Statistical Methods for single cell data analysis 3

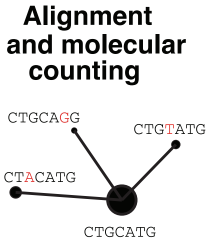
Yongjin Park, UBC Path + Stat, BC Cancer

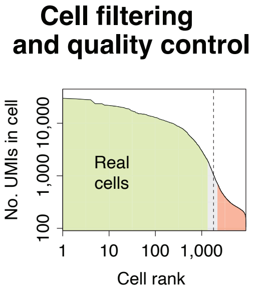
18 March 2024

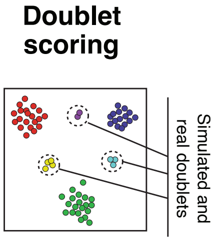
Source code available:

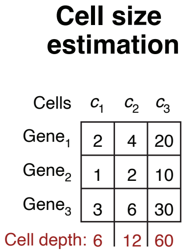
`https://github.com/stat540-UBC/lectures`

Overview of single-cell data analysis

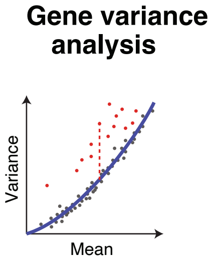







Cell size estimation

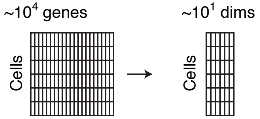
Cells	c_1	c_2	c_3
Gene_1	2	4	20
Gene_2	1	2	10
Gene_3	3	6	30
Cell depth:	6	12	60



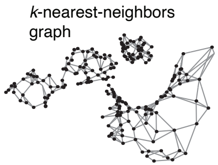
Karchenko, *Nature Methods* (2021)

Overview of single-cell data analysis cont’d

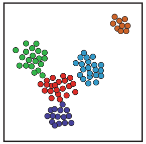
Reduction to a medium-dimensional space



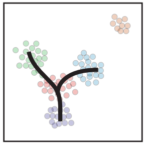
Manifold representation



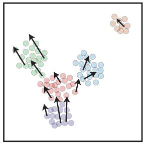
Clustering and differential expression



Trajectories



Velocity estimation



Karchenko, *Nature Methods* (2021)

The goal of modelling in high-dimensional space

The goal is to model two types of relationships:

1. Relationship between dimensions/features (genes)
2. Relationship between samples/data points (cells)

- One of the most fundamental relationships is co-variation.

Covariance calculation in high-dimensional space

Covariance between cell i and j across genes:

$$\text{cov}(X_i, X_j) = \mathbb{E}\left[X_i\, X_j\right] - \mathbb{E}[X_i]\,\mathbb{E}\left[X_j\right]$$

If $\mathbb{E}[X_i] = \mathbb{E}\left[X_j\right] = 0$ (e.g., standardization),

$$\text{cov}(X_i, X_j) = \mathbb{E}\left[X_i\, X_j\right] \approx \frac{1}{p}\sum_{g=1}^{p} X_{gi} X_{gj}$$

Covariance calculation in high-dimensional space - 2

Letting $\mathbf{x}_i^\top \equiv \left(X_{1i}, \ldots, X_{gi}, \ldots, X_{pi}\right)$ and $\mathbb{E}\left[X_{gi}\right] \approx p^{-1} \sum_g X_{gi} = 0$

Covariance calculation in high-dimensional space - 2

Letting $\mathbf{x}_i^\top \equiv \left(X_{1i}, \ldots, X_{gi}, \ldots, X_{pi}\right)$ and $\mathbb{E}\left[X_{gi}\right] \approx p^{-1} \sum_g X_{gi} = 0$

$$\text{cov}(X_i, X_j) = \mathbb{E}\left[X_i X_j\right] \approx \frac{1}{p} \sum_{g=1}^{p} X_{gi} X_{gj} = \frac{1}{p} \mathbf{x}_i^\top \mathbf{x}_j$$

Covariance calculation in high-dimensional space - 2

Letting $\mathbf{x}_i^\top \equiv (X_{1i}, \ldots, X_{gi}, \ldots, X_{pi})$ and $\mathbb{E}[X_{gi}] \approx p^{-1} \sum_g X_{gi} = 0$

$$\text{cov}(X_i, X_j) = \mathbb{E}[X_i X_j] \approx \frac{1}{p} \sum_{g=1}^{p} X_{gi} X_{gj} = \frac{1}{p} \mathbf{x}_i^\top \mathbf{x}_j$$

Full sample covariance:

$$\frac{1}{p} X^\top X = \frac{1}{p} \begin{pmatrix} \mathbf{x}_1^\top \mathbf{x}_1 & \cdots & \mathbf{x}_1^\top \mathbf{x}_n \\ \mathbf{x}_2^\top \mathbf{x}_1 & \cdots & \mathbf{x}_2^\top \mathbf{x}_n \\ & \vdots & \\ \mathbf{x}_n^\top \mathbf{x}_1 & \cdots & \mathbf{x}_n^\top \mathbf{x}_n \end{pmatrix}$$

Singular value decomposition simplifies covariance

$$X = UDV^\top$$

Gene covarince

$$\begin{aligned} XX^\top &= UDV^\top(UDV^\top)^\top \\ &= UDV^\top((V^\top)^\top D^\top U^\top) \\ &= UD\underbrace{\cancel{V^\top V}}_{}\overset{I}{}DU^\top \\ &= UD^2U^\top \end{aligned}$$

- The Matrix Cookbook: https://www2.imm.dtu.dk/pubdb/pubs/3274-full.html

Singular value decomposition simplifies covariance

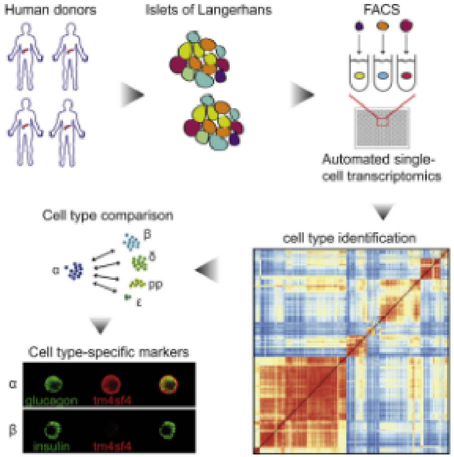
$$X = UDV^{\top}$$

Sample covarince

$$\begin{aligned} X^{\top}X &= (UDV^{\top})^{\top}UDV^{\top} \\ &= ((V^{\top})^{\top}D^{\top}U^{\top})UDV^{\top} \\ &= VD\underbrace{\cancel{U^{\top}U}}_{}\overset{I}{}DV^{\top} \\ &= VD^{2}V^{\top} \end{aligned}$$

- The Matrix Cookbook: https://www2.imm.dtu.dk/pubdb/pubs/3274-full.html

Example: scRNA-seq data of human pancreatic cells



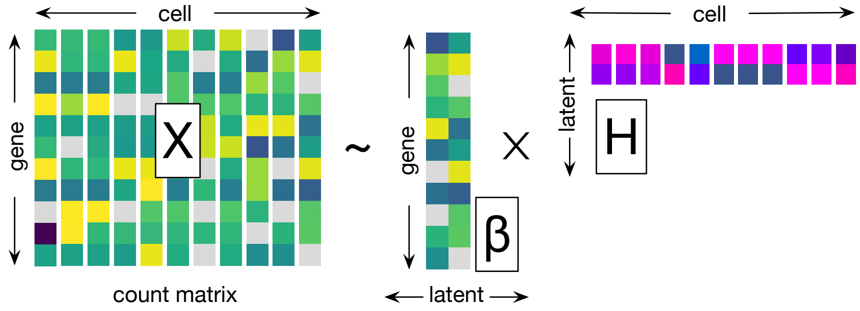
Muraro *et al.* *Cell Systems* (2016)

- 19,140 genes/features/rows
- 3,072 cells/columns
- 12,442,034 non-zero elements
- $\approx$ 21 % non-zero elements

Today’s lecture

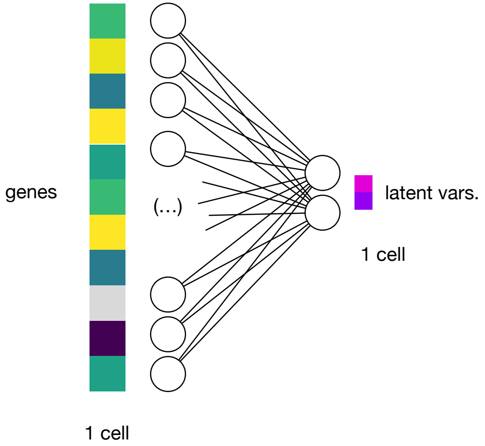
1. Recent developments in latent factor modelling

Recap: How can we learn patterns in unsupervised learning from single cell count data?



- Goal: Find the factor-specific gene dictionary β and hidden factor loading H.

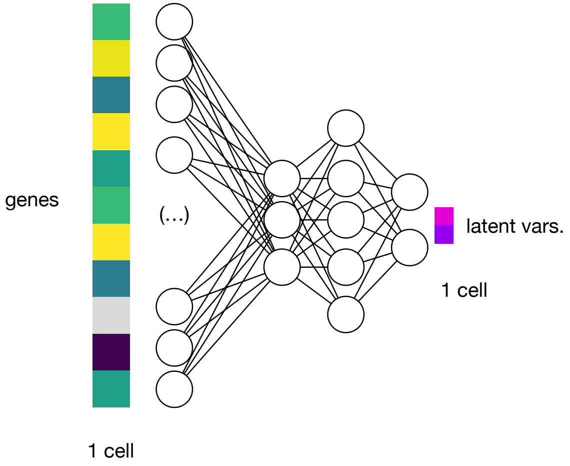
Can we "encode" high-dim data to low-dim hidden variables?



- Take one cell as a long vector $\mathbf{x}_i$
- Apply an encoding function $f(\mathbf{x}_i)$
- Neural network architectures capture relationships between variables

- no connection within each layer

Can we "encode" high-dim data to low-dim hidden variables?



- Take one cell as a long vector $\mathbf{x}_i$
- Apply an encoding function $f(\mathbf{x}_i)$
- Neural network architectures capture relationships between variables

$$h_i^{(l)} \leftarrow f(\sum_j \underbrace{W_{ij}}_{\text{connections}} \; h_j^{(l-1)} + \underbrace{b_i}_{\text{bias}})$$

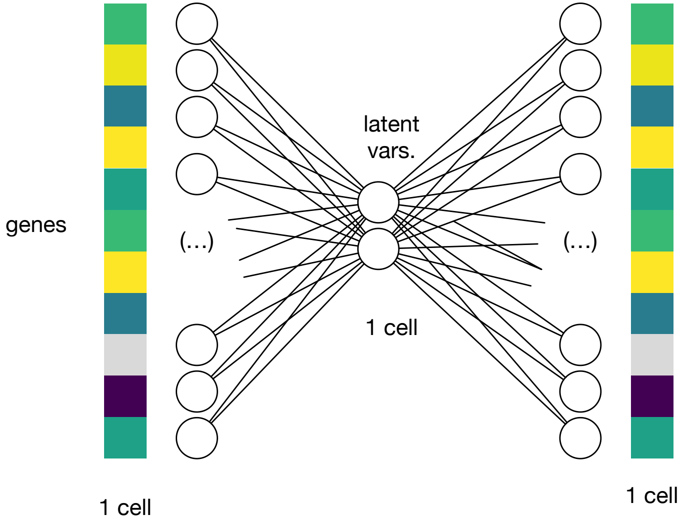
- no connection within each layer

Unsupervised learning of a good encoder model is fundamentally challenging because …

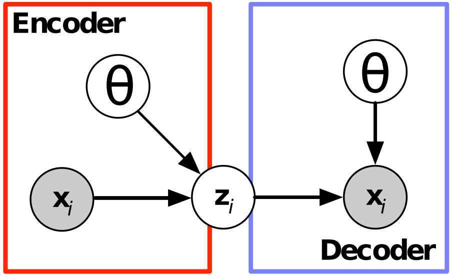
Unsupervised learning of a good encoder model is fundamentally challenging because …

we don’t have good encoding “examples” beforehand.

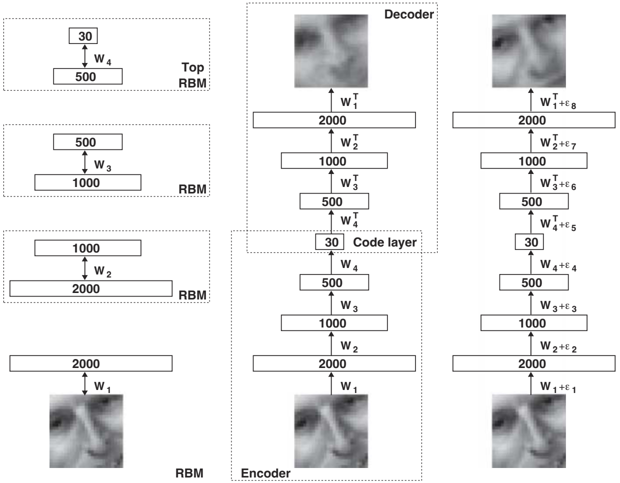
How can we "supervise" how well we encoded?



“Supervise” unsupervised learning by self reconstruction



Deep autoencoder first proposed in computer vision



- Hinton & Salakhutdinov, Science (2006)
- One of the first "deep learning" idea
- Learning the latent representation of an image helps subsequent classification tasks.
- To train a deep autoencoder architecture, they pretrained the model layer by layer.

Let’s build a simple autoencoder model using `torch`

```r
################################################################
## R wrapper library for `libtorch` (C++ engine for PyTorch) ##
################################################################

library(torch)

############################################
## We can use GPU (NVIDIA Cuda), optional ##
############################################

GPU <- torch_device("cuda")
```

- See this online book if you are interested in building your own deep learning model: **Deep Learning and Scientific Computing with R torch**

- https://skeydan.github.io/Deep-Learning-and-Scientific-Computing-with-R-torch/

Why do we use some Deep Learning library?

- Built-in functions for scientific computing
  - `log`, `log1p`, `exp`, `softmax`, `sigmoid`, `softplus`
- Faster computation both in CPU and GPU (in general)
- **No need to work on differentiation** by hands
- Trouble shooting by ML community

Encoder: high-dim data $\longrightarrow$ low-dim latent space

```r
build.linear.encoder <-
    nn_module(
        ## How do we want to build this module ##
        classname = "linear encoder",
        initialize = function(d.in, K){
            self$K <- K # number of hidden factors
            self$func.z <- nn_linear(d.in, K) # a linear neural net layer
            self$bn <- nn_batch_norm1d(d.in) # batch norm
        },
        ## Define how do we propate infomration ##
        forward = function(x.b){
            self$get.latent(x.b)
        },
        get.latent = function(x.b){
            x.b <- torch_log1p(x.b) # log1p transformation
            x.b <- self$bn(x.b) # to expedite training
            self$func.z(x.b) # take linear combinations
        })
```

Check if this encoder module works okay

```
lin.encoder <- build.linear.encoder(ncol(x.torch), 7)$to(dev = GPU)
lin.encoder
```

```
## An `nn_module` containing 172,267 parameters.
##
## -- Modules ---------------------------------------------------------------
## * func.z: <nn_linear> #133,987 parameters
## * bn: <nn_batch_norm1d> #38,280 parameters
```

```
#########################################
## Test using the first five cells/data ##
#########################################

lin.encoder(x.torch[1:5, ])
```

```
## torch_tensor
## -0.3672  0.5029  0.6738 -0.0900  0.2418 -0.8185 -0.0766
##  0.9078 -0.7000 -0.1856  0.4843 -0.7655  0.0200  0.2828
##  0.1803  0.1363 -0.5774  0.0312  0.2056  0.0023 -0.3558
## -0.5873  0.3323  1.1573 -0.2551 -0.5147 -0.1735 -0.6886
## -0.1098 -0.2509 -1.0951 -0.1718  0.8373  0.9424  0.8035
## [ CUDAFloatType{5,7} ][ grad_fn = <AddmmBackward0> ]
```

Decoder: low-dim latent space $\longrightarrow$ high-dim data

```r
build.linear.decoder <-
    nn_module(
        classname = "linear decoder",
        ## Define how we want to build this module
        initialize = function(n.out, K) {
            self$K <- K
            self$func.logX <- nn_linear(K, n.out) # latent dim -> data dim
        },
        ## Define how do get back high-dim data
        forward = function(z.b){
            .num <- self$func.logX(z.b)
            .denom <- torch_logsumexp(.num, dim = -1, keepdim = T)
            return(.num - .denom)
        },
        ## Helper function
        get.weight = function(){
            self$func.logX$weight
        })
```

Check flow from the encoder to decoder

```r
lin.decoder <- build.linear.decoder(ncol(x.torch), K=7)$to(device=GPU)

x.input <- x.torch[1:5, ]
z.b <- lin.encoder(x.input)

#############################################
## reconstruction of x based on the latent ##
#############################################
logx.recon <- lin.decoder(z.b)
logx.recon[, 1:7]
```

```
## torch_tensor
## -10.2171 -10.0874  -9.8182  -9.4994 -10.5867 -10.1833  -9.9308
##  -9.6669  -9.3208  -9.3459  -9.8067  -9.5030 -10.4431  -9.0368
## -10.2895 -10.0550  -9.9206  -9.7503  -9.9255 -10.2100  -9.8453
##  -9.9890 -10.2931 -10.0914  -9.2689 -10.5573 -10.2492 -10.2278
## -10.2378 -10.0500 -10.2929  -9.8909  -9.1810  -9.7929 -10.0502
## [ CUDAFloatType{5,7} ][ grad_fn = <SliceBackward0> ]
```

- Note: the reconstructed data matrix is in logarithm scale

What's really going on in terms of functions?

For each sample/cell i,

- Encoder:

$$\mathbf{z}_i \leftarrow \log\left(\text{normalize}\left(\mathbf{x}_i\right) + 1\right) \cdot \mathbf{W}^{(z)} + \mathbf{b}^{(z)}$$

- Decoder:

$$\hat{\mathbf{x}}_i \leftarrow \mathbf{z}_i \cdot \mathbf{W}^{(x)} + \mathbf{b}^{(x)}$$

Goal: to make the reconstructed data $\approx$ the input

- Gene expression frequency:

$$\rho_{ig} = \frac{\exp\left(\widehat{\log X}_{ig}\right)}{\sum_{g'} \exp\left(\widehat{\log X}_{ig'}\right)}$$

- Multinomial log-likelihood:

$$L_i \overset{\text{def}}{=} \sum_{g=1}^{p} X_{ig} \log \rho_{ig}$$

$$\log \rho_{ig} = \widehat{\log X}_{ig} - \log \sum_{g'} \exp\left(\widehat{\log X}_{ig'}\right)$$

Goal: to make the reconstructed data $\approx$ the input

- Gene expression frequency:

$$\rho_{ig} = \frac{\exp\left(\widehat{\log X}_{ig}\right)}{\sum_{g'} \exp\left(\widehat{\log X}_{ig'}\right)}$$

- Multinomial log-likelihood:

$$L_i \overset{\text{def}}{=} \sum_{g=1}^{p} X_{ig} \log \rho_{ig}$$

$$L_i = \sum_{g=1}^{p} X_{ig} \left[\widehat{\log X}_{ig} - \text{logSumExp}\left(\widehat{\log \mathbf{x}}_{i}\right)\right]$$

Goal: to make the reconstructed data $\approx$ the input

```r
multinom.llik <- function(x.input, logx.recon){
    torch_sum(x.input * logx.recon, dim = -1)
}

multinom.llik(x.input, logx.recon)
```

```
## torch_tensor
## 1e+05 *
## -2.3127
## -3.1764
## -1.7012
## -3.4007
## -1.6081
## [ CUDAFloatType{5} ][ grad_fn = <SumBackward1> ]
```

- We need to maximize this with respect to all the parameters in the encoder and decoder layers.
- Maximizing log-likelihood $\iff$ minimizing *negative* log-likelihood

Torch provides a convenient way to "differentiate"

```
loss <- -torch_mean(multinom.llik(x.input, logx.recon))
loss
```

```
## torch_tensor
## 243982
## [ CUDAFloatType{} ][ grad_fn = <SubBackward0> ]
```

```
loss$backward()
```

Put the encoder and decoder together (so that gradients flow)

```r
build.linear.autoenc <-
    nn_module(
        classname = "linear autoencoder",
        initialize = function(d.data, K){
            self$enc <- build.linear.encoder(d.data, K)
            self$dec <- build.linear.decoder(d.data, K)
        },
        forward = function(x){
            z <- self$enc(x)
            x.hat <- self$dec(z)
            .loss <- - multinom.llik(x, x.hat)
            list(loss = .loss)
        })
```

A bit more formal definition

Minimize the total loss

$$L \overset{\text{def}}{=} \sum_{i=1}^{n} \mathsf{Loss}\left(\mathbf{x}_i, \hat{\mathbf{x}}_i\right)$$

where

$$\hat{\mathbf{x}}_i = \mathsf{Decoder}\left(\mathsf{Encoder}\left(\mathbf{x}_i; \theta^{(\mathsf{enc})}\right); \theta^{(\mathsf{dec})}\right)$$

with respect to the parameters θ.

Update by stochastic gradient steps

Take gradient (direction to minimize the loss) for each parameter:

$$\nabla L \equiv \left(\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, \dots \right)$$

Update parameters:

$$\theta^{(t)} \leftarrow \underset{\text{the previous}}{\theta^{(t-1)}} - \underset{\text{learning rate}}{\rho_t} \underbrace{\nabla L}_{\text{gradient at } t-1}$$

Update by stochastic gradient steps

```r
## register parameters to ADAM optimizer
.params <- linear.autoenc$parameters
adam <- optim_adam(.params, lr = 1e-2)
adam$zero_grad()
```

```r
x.b <- x.torch[1:3, ]       # Select minibatch data
out <- linear.autoenc(x.b) # data -> latent -> recon.
out$loss                   # loss evaluated on X.b
```

```
## torch_tensor
## 1e+05 *
##  2.3200
##  3.1856
##  1.7112
## [ CUDAFloatType{3} ][ grad_fn = <SubBackward0> ]
```

1. Take minibatch $X^{(b)}$
2. Recon. $\hat{X}^{(b)} = \text{Dec}(\text{Enc}(X^{(b)}))$
3. Evaluate $\text{Loss}(X^{(b)}, \hat{X}^{(b)})$

Update by stochastic gradient steps

```r
loss.b <- torch_sum(out$loss) #
loss.b$backward()             # numerically
                              # differentiate


## Before we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.0001 *
##  5.1905
## -13.0841
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

```r
adam$step() -> .
## After we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.001 *
## -9.4810
##  8.6916
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

Update by stochastic gradient steps

```r
loss.b <- torch_sum(out$loss) #
loss.b$backward()             # numerically
                              # differentiate

## Before we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.0001 *
##  5.1905
## -13.0841
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

```r
adam$step() -> .
## After we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.001 *
## -9.4810
##  8.6916
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

⋆ Aggregate training loss across samples in this minibatch: $\sum_{i \in \text{minibatch}(b)} \text{loss}\left(\mathbf{x}_i, \hat{\mathbf{x}}_i\right)$

Update by stochastic gradient steps

```
loss.b <- torch_sum(out$loss) #
loss.b$backward()             # numerically
                              # differentiate


## Before we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.0001 *
##  5.1905
## -13.0841
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

```
adam$step() -> .
## After we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.001 *
## -9.4810
##  8.6916
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

⋆ Aggregate training loss across samples in this minibatch: $\sum_{i \in \text{minibatch}(b)} \text{loss}\left(\mathbf{x}_i, \hat{\mathbf{x}}_i\right)$

⋆ Take gradient with respect to encoder and decoder parameters

Update by stochastic gradient steps

```r
loss.b <- torch_sum(out$loss) #
loss.b$backward()             # numerically
                              # differentiate

## Before we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.0001 *
##  5.1905
## -13.0841
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

```r
adam$step() -> .
## After we take one SGD step
head(adam$param_groups[[1]]$params$enc.func.z.bias, 2)
```

```
## torch_tensor
## 0.001 *
## -9.4810
##  8.6916
## [ CUDAFloatType{2} ][ grad_fn = <IndexBackward0> ]
```

⋆ Aggregate training loss across samples in this minibatch: $\sum_{i \in \text{minibatch}(b)} \text{loss}(\mathbf{x}_i, \hat{\mathbf{x}}_i)$

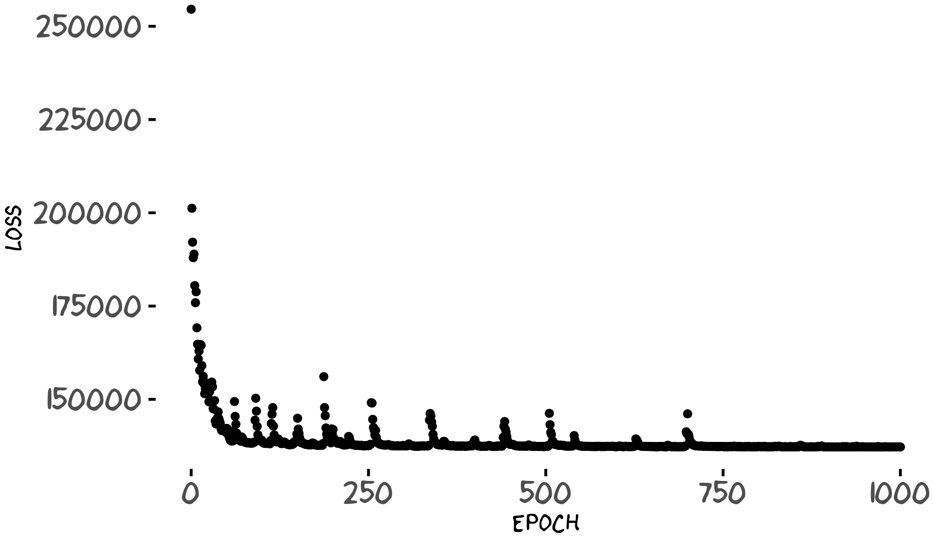
⋆ Take gradient with respect to encoder and decoder parameters

⋆ Update the parameters by taking one (stochastic) gradient descent step
$\theta^{(t)} \leftarrow \theta^{(t-1)} + \rho \nabla L$

Training algorithm: Repeat SGD steps many times

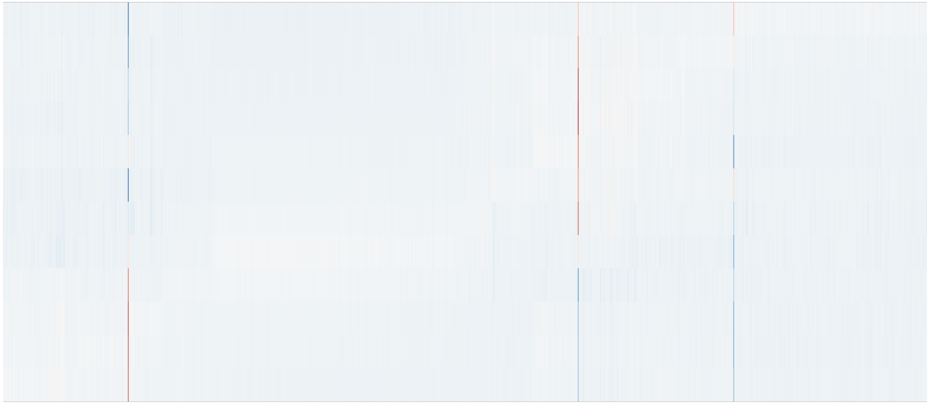
- For many epochs
  1. Sample mini batch data
  2. Evaluate loss function $L(\mathbf{x}_i, \widehat{\mathbf{x}}_i)$
  3. Compute gradient $\nabla_{\theta} L$
  4. Update parameters by SGD
- Report encoding results

Results: SGD minimized the loss function



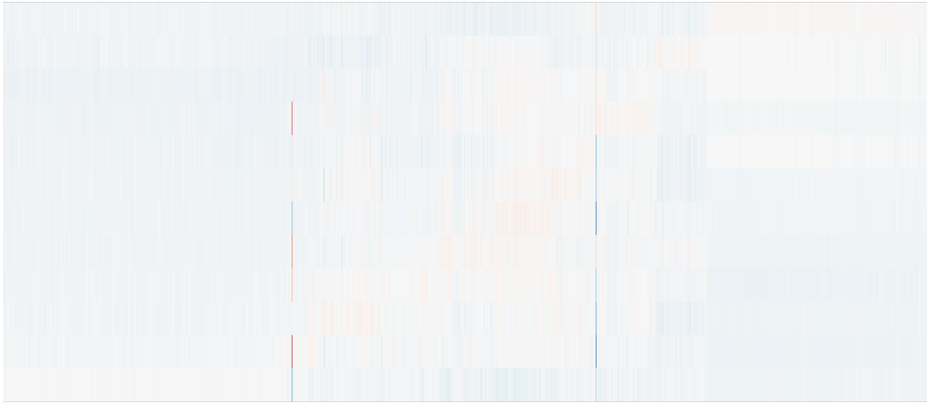
Hidden representations (factor $\times$ cell)

EPOCH = 1



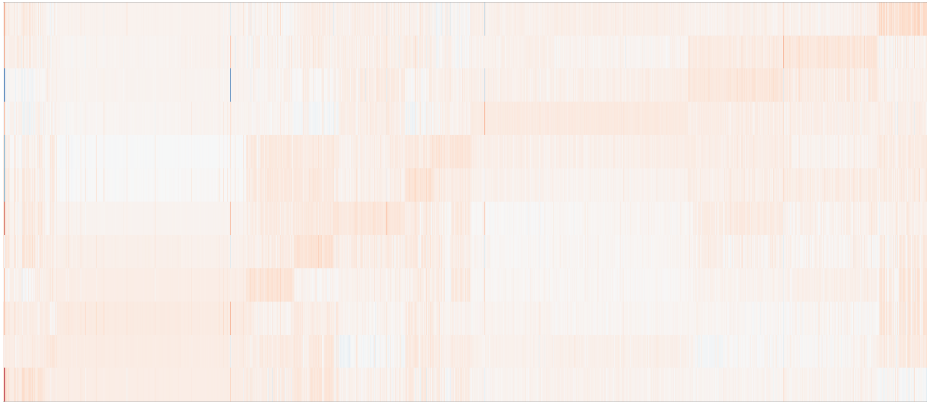
Hidden representations (factor $\times$ cell)

EPOCH = 26



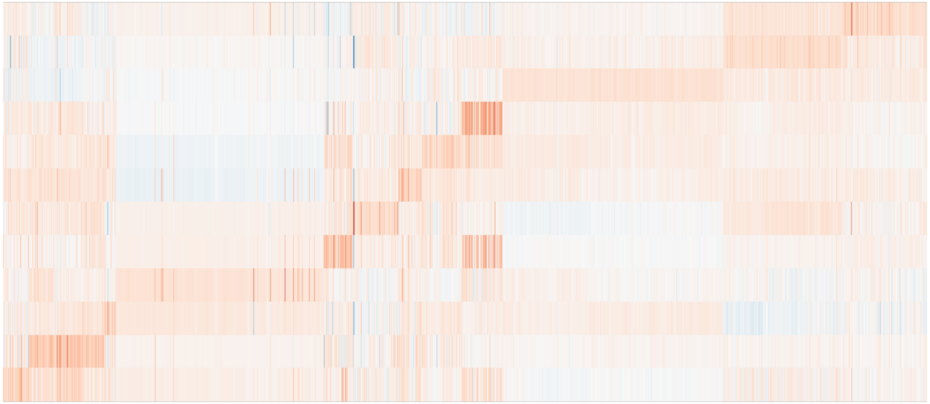
Hidden representations (factor $\times$ cell)

EPOCH = 226



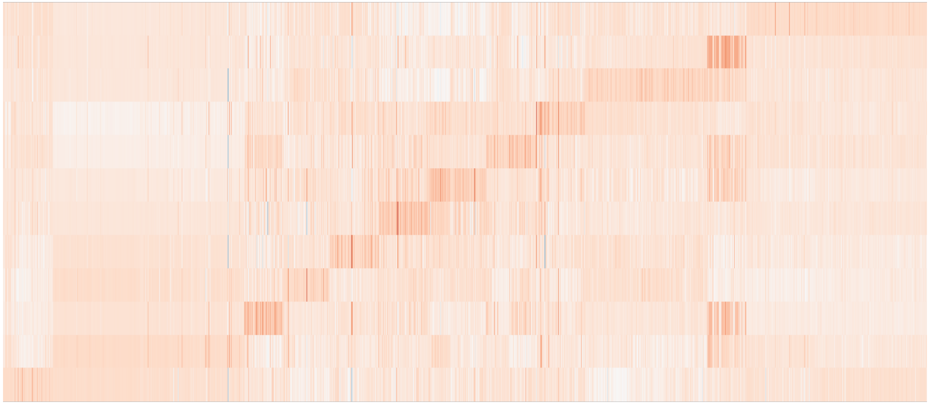
Hidden representations (factor $\times$ cell)

EPOCH = 476



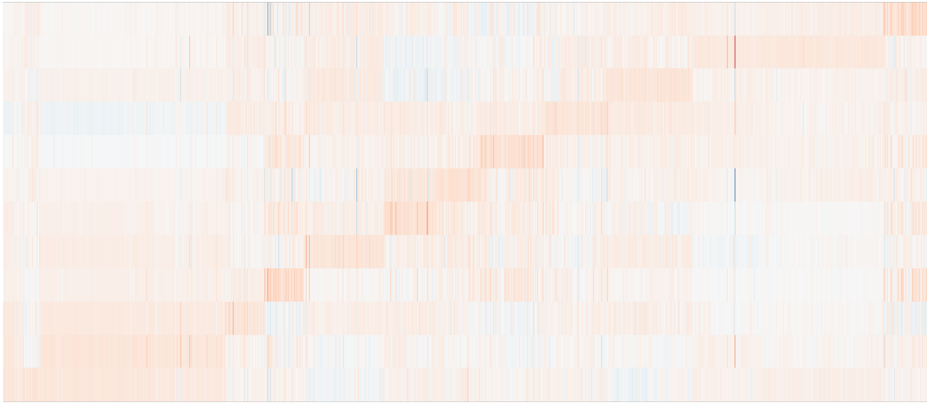
Hidden representations (factor $\times$ cell)

EPOCH = 726

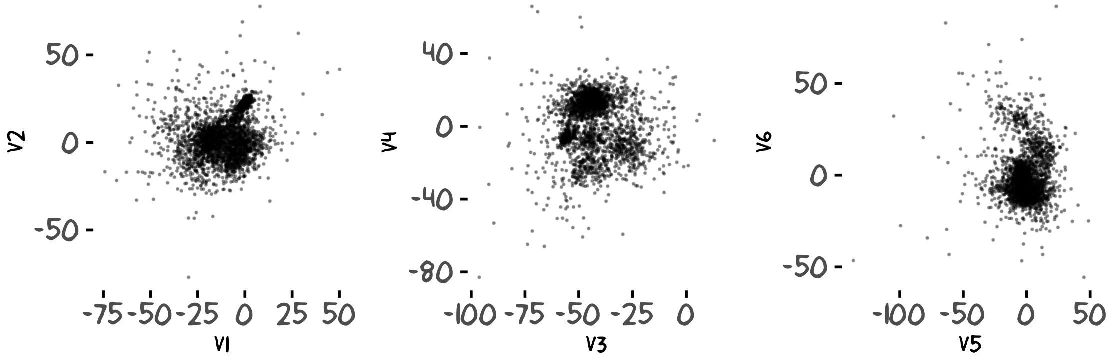


Hidden representations (factor $\times$ cell)

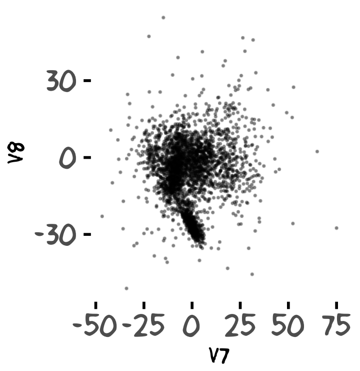
EPOCH = 976

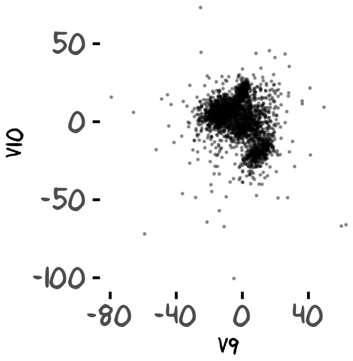


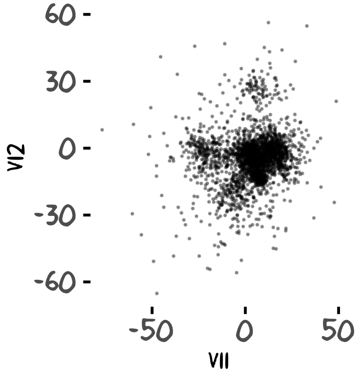
Latent dimensions estimated by the encoder model



Latent dimensions estimated by the encoder model

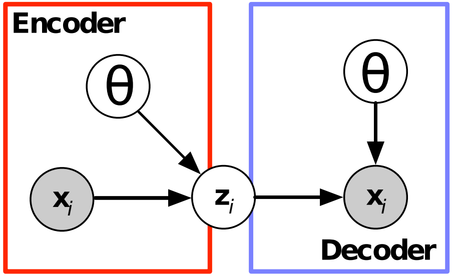




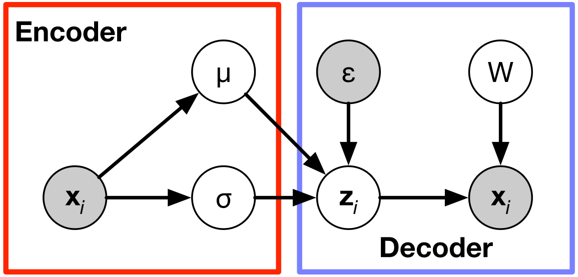


Variational autoencoder (VAE)

A classical autoencoder:

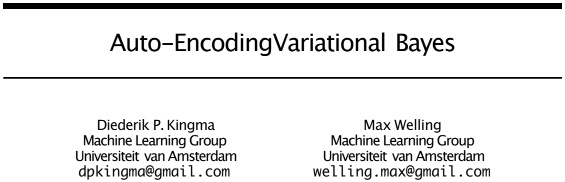


Variational autoencoder:



- Define relationships between variables (auto generative process)
- Usually, the decoder side captures our scientific hypothesis

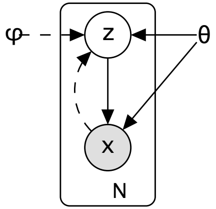
VAE approximates Bayesian inference



Auto-Encoding Variational Bayes

Diederik P. Kingma
Machine Learning Group
Universiteit van Amsterdam
dpkingma@gmail.com

Max Welling
Machine Learning Group
Universiteit van Amsterdam
welling.max@gmail.com



Prior:

$$Z \sim p(Z|\psi)$$

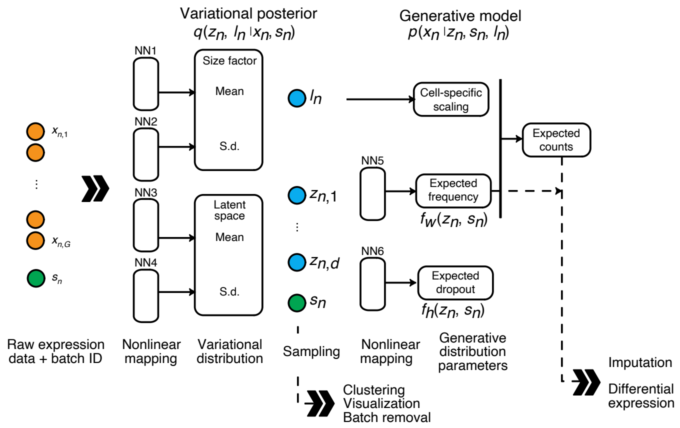
Data likelihood:

$$X \sim p(X|Z,\theta)$$

Variational Encoder:

$$q(Z|X) = \mathsf{NN}(X)$$

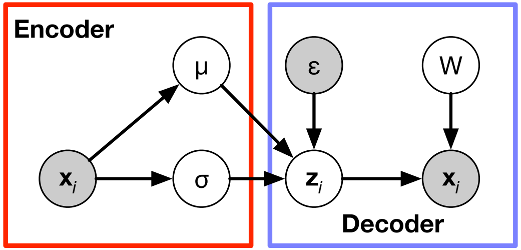
SCVI: deep generative model for scRNA-seq



Generative model: zero-inflated negative binomial distribution

Lopez, .., Jordan, Yosef, *Nature Methods* (2018)

A new encoder as a posterior inference machine



We need to define functions (neural networks) that maps from the data vector **x** to

- the mean of latent embedding: μ
- the log variance of latent embedding: σ
- $z_{ig} \leftarrow \mu_{ig} + \sigma_{ig}\epsilon, \quad \epsilon \sim \mathcal{N}(0,1)$

A new encoder as a posterior inference machine

```r
build.vae.encoder <- nn_module(
        classname = "vae encoder",
        initialize = function(d.in, K){
            self$K <- K                                         # number of hidden vars.
            self$z.mean <- nn_linear(d.in, K)                   # mean function
            self$z.logvar <- nn_linear(d.in, K)                 # log variance function
            self$bn <- nn_batch_norm1d(d.in)                    # batch norm
        },
        forward = function(x.b){
            x.b <- self$normalize(x.b)
            mm <- self$z.mean(x.b)                              # mean evaluated
            lv <- torch_clamp(self$z.logvar(x.b), -4.0, 4.0)    # log-var evaluated
            z <- mm + torch_randn_like(lv) * torch_exp(lv/ 2.) # stochastic z
            list(z = z, z.mean = mm, z.logvar = lv)
        },
        normalize = function(x.b){                              # normalization
            x.b <- torch_log1p(x.b)                             # log1p transformation
            self$bn(x.b)                                        # to expedite training
        }, ## helper function
        get.latent = function(x.b){ self$z.mean(self$normalize(x.b)) })
```

We will have two types of loss functions

Data log likelihood (a generative model)

$$\log \prod_{i=1}^{n} \underbrace{p(\mathbf{x}_i | \mathbf{z}_i)}_{\text{multinomial}}$$

```
multinom.llik <- function(x.input, logx.recon){
    torch_sum(x.input * logx.recon, dim = -1)
}
```

- The log-likelihood is the same as before
- KL loss will work like regularization

Divergence between prior and posterior

$$D_{\mathrm{KL}}\left(\underbrace{q(\mathbf{z})}_{\text{posterior}} \middle\| \underbrace{p(\mathbf{z})}_{\text{prior}} \right)$$

```
kl.loss <- function(.mean, .lnvar) {
    -0.5 * torch_sum(1. + .lnvar
                     - torch_pow(.mean, 2.)
                     - torch_exp(.lnvar),
                     dim = -1)
}
```

- We assume both q and p follows Gaussian distribution

A complete definition of our VAE model

```r
build.vae <-
    nn_module(
        classname = "variational autoencoder",
        initialize = function(d.data, K){
            self$enc <- build.vae.encoder(d.data, K)     # encoder model
            self$dec <- build.linear.decoder(d.data, K) # decoder model
        },
        forward = function(x){
            .enc <- self$enc(x)                          #
            x.hat <- self$dec(.enc$z)                    # reconstruction
            .llik <- multinom.llik(x, x.hat)             # data likelihood
            .kl <- kl.loss(.enc$z.mean, .enc$z.logvar)   # KL divergence
            .loss <- .kl - .llik                         # combined loss
            list(loss = .loss, kl=.kl)
        }
    )
```

- What do you want to change?

VAE: Check flow from the encoder to decoder

```
vae <- build.vae(ncol(x.torch), K=12)
vae$to(device = GPU)

x.input <- x.torch[1:5, ]
out <- vae(x.input)
out$loss
```

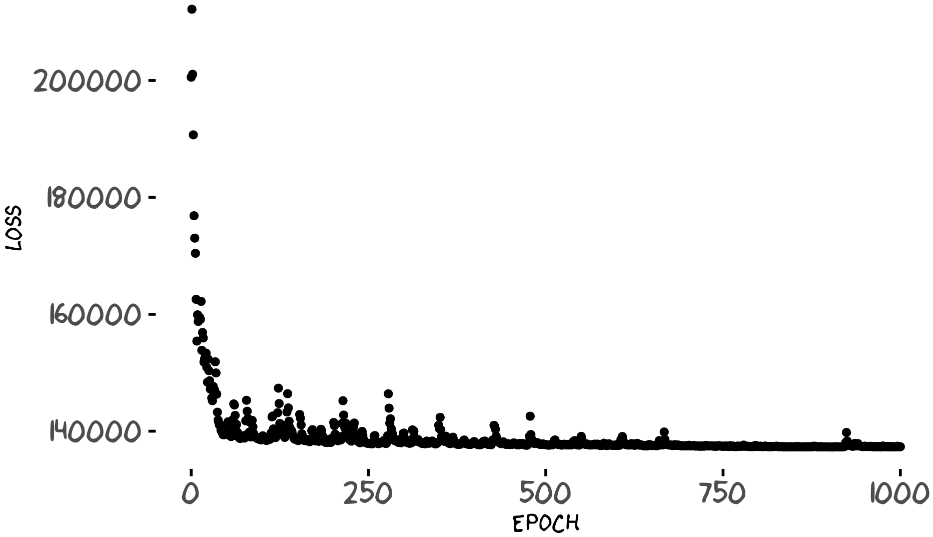
```
## torch_tensor
## 1e+05 *
##  2.3248
##  3.2764
##  1.7317
##  3.4227
##  1.6213
## [ CUDAFloatType{5} ][ grad_fn = <SubBackward0> ]
```

```
############################################
## reconstruction of x based on the latent ##
############################################
z.b <- vae$enc(x.input)
logx.recon <- vae$dec(z.b$z)
logx.recon[, 1:5]
```

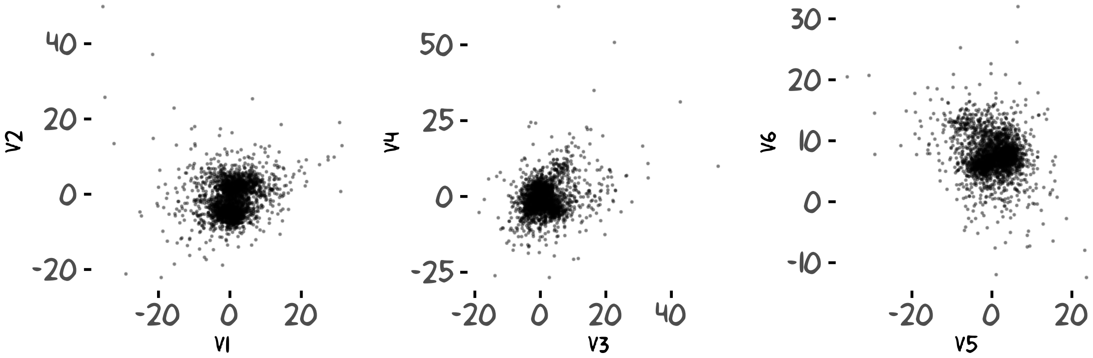
```
## torch_tensor
## -9.2107 -10.8502 -10.5020 -10.3259 -9.6075
## -11.4052 -10.0367 -10.2286 -9.9166 -8.6251
## -9.8550 -10.3307 -10.2863 -10.2946 -10.5433
## -10.7300 -10.5196 -10.3045 -10.0491 -9.5219
## -10.3972 -10.3624 -10.2260 -9.7453 -9.8102
## [ CUDAFloatType{5,5} ][ grad_fn = <SliceBackward0>
```

- Note: the reconstructed data matrix is in logarithm scale

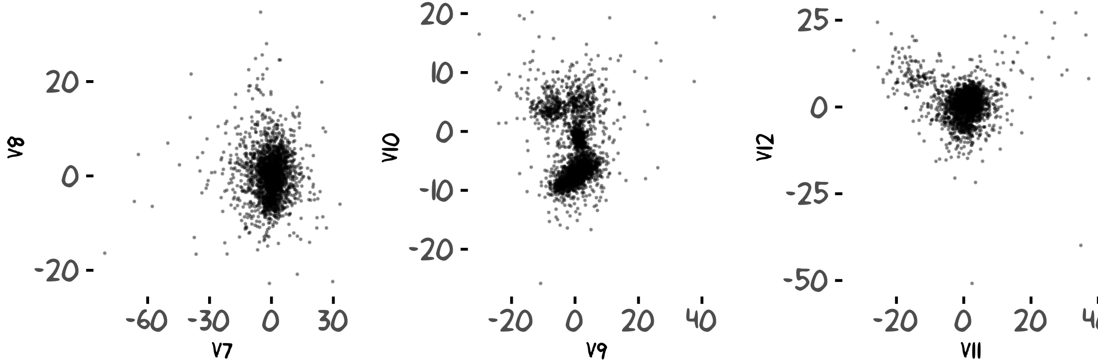
Results: SGD minimized the VAE loss function



Latent dimensions in the VAE model

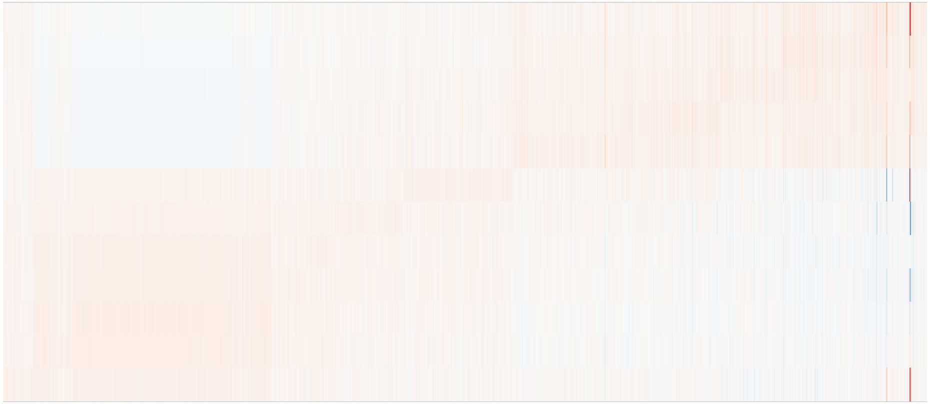


Latent dimensions in the VAE model



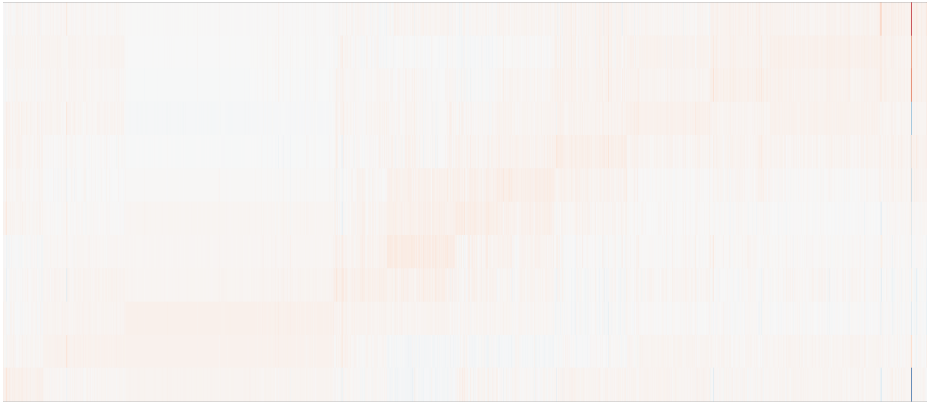
Hidden representations (factor $\times$ cell)

EPOCH = 1



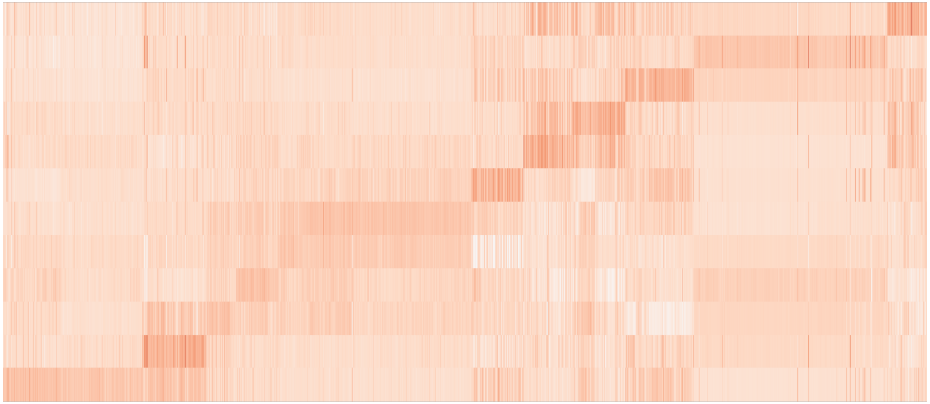
Hidden representations (factor $\times$ cell)

EPOCH = 26



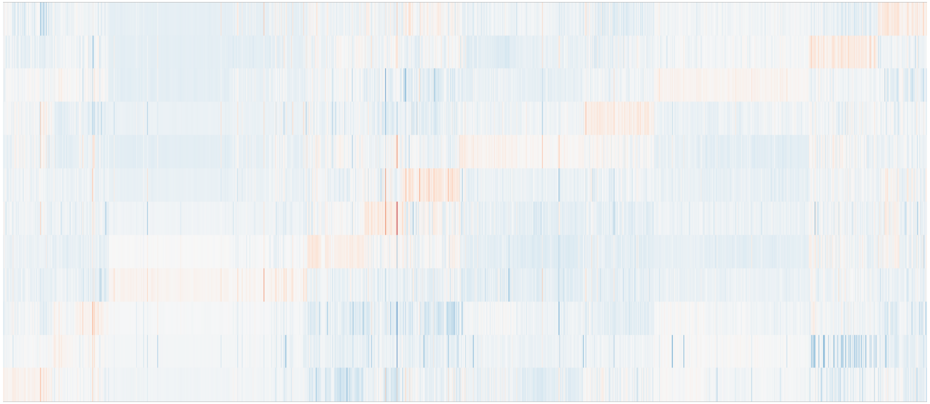
Hidden representations (factor $\times$ cell)

EPOCH = 226



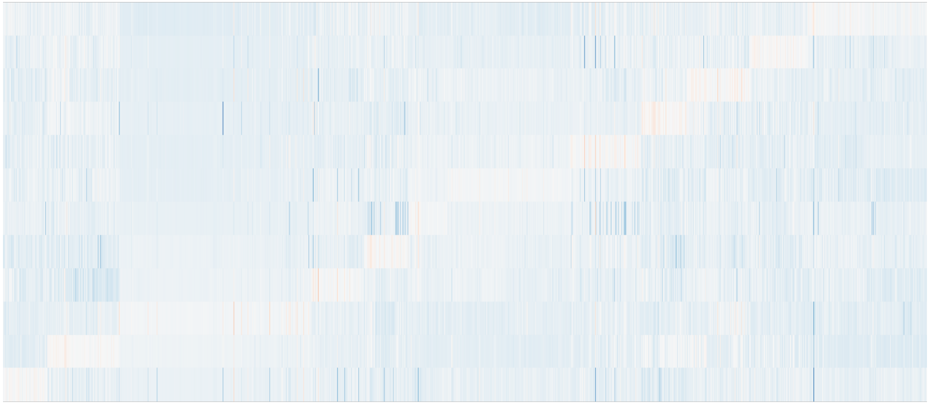
Hidden representations (factor $\times$ cell)

EPOCH = 476



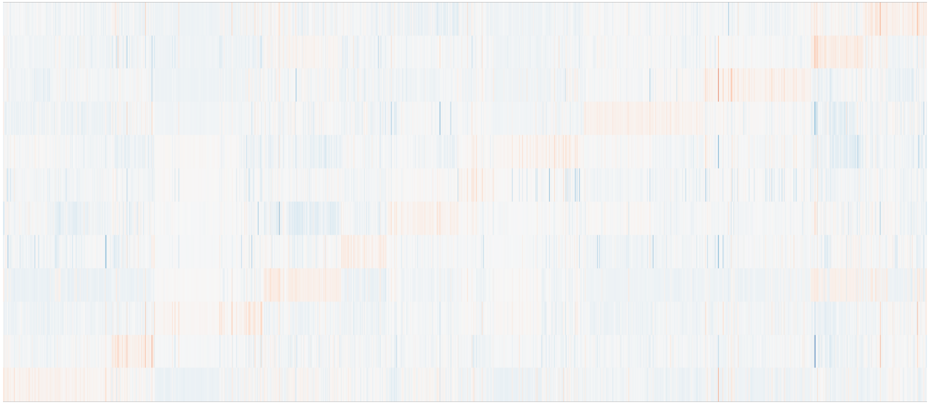
Hidden representations (factor $\times$ cell)

EPOCH = 726

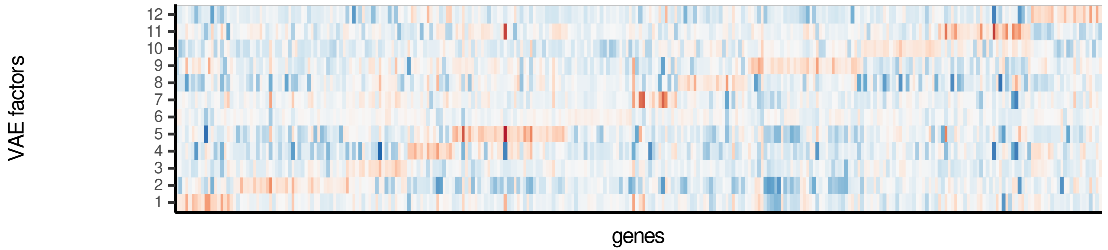


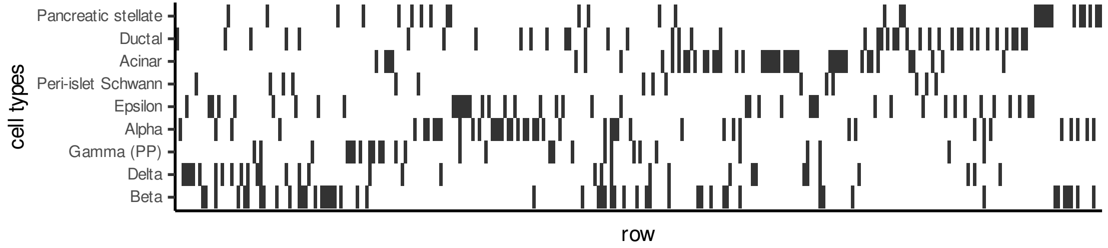
Hidden representations (factor $\times$ cell)

EPOCH = 976

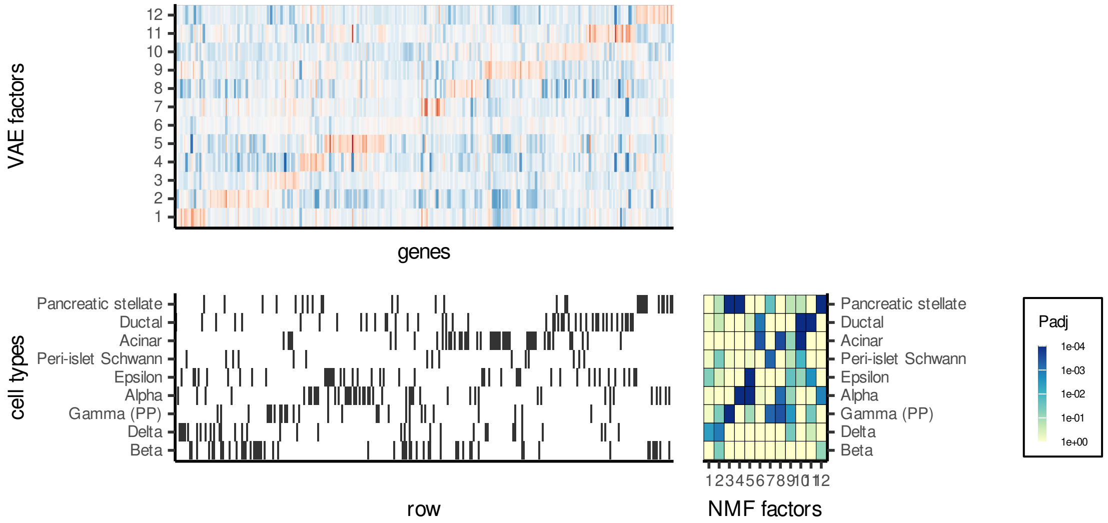


Annotate factors to cell types by enrichment (`fgsea`)





Annotate factors to cell types by enrichment (`fgsea`)



Can we improve model interpretation?

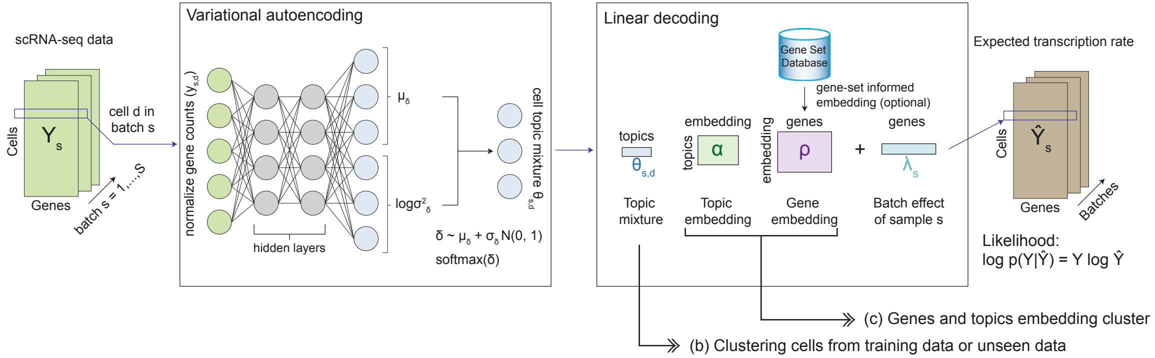
Can we improve model interpretation?

1. The decoder part is open to modelling in many different ways.

Can we improve model interpretation?

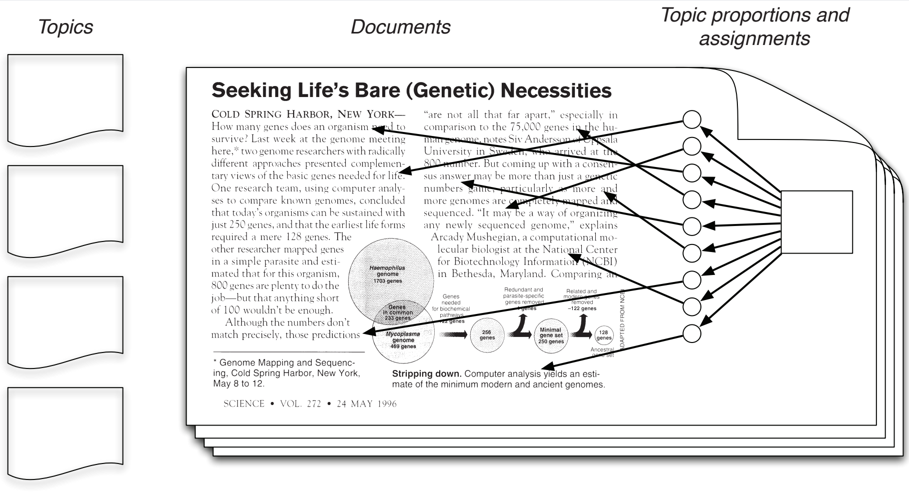
1. The decoder part is open to modelling in many different ways.
2. Can we define latent factors by gene expression frequencies?

Single-cell Embedded Topic Model



Zhao .. Li, Nature Comm. (2021)

Document topic modelling


Topics
Documents
Topic proportions and assignments
Haemophilus genome 1703 genes
Genes in common 233 genes
Mycoplasma genome 469 genes
Genes needed for biochemical pathways +22 genes
256 genes
Redundant and parasite-specific genes removed −4 genes
Minimal gene set 250 genes
Related and modern genes removed −122 genes
128 genes
Ancestral gene set
ADAPTED FROM NCBI

Seeking Life's Bare (Genetic) Necessities

COLD SPRING HARBOR, NEW YORK—How many genes does an organism need to survive? Last week at the genome meeting here,* two genome researchers with radically different approaches presented complementary views of the basic genes needed for life. One research team, using computer analyses to compare known genomes, concluded that today's organisms can be sustained with just 250 genes, and that the earliest life forms required a mere 128 genes. The other researcher mapped genes in a simple parasite and estimated that for this organism, 800 genes are plenty to do the job—but that anything short of 100 wouldn't be enough.

Although the numbers don't match precisely, those predictions "are not all that far apart," especially in comparison to the 75,000 genes in the human genome, notes Siv Andersson of Uppsala University in Sweden, who arrived at the 800 number. But coming up with a consensus answer may be more than just a genetic numbers game, particularly as more and more genomes are completely mapped and sequenced. "It may be a way of organizing any newly sequenced genome," explains Arcady Mushegian, a computational molecular biologist at the National Center for Biotechnology Information (NCBI) in Bethesda, Maryland. Comparing an

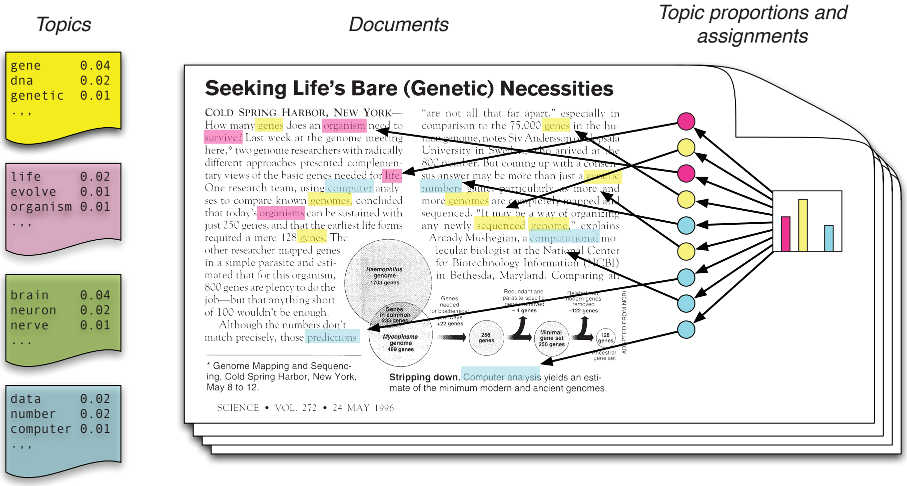
* Genome Mapping and Sequencing, Cold Spring Harbor, New York, May 8 to 12.

Stripping down. Computer analysis yields an estimate of the minimum modern and ancient genomes.

SCIENCE • VOL. 272 • 24 MAY 1996

Slide credit: David Blei

Word frequencies define topics in documents



gene	0.04
dna	0.02
genetic	0.01
,,,	

life	0.02
evolve	0.01
organism	0.01
,,,	

brain	0.04
neuron	0.02
nerve	0.01
...	

data	0.02
number	0.02
computer	0.01
,,,	

Seeking Life's Bare (Genetic) Necessities

COLD SPRING HARBOR, NEW YORK— How many genes does an organism need to survive? Last week at the genome meeting here,* two genome researchers with radically different approaches presented complementary views of the basic genes needed for life. One research team, using computer analyses to compare known genomes, concluded that today's organisms can be sustained with just 250 genes, and that the earliest life forms required a mere 128 genes. The other researcher mapped genes in a simple parasite and estimated that for this organism, 800 genes are plenty to do the job—but that anything short of 100 wouldn't be enough.

Although the numbers don't match precisely, those predictions "are not all that far apart," especially in comparison to the 75,000 genes in the human genome, notes Siv Andersson of Uppsala University in Sweden, who arrived at the 800 number. But coming up with a consensus answer may be more than just a genetic numbers game, particularly as more and more genomes are completely mapped and sequenced. "It may be a way of organizing any newly sequenced genome," explains Arcady Mushegian, a computational molecular biologist at the National Center for Biotechnology Information (NCBI) in Bethesda, Maryland. Comparing an

* Genome Mapping and Sequencing, Cold Spring Harbor, New York, May 8 to 12.

Haemophilus genome 1703 genes
Genes in common 233 genes
Mycoplasma genome 469 genes
Genes needed for biochemical pathways +22 genes
256 genes
Redundant and parasite-specific genes removed −4 genes
Minimal gene set 250 genes
Related and modern genes removed −122 genes
128 genes
Ancestral gene set
ADAPTED FROM NCBI

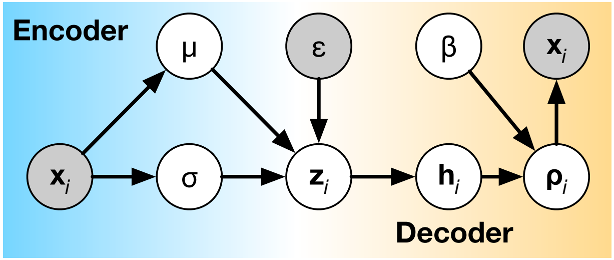
Stripping down. Computer analysis yields an estimate of the minimum modern and ancient genomes.

SCIENCE • VOL. 272 • 24 MAY 1996

Slide credit: David Blei

Multinomial topic model for scRNA-seq data

Can we simply model scRNA-seq counts by multinomial distribution?



- X_{ig}: gene expression of a gene g in a single cell i
- H_{ik}: latent topic proportion of a cell i to a topic k
- β_{kg}: topic k-specific gene probability

Multinomial topic model for scRNA-seq data

Can we simply model scRNA-seq counts by multinomial distribution?

Likelihood model:

$$\mathcal{L} = \prod_{i=1}^{n} \prod_{g=1}^{\text{genes}} \left(\sum_{k} H_{ik} \beta_{kg} \right)^{X_{ij}}$$

- X_{ig}: gene expression of a gene g in a single cell i
- H_{ik}: latent topic proportion of a cell i to a topic k
- β_{kg}: topic k-specific gene probability

Multinomial topic model for scRNA-seq data

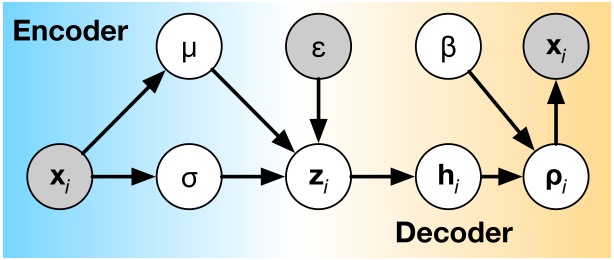
Can we simply model scRNA-seq counts by multinomial distribution?

Likelihood model:

$$\mathcal{L} = \prod_{i=1}^{n} \prod_{g=1}^{\text{genes}} \left(\underbrace{\sum_{k} H_{ik}\beta_{kg}}_{\text{a gene } g\text{'s probability in a cell } i \equiv \rho_{ig}} \right)^{X_{ij}}$$

- X_{ig}: gene expression of a gene g in a single cell i
- H_{ik}: latent topic proportion of a cell i to a topic k
- β_{kg}: topic k-specific gene probability

Topic modelling for single-cell data



Probability of gene g in a cell i:

$$\rho_{ig} = \sum_{k \in \text{topics}} H_{ik} \beta_{kg}$$

By **not** normalizing the probability of each cell, we do not worry about modelling sequencing depths.

Document topic modelling vs. single-cell ETM

variables	in document topic model	in single cell ETM
D	Total number of documents (corpus)	Total number of cells
d	Document index	Cell index
N_d	Number of words in a document d	Number of read counts in a cell d
j	Word index, $j \in [N_d]$	Read index
K	Total number of topics	Total number of cell type topics
k	Topic index, $k \in [K]$	Cell topic index
V	Size of vocabulary	Total number of genes
v	Vocabulary index $v \in [V]$	Gene index
W_{dj}^{v}	Indicator for a word to vocabulary $\in \{0, 1\}$	Indicator for a read to a gene $\in \{0, 1\}$
X_{dv}	Vocabulary v occurrence in a document d	Gene expression of a gene v in a cell $d \in [0, N_d]$

variables	in document topic model	in single cell ETM
Z_{dj}^{k}	Indicator for assigning a word to a topic k	Indicator for assigning a read to a topic k
H_{dk}	Hidden state k of a document d	Hidden state k of a cell d
β_{kv}	topic k-specific vocabulary v frequency	topic k-specific, a gene v's exression

- In Latent Dirichlet Allocation: $\sum_{k=1}^{K} H_{dk} = 1$ and $H_{dk} > 0$, and $\mathbf{h}_d \sim \text{Dirichlet}(\alpha/K, \ldots, \alpha/K)$ *a priori*. Approximately, we have $\hat{H}_{dk} = \sum_{j}^{N_d} Z_{dj}^{k}/N_d$.
- In Embedded Topic model, H_{dk} with the simplex constraints; $H_{dk} = \exp(\delta_{dk})/\sum_{k'} \exp(\delta_{dk'})$ where $\delta_{dk} \sim \mathcal{N}(0,1)$ *a priori*.
- Additional constraints: $\beta_{kv} > 0$ and $\sum_{v} \beta_{kv} = 1$, meaning that only a handful of vocabulary v contribute to a topic k.

Let’s modify the decoder part

```r
build.etm.decoder <-
    nn_module(classname = "ETM decoder",
        initialize = function(n.out, K, jitter = 1e-2) {
            self$lbeta <- nn_parameter(torch_randn(K, n.out) * jitter)
            self$beta <- nn_log_softmax(2) # topic x variant (softmax for each topic)
            self$hid <- nn_log_softmax(2)  # sample x topic (softmax for each sample)
        },
        ## Define how do get back high-dim data
        forward = function(z.b, eps = 1e-8){
            .beta <- self$get.weight()
            h.b <- self$hid(z.b)
            torch_log(torch_mm(torch_exp(h.b), torch_exp(.beta)) + eps)
        },
        ## Helper function
        get.weight = function(){
            self$beta(self$lbeta)
        })
```

ETM: putting the encoder and decoder together

```r
build.etm <-
    nn_module(
        classname = "embedded topic model",
        initialize = function(d.data, K){
            self$enc <- build.vae.encoder(d.data, K)
            self$dec <- build.etm.decoder(d.data, K)
        },
        forward = function(x){
            .enc <- self$enc(x)
            x.hat <- self$dec(.enc$z)
            .llik <- multinom.llik(x, x.hat)
            .kl <- kl.loss(.enc$z.mean, .enc$z.logvar)
            .loss <- .kl - .llik
            list(loss = .loss, kl = .kl, latent = .enc$z.mean)
        })
```

ETM: Check flow from the encoder to decoder

```r
etm <- build.etm(ncol(x.torch), K=12)
etm$to(device = GPU)

x.input <- x.torch[1:5, ]
out <- etm(x.input)

out <- vae(x.input)
out$loss
```

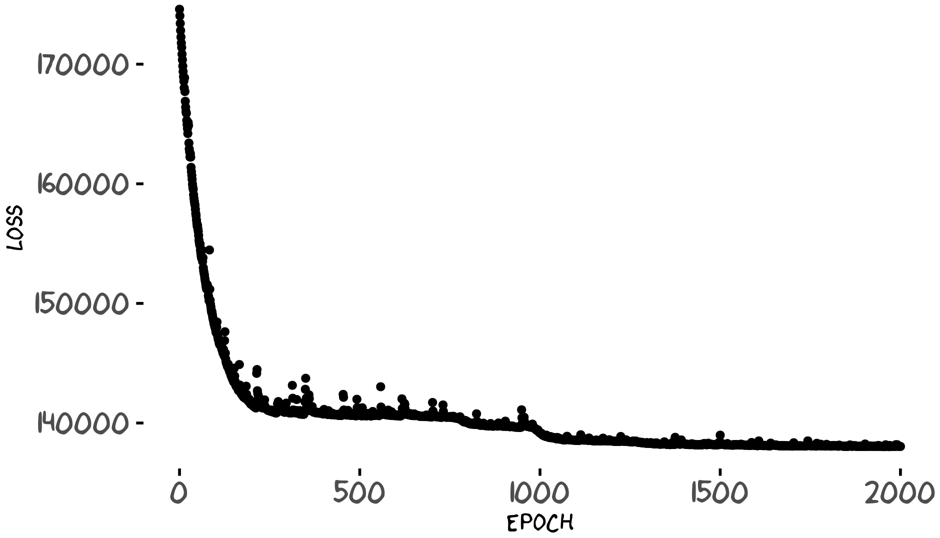
```
## torch_tensor
## 1e+05 *
##  2.3399
##  3.2311
##  1.7402
##  3.4814
##  1.6287
## [ CUDAFloatType{5} ][ grad_fn = <SubBackward0> ]
```

```r
###########################################
## reconstruction of x based on the latent ##
###########################################
z.b <- etm$enc(x.input)
logx.recon <- etm$dec(z.b$z)
logx.recon[, 1:5]
```

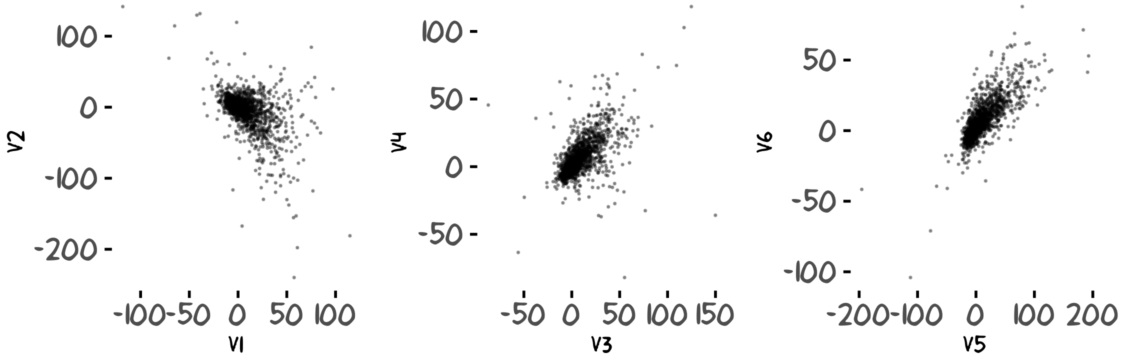
```
## torch_tensor
## -9.8607 -9.8591 -9.8561 -9.8585 -9.8618
## -9.8593 -9.8634 -9.8592 -9.8578 -9.8654
## -9.8550 -9.8600 -9.8584 -9.8569 -9.8566
## -9.8584 -9.8653 -9.8614 -9.8545 -9.8666
## -9.8580 -9.8616 -9.8597 -9.8564 -9.8608
## [ CUDAFloatType{5,5} ][ grad_fn = <SliceBackward0>
```

- Note: the reconstructed data matrix is in logarithm scale

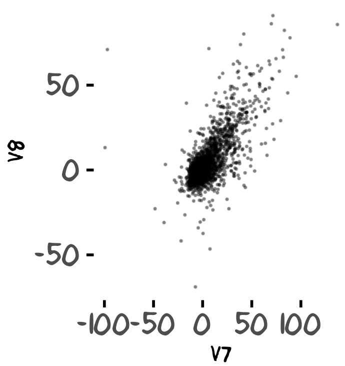
Results: SGD minimized the loss function

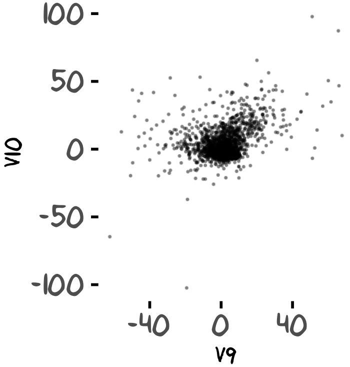


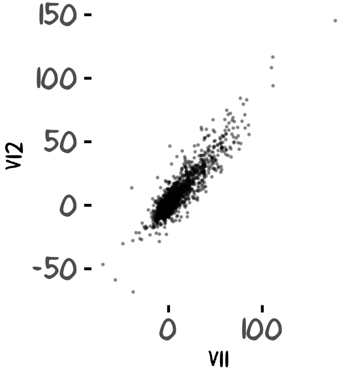
Latent dimensions in the ETM model



Latent dimensions in the ETM model





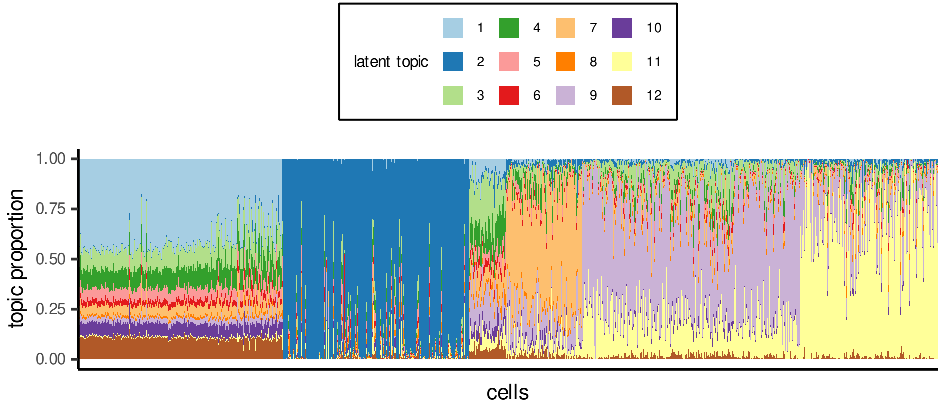


In ETM, the hidden dimensions are not independent

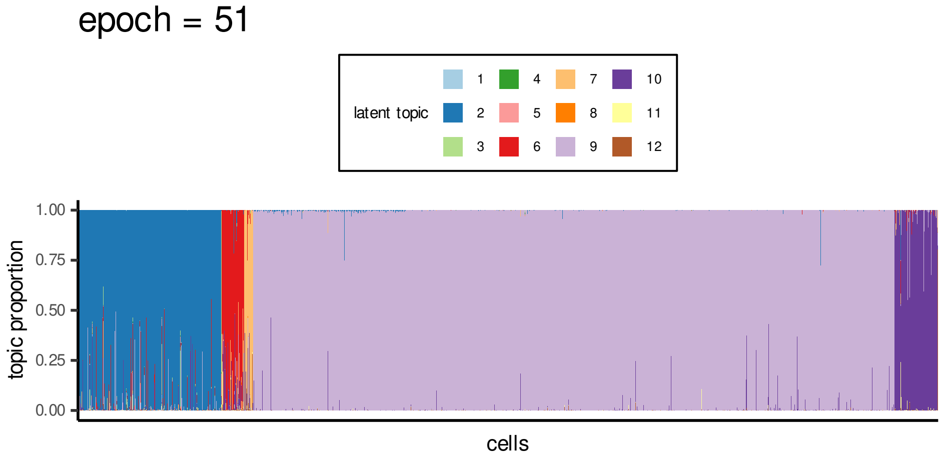
$$H_{ik} = \frac{\exp(Z_{ik})}{\sum_{k'} \exp(Z_{ik'})}$$

Hidden representations (mixing proportions)

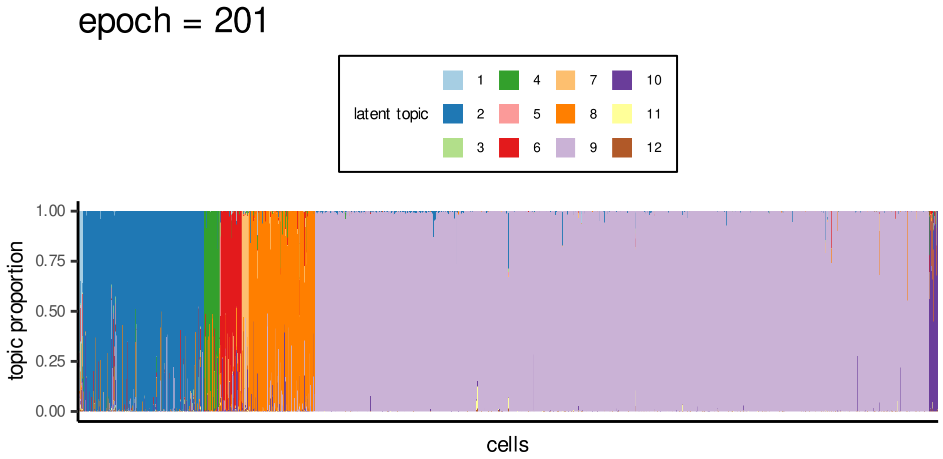
epoch = 1



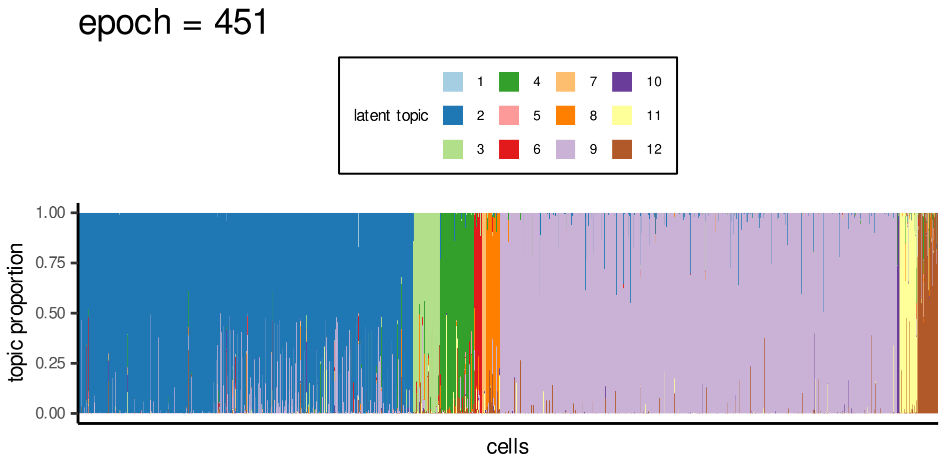
Hidden representations (mixing proportions)



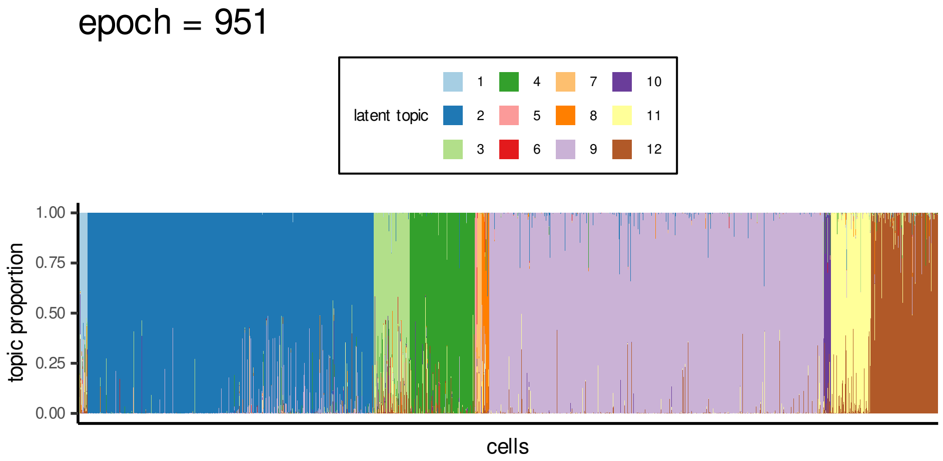
Hidden representations (mixing proportions)



Hidden representations (mixing proportions)

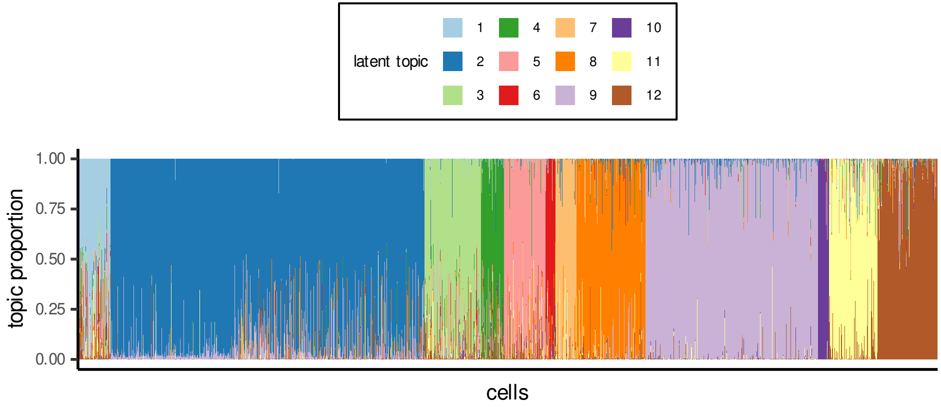


Hidden representations (mixing proportions)

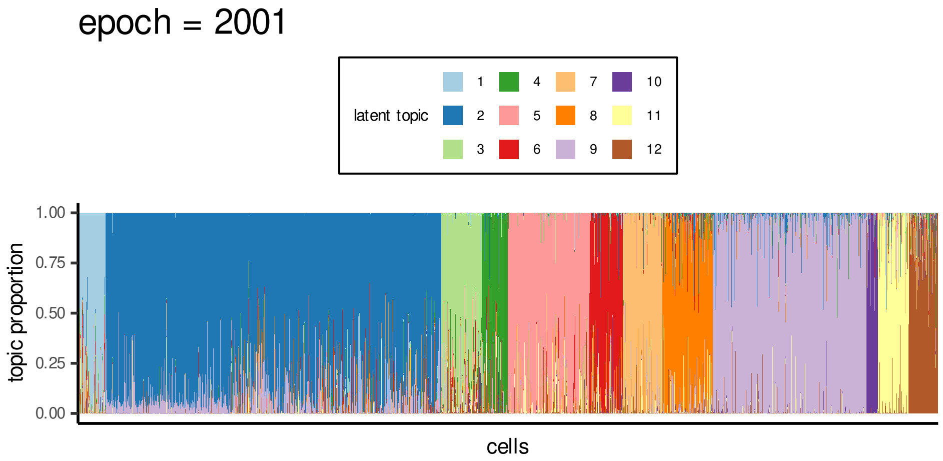


Hidden representations (mixing proportions)

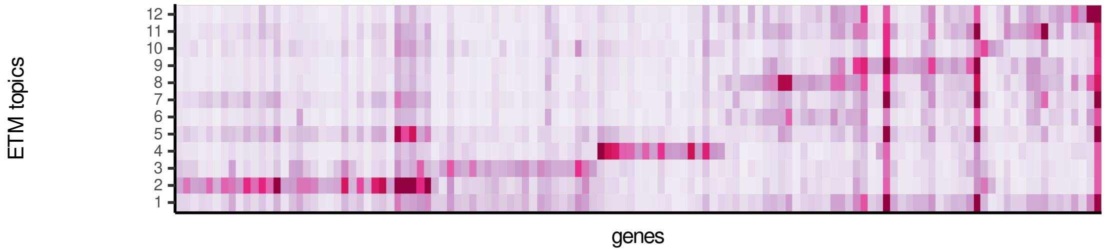
epoch = 1451

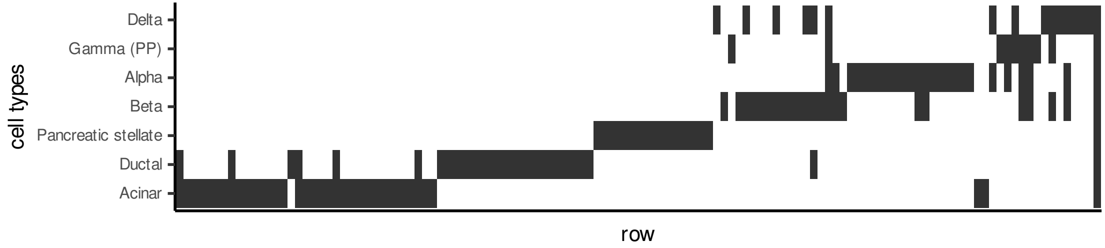


Hidden representations (mixing proportions)

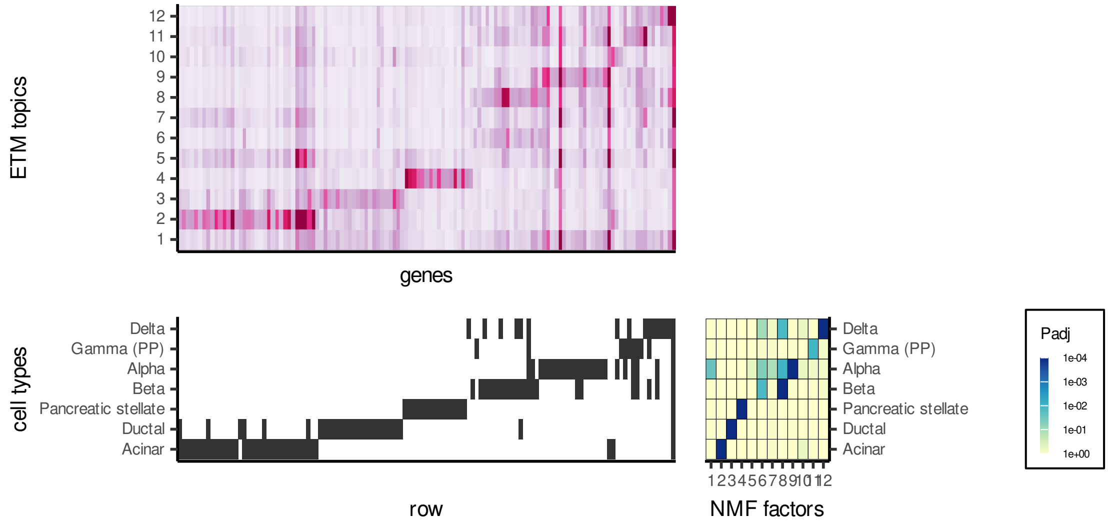


Annotate factors to cell types by enrichment (`fgsea`)





Annotate factors to cell types by enrichment (`fgsea`)



Discussions on latent topic modelling

- Most cells predominantly belong to one topic (one colour). Why?
- If we model cells as a mixture of cell topics, we can capture doublets or triplets.
- The underlying generative model assumes no sequencing depth! This can help avoid batch-specific differences in practice.
- VAE offers a flexible framework with which our scientific hypothesis can be formulated in a probabilistic language (`torch`).
- Potentially, this purely-unsupervised learning framework can be combined with supervised, semi-supervised learning models.