STA414/2104: Week 8

Continuous Latent Variable Models

Alex Stringer

March 6th, 2018

Recap

Last lecture we taked about *discrete* latent variable models.

We motivated the use of latent variables as a convenient way to introduce *structure* into our models.

We said there are some common types of structure we might want to induce:

- Observations are **dependent**
- The distribution of $\mathbf{x}$ lies close to some $d < p$-dimensional subspace of the p-dimensional feature space
- Not identically distributed; observations come from one of $K > 1$ *component distributions*

Recap

We approached the clustering scenario using a *discrete latent variable model*.

We assumed that in addition to observing $\mathbf{x}$, there was some unobserved discrete quantity $\mathbf{z}$ which described the *group membership* of $\mathbf{x}$.

We then put a distribution on $\mathbf{z}$, specified a conditional distribution for $\mathbf{x}|\mathbf{z}$, and proceeded to model both $(\mathbf{x}, \mathbf{z})$ and $\mathbf{z}|\mathbf{x}$.

We can try this approach in other situations as well.

Correlated Data

Suppose we observe a dataset $(t_n, \mathbf{x}_n), n = 1 \dots N$, and we are interested in building a linear model for $t_n|\mathbf{x}_n$:

$$t_n = \mathbf{x}_n' \mathbf{w} + \epsilon_n$$
$$\boldsymbol{\epsilon} = (\epsilon_1, \dots, \epsilon_N) \sim N(\mathbf{0}, \sigma^2 \mathbf{I}_p)$$

This implies that for $n \neq k$,

$$Cor(t_n, t_k) = Cor(\epsilon_n, \epsilon_k) = \left(\sigma^2 \mathbf{I}\right)_{nk} = 0$$

which in turn (due to the multivariate normality of $\boldsymbol{\epsilon}$), implies that $t_n \perp t_k$. Our model, as specified, assumes that observations are independent.

Important Note

We have discussed differences between the inferential, "classical" approach in statistics, and the prediction-focussed, machine learning approach, which often derive the same model.

The previous slide illustrates this conceptual difference. In classical linear model theory, we make an **explicit** assumption of independence between the observations, then code that into our model.

Here, we write down a sensible model, i.e. a sensible relationship between features and targets that incorporates randomness/variability. We then showed that what we wrote down **implies** independence between the observations.

Now, we can modfiy our model to relax this implicit assumption.

Correlated Data

How can we relax this? The most apparent way would be to change the model itself. We could write, for example,

$$t_n = \mathbf{x}_n' \mathbf{w} + \epsilon_n$$
$$\boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_N) \sim N(\mathbf{0}, \Sigma)$$

where Σ is not diagonal. Then $Cor(t_n, t_k)$ would not be 0, in general.

Correlated Data

Why not?

If we wanted to estimate Σ fully from the data, we would be out of luck, because we have N training observations but this matrix has $N(N+1)/2$ unique elements.

We could solve this by changing the model further, and assuming some structure on Σ (d-th order autoregressive, or something else).

Or, using regularization.

Another Way

However, we usually have some *actual reason* to believe the observations in our training data are correlated.

For example, our data could consist of monthly credit card balances for customers. Balances from the same customer will be correlated; balances from different customers (conditional on our features) will be independent.

Another Way

Or, our data might be colour images. For greyscale images, we code the image as being a big vector of pixel intensities. For a colour image, each pixel has three values (R, G, B). One way to code these is just as additional features (make the vector $3\times$ longer), but a potentially more powerful way is to give each pixel three rows in the dataset, then modify the model to account for the correlation that this induces- see Demidenko (2013) *Mixed Models, Theory and Applications*.

You could make Σ block-diagonal.

Or, work this directly into the model.

Random Effects Models

Partition $\mathbf{t} = (t_1, \ldots, t_N)$ into $(\mathbf{t}_1, \ldots, \mathbf{t}_K)$ according to which observations are correlated and which aren't (e.g. each $\mathbf{t}_k$ could be one customer's sequence of monthly credit card balances).

Define a linear model for t_{kt} as follows:

$$
\begin{aligned}
t_{kt} &= \mathbf{x}_{kt}'\mathbf{w} + b_k + \epsilon_{kt} \\
\mathbf{b} &= (b_1, \ldots, b_K) \sim N(\mathbf{0}, \sigma_b^2 \mathbf{I}) \\
\boldsymbol{\epsilon} &= (\epsilon_{11}, \ldots, \epsilon_{KT}) \sim N(\mathbf{0}, \sigma_\epsilon^2 \mathbf{I}) \\
\mathbf{b} &\perp \boldsymbol{\epsilon}
\end{aligned}
$$

Inducing Correlation

We have done the same thing we did before: added a latent variable ($\mathbf{b}$) into our model and given it a distribution. We also indirectly specified the conditional distribution of $\mathbf{t}|\mathbf{b}$:

$$\mathbf{t}_k|\mathbf{b} \sim N(\mathbf{X}_k'\mathbf{w} + \mathbf{b}_k, \Sigma_k)$$

where Σ_k is not diagonal. We have *induced correlation* between the elements of $\mathbf{t}_k$. More explicitly,

$$Cov(t_{ki}, t_{kj}) = \sigma_b^2$$

Inducing Correlation

Why is this any different than just specifying the correlation structure between datapoints directly? There are several key advantages:

- The model is *interpretable*. We interpret b_k as the baseline measurement of any observation belonging to $\mathbf{t}_k$. For example, it is a customer's global average credit card balance, or the average intensity of a pixel (across the three colours)

Inducing Correlation

- This modelling procedure is readily *extendable*. For example, what if we thought that in addition to customer's balances being correlated, balances from the same *month* across customers were also correlated? Using latent variables, we just add an a_t term to the model:

$$\mathbf{t}_{kt} = \mathbf{x}'_{kt}\mathbf{w} + b_k + a_t + \epsilon_{kt}$$

  and everything works out the same.
- Or perhaps you think customers' balances follow a seasonal pattern?

$$t_{kt} = \mathbf{x}'_{kt}\mathbf{w} + b_k + a_t + q_k \sin(2\pi t) + \epsilon_{kt}$$

Dimension Reduction

We can apply this same reasoning to the problem of *dimensionality reduction*.

Suppose we observe our usual dataset, but our features are very high-dimensional.

For example, images, where each feature is a pixel intensity (greyscale) or even a value of R/G/B (colour), giving three features per pixel.

For MNIST, in which the images contain only hand-written digits, 28×28 pixel greyscale images (small) correspond to a 784-dimensional feature space (big).

Dimension Reduction

It doesn't seem reasonable to suggest that 28×28 pixel greyscale images contain 784 unique pieces of information.

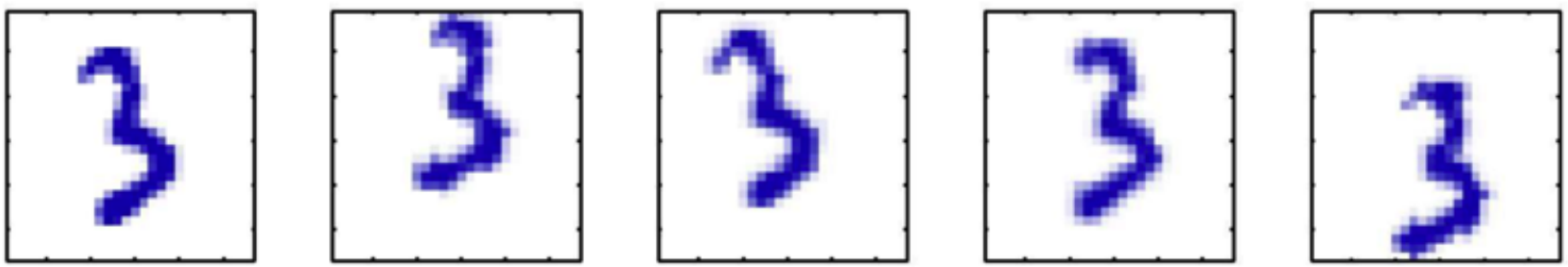


Figure 1: MNIST 3's, Bishop (2006)

Dimension Reduction

We can formulate this notion using our language of modelling as follows: we assume that each training case was generated by randomly selecting a point in the *latent space*, and then generating a point from the conditional distribution in the *input space*.

The latent space we invent will be simpler (lower dimension) than the input space.

Dimension Reduction

This is the idea behind *Factor Analysis*:

$$\begin{aligned}
\mathbf{x} &= \boldsymbol{\mu} + \boldsymbol{\Lambda}\mathbf{u} + \boldsymbol{\epsilon} \\
\boldsymbol{\Lambda} &\in \mathbb{R}^{p \times q} \\
\mathbf{u} &\sim N(\mathbf{0}, \mathbf{I}_q) \\
\boldsymbol{\epsilon} &\sim N(\mathbf{0}, \sigma^2 \mathbf{I}_p) \\
\mathbf{x}|\mathbf{u} &\sim N(\boldsymbol{\mu}, \boldsymbol{\Lambda}'\boldsymbol{\Lambda} + \sigma^2 \mathbf{I}_p)
\end{aligned}$$

Although it's not worth talking about in detail, since the rotational invariance of the multivariate Gaussian distribution renders the model unidentifiable.

Still, it's an interesting idea...

Principal Components Analysis

. . . and it's the same motivating idea behind the extremely popular **Principal Components Analysis**.

Principal Components Analysis tries to find a low-dimensional factorization of the data matrix that preserves the maximum possible variance:

$$\mathbf{X} = \mathbf{U}\mathbf{Z}$$

$$\mathbf{X} \in \mathbb{R}^{n \times p}$$

$$\mathbf{Z} \in \mathbb{R}^{d \times p}$$

$$\mathbf{U} \in \mathbb{R}^{n \times d}$$

$$d < p$$

There is an equivalent probabilistic formulation that directly uses latent variables, but we won't cover it here.

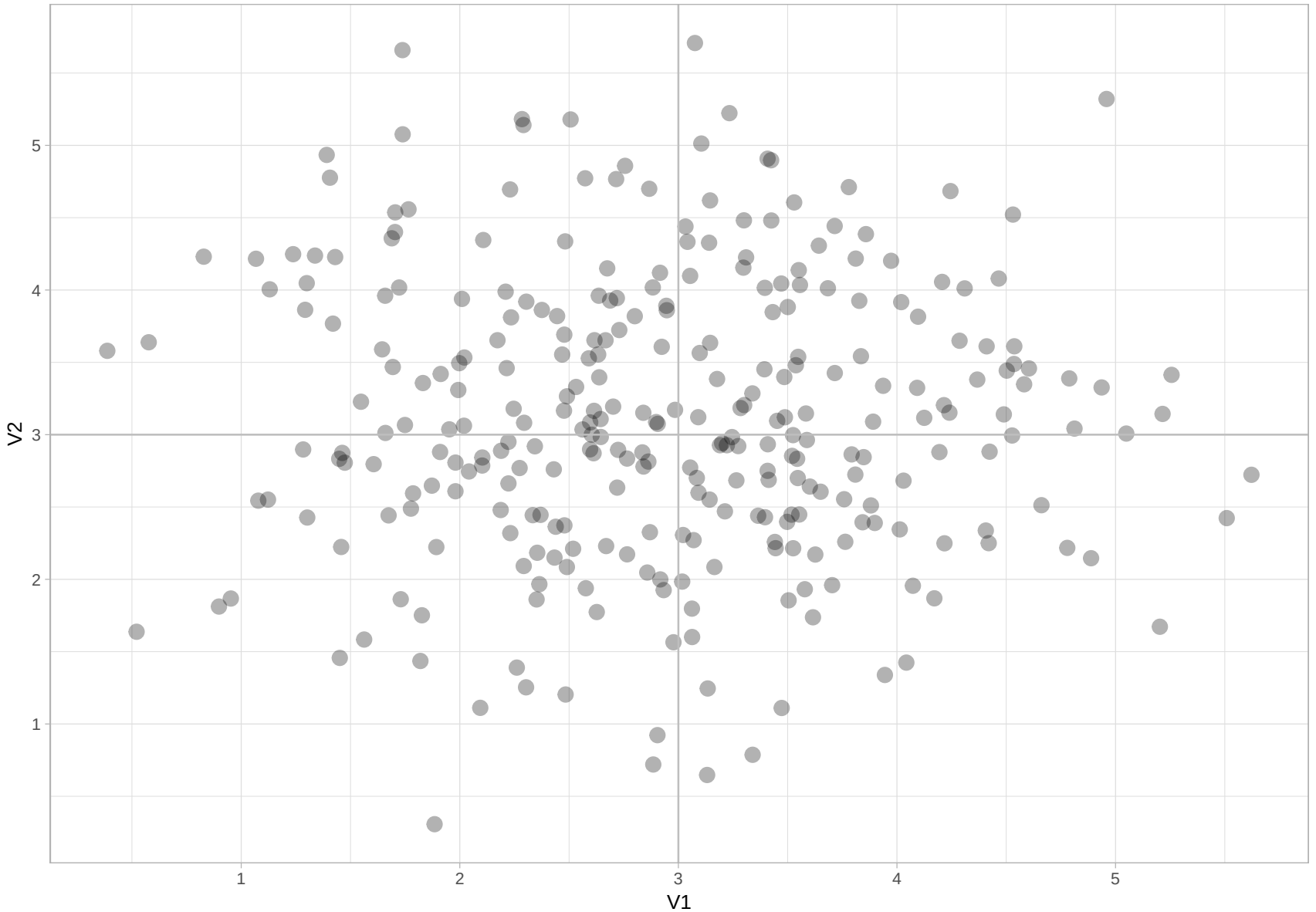
Principal Components Analysis

We estimate $\mathbf{U}$ from the data, and call the associated $\mathbf{Z}$ the *principal components* of $\mathbf{X}$.

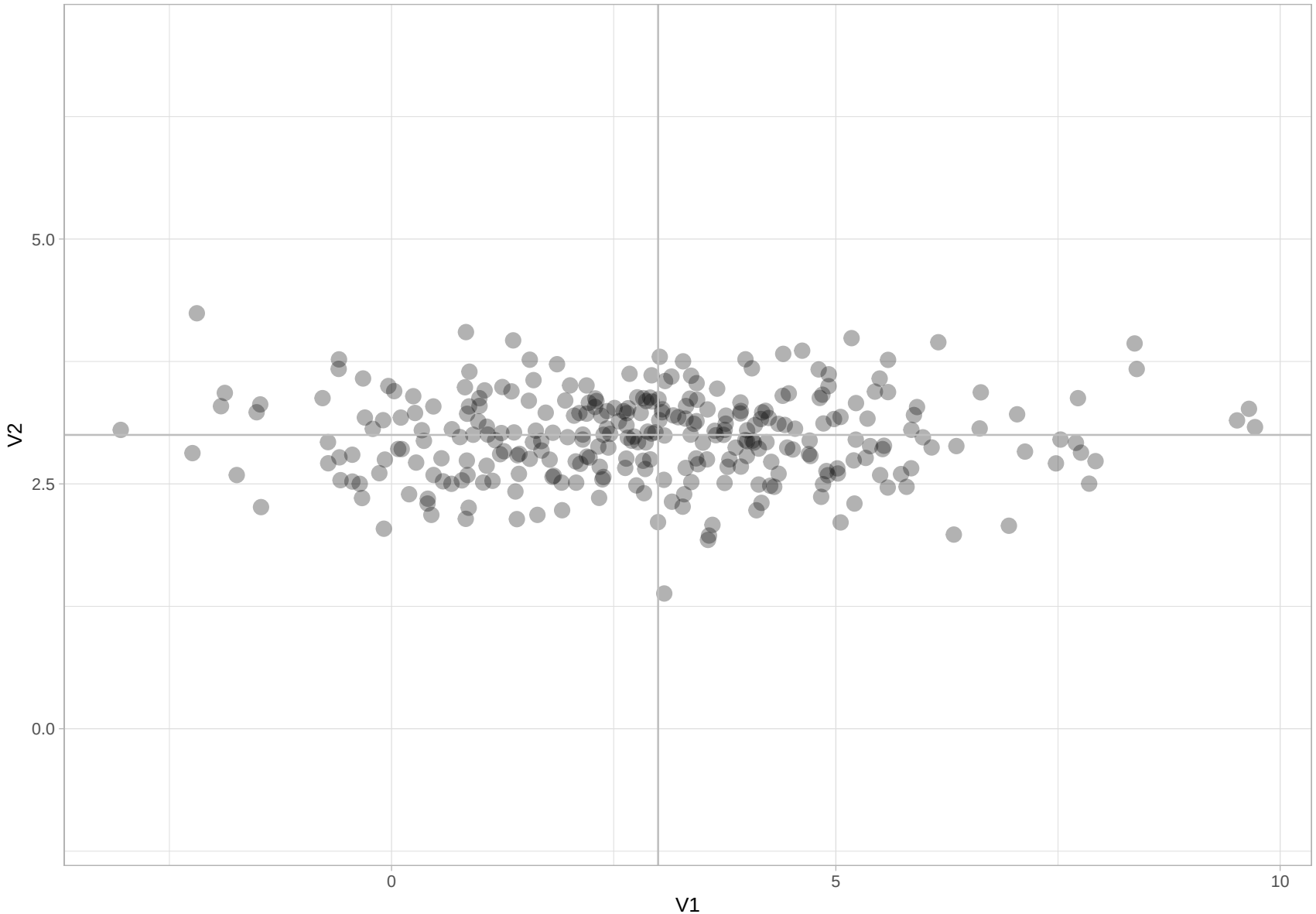
These represent the intristic latent features in our data.

Variance is analagous to information.

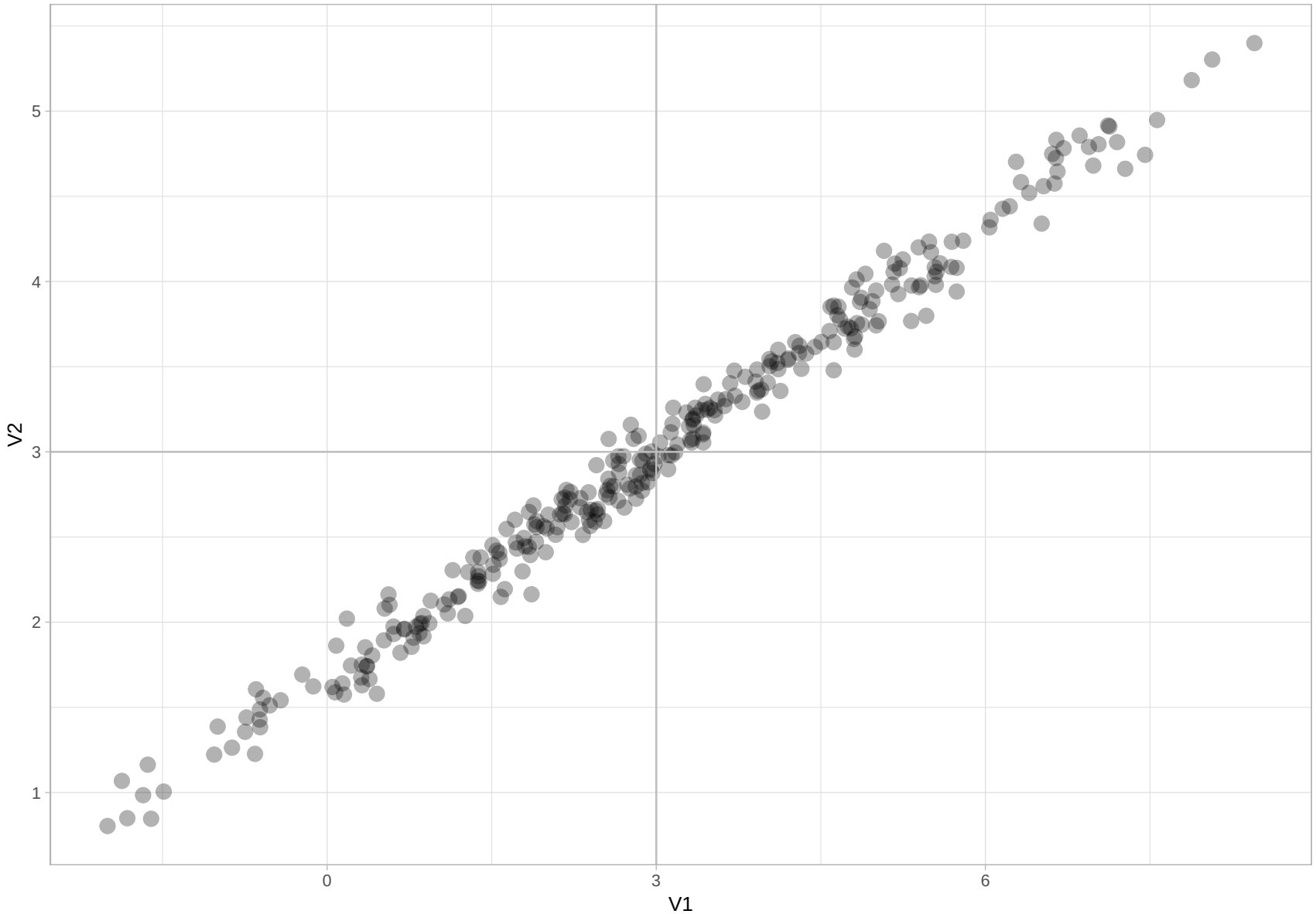
Variance is Information



Variance is Information



Variance is Information



Variance is Information

Directions along which $\mathbf{x}$ doesn't vary don't provide as much information about the distribution of $\mathbf{x}$.

In the digits data, look at all that whitespace in the corners. The presence of whitespace in the corners tells us nothing about the pixel intensities in the centre of the image, where the digit is. All of the images have whitespace in the corners- $\mathbf{x}$ doesn't *vary* in these dimensions.

However, even if $\mathbf{x}$ varies in all dimensions, it may be the case that there still aren't p distinct pieces of information.

If we rotated the distribution of $\mathbf{x}$, somehow, we could isolate only those directions that provide information.

Principal Components Analysis

PCA is thus stated as follows: for $j = 1 \dots d$,

$$\mathbf{u}_j = \text{argmax } Var(\mathbf{u}'\mathbf{x}) = \text{argmax } \mathbf{u}'\mathbf{S}\mathbf{u}$$
$$\text{subject to } \mathbf{u}'\mathbf{u} = 1$$
$$\text{and } \mathbf{u} \perp \mathbf{u}_k \text{ for } k < j$$

where

$$\mathbf{S} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})'$$

is the sample covariance matrix.

Principal Components Analysis

Solving this optimization problem gives us an orthonormal basis where the basis vectors point in the directions of the principal axes of the sample covariance matrix, in decreasing order of length.

It's a rotation of the original input space.

We then chop off the $p - d$ smallest directions, and call this the basis for our d-dimensional latent space.

Principal Components Analysis

Using lagrange multipliers, we see the solution to the above problem must satisfy

$$\mathbf{S}\mathbf{u}_1 = \lambda \mathbf{u}_1$$

which means that $\mathbf{u}_1$ is an eigenvector of $\mathbf{S}$. Further, since

$$Var(\mathbf{u}_1'\mathbf{x}) = \mathbf{u}_1'\mathbf{S}\mathbf{u}_1 = \lambda$$

we see that $Var(\mathbf{u}_1'\mathbf{x}) = \lambda$ is an eigenvalue of $\mathbf{S}$. By definition of the problem, it must be the largest eigenvalue (why?).

Principal Components Analysis

So, the solution to the principal components analysis problem is:

- Choose $\mathbf{u}_j$ to be the normalized eigenvector of $\mathbf{S}$ corresponding to the j-th highest eigenvalue
- Choose $\mathbf{U}$ to be the matrix of orthonormal eigenvectors of $\mathbf{S}$, so that $\mathbf{U}'\mathbf{U} = \mathbf{I}$
- Then $\mathbf{Z} = \mathbf{X}\mathbf{U}'$
- Keep only the first d columns of $\mathbf{Z}$ and corresponding $d \times d$ submatrix of $\mathbf{U}$
- Reconstruct the data as $\mathbf{X}^* = \mathbf{Z}^*\mathbf{U}^*$

Eigenanalysis of Previous Example

Compute the eigenvalues and vectors of the covariance matrices for the datasets in the three plots shown before.

Dataset 1:

```
## eigen() decomposition
## $values
## [1] 1.068101 1.022009
##
## $vectors
##             [,1]        [,2]
## [1,] -0.9100574  0.4144823
## [2,] -0.4144823 -0.9100574
```

Eigenanalysis of Previous Example

Compute the eigenvalues and vectors of the covariance matrices for the datasets in the three plots shown before.

Dataset 2:

```
## eigen() decomposition
## $values
## [1] 4.399837 0.213023
##
## $vectors
##              [,1]        [,2]
## [1,] -0.99987302 -0.01593539
## [2,]  0.01593539 -0.99987302
```

Eigenanalysis of Previous Example

Compute the eigenvalues and vectors of the covariance matrices for the datasets in the three plots shown before.

Dataset 3:

```
## eigen() decomposition
## $values
## [1] 5.64323316 0.01201601
##
## $vectors
##             [,1]       [,2]
## [1,] -0.9145276  0.4045236
## [2,] -0.4045236 -0.9145276
```

How Many Components?

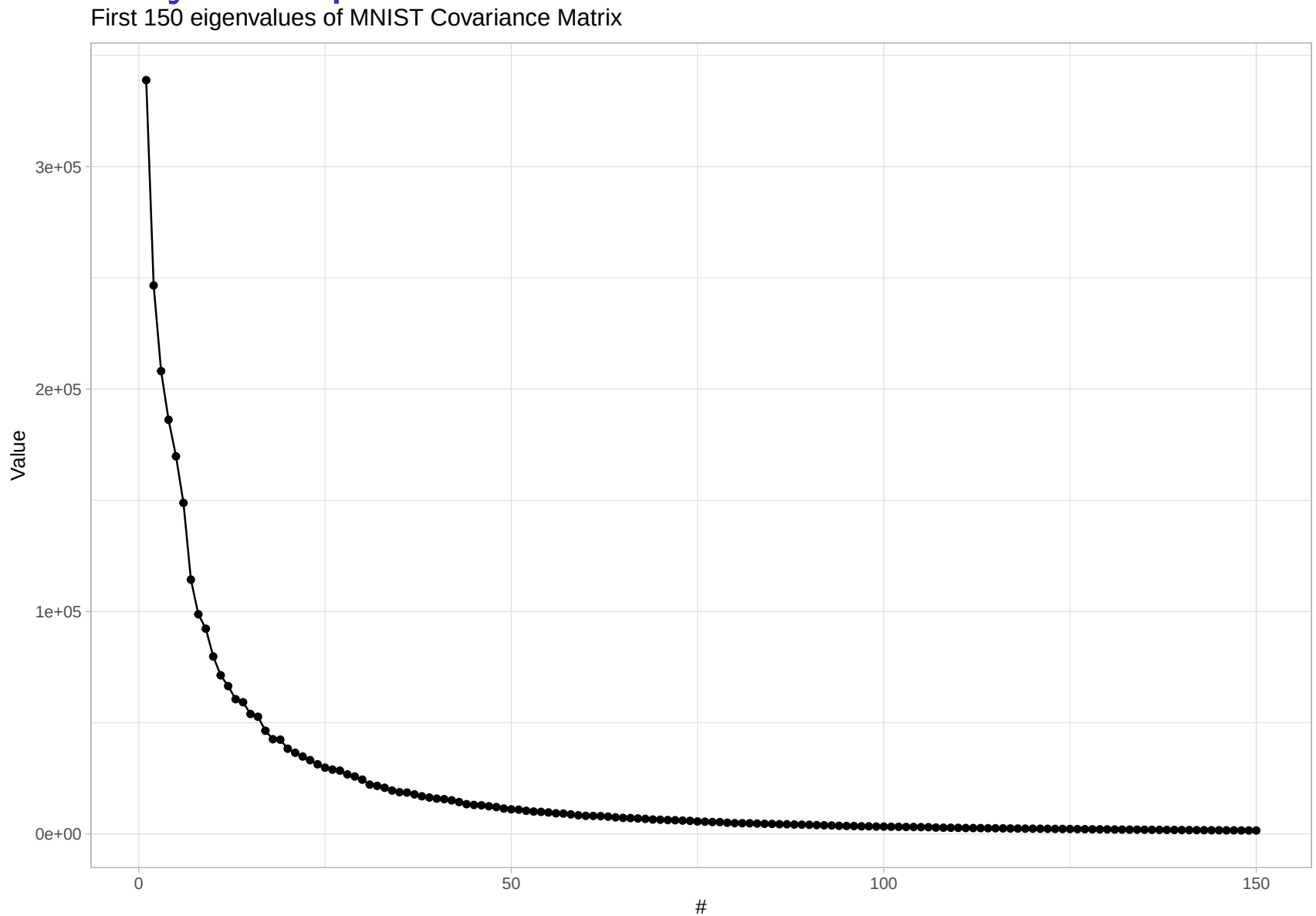
How many components to keep? Two approaches:

- Plot the eigenvalues in decreasing order. Pick the number where the plot "levels off"
- Pick the number of components that preserve some predetermined proportion of variance, i.e. choose the smallest d so that

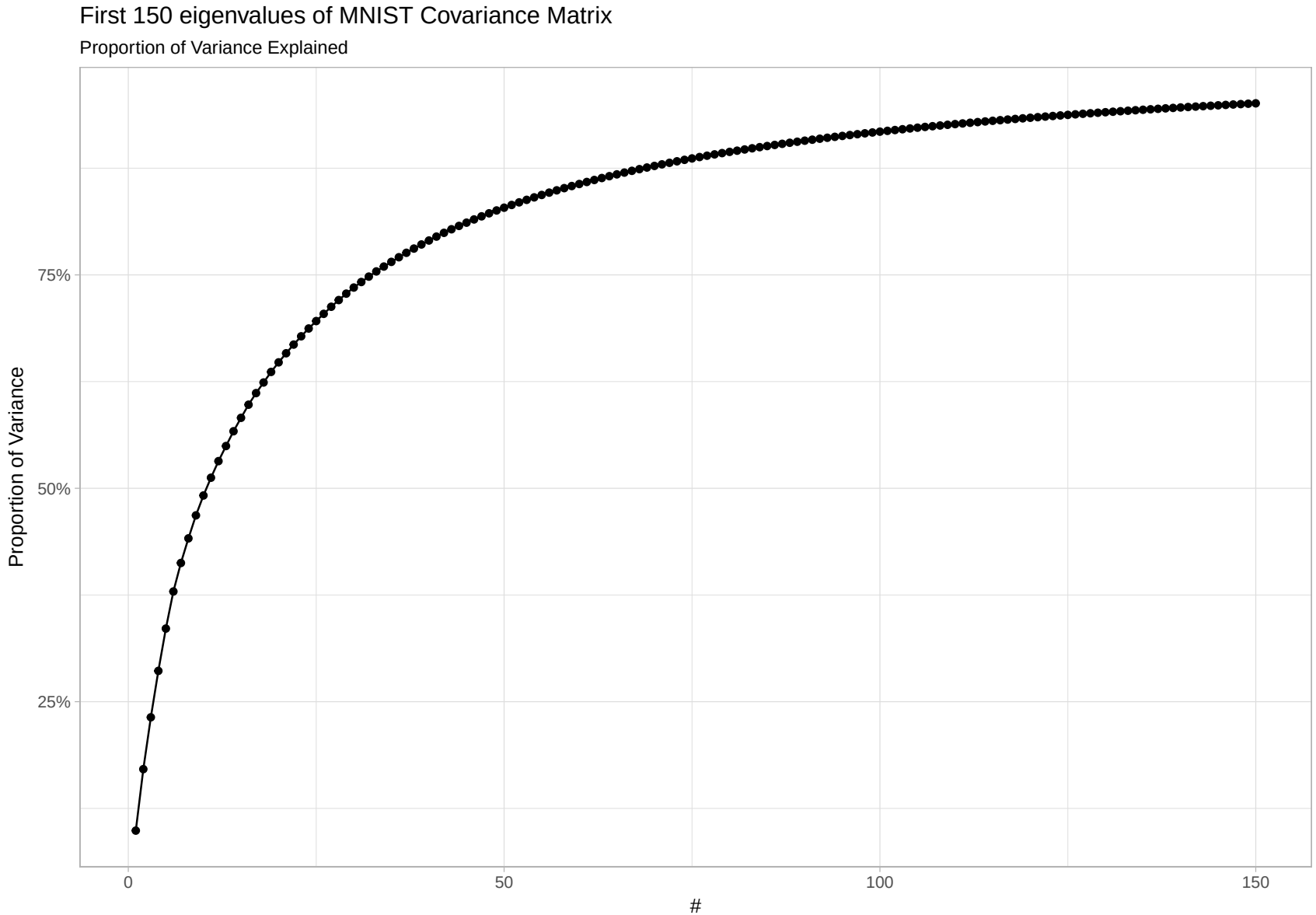
$$\frac{\sum_{i=1}^{d} \lambda_i}{tr(\mathbf{S})} > \gamma$$

for some $0 < \gamma < 1$

How Many Components



How Many Components

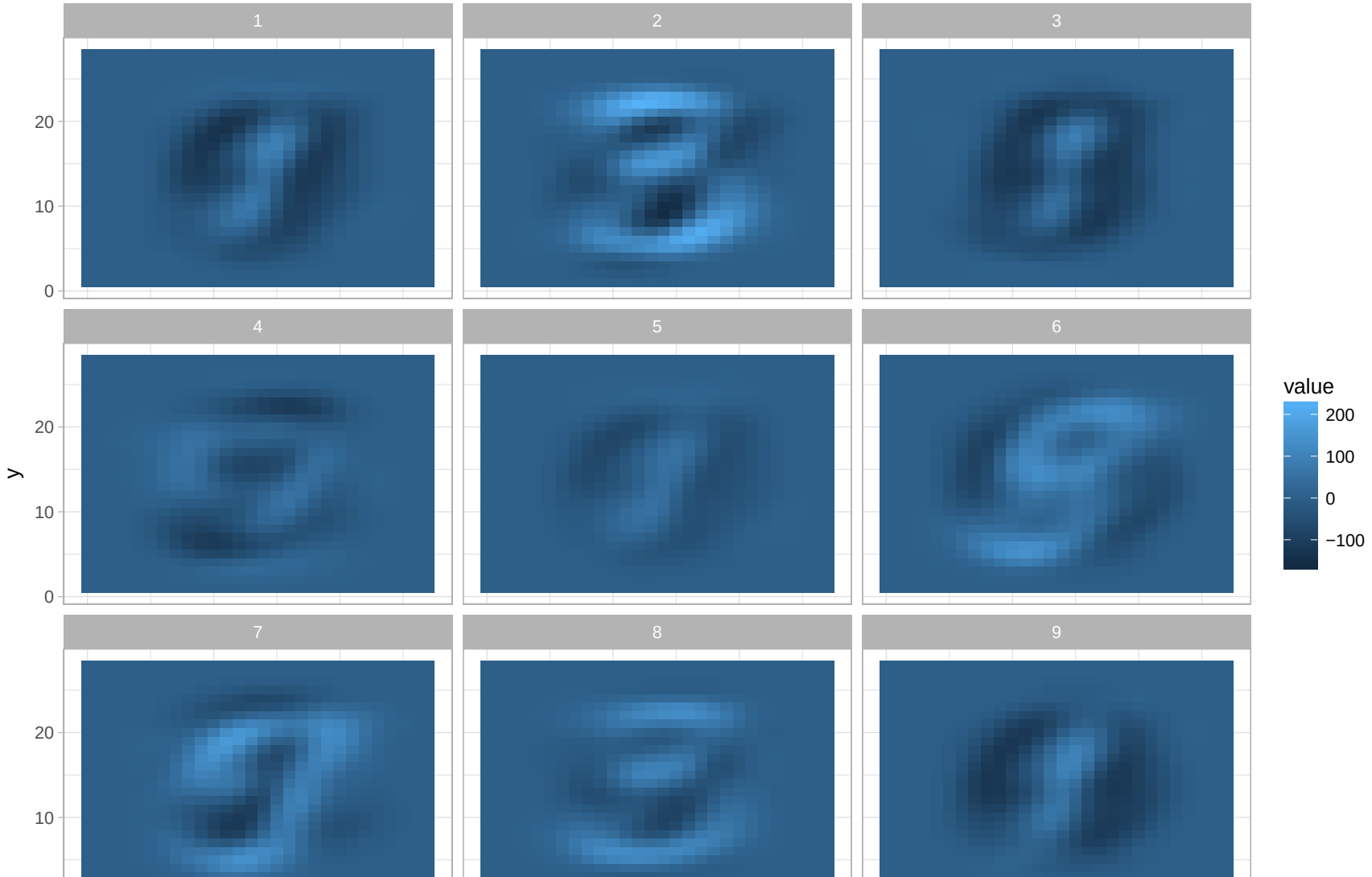


How Many Components

Original MNIST:

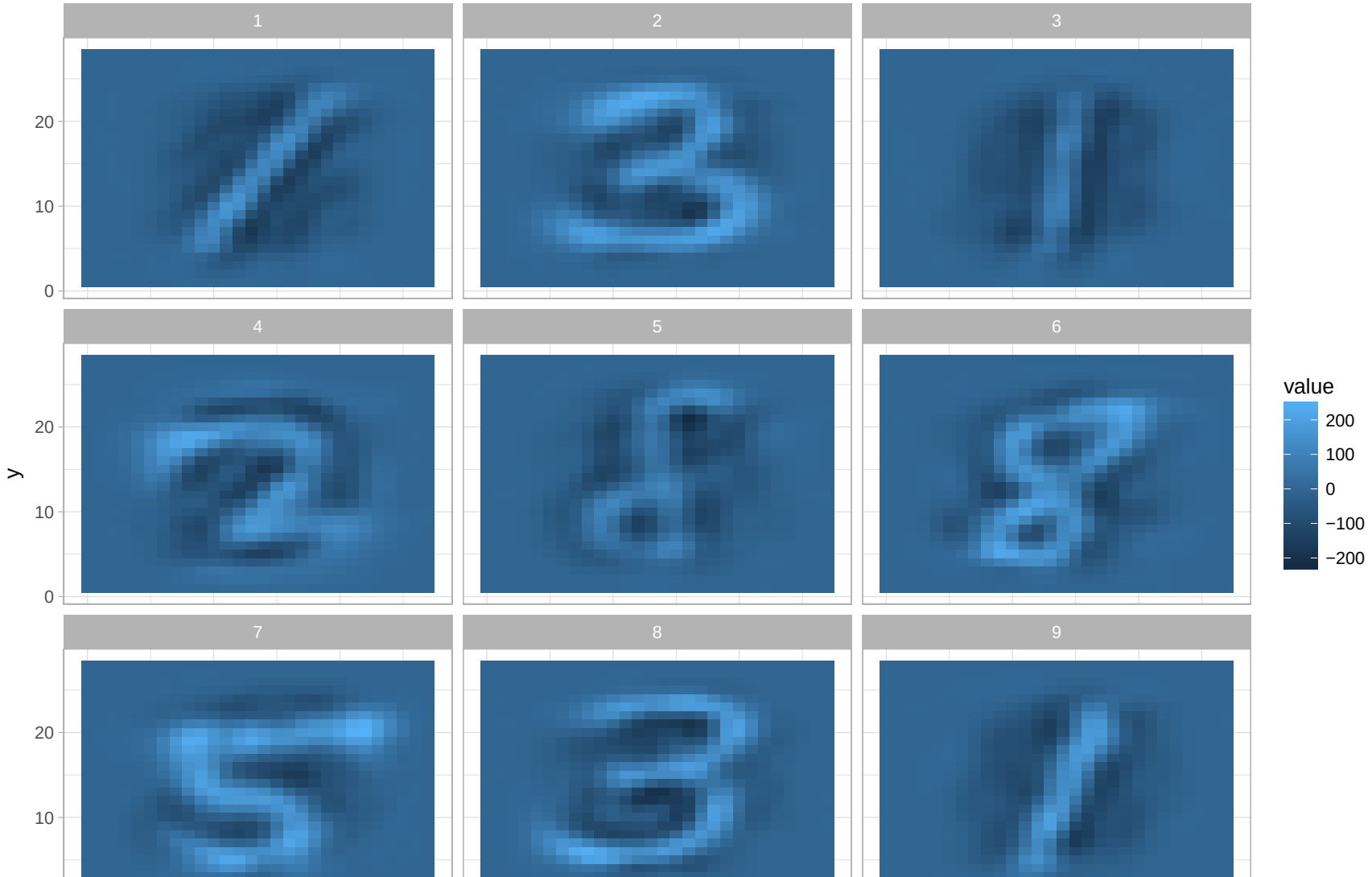
How Many Components

Reconstruction using 5 PCs:



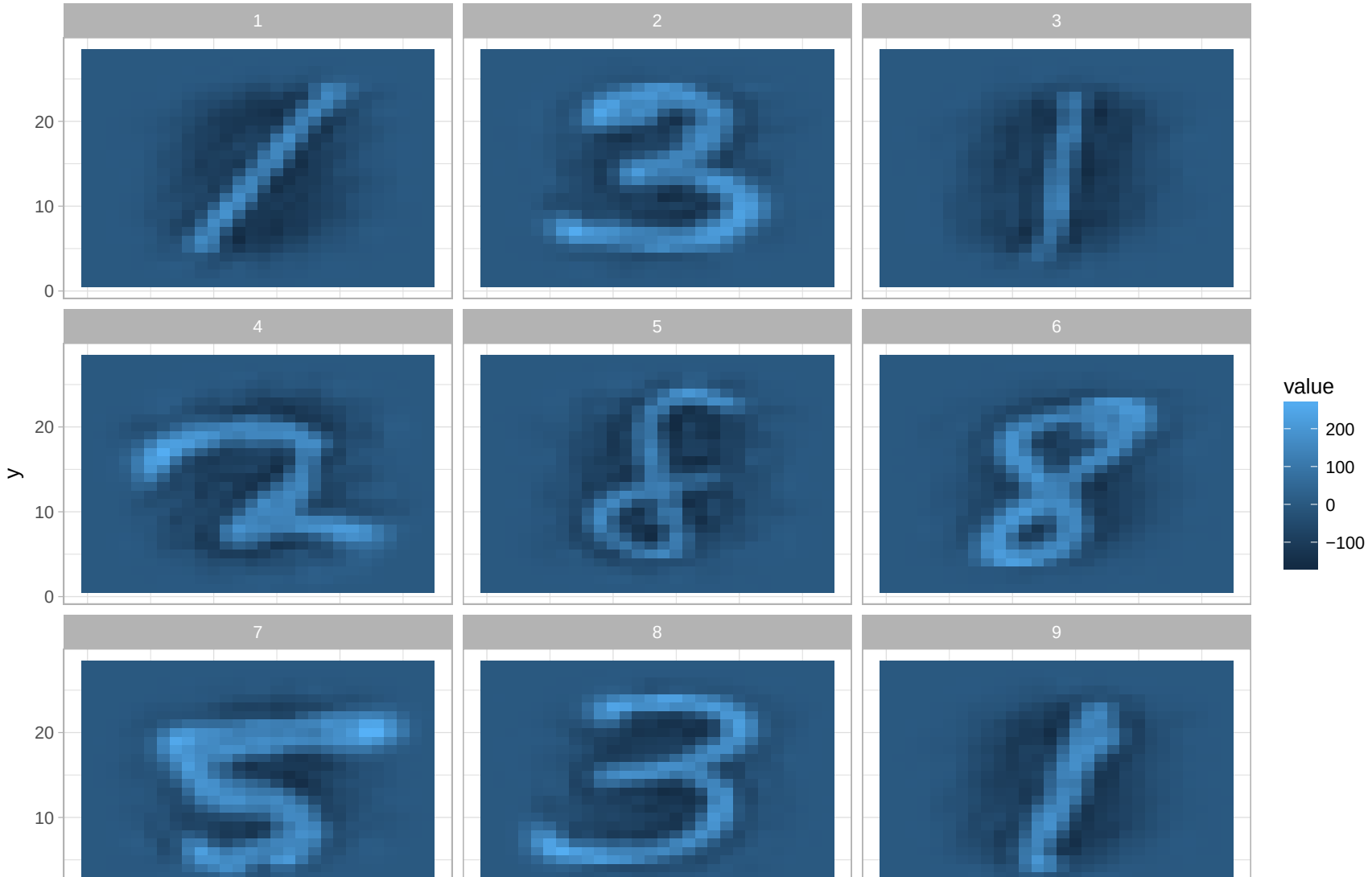
How Many Components

Reconstruction using 50 PCs:



How Many Components

Reconstruction using 150 PCs:



Latent Variable Models

Let's look at one more application of latent variables. We'll use them to make a complex model interpretable.

Back to the supervised case: we have a dataset $(t_n, \mathbf{x}_n), n = 1 \dots N$.

Suppose we model our functional relationship between features and parameters as $\mathbf{x}_n' \mathbf{w}$.

Then we add a nonlinear wrapper around this, and say $y(\mathbf{x}_n, \mathbf{w}) = g(\mathbf{x}_n' \mathbf{w})$.

Latent Variable Models

This gives us a flexible model that we have seen before, e.g. logistic regression with

$$g(x) = \frac{1}{1 + \exp(-x)}$$

What if this model is too simple? What if there are additional nonlinear interactions between elements of $\mathbf{x}$ that we want to capture?

Latent Variable Models

We could define a new vector $\mathbf{h} = (h_1, \ldots, h_K)$ for some K, and say

$$h_k = g_k(\mathbf{x}_n' \mathbf{w}_k)$$

and then model

$$t_n = \sigma(\mathbf{h}' \boldsymbol{\beta})$$

Note the notation $\sigma(\cdot)$ for the *activation function* is common- don't get confused with using the same Greek letter to represent standard deviation, and the sigmoid function (though even more confusingly, often the $\sigma(\cdot)$ function above actually will be the sigmoid function).

Latent Variable Models

What we have done is modelled

$$t_n = \sigma\left(\begin{pmatrix} g_1(\mathbf{x}_n'\mathbf{w}_1) \\ \vdots \\ g_K(\mathbf{x}_n'\mathbf{w}_K) \end{pmatrix}' \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_K \end{pmatrix}\right)$$

which is not pretty. Already we we would be very hard-pressed to interpret this model physically.

It is better to interpret the intermediate variable $\mathbf{h}$ as a **latent variable**.

Neural Networks (ESL, Chapter 11)

We have just specified a single-layer, fully-connected **Neural Network**.

A Neural Network is a non-linear statistical model that is built by composing many non-linear transformations of linear combinations of features and parameters.

> *The central idea is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features.*

- Hastie/Tibshirani/Friedman, ESL, 2009, pg 389

Neural Networks (ESL, Chapter 11)

Formally, suppose we are performing K-class classification. We define a model as

$$h_m = g_m(\mathbf{x}_n' \mathbf{w}_m), m = 1 \dots M$$
$$T_k = \mathbf{h}' \boldsymbol{\beta}_k, k = 1 \dots K$$
$$\hat{t}_k = \sigma(T_k)$$

We could also define a similar model for regression.

Neural Networks (ESL, Chapter 11)

This is a fully-connected, single-layer Neural Network. It's sort of clear what's happening:

- Take some $M > 0$ linear combinations of the p predictors
- Apply M (possibly different) non-linear transformations of these; these are "derived" features
- Predict our target using a non-linear transformation of a linear combination of these derived features

This last step, again, is what we were doing before (e.g. logistic regression).

It seems similar in spirit to the basis-function approach we discussed in week 3.

Neural Networks (ESL, Chapter 11)

We can do better at interpreting the model here by viewing the h_k as **latent variables**.

We are thus modelling our latent variables, then modelling our observed variables as a function of these latent variables, like before.

These let us visualize the model using a *network diagram*, which you have likely seen before at least in passing.

Neural Networks (ESL, Chapter 11)

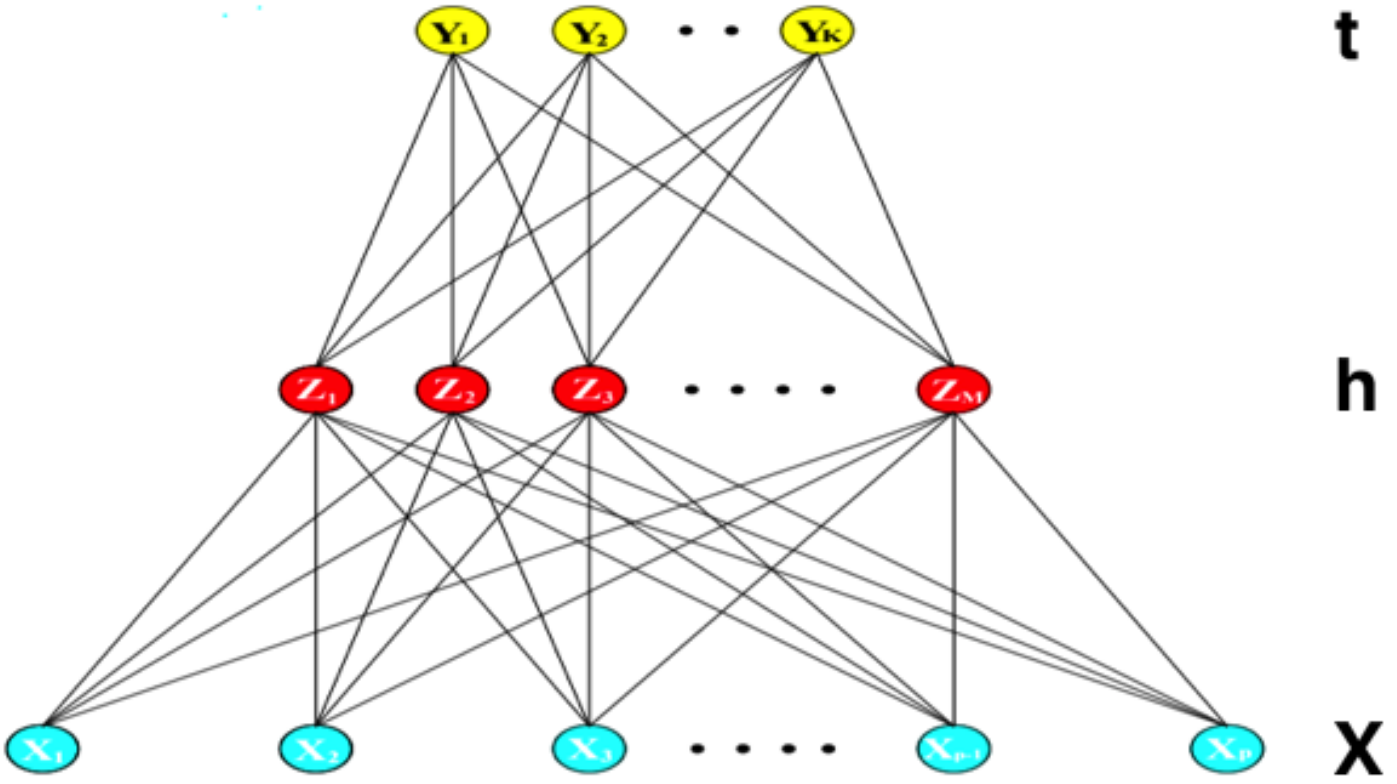


Figure 2: Network Diagram from ESL, page 393

Neural Networks (ESL, Chapter 11)

The two main advantages here are, like with Random Effects Models, *interpretability* and *extendability*.

We can interpret the $\mathbf{h}$ as **hidden units** or **hidden layers**.

We can then alter the structure of the model graphically by changing the number and size of the hidden layers, and how everything is connected (inputs to hidden layer 1, the hidden layers to each other, and the last hidden layer to the output layer).

Neural Network Terminology

Doing this leads to a variety of *network architectures*, which are tailored to individual problems.

Examples:

- **Autoencoders** have a "squeeze-box" or "accordion" shape, with layers near the inputs/outputs being large and layers near the middle being small
- **Convolutional** Neural Networks specify that the input units are connected to the first hidden layer in *groups*, which could i.e. correspond to physical locations in an image
- **Recurrant** Neural Networks allow connections between hidden units other than the forward-only connections we have discussed so far

. . . and many more:
http://www.asimovinstitute.org/neural-network-zoo/

Training Neural Networks

Choosing an architecture that is suited to your problem is an art, and there is usually a massive amount of problem-specific literature on the topic.

All of it is made possible by the latent variable formulation of the problem.

These models are so popular in part because their flexibility allows for application of this art (as long as the data is massive).

But this same complexity makes them very difficult to train.

Training Neural Networks

Neural Networks are trained using Gradient Descent, which is difficult due to the high-dimensionality of the problem.

When implemented in a particular efficient way, gradient descent for neural networks is called **backpropagation**, because the chain rule from calculus lets gradients propagate backwards through the network.

This was a major advancement in AI, and was developed in part at U of T:

> *Rumelhart, D. E., **Hinton, G. E.**, and Williams, R. J. (1986) Learning representations by back-propagating errors. Nature, 323, 533–536*

Training Neural Networks

Define the loss function

$$L(\theta) = \sum_{n=1}^{N} L_n(\theta)$$

where θ is the vector of all parameters in the network, and $L_n(\theta)$ is the contribution to the loss of the n^{th} input feature $\mathbf{x}_n$.

We wish to compute the gradient, $\nabla L = \frac{\partial L}{\partial \theta}$, which will have elements

$$\nabla L_j = \sum_{n=1}^{N} \frac{\partial L_n}{\partial \theta_j}$$

which we update according to

$$\theta_j^{t+1} = \theta_j^t - \gamma \sum_{n=1}^{N} \frac{\partial L_n}{\partial \theta_j}$$

Training Neural Networks

The loss function for a single datapoint L_n depends on θ through the predicted values of $\mathbf{t}_k$,

$$L_n(\theta) \equiv L_n(\hat{\mathbf{t}})$$

In turn,

$$\hat{t}_k = \sigma(T_k)$$

$$T_k = \sum_{m=1}^{M} \beta_m h_m(\mathbf{w}_m)$$

$$h_m(\mathbf{w}_m) = g_m(\mathbf{x}_n' \mathbf{w}_m)$$

Training Neural Networks

The chain rule gives:

$$\frac{\partial L_n}{\partial \mathbf{w}_m} = \frac{\partial L_n}{\partial \mathbf{t}} \times \frac{\partial \mathbf{t}}{\partial \mathbf{T}} \times \frac{\partial \mathbf{T}}{\partial h_m} \times \frac{\partial h_m}{\partial \mathbf{w}_m}$$

Training Neural Networks

For concreteness, suppose $K = 2$ (so we only have one predicted t for each $\mathbf{x}_n$) and let

$$\sigma(x) = g_m(x) = \frac{1}{1 + e^{-x}}$$

so all the activation functions involved are sigmoids. Recall that

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Training Neural Networks

We then have

$$\frac{\partial \hat{t}}{\partial T} = \sigma(T)(1-\sigma(T))$$
$$\frac{\partial T}{\partial h_m} = \beta$$
$$\frac{\partial h_m}{\partial \mathbf{w}_m} = g_m'(\mathbf{x}_n'\mathbf{w}_m)\mathbf{w}_m = \sigma(\mathbf{x}_n'\mathbf{w}_m)(1-\sigma(\mathbf{x}_n'\mathbf{w}_m))\mathbf{w}_m$$

All that remains is to specify the loss function- cross-entropy is popular,

$$L_n = -\sum_{k=1}^{K} t_{nk} \log \hat{t}_{nk}$$
$$\equiv -\left(t_n \log \hat{t}_n + (1-t_{nk}) \log (1-\hat{t}_{nk})\right) \text{ when } k=2$$

Training Neural Networks

Coding up gradient descent using this approach would require many redundant passes over the full network, making "deep learning" computationally infeasible.

The backpropagation algorithm describes how to compute **all** the elements of the gradient of the loss with respect to θ using only two passes over the full network.

This generalizes to the case where there is more than one hidden layer.

Stochastic Gradient Descent is very popular for training neural networks.

Training Neural Networks: Practical Issues

Overparametrization. Neural Networks can have millions of parameters.

As $p \to n$ and beyond, optimization becomes very unstable and there exist many solutions (local minima of the loss function).

This can (and is) solved using regularization, in the following ways:

- Adding a direct penalty to the loss function
- "Dropout", which sets a pre-determined number of weights in each layer to zero at each iteration of gradient descent
- Early Stopping, which stops gradient descent before reaching a minimum of the loss function on the training set, to mitigate overfitting

Training Neural Networks: Practical Issues

Zero Weights. This one is more subtle, but happens a lot when using "out of the box" neural network software.

Much software allows the user to specify the architecture by specifying the number of layers, and the number of units per layer.

A natural thing when naively building models is to go deeper (more layers) and bigger (more units).

Training Neural Networks: Practical Issues

Zero Weights. This one is more subtle, but happens a lot when using "out of the box" neural network software.

But adding more parameters, like we saw with the polynomial curve fitting, can lead to estimates that are very large in absolute value, or within machine-precision distance of zero.

When adding more and more layers, always do statistical summaries (histograms, etc) of the weights in each layer.

You could be telling your boss you're doing "deep learning with stochastic gradient descent", when you're actually just fitting a logistic regression model.