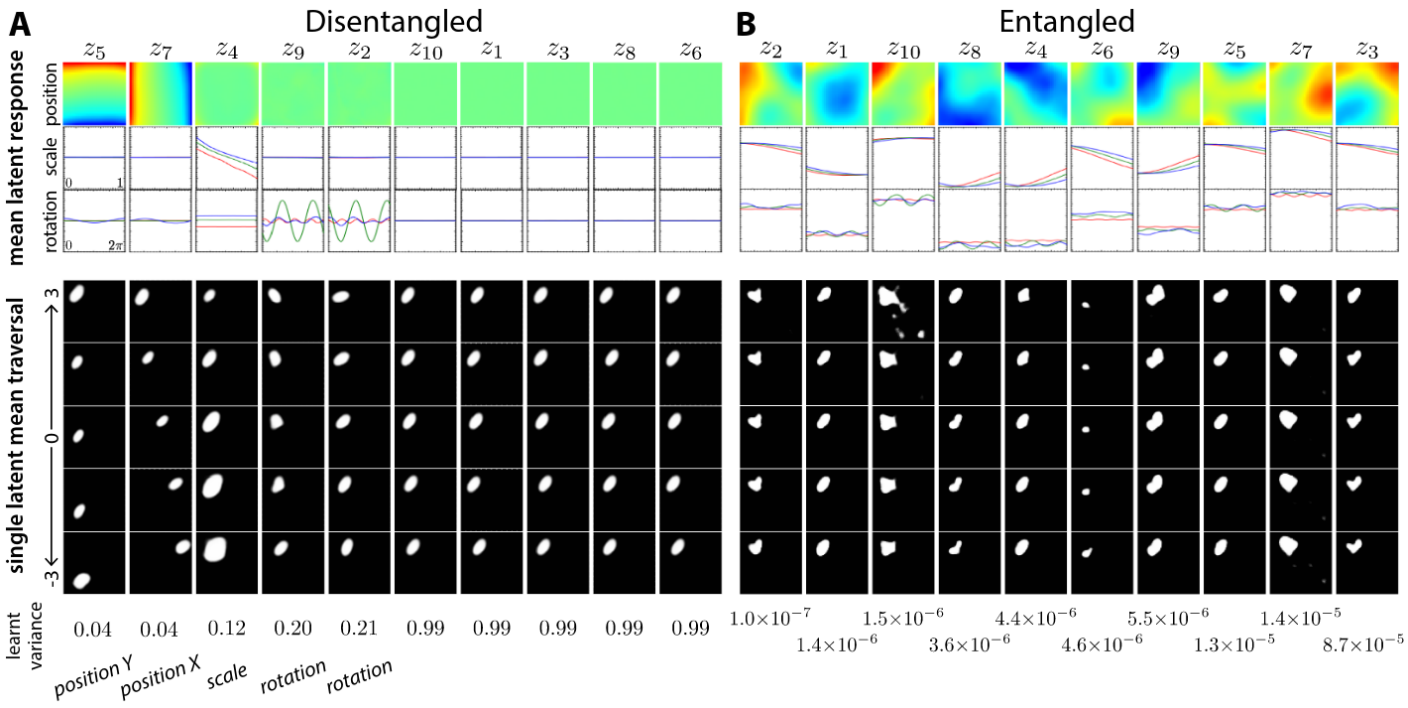
$$-\mathrm{D}_{\mathrm{KL}}\left(\mathrm{q}_{\phi}(\mathrm{z})||\mathrm{p}_{\theta}(\mathrm{z})\right) = \frac{1}{2}\sum_{\mathrm{j}=1}^{\mathrm{J}}\left(1+\log(\sigma_{\mathrm{j}}^{2})-\mu_{\mathrm{j}}^{2}-\sigma_{\mathrm{j}}^{2}\right).$$

paper,

Therefore, the complete VAE loss is

```
vaeLoss :: Float -> Tensor -> Tensor -> Tensor -> Tensor -> Tensor
vaeLoss beta recon_x x mu logSigma = reconLoss + asTensor beta * kld
  where
    reconLoss = bceLoss recon_x x
    kld = -0.5 * sumAll (1 + logSigma - pow (2 :: Int) mu - exp logSigma)
```

We also include the β≥0 term. When β=0 the networks is trained as an ordinary autoencoder. When β=1, we have a classical VAE. And when β>1, we force latent vector representations disentanglement. As we can see from the image below, in case of disentanglement, there are separate latent variables that encode position, rotation, and scale. Whereas entangled variables tend to encode all object properties at the same time. I recommend the excellent article by Higgins et al., which is featuring some insights from neuroscience.



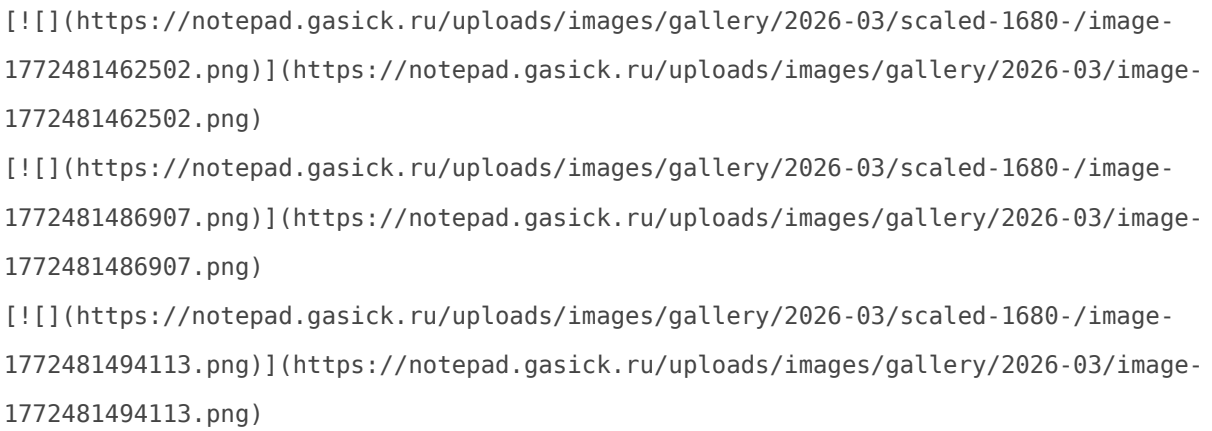
Disentangled vs entangled latent representations. Disentangled vs entangled latent representations. Image source: Higgins et al. 2016.

The binary cross-entropy loss is defined as

```
bceLoss target x =
  -sumAll (x * Torch.log(1e-10 + target) + (1 - x) * Torch.log(1e-10 + 1 - target))
```
We add a small term 10−10 to avoid numerical errors due to log(0).

Visualizing the Latent Space

Here is how our latent space looks for different values of β.



Test data distributions for different β values.

And here is how we compute that

```haskell
main = do
    (trainData, testData) <- initMnist "data"
    net0 <- toLocalModel' <$> sample myConfig

    beta_: _ <- getArgs
    putStrLn $ "beta = " ++ beta_

    let beta = read beta_
        trainMnistStream = V.MNIST { batchSize = 128, mnistData = trainData }
        testMnistStream = V.MNIST { batchSize = 128, mnistData = testData }
        epochs = 20
        cpt = printf "VAE-Aug2022-beta_%s.ht" beta_
        logname = printf "beta_%s.log" beta_

    putStrLn "Starting training..."
    net' <- time $ train beta trainMnistStream epochs net0
    putStrLn "Done"
```

```
    -- Saving the trained model
    save' net' cpt

    -- Restoring the model
    net <- load' cpt

    -- Test data distribution in the latent space
    putStrLn $ "Saving test dataset distribution to " ++ logname
    _ <- testLatentSpace logname testMnistStream net
```

Where

```
testLatentSpace :: FilePath -> V.MNIST IO -> VAE -> IO ()
testLatentSpace fn testStream net = do
      runContT (streamFromMap (datasetOpts 2) testStream) $ recordPoints fn net. fst

recordPoints :: FilePath -> VAE -> ListT IO (Tensor, Tensor) -> IO ()
recordPoints logname net = P.foldM step begin done. enumerateData
  where
    step :: () -> ((Tensor, Tensor), Int) -> IO ()
    step () args = do
      let ((input, labels), i) = toLocalModel' args
          (encMu, _) = encode net input
          batchSize = head $ shape encMu

      let s = toStr $ Torch.cat (Dim 1) [reshape [-1, 1] labels, encMu]
      appendFile logname s

      return ()

    done () = pure ()
    begin = pure ()

toStr :: Tensor -> String
toStr dec =
    let a = asValue dec :: [[Float]]
        b = map (unwords. map show) a
     in unlines b
```

Now, let's take a walk around our latent space to get an idea how it looks like inside.

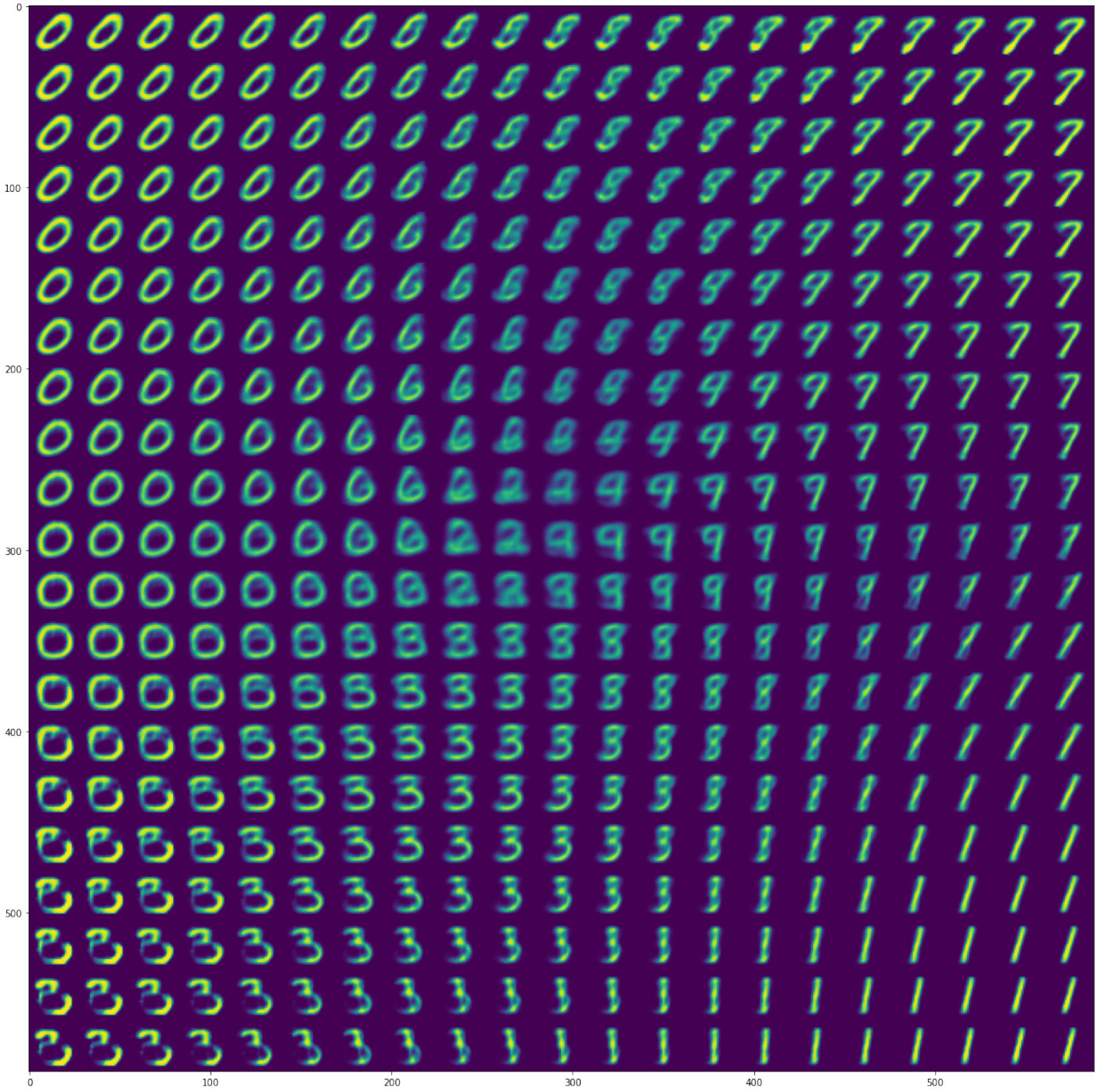
```
main = do
    net <- load' "VAE-Aug2022-beta_1.ht"

    let xs = [-3,-2.7..3::Float]

        -- 2D latent space as a Cartesian product
        zs = [ [x,y] | x<-xs, y<-xs ]

        decoded = Torch.cat (Dim 0) $
                    map (decode net. toLocalModel'. asTensor. (:[])) zs

    writeFile "latent_space_2D.txt" (toStr decoded)
```



Some examples from 2D latent space. Some examples from 2D latent space.

Pretty neat! We see gradual transitions between different digits. Note that these digits are actually generated by VAE.

ConvNet VAE

While we have built a simple variational autoencoder based on MLP (multilayer perceptron), nothing prevents us from using other architectures. In fact, let's build a convolutional VAE!

```
data VAESpec = VAESpec
  {
    -- Encoder trainable parameters
    conv1 :: Conv2dSpec,
    conv2 :: Conv2dSpec,
    conv3 :: Conv2dSpec,
    fcMu :: LinearSpec,
    fcSigma :: LinearSpec,

    -- Decoder trainable parameters
    fc :: LinearSpec,
    deconv1 :: ConvTranspose2dSpec,
    deconv2 :: ConvTranspose2dSpec,
    deconv3 :: ConvTranspose2dSpec
  }
  deriving (Show, Eq)

myConfig =
  VAESpec
    (Conv2dSpec 1 32 4 4)    -- 1 -> 32 channels; 4 x 4 kernel
    (Conv2dSpec 32 64 4 4)   -- 32 -> 64 channels; 4 x 4 kernel
    (Conv2dSpec 64 128 3 3)  -- 64 -> 128 channels; 3 x 3 kernel
    (LinearSpec (2 * 2 * 128) latent_size)
    (LinearSpec (2 * 2 * 128) latent_size)
    (LinearSpec latent_size 1024)
    (ConvTranspose2dSpec 1024 256 4 4)
    (ConvTranspose2dSpec 256 128 6 6)
    (ConvTranspose2dSpec 128 1 6 6)

data VAE = VAE
  { c1 :: Conv2d,
    c2 :: Conv2d,
    c3 :: Conv2d,
    lMu :: Linear,
    lSigma :: Linear,
    l :: Linear,
    t1 :: ConvTranspose2d,
    t2 :: ConvTranspose2d,
    t3 :: ConvTranspose2d
  }
```

```
  deriving (Generic, Show, Parameterized)

instance Randomizable VAESpec VAE where
  sample VAESpec {..} =
    VAE
      <$> sample conv1
      <*> sample conv2
      <*> sample conv3
      <*> sample fcMu
      <*> sample fcSigma
      <*> sample fc
      <*> sample deconv1
      <*> sample deconv2
      <*> sample deconv3

encode :: VAE -> Tensor -> (Tensor, Tensor)
encode VAE {..} x0 =
  let enc_ =
          -- Reshape vectors [batch_size x 784]
          -- into grayscale images of [batch_size x 1 x 28 x 28]
          reshape [-1, 1, 28, 28]
          -- Stride 2, padding 0
          ~> conv2dForward c1 (2, 2) (0, 0)
          ~> relu
          ~> conv2dForward c2 (2, 2) (0, 0)
          ~> relu
          ~> conv2dForward c3 (2, 2) (0, 0)
          ~> relu
          ~> flatten (Dim 1) (Dim (-1))

      x1 = enc_ x0
      mu = linear lMu x1
      logSigma = linear lSigma x1
   in (mu, logSigma)

decode :: VAE -> Tensor -> Tensor
decode VAE {..} =
         linear l
         ~> relu
         ~> reshape [-1, 1024, 1, 1]
```

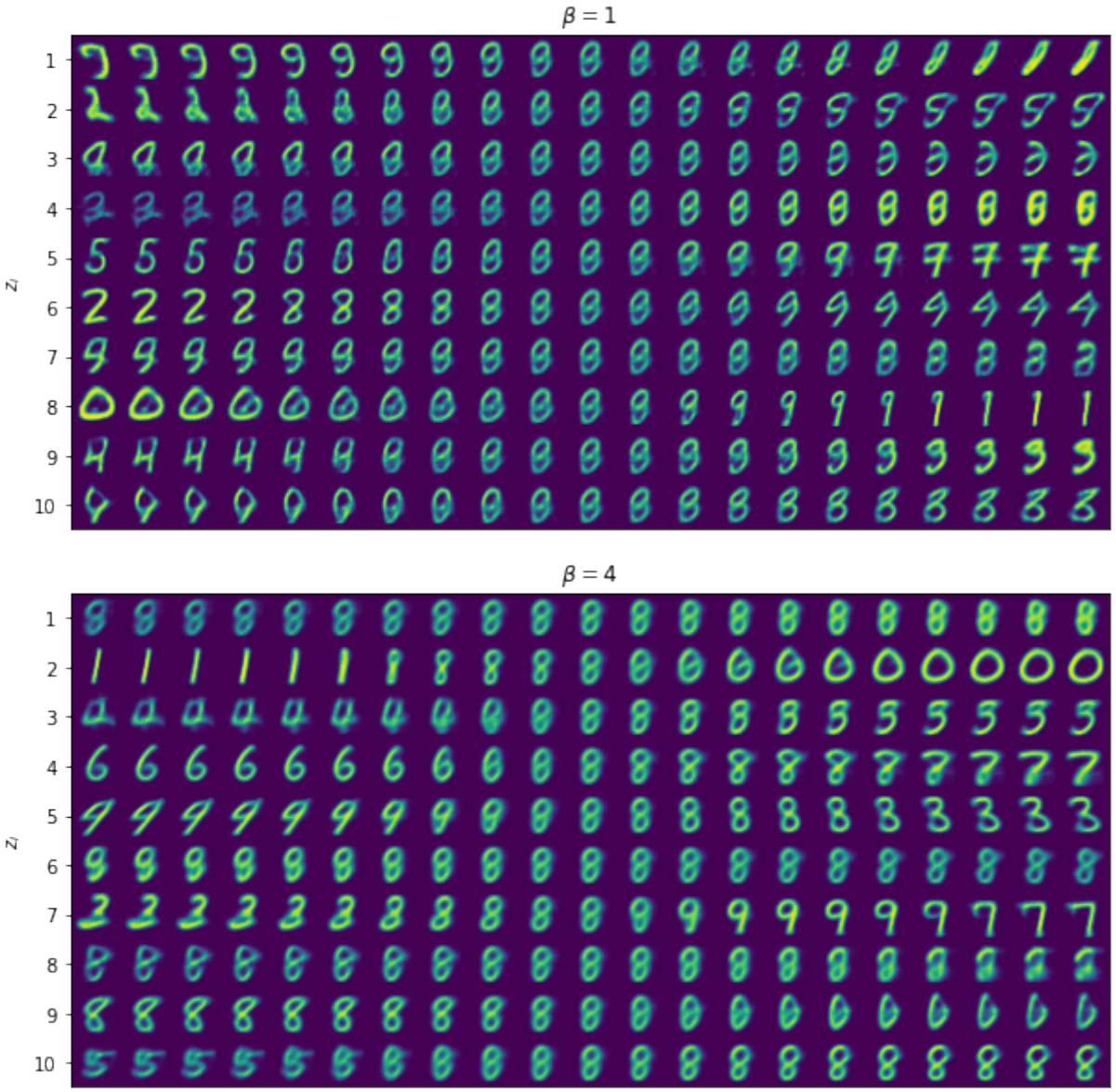
```
        -- Stride 2, padding 0
        ~> convTranspose2dForward t1 (2, 2) (0, 0)
        ~> relu
        ~> convTranspose2dForward t2 (2, 2) (0, 0)
        ~> relu
        ~> convTranspose2dForward t3 (2, 2) (0, 0)
        ~> sigmoid
        ~> reshape [-1, 784]  -- Reshape back
```

And that is all we need.

Here is the latent space for β=1 (normal VAE) using our CNN architecture:

Disentanglement

To better illustrate how parameter β encourages disentanglement between latent representations, let us first increase the latent dimension to L=10. For β=1 and β=4 we perform scan along each individual z coordinate.



Indeed, the latent space under β=4 looks more disentangled compared to β=1. We can see e.g. that z1 is the parameter that defines how "light" or how "bold" is the digit, whereas z2 controls how wide is the digit. Whereas such individual components for β=1 are hard to identify. For instance when β=1, z4 controls not only how "bold" is the digit, but also its shape.

For more details, see this notebook.

Find the complete project and associated data on Github. For suggestions about the content feel free to open a new issue.

Summary

Variational autoencoder is a great tool in modern deep learning. Manipulating the latent space allows us not only to "interpolate" between different images or other objects, but also to perform inpainting (adding details to incomplete images) or even zero shot learning. The last one is crucial for the so-called artificial general intelligence.

The loss function is important for VAE training. If your VAE does not work as expected, the odds are that the loss function is not implemented correctly. Also check if the random noise ε is drawn from the normal distribution.

Citation

```
@article{penkovsky2022VAE,
 title   = "Roaming The Latent Space",
 author  = "Penkovsky, Bogdan",
 journal = "penkovsky.com",
 year    = "2022",
 month   = "August",
 url     = "https://penkovsky.com/neural-networks/day9/"
}
```

Learn More

Auto-Encoding Variational Bayes https://arxiv.org/abs/1312.6114 A Beginner's Guide to Variational Methods: Mean-Field Approximation https://blog.evjang.com/2016/08/variational-bayes.html Early Visual Concept Learning with Unsupervised Deep Learning https://arxiv.org/abs/1606.05579 Pytorch VAE Implementation https://github.com/pytorch/examples/blob/main/vae/main.py Interpolating Music https://magenta.tensorflow.org/music-vae β-VAE: Learning Basic Visual Concepts With A Constrained Variational Framework https://openreview.net/pdf?id=Sy2fzU9gl The Reparameterization Trick https://gregorygundersen.com/blog/2018/04/29/reparameterization/ KL is All You Need https://blog.alexalemi.com/kl-is-all-you-need.html Generating Diverse High-Fidelity Images with VQ-VAE-2 https://arxiv.org/abs/1906.00446 Generating Diverse Structure for Image Inpainting With Hierarchical VQ-VAE https://openaccess.thecvf.com/content/CVPR2021/papers/Peng_Generating_Diverse_Structure_for_I

mage_Inpainting_With_Hierarchical_VQ-VAE_CVPR_2021_paper.pdf Variational Image Inpainting https://vincentcartillier.github.io/papers/variational-image-inpainting.pdf

A Technical Sidenote

Compared to the previous day, the trainLoop is slightly modified: First, we rescale the images between 0 and 1. Second, we include our new loss function in the step function.

```
    step :: Optimizer o => (VAE, o) -> ((Tensor, Tensor), Int) -> IO (VAE, o)
    step (model, opt) args = do
      let ((x, _), iter) = toLocalModel' args
          -- Rescale pixel values [0, 255] -> [0, 1.0].
          -- This is important as the sigmoid activation in decoder can
          -- reach values only between 0 and 1.
          x' = x / 255.0
      (recon_x, mu, logSigma) <- vaeForward model False x'
      let loss = vaeLoss beta recon_x x' mu logSigma
      -- Print loss every 100 batches
      when (iter `mod` 100 == 0) $ do
        putStrLn
          $ printf "Batch: %d | Loss: %.2f" iter (asValue loss :: Float)
      runStep model opt loss lr
```

The train function now uses Adam optimizer from Torch.Optim.CppOptim, which tends to be faster compared to mkAdam we used previously. This is not very different from mkAdam-based training, except that the learning rate is specified as Cpp.adamLr parameter and not as a trainLoop parameter (ignored when passed to runStep).

```
train :: Float -> V.MNIST IO -> Int -> VAE -> IO VAE
train beta trainMnist epochs net0 = do
    optimizer <- Cpp.initOptimizer adamOpt net0

    (net', _) <- foldLoop (net0, optimizer) epochs $ \(net', optState) _ ->
      runContT (streamFromMap dsetOpt trainMnist)
      $ trainLoop beta (net', optState) 0.0 . fst
```

```
    return net'
  where
    dsetOpt = datasetOpts workers
    workers = 2
    -- Adam optimizer parameters
    adamOpt =
        def
          { Cpp.adamLr = learningRate,
            Cpp.adamBetas = (0.9, 0.999),
            Cpp.adamEps = 1e-8,
            Cpp.adamWeightDecay = 0,
            Cpp.adamAmsgrad = False
          } ::
          Cpp.AdamOptions
```

```
learningRate :: Double
learningRate = 1e-3
```

I used a compiled version instead of a notebook since the network training worked much faster (the bottleneck was in training data mini-batches loading). Also I have trained networks with Torch.Optim.CppOptim. It is slightly faster compared to mkAdam.

I was also wondering why I get large values out of the encoder. It turns out that this is because relu function is unbounded. You may want to replace relu with Torch.tanh and visualize the latent space again.

Revision #5
Created 2022-09-03 08:54:01 UTC by gasick
Updated 2026-03-02 20:02:20 UTC by gasick