STA 414/2104:
Lecture 8

5 March 2018:
Continuous Latent Variable Models,
Neural networks

Delivered by Mark Ebden
With thanks to Russ Salakhutdinov,
Jimmy Ba and others

Announcements

- Midterm regrading requests:
  - You should receive a reply later this week
  - Today (5 March) is the request deadline
- Next week’s topics:
  - Graphical Models
  - Modelling Sequential Data

Outline

- Continuous latent variable models
  - Background
  - PCA
- Neural networks
  - Introduction
  - Autoencoders
  - Learning neural networks

Reminder from last week: latent variables

"Latent variables are entities that we invent to explain patterns we see in observable variables – for instance, doctors have invented *diseases* to explain commonalities of symptoms seen in patients."

– Radford Neal

The values of these latent variables are inferred from those of observable variables.

Continuous Latent Variable Models

- Often there are some unknown underlying causes of the data

- So far we have looked at models with discrete latent variables, such as the mixture of Gaussians

- Sometimes, it is more appropriate to think in terms of continuous factors which control the data we observe

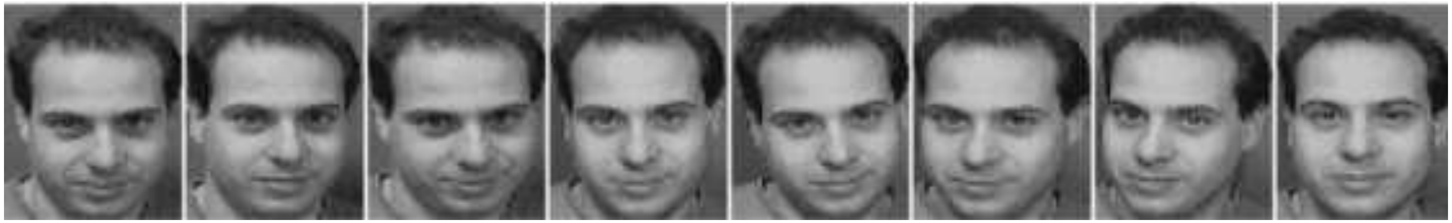
- Motivation: for many datasets, data points lie close to a manifold of much lower dimensionality compared to that of the original data space

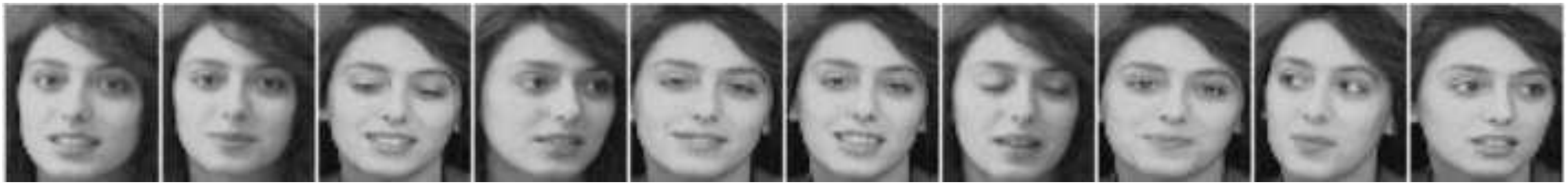
- Training continuous latent variable models is often called **dimensionality reduction,** since there are typically fewer latent dimensions

- Examples: Principal Component Analysis, Factor Analysis, Independent Component Analysis

Intrinsic Latent Dimensions

- What are the intrinsic latent dimensions in these two datasets?

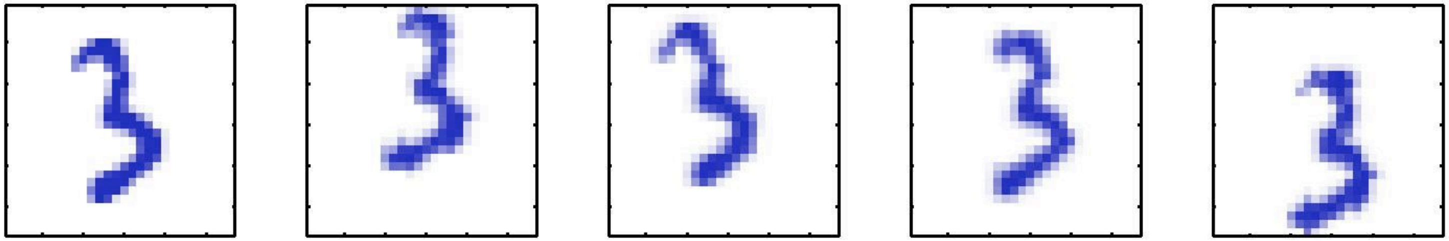




- How can we find the latent dimensions from this high-dimensional data?

Intrinsic Latent Dimensions

- In this dataset, there are only 3 degrees of freedom of variability, corresponding to vertical- and horizontal translations, and the rotations



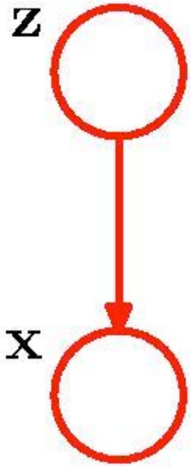
- Each image undergoes a random displacement and rotation within some larger image field

- The resulting images have 100 x 100 = 10,000 pixels

Generative View

- Each data example was generated by first selecting a point from a distribution in the latent space, then generating a point from the conditional distribution in the input space

- Simple example of a latent variable model: Assume a Gaussian distribution for both the latent and observed variables



- This can lead to a probabilistic formulation of Principal Component Analysis and Factor Analysis

- We will look at standard PCA, then briefly note its probabilistic formation.

- Advantages of the probabilistic formulation: use of EM for parameter estimation, mixture of PCAs, Bayesian PCA, etc

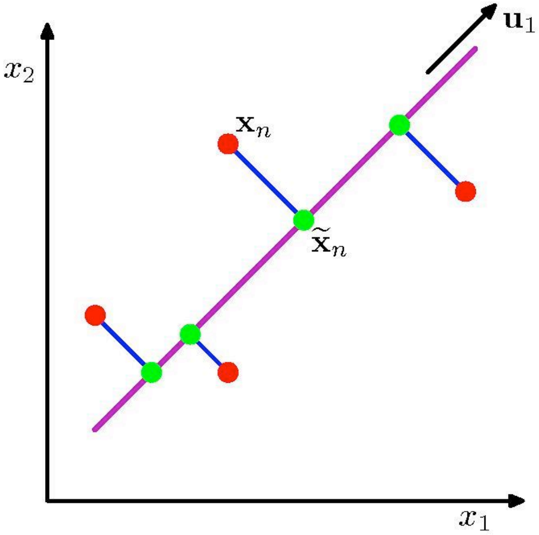
Principal Component Analysis

- Used for data compression, visualization, feature extraction, dimensionality reduction

- The goal is to:
  - Find the *M* principal components underlying the *D*-dimensional data: select the top *M* eigenvectors of **S** (data covariance matrix):

$$\{\mathbf{u}_1, \ldots, \mathbf{u}_M\}.$$

  - Project each input vector $\mathbf{x}$ into this subspace, e.g. $z_{n1} = \mathbf{x}_n^T \mathbf{u}_1.$



- The full projection into *M* dimensions takes the form:

$$\begin{bmatrix} \mathbf{u}_1^\top \\ \cdots \\ \mathbf{u}_M^\top \end{bmatrix} [\mathbf{x}_1 \cdots \mathbf{x}_N] = [\mathbf{z}_1 \cdots \mathbf{z}_N]$$

Two equivalent views / derivations:

- PCA maximizes the variance of the projected data (the scatter of the green points)
- PCA minimizes the error of the projected data (the mean of the squared blue lines)

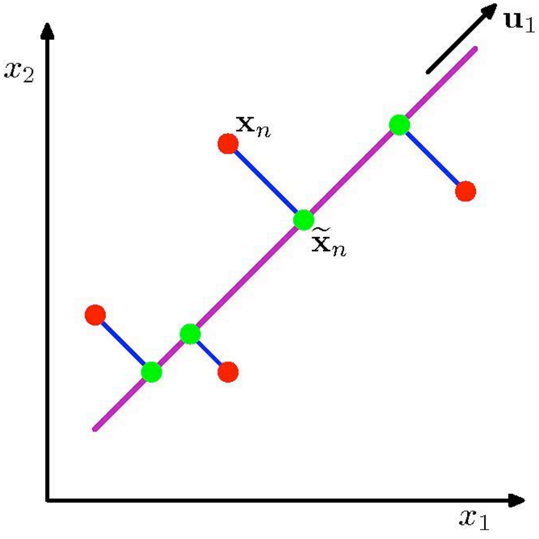
Maximum Variance Formulation

- Consider a dataset $\{\mathbf{x}_1,\ldots,\mathbf{x}_N\}$, $\mathbf{x}_n$ ε R^D. Our goal is to project data onto a space having dimensionality $M < D$

- Consider the projection into M=1 dimensional space

- Define the direction of this space using a D-dimensional unit vector $\mathbf{u}_1$, so that $\mathbf{u}_1^T\mathbf{u}_1 = 1.$

- Objective: maximize the variance of the projected data with respect to $\mathbf{u}_1$



$$\frac{1}{N}\sum_{n=1}^{N}\{\mathbf{u}_1^T\mathbf{x}_n - \mathbf{u}_1^T\overline{\mathbf{x}}\}^2 = \mathbf{u}_1^T\mathbf{S}\mathbf{u}_1$$

where sample mean and data covariance is given by:

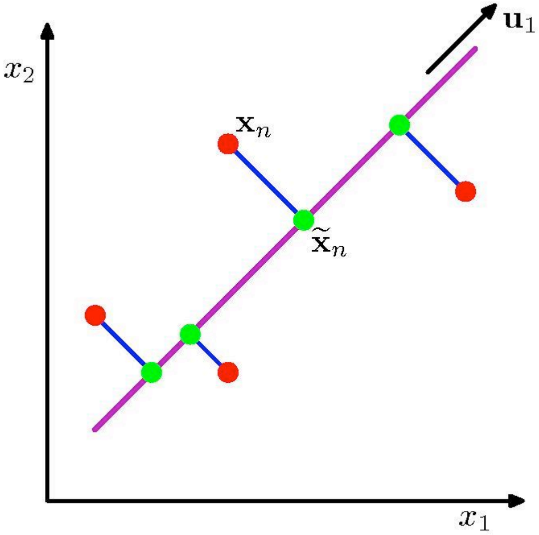
$$\overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$$

$$\mathbf{S} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^T$$

Maximum Variance Formulation

- Maximize the variance of the projected data:

$$\frac{1}{N}\sum_{n=1}^{N}\{\mathrm{u}_1^T\mathrm{x}_n - \mathrm{u}_1^T\overline{\mathrm{x}}\}^2 = \mathrm{u}_1^T\mathrm{S}\mathrm{u}_1$$



- Must constrain $||\mathbf{u}_1||$, and we choose $||\mathbf{u}_1|| = 1$.

Using a Lagrange multiplier, maximize:

$$\mathrm{u}_1^T\mathrm{S}\mathrm{u}_1 + \lambda(1 - \mathrm{u}_1^T\mathrm{u}_1)$$

- Setting the derivative with respect to $\mathbf{u}_1$ to zero:

$$\mathbf{S}\mathbf{u}_1 = \lambda_1\mathbf{u}_1.$$

- Hence $\mathbf{u}_1$ must be an eigenvector of $\mathbf{S}$
- The maximum variance of the projected data is given by:

$$\mathbf{u}_1^T\mathbf{S}\mathbf{u}_1 = \lambda_1.$$

- Optimal $\mathbf{u}_1$ is the principal component (eigenvector with maximal eigenvalue)

Minimum Error Formulation

- Introduce a complete orthonormal set of *D*-dimensional basis vectors: $\{\mathbf{u}_1, ..., \mathbf{u}_D\}$:

$$\mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}.$$

- Without loss of generality, we can write:

$$\mathbf{x}_n = \sum_{i=1}^{D} \alpha_{ni} \mathbf{u}_i, \quad \alpha_{ni} = \mathbf{x}_n^T \mathbf{u}_i.$$

Rotation of the coordinate system to a new system defined by $\mathbf{u}_i$.

- Our goal is to represent data points by the projection into an *M*-dimensional subspace (plus some distortion)
- Represent the *M*-dimensional linear subspace by the first *M* basis vectors:

$$\tilde{\mathbf{x}}_n = \sum_{i=1}^{M} z_{ni} \mathbf{u}_i + \sum_{i=M+1}^{D} b_i \mathbf{u}_i.$$

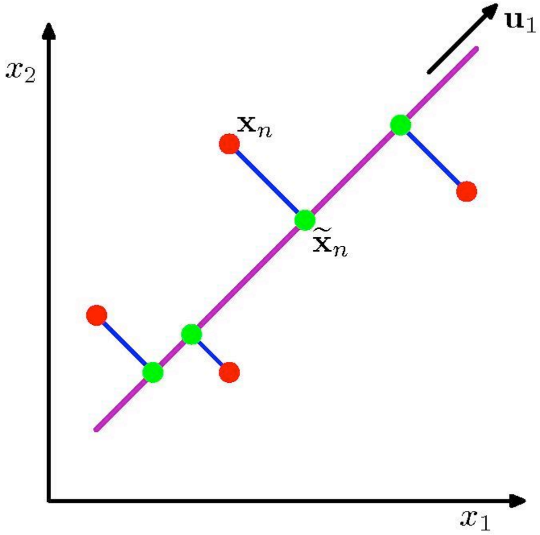
first M basis are principle components

Minimum Error Formulation

- Represent the *M*-dimensional linear subspace by the first *M* basis vectors:

$$\tilde{\mathbf{x}}_n = \sum_{i=1}^{M} z_{ni}\mathbf{u}_i + \sum_{i=M+1}^{D} b_i\mathbf{u}_i.$$

where z_{ni} depend on the particular data point and b_i are constants



- Objective: minimize distortion with respect to $\mathbf{u}_i$, z_{ni}, and b_i

$$J = \frac{1}{N}\sum_{n=1}^{N} ||\mathbf{x}_n - \tilde{\mathbf{x}}_n||^2.$$

- Minimizing with respect to z_{nj}, b_j:

$$z_{nj} = x_n^T u_j$$
$$b_j = \bar{x}^T u_j$$

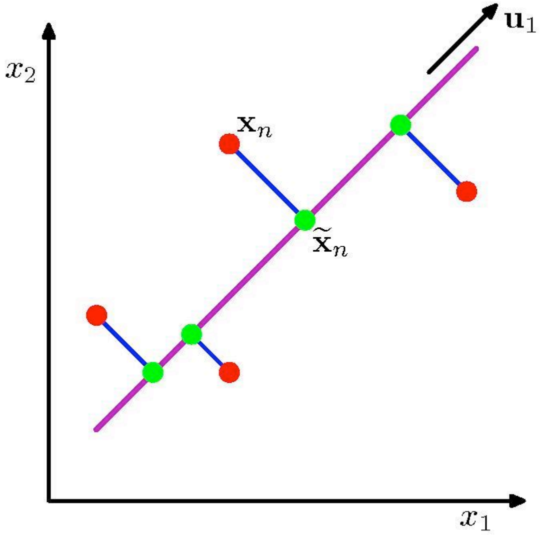
- Hence, the objective reduces to:

$$J = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=M+1}^{D}(\mathbf{x}_n^T\mathbf{u}_i - \bar{\mathbf{x}}^T\mathbf{u}_i)^2 = \sum_{i=M+1}^{D}\mathbf{u}_i^T\mathbf{S}\mathbf{u}_i.$$

Minimum Error Formulation

- Minimizing distortion with respect to $\mathbf{u}_i$ is a constrained minimization problem:

$$J = \frac{1}{N}\sum_{n=1}^{N} ||\mathbf{x}_n - \tilde{\mathbf{x}}_n||^2 = \sum_{i=M+1}^{D} \mathbf{u}_i^T \mathbf{S} \mathbf{u}_i.$$



- You are not responsible for showing that the *general* solution is to choose $\mathbf{u}_i$ to be eigenvectors of the covariance matrix: $\mathbf{S}\mathbf{u}_i = \lambda_i \mathbf{u}_i.$

- The distortion is then given by: $J = \sum_{i=M+1}^{D} \lambda_i.$

- The objective is minimized when the remaining *D* – *M* components are the eigenvectors of **S** with ***lowest eigenvalues*** → same result

- **Exercise:** show that for *D=2*, *M=1*, $\mathbf{u}_2$ is the eigenvector of **S** corresponding to the second-largest eigenvalue

- We will later see a generalization of PCA: deep autoencoders

Applications of PCA

- Run PCA on 2429 19x19 grayscale images (CBCL database)

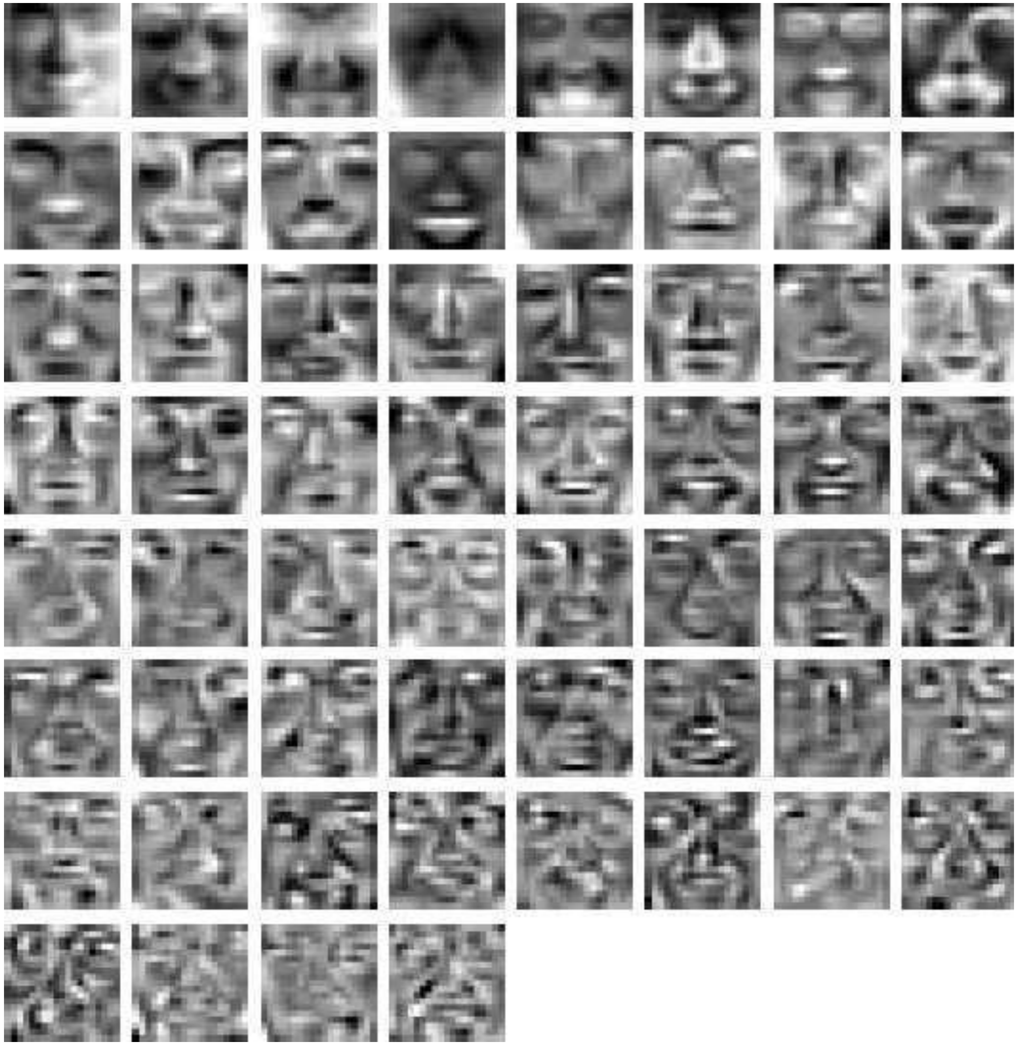
- Data compression: We can get good reconstructions with only 3 components

- Pre-processing: We can apply a standard classifier to latent representation – PCA with 3 components obtains 79% accuracy on face/non-face discrimination in test data, vs. 76.8% for a mixture of Gaussians with 84 components

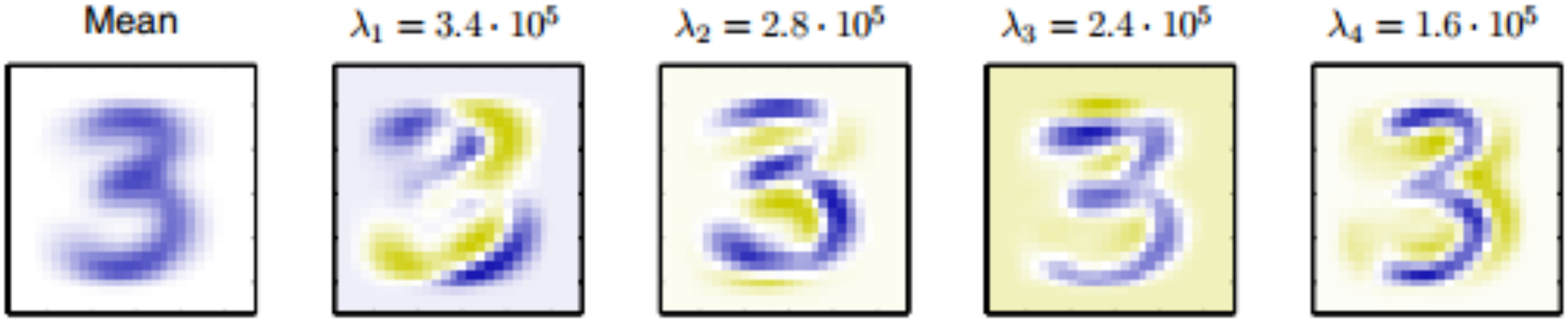
- Data visualization: by projecting the data onto the first two principal components

Learned Basis

- Run PCA on 2429 19x19 grayscale images (CBCL database)



Eigenvectors for 3’s



Mean

$\lambda_1 = 3.4 \cdot 10^5$

$\lambda_2 = 2.8 \cdot 10^5$

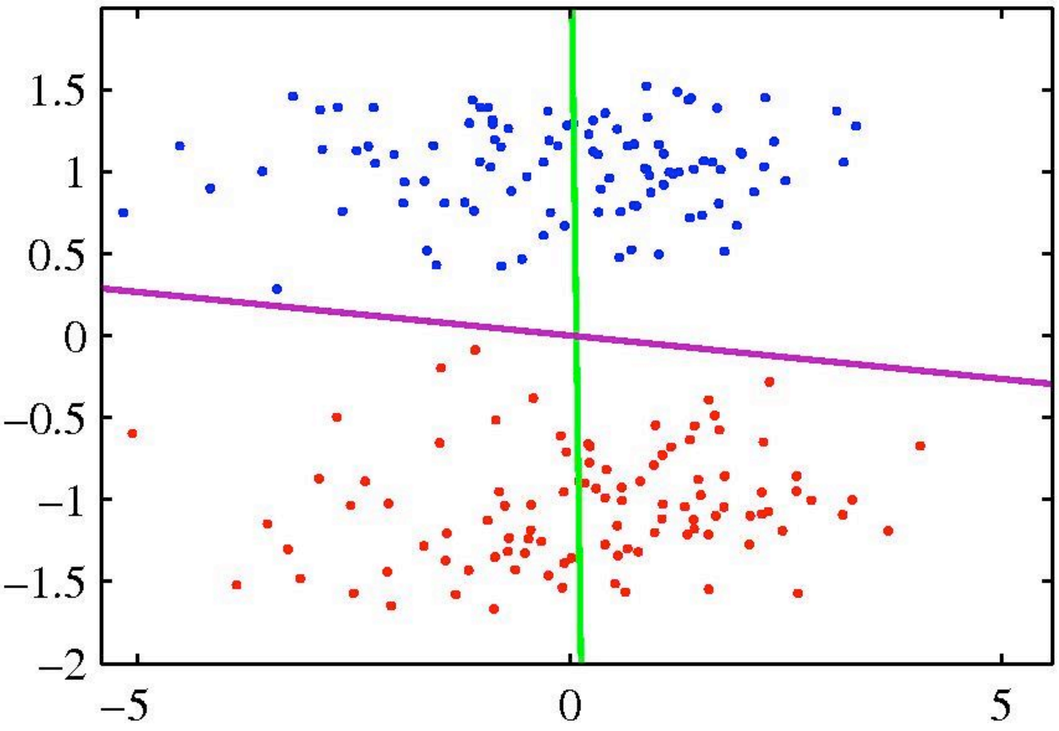
$\lambda_3 = 2.4 \cdot 10^5$

$\lambda_4 = 1.6 \cdot 10^5$

The mean vector $\overline{\mathbf{x}}$ along with the first four PCA eigenvectors $\mathbf{u}_1, \ldots, \mathbf{u}_4$ for the off-line digits data set, together with the corresponding eigenvalues.

PCA vs. Fisher’s LDA

- A comparison of PCA with Fisher’s LDA for linear dimensionality reduction



- PCA chooses the direction of maximum variance (magenta curve) leading to strong class overlap (unsupervised)

- LDA takes into account the class labels (supervised), leading to a projection into the green curve

PCA for High-Dimensional Data

- In some applications of PCA, the number of data points is smaller than the dimensionality of the data space, i.e. $N<D$
- So far, we've needed to find the eigenvectors of the D x D data covariance matrix **S**, which scales as $\mathcal{O}(D^3)$
- Direct application of PCA may be computationally infeasible
- Solution: Let $\mathbf{X}$ be the N x D centred data matrix. The corresponding eigenvector equation becomes:

$$\frac{1}{N}\mathbf{X}^T\mathbf{X}\mathbf{u}_i = \lambda_i\mathbf{u}_i.$$

- Pre-multiply by $\mathbf{X}$:

$$\frac{1}{N}\mathbf{X}\mathbf{X}^T(\mathbf{X}\mathbf{u}_i) = \lambda_i(\mathbf{X}\mathbf{u}_i).$$

PCA for High-Dimensional Data

- Define $\mathbf{v}_i = \mathbf{X}\mathbf{u}_i$, and hence we have:

$$\frac{1}{N}\mathbf{X}\mathbf{X}^T\mathbf{v}_i = \lambda_i \mathbf{v}_i.$$

- This is an eigenvector equation for the N x N matrix

- It has the same N – 1 eigenvalues as the original data covariance matrix **S** (which itself has an additional D – N + 1 zero eigenvalues).

- Computational cost scales as $\mathcal{O}(N^3)$ rather than $\mathcal{O}(D^3)$

- To determine eigenvectors, we pre-multiply by $\mathbf{X}^T$:

$$\left(\frac{1}{N}\mathbf{X}^T\mathbf{X}\right)(\mathbf{X}^T\mathbf{v}_i) = \lambda_i \mathbf{X}^T\mathbf{v}_i.$$

- Hence $\mathbf{X}^T \mathbf{v}_i$ is an eigenvector of **S** with eigenvalue λ_i

- These eigenvectors may not be normalized. **Exercise:** How do we ensure that they are normalized? (Answer: Bishop exercise 12.3)

Bonus

Probabilistic PCA

- A probabilistic, generative view of data which is not covered in STA414

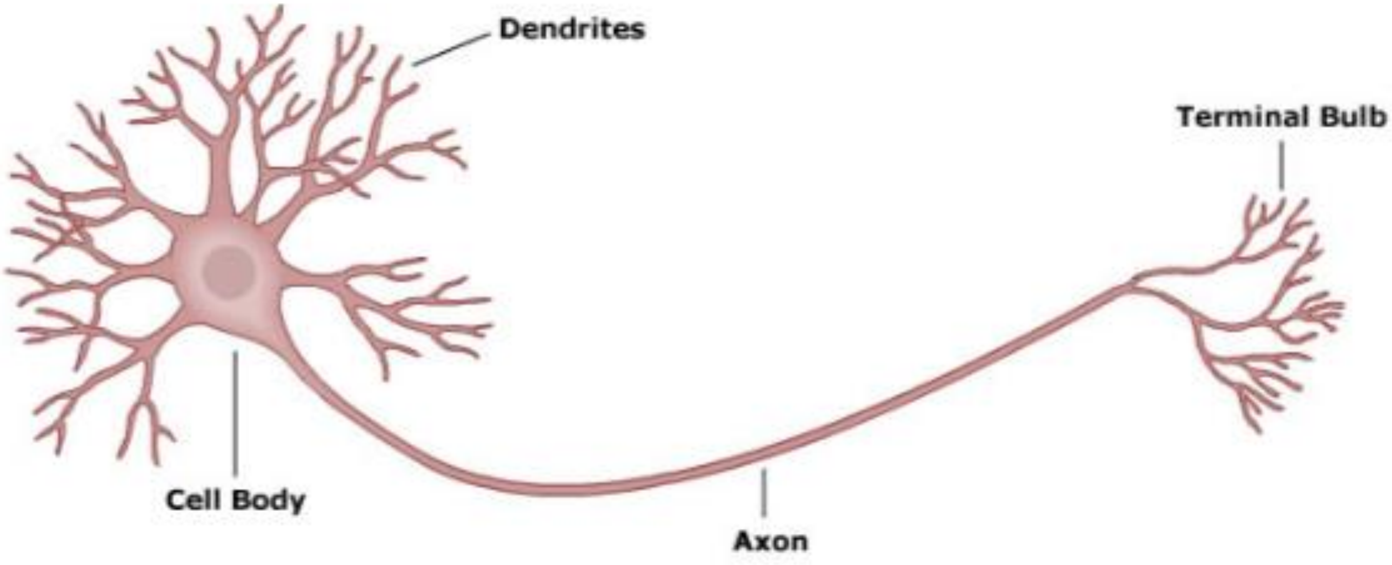
- Advantages of probabilistic PCA (PPCA):

  - We can derive an EM algorithm for PCA, which in some settings is more computationally efficient
  - PPCA allows us to deal with missing values in the data set
  - We can formulate a mixture of PPCAs in a principled way
  - PPCA forms the basis for Bayesian PCA, in which the dimensionality of the principal subspace can be determined from the data
  - The existence of a likelihood function allows direct comparisons with other probabilistic density models
  - And more

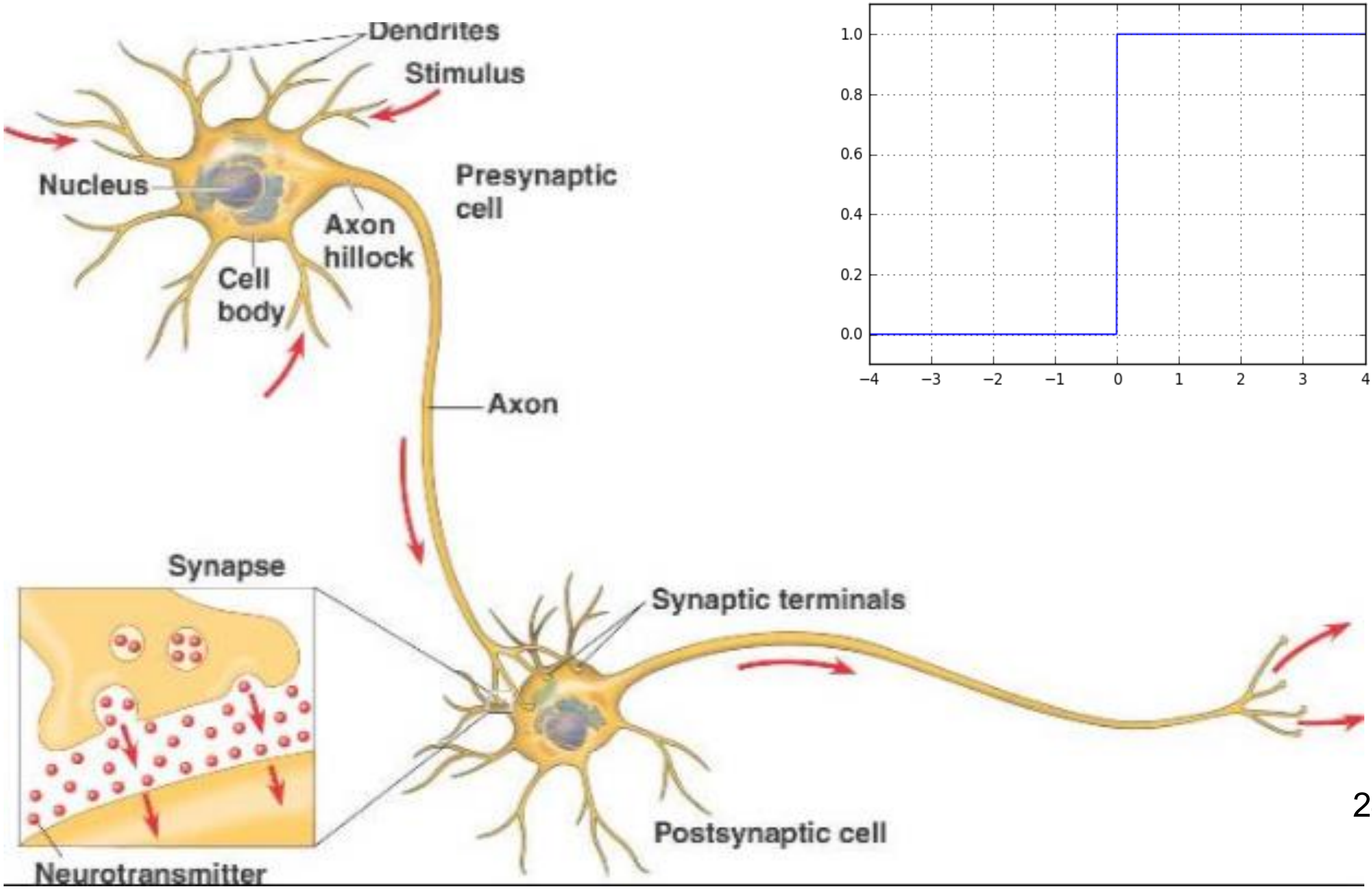
Outline

- Continuous latent variable models
  - – Background
  - – PCA
- Neural networks
  - – Introduction
  - – Autoencoders
  - – Learning neural networks

One neuron (biological)

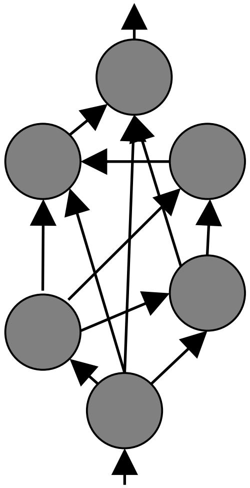


Two neurons


Dendrites
Stimulus
Nucleus
Presynaptic cell
Axon hillock
Cell body
Axon
Synapse
Synaptic terminals
Postsynaptic cell
Neurotransmitter
1.0
0.8
0.6
0.4
0.2
0.0
−4
−3
−2
−1
0
1
2
3
4

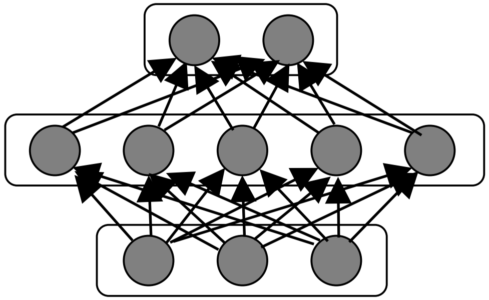
Neural networks

- Neural networks are flexible computation models that consist of many smaller computational modules called neurons or hidden units:
  - Neural networks are real-valued
  - It is very modular and some special modules are designed for reusability and abstraction
  - All continuous functions can be represented as neural networks
  - All the learnt knowledge of a neural network is stored in its weight connections; it is also called "connectionism" (a name popular before the first AI Winter)



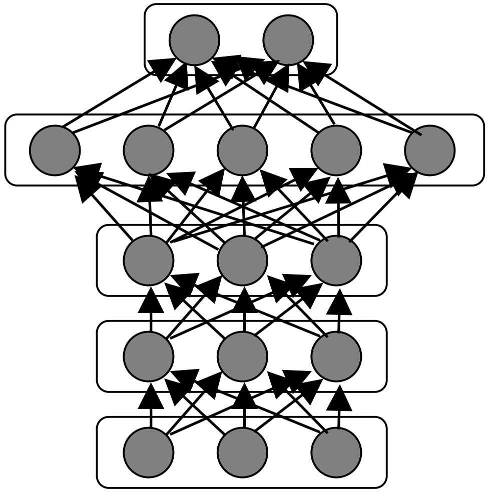
Neural networks

- One very useful abstraction is the concept of a “layer”:
  - A hidden layer is a group of hidden units that have connections one layer above and one layer below
  - There is no connection among the hidden units within a layer
  - This abstraction is computationally efficient because all the hidden units within a layer can be computed in parallel



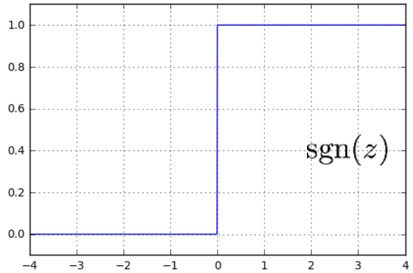
Neural networks

- Deep learning typically refers to a neural network with more than three hidden layers
  - Deep neural networks can mathematically represent any continuous function given enough layers, but they also require additional tricks to learn useful representations for any tasks
  - They work really well in supervised learning, given enough data
  - A deep neural network is like a complex system in biology: we understand a lot about what the simple module does, but it quickly becomes hard to understand what the system does, i.e. a “black box”



An artificial neuron

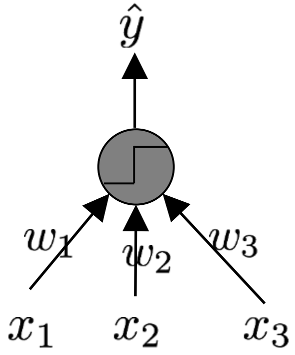
- An artificial neuron is a simple computation unit that receives inputs from other simple computation units:
  - The effect of each input on the final output of the neuron is controlled by a weight
  - The weights can be positive or negative values for encoding +ve or -ve contributions from the inputs
  - A weighted sum of the inputs was first proposed by McCulloch-Pitts (1943)



$$\mathrm{sgn}(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

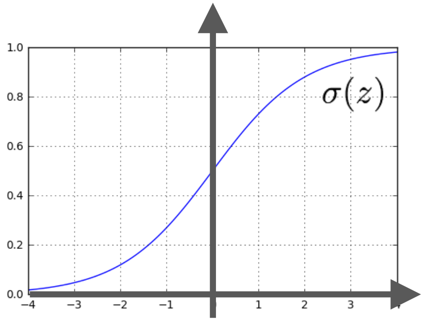
$$\frac{\partial sgn}{\partial z} = 0, \forall z \neq 0$$

$$\hat{y} = \mathrm{sgn}(\sum_n w_n x_n + b)$$



Some simple neurons: sigmoid neurons

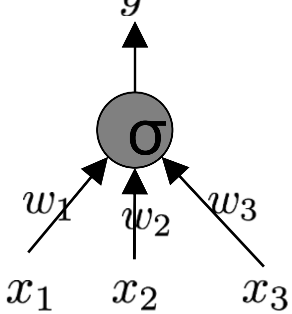
- An alternative to a hard step function is a soft, differentiable function; desirable when using a gradient-descent algorithm (e.g. Week 3) to learn our model
  - Sigmoid neurons can be thought of as soft-thresholding units
  - Logistic regression models are simply neural networks with a single logistic neuron
  - With high enough *w* values, the sigmoid can behave like the *sgn* function (slides 47-48)



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

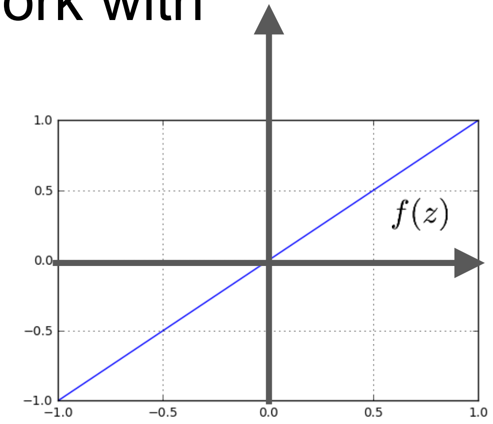
$$\frac{\partial \sigma}{\partial z} = \sigma(z)(1-\sigma(z))$$

$$\hat{y} = \sigma(\sum_n w_n x_n + b)$$



Some simple neurons: linear neurons

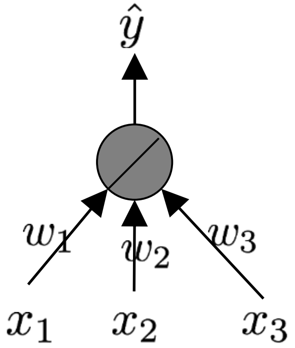
- A Linear neuron directly outputs the weighted sum of the inputs:
  - Linear regression is the simplest neural network with a single linear neuron
  - It has a constant partial derivative which is great for gradient descent
  - However, stacking layers of linear neurons does not increase the representational power of a model. Nonlinearity is important for building richer and more flexible models



$$f(z) = z$$

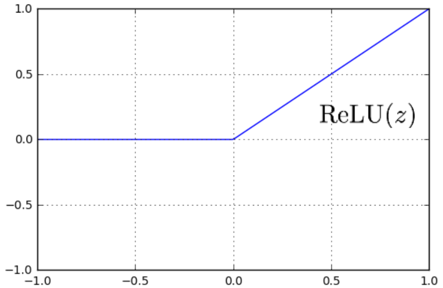
$$\frac{\partial f}{\partial z} = 1$$

$$\hat{y} = \sum_n w_n x_n + b$$



Some simple neurons: rectified linear units (ReLU)

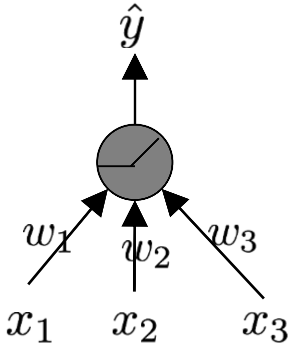
- Linear neurons can be modified to exhibit nonlinear behaviour:
  - The non-positive values are forced to be zero
  - ReLU neurons still have a constant gradient if the weighted sum of the inputs is positive
  - It is mathematically non-differentiable at zero, but we ignore that and use gradient descent anyways. It works well (numerically, we tend not to get exactly zero summed inputs)



$$\mathrm{ReLU}(z) = \max(0, z)$$

$$\frac{\partial \mathrm{ReLU}}{\partial z} = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

$$\hat{y} = \mathrm{ReLU}(\sum_n w_n x_n + b)$$

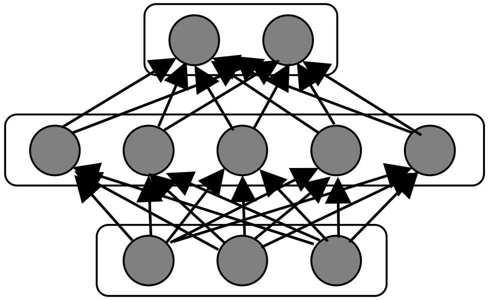


Context of neural networks

Two common ways to solve a problem:
1. Hire people to hard-code a program
2. Gather a huge dataset and learn the program from the data

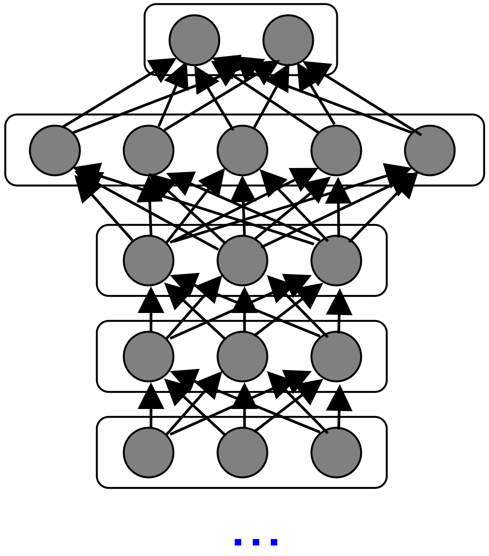
Deep neural networks avoid time-consuming feature-engineering by hand, and as the datasets grow larger they can discover better and better features without human intervention.

Neural networks can also be understood as a form of *adaptive basis function model* in which the model learns layers of basis functions. The activation function used for a neuron is similar to the nonlinear basis functions.



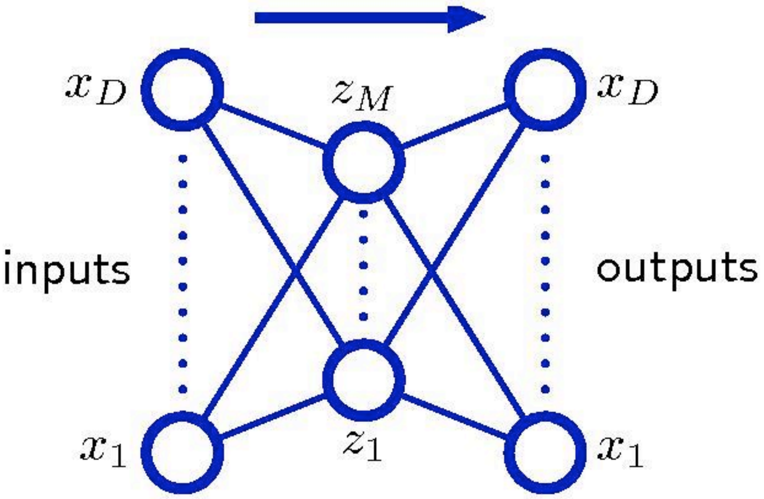
Hyper-parameters

- Choices:
  - How many hidden units to use in each hidden layer?
  - How many layers in total?
  - Which hidden activation function?
- Good answer: decide these hyper-parameters using a **validation set**
- *One* common, practical answer:
  - Around 500-2000 hidden units
  - 2-3 layers
  - Choosing ReLU often leads to fast convergence



Autoencoders

- Neural networks can be used for nonlinear dimensionality reduction
- This is achieved by having the same number of outputs as inputs. These models are called autoencoders
- Consider a multilayer perceptron that has *D* inputs, *D* outputs, and *M* hidden units such that *M* < *D*
- It is useful if we can squeeze the information through some kind of bottleneck



- If we use a linear network, this is very similar to Principal Component Analysis

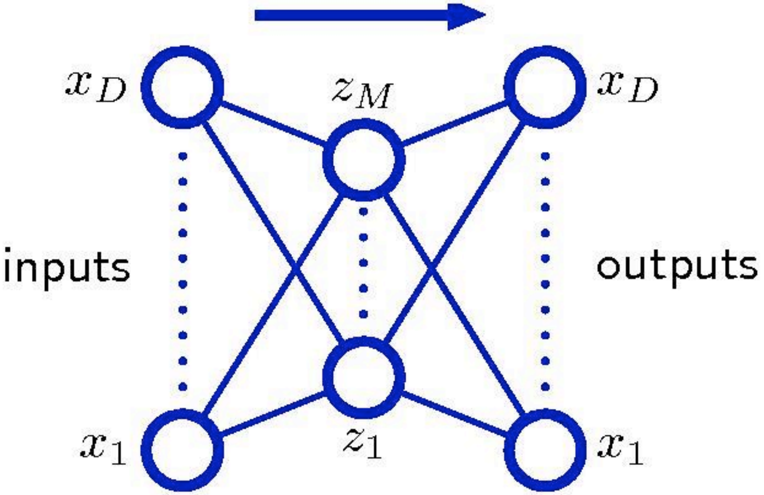
Autoencoders and PCA

- Given an input **x**, its corresponding reconstruction is given by:

$$y_k(\mathbf{x}, \mathbf{w}) = \sum_{j=1}^{M} w_{kj}^{(2)} \sigma \left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i \right), \quad k = 1, .., D.$$

- We can determine the network parameters **w** by minimizing the reconstruction error:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} ||y(\mathbf{x}_n, \mathbf{w}) - \mathbf{x}_n||^2.$$



- If the hidden and output layers are linear, we will learn hidden units that are a linear function of the data and minimize the squared error

- The *M* hidden units will span the same space as the first *m* principal components. The weight vectors may not be orthogonal

Deep Autoencoders

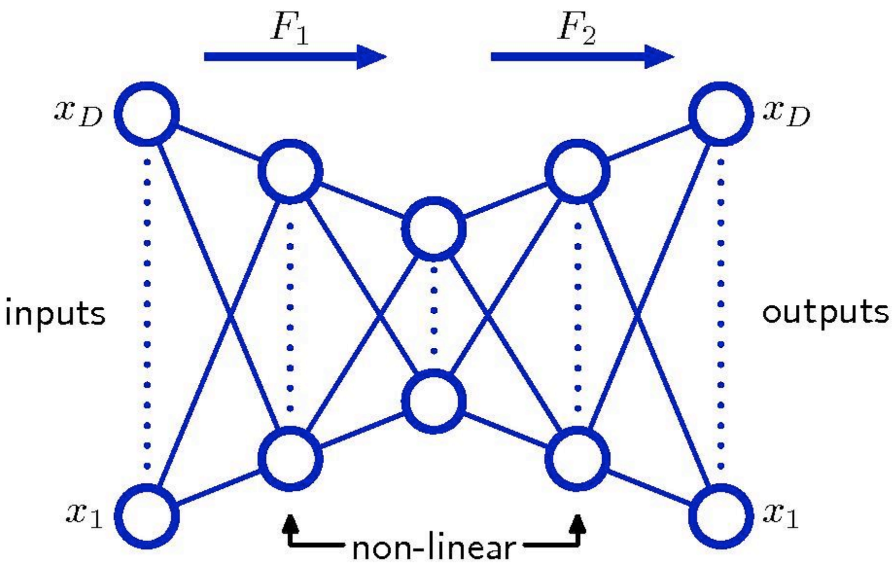
- We can put extra nonlinear hidden layers between the input and the bottleneck and between the bottleneck and the output

- This gives a nonlinear generalization of PCA

- It is good for nonlinear dimensionality reduction

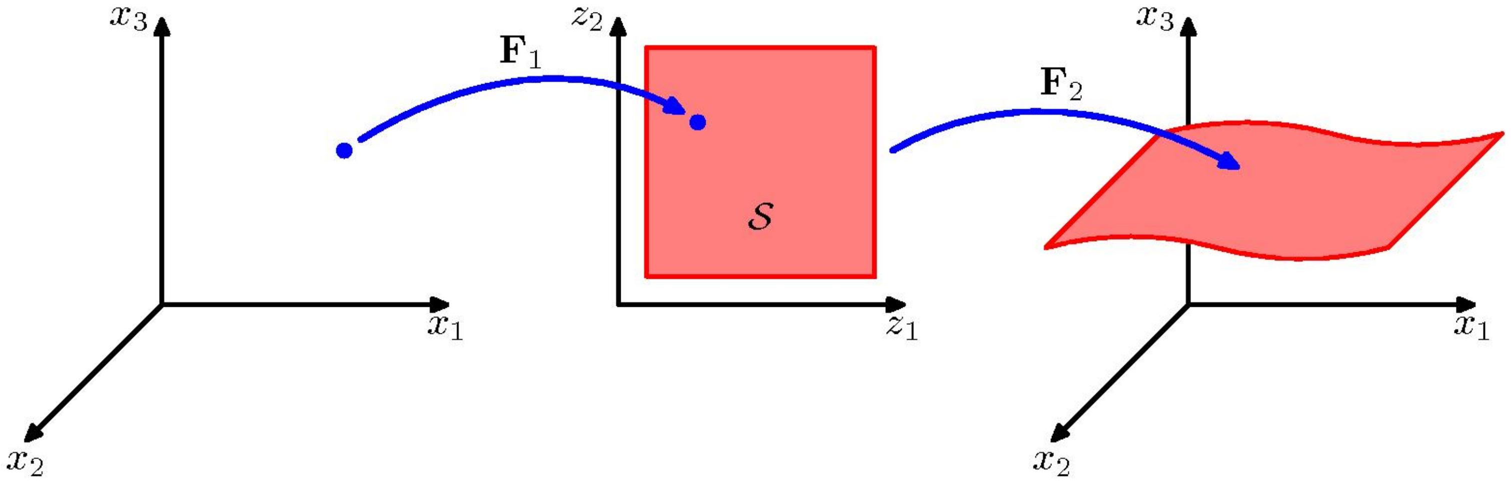
- The network can be trained by the minimization of the reconstruction error function

- Much harder to train



Geometrical Interpretation

- Geometrical interpretation of the mappings performed by the network with 2 hidden layers, for the case of *D*=3 and *M*=2 units in the middle layer

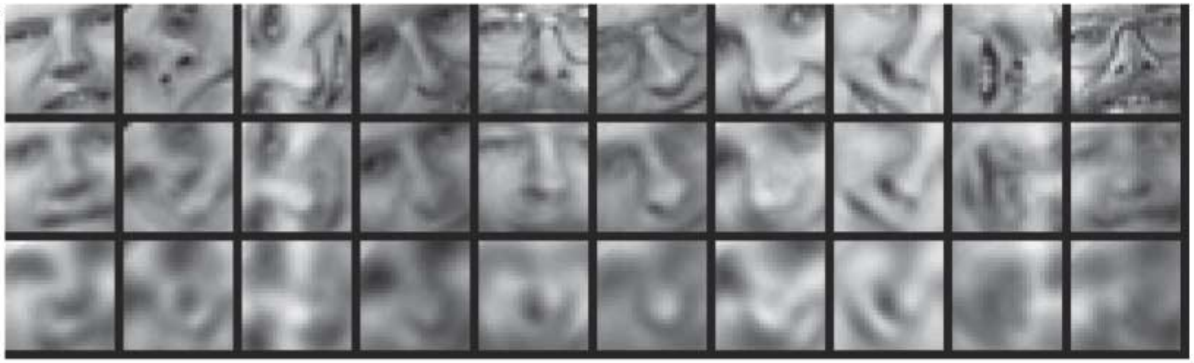


- The mapping $\mathbf{F_1}$ defines a nonlinear projection of points in the original *D*-space into the *M*-dimensional subspace

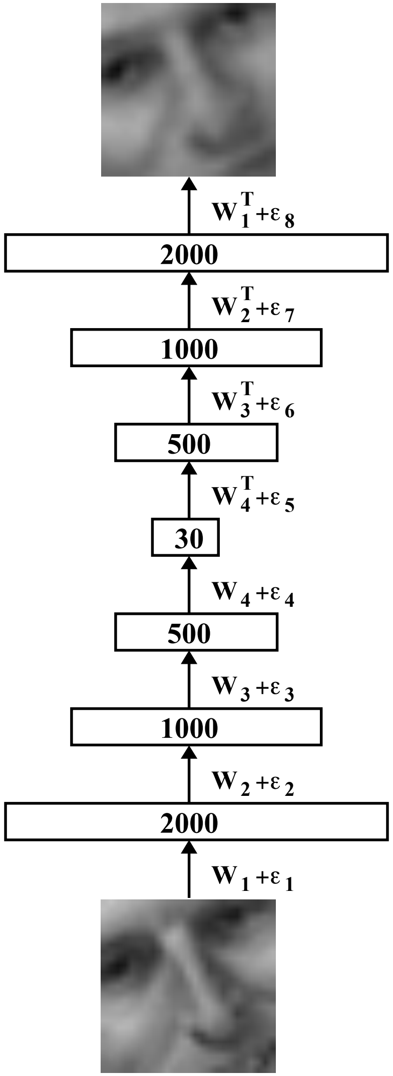
- The mapping $\mathbf{F_2}$ maps from an *M*-dimensional space into *D*-dimensional space

Deep Autoencoders

- We can consider deep autoencoders
- There is an efficient way to learn these deep autoencoders

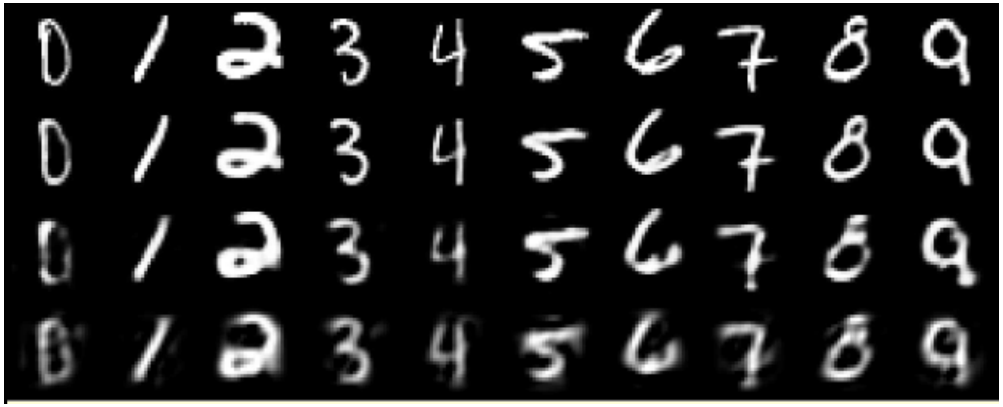


- By row: Real data, Deep autoencoder with a bottleneck of 30 linear units, and 30-d PCA



Deep Autoencoders

- We can consider deep autoencoders
- Similar model for MNIST handwritten digits:



Real data

30-d deep autoencoder

30-d logistic PCA

30-d PCA

- The Deep autoencoder produces much better reconstructions

Class Structure of the Data

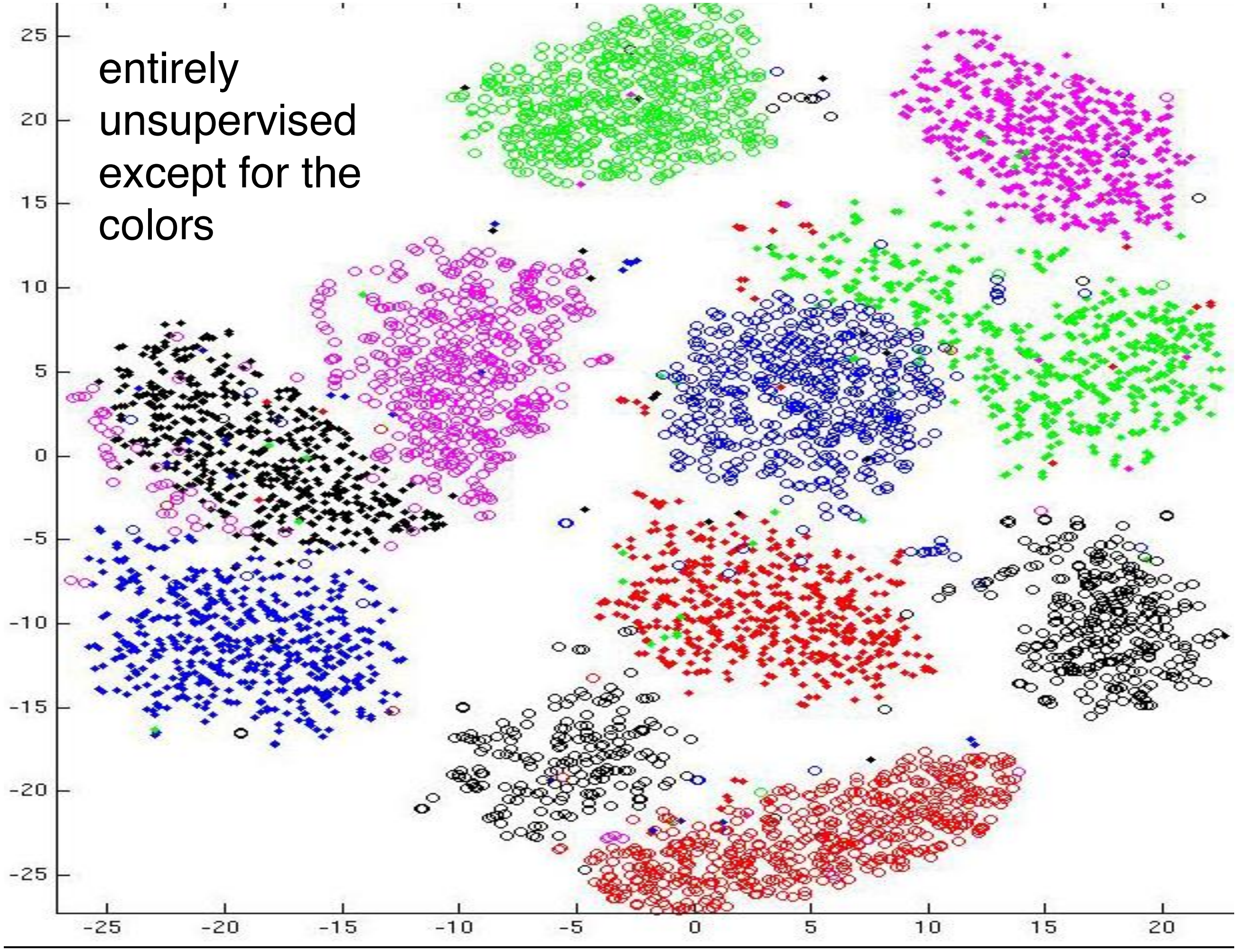
- Do the 30-D codes found by the deep autoencoder preserve the class structure of the data?

- Take the 30-D activity patterns in the code layer and display them in 2-D using nonlinear multi-dimensional scaling (UNI-SNE)

- Will the learning find the natural classes?

Class Structure of the Data

- Do the 30-D codes found by the deep autoencoder preserve the class structure of the data?



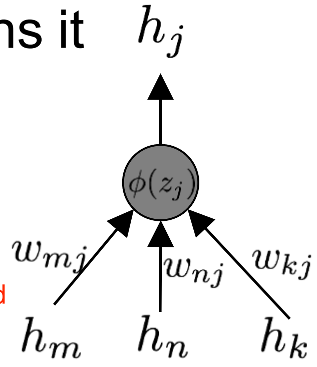
entirely unsupervised except for the colors

Outline

- Continuous latent variable models
  - Background
  - PCA
- Neural networks
  - Introduction
  - Autoencoders
  - Learning neural networks

Notation for neural networks

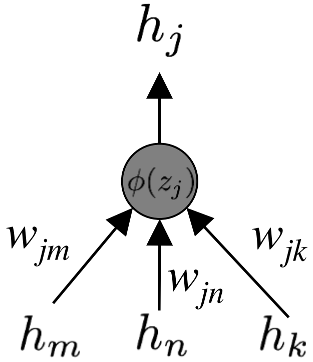
- The model now consists of many artificial neurons wired together into a large network. For clarity, we will use the following notation for our algorithms:
  - The output of a neuron or the hidden activation is denoted as *h*
  - Scalar weight connections are indexed by the two neurons it connects
  - The input to the network is denoted *x*
  - The output of the network is denoted as $\hat{y}$
  - The element-wise hidden activation function or the activation function or nonlinearity, denoted as $\phi(\cdot)$, is the nonlinear transformation for the weighted sum of the inputs of a neuron, e.g. sigmoid, ReLU...
  - The weighted sum of a neuron's inputs is denoted as *z*

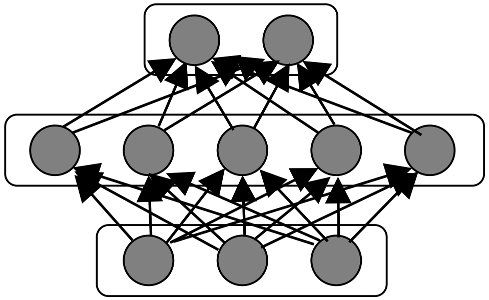


m,j switched

Forward propagation

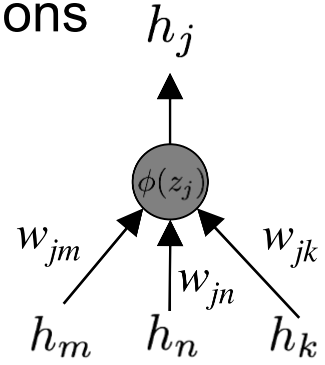
- Forward propagation computes all the hidden activations *h* and the output of the neural network $\hat{y}$
  - $h_j = \phi(z_j) = \phi(\sum_n w_{jn} h_n + b_j)$
  - This requires computing all the hidden activations that are the inputs to the current hidden units.
  - The forward propagation can be written as a recursive algorithm
  - The naive recursive algorithm is bad because there are a lot of redundant computations. We would like to cache the appropriate intermediate values and reuse them

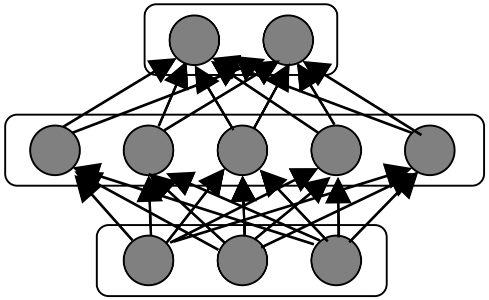




Back-propagation

- Back-propagation (Rumelhart, Hinton and Williams, 1986) is a dynamic programming method to reuse previous computations when computing the gradient of some variable using the chain rule from calculus
  - $h_j = \phi(z_j) = \phi(\sum_n w_{jn} h_n + b_j)$
  - In its simplest form: $\frac{\partial \mathcal{L}}{\partial w_{jn}} = \frac{\partial \mathcal{L}}{\partial h_j} \frac{\partial h_j}{\partial z_j} \frac{\partial z_j}{\partial w_{jn}}$
  - $\frac{\partial \mathcal{L}}{\partial h_j}$ can be further expanded until the output of the neural network
  - The key observation here is that the gradient of a connection is a product of the input and the partial derivative of the weighted sum of that neuron





$$\frac{\partial \mathcal{L}}{\partial w_{jn}} = \frac{\partial \mathcal{L}}{\partial z_j} h_n$$

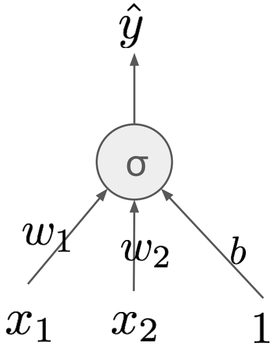
Appendix

- Slides 47-48: an example of a simple neural network
- Slides 49-54: backpropagation, in which w_{ij} becomes w_{ji}

Example 1: representing digital circuits with neural networks

- Let us look at a simple example of a soft OR gate simulated by a neural network:
  - Use a single sigmoid neuron with two inputs and a bias unit.

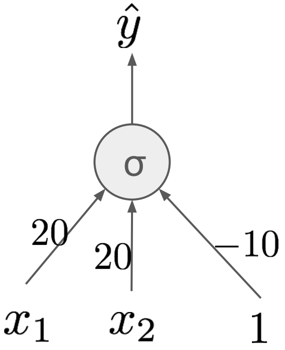
	x2=0	x2=1
x1=0	0	1
x1=1	1	1



Example 1: representing digital circuits with neural networks

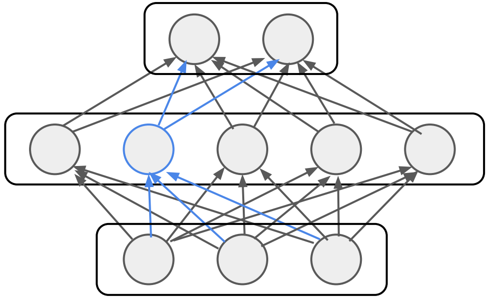
- Let us look at a simple example of a soft OR gate simulated by a neural network:
  - Use a single sigmoid neuron with two inputs and a bias unit.
  - One possible solution is to use the bias as a threshold while setting w_1 and w_2 to be large positive values. When either of the inputs is non-zero, the sigmoid neuron will be turned on and the output will be 1.

	x2=0	x2=1
x1=0	0	1
x1=1	1	1



Learning fully connected multi-layer neural networks

- There are many choices when “crafting” the architecture of a neural network. The fully connected multi-layer NN is the most general multi-layer NN:
  - Each neuron has its incoming weights connected to *all* the neurons from the previous layer and its outgoing weights connected to *all* the neurons in the next layer.
- Fully connected network is the go-to architecture choice if we do not have any additional information about the dataset.
  - After choosing the network architecture, there are a few more engineering choices: #hidden units, #layers, the type of activation function.
  - The output units type: linear, logistic or softmax are determined by output tasks, i.e. regression or classification task



Learning fully connected multi-layer neural networks

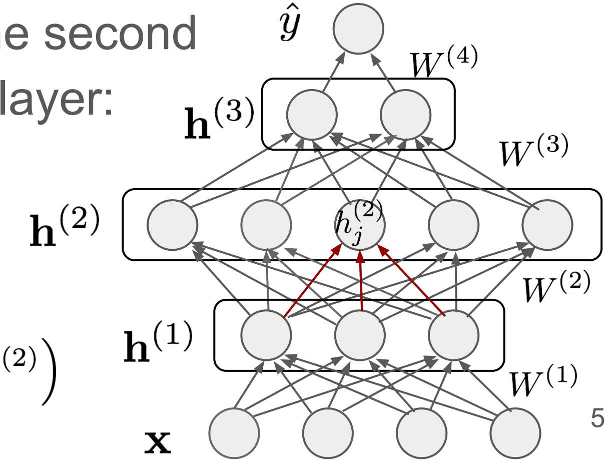
- Consider a fully connected neural network with 3 hidden layers:
  - The input to the neural network is an *N*-dimensional vector $\mathbf{x}$. There are *H1, H2,* and *H3* hidden units in the three hidden layers. We use superscript to index the layers.
  - There are four weight matrices among the hidden layers, e.g. $W^{(2)} \in \mathbb{R}^{H_2 \times H_1}, b^{(2)} \in \mathbb{R}^{H_2}$
  - The *j*th row of the weight matrix $W^{(2)}$ is denoted as $W_j^{(2)} \in \mathbb{R}^{H_1}$

- The hidden activation of the *j*th hidden unit $h_j^{(2)}$ in the second hidden layer is the weighted sum of the first hidden layer:

$$h_j^{(2)} = \phi\left(z_j^{(2)}\right) = \phi\left(\sum_i w_{ij}^{(2)} h_i^{(1)} + b_j^{(2)}\right) = \phi\left({W_j^{(2)}}^T \mathbf{h}^{(1)} + b_j^{(2)}\right)$$

  - We can use vector notation to express the hidden vector:

$$\mathbf{h}^{(2)} = \begin{bmatrix} h_1^{(2)} \\ \vdots \\ h_{H_2}^{(2)} \end{bmatrix} = \begin{bmatrix} \phi\left(z_1^{(2)}\right) \\ \vdots \\ \phi\left(z_{H_2}^{(2)}\right) \end{bmatrix} = \phi\left(\begin{bmatrix} {W_1^{(2)}}^T \\ \vdots \\ {W_{H_2}^{(2)}}^T \end{bmatrix} \mathbf{h}^{(1)} + b^{(2)}\right) = \phi\left(W^{(2)}\mathbf{h}^{(1)} + b^{(2)}\right)$$



Learning fully connected multi-layer neural networks

- For a single data point, we can write the the hidden activations of the fully connected neural network as a recursive computation using the vector notation:

$$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + b^{(1)}, \quad \mathbf{h}^{(1)} = \phi\left(\mathbf{z}^{(1)}\right)$$

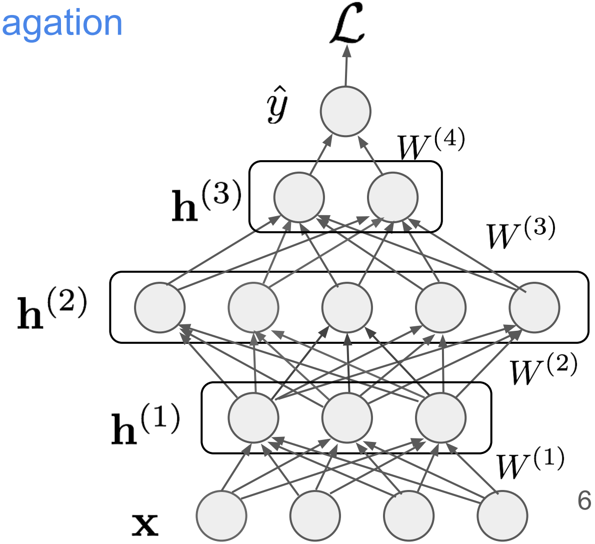
$$\mathbf{z}^{(2)} = W^{(2)}\mathbf{h}^{(1)} + b^{(2)}, \quad \mathbf{h}^{(2)} = \phi\left(\mathbf{z}^{(2)}\right)$$

$$\mathbf{z}^{(3)} = W^{(3)}\mathbf{h}^{(2)} + b^{(3)}, \quad \mathbf{h}^{(3)} = \phi\left(\mathbf{z}^{(3)}\right)$$

$$\mathbf{z}^{(4)} = W^{(4)}\mathbf{h}^{(3)} + b^{(4)}, \quad \hat{y} = f\left(\mathbf{z}^{(4)}\right)$$

  - *f()* is the output activation function
  - The output of the network is then used to compute the loss function on the training data

forward propagation

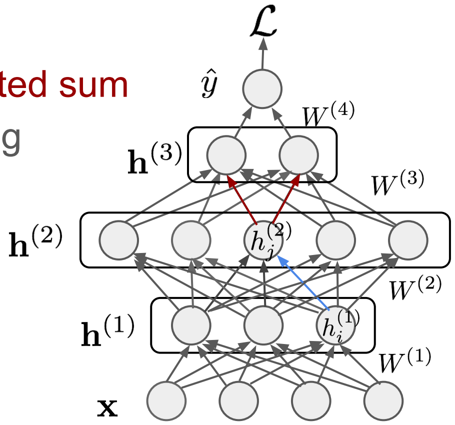


Learning fully connected multi-layer neural networks

- Learning neural networks using stochastic gradient descent requires the gradient of the weight matrices from each hidden layer.
  - Let us consider the gradient of the loss for a single training example. The gradient w.r.t. the incoming weights $w_{ij}^{(2)}$ of the *j*th hidden unit in the second layer is the *product* of the hidden activation from layer 1 *and* the partial derivative w.r.t. z_j. Remember: $h_j^{(2)} = \phi\left(z_j^{(2)}\right) = \phi\left(\sum_i w_{ij}^{(2)} h_i^{(1)} + b_j^{(2)}\right)$

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(2)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial w_{ij}^{(2)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(2)}} h_i^{(1)}$$

  - The partial derivative w.r.t. z_j in the second hidden layer is the weighted sum of the partial derivatives from the third layer, weighted by the outgoing weights of the *j*th hidden units:



$$\frac{\partial \mathcal{L}}{\partial z_j^{(2)}} = \frac{\partial \mathcal{L}}{\partial h_j^{(2)}} \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} = \left(\sum_i \frac{\partial \mathcal{L}}{\partial z_i^{(3)}} \frac{\partial z_i^{(3)}}{\partial h_j^{(2)}}\right) \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} = \boxed{\left(\sum_i \frac{\partial \mathcal{L}}{\partial z_i^{(3)}} w_{ji}^{(3)}\right)} \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}}$$

Learning fully connected multi-layer neural networks

- Similar to the hidden-activation computation (slide 10), the weighted sum of the partial derivatives can be rewritten using vector notation:

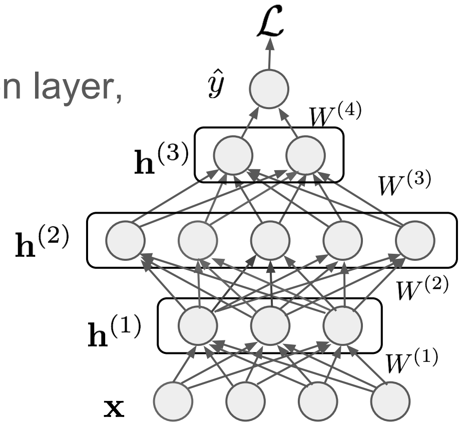
$$\frac{\partial \mathcal{L}}{\partial z_j^{(2)}} = \left(\sum_i \frac{\partial \mathcal{L}}{\partial z_i^{(3)}} w_{ji}^{(3)} \right) \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} = \left(\mathcal{W}_{j\cdot}{}^{(3)\,T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right) \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}}$$

$$\frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} = \begin{bmatrix} \frac{\partial h_1^{(2)}}{\partial z_1^{(2)}}0 & \cdots & 0 \\ \vdots & \frac{\partial h_j^{(2)}}{\partial z_j^{(2)}} & \vdots \\ 0 & \cdots & \frac{\partial h_{H_2}^{(2)}}{\partial z_{H_2}^{(2)}} \end{bmatrix} = \operatorname{diag} \left\{ \begin{bmatrix} \frac{\partial h_1^{(2)}}{\partial z_1^{(2)}} \\ \vdots \\ \frac{\partial h_{H_2}^{(2)}}{\partial z_{H_2}^{(2)}} \end{bmatrix} \right\}$$

  - Here, $\mathcal{W}_{j\cdot}^{(3)}$ is the *j*th column of the weight matrix $W^{(3)}$

  - To express the partial derivatives w.r.t. *z* for the entire second hidden layer, we can use a matrix-vector product:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z_1^{(2)}} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial z_{H_2}^{(2)}} \end{bmatrix} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} \left(\begin{bmatrix} \mathcal{W}_{1\cdot}^{(3)\,T} \\ \vdots \\ \mathcal{W}_{H_2\cdot}^{(3)\,T} \end{bmatrix} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right) = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} \left(W^{(3)\,T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right)$$



Learning fully connected multi-layer neural networks

- For a single training datum, computing the gradient w.r.t. the weight matrices is also a recursive procedure:
    - Remember: $\mathbf{z}^{(4)} = W^{(4)}\mathbf{h}^{(3)} + b^{(4)}, \quad \hat{y} = f\left(\mathbf{z}^{(4)}\right)$
    - Back-propagation is similar to running the neural network backwards using the transpose of the weight matrices

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}} = \frac{\partial \hat{y}}{\partial \mathbf{z}^{(4)}} \frac{\partial \mathcal{L}}{\partial \hat{y}}, \quad \frac{\partial \mathcal{L}}{\partial W^{(4)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}} \mathbf{h}^{(3)T}$$

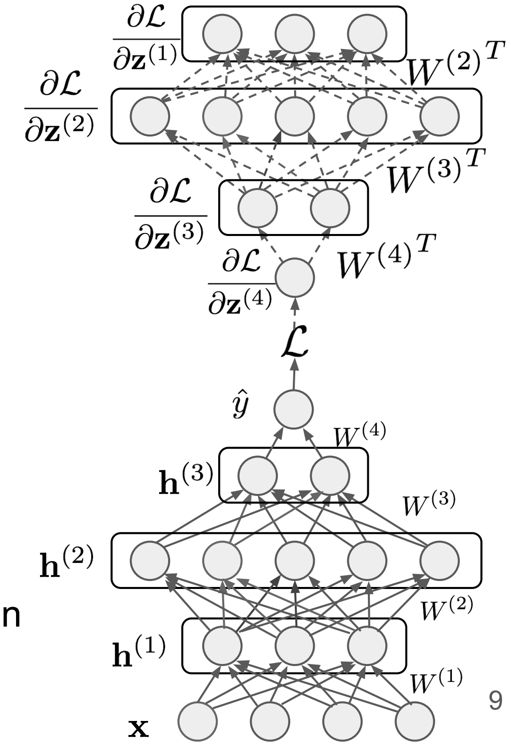
$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} = \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{z}^{(3)}} \left(W^{(4)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}} \right), \quad \frac{\partial \mathcal{L}}{\partial W^{(3)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \mathbf{h}^{(2)T}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}} \left(W^{(3)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} \right), \quad \frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} \mathbf{h}^{(1)T}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{z}^{(1)}} \left(W^{(2)T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} \right), \quad \frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \mathbf{x}^{T}$$

back-propagation

What about the expression for the bias units?



$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}}$
$W^{(2)T}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}}$
$W^{(3)T}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}}$
$W^{(4)T}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}}$
$\mathcal{L}$
$\hat{y}$
$W^{(4)}$
$\mathbf{h}^{(3)}$
$W^{(3)}$
$\mathbf{h}^{(2)}$
$W^{(2)}$
$\mathbf{h}^{(1)}$
$W^{(1)}$
$\mathbf{x}$