Business Analytics
Neural Networks

Prof. Bichler

Decision Sciences & Systems

Department of Informatics

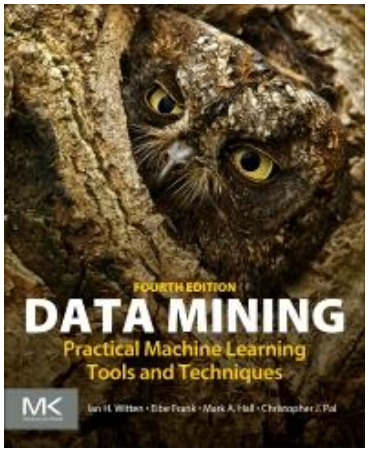
Technische Universität München

Course Content

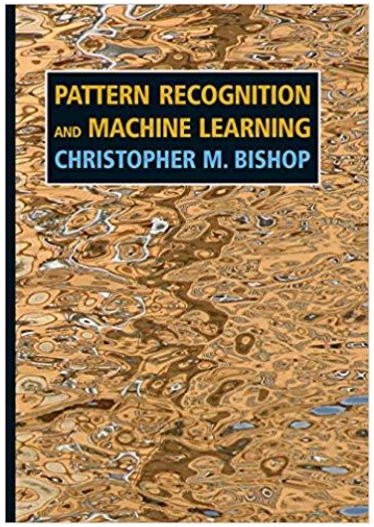
- Introduction
- Regression Analysis
- Regression Diagnostics
- Logistic and Poisson Regression
- Naive Bayes and Bayesian Networks
- Decision Tree Classifiers
- Data Preparation and Causal Inference
- Model Selection and Learning Theory
- Ensemble Methods and Clustering
- High-Dimensional Problems
- Association Rules and Recommenders
- **Neural Networks (gradient descent, backpropagation)**

Recommended Literature

- **Data Mining: Practical Machine Learning Tools and Techniques**
  - Ian H. Witten, Eibe Frank, Mark Hall, Christopher Pal
  - http://www.cs.waikato.ac.nz/ml/weka/book.html
  - Section: 7, 10


FOURTH EDITION
DATA MINING
Practical Machine Learning Tools and Techniques
MK
Ian H. Witten · Eibe Frank · Mark A. Hall · Christopher J. Pal

- **Pattern Recognition and Machine Learning**
  - Christopher M. Bishop
  - Section: 5



Training Neural Networks

- Each *training example* has the form $\langle x_n, y_n \rangle$, were x_n is the vector of inputs, and y_n is the desired corresponding output vector.
- A *loss function* or *cost function* is a function that maps values of one or more variables onto a real number intuitively representing some "cost" associated with the event.
- The *risk function* R associated with hypothesis is defined as the expectation of the loss function.
- *Backpropagation* is a method used in artificial neural networks to calculate a gradient that is needed in the calculation of the weights to be used in the network. It describes *how a **single** training example*, starting from the output neurons, dertermines the goal for the neurons on the next layer and steps backwards recursively.
- An *epoch* is one pass through the training set, with an adjustment via backpropagation to the network weights for each training example. Perform as many epochs of training as needed to reduce the classification error to the required level.

Reminder

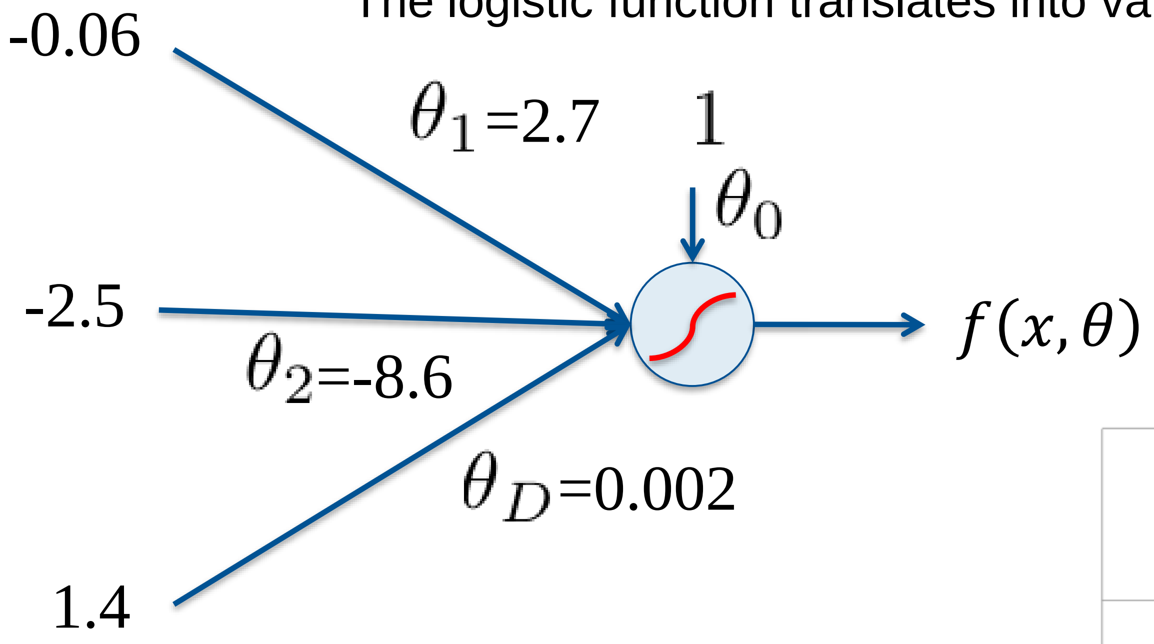
Activation function:

$$f(x,\theta) = g(\theta^T x)$$

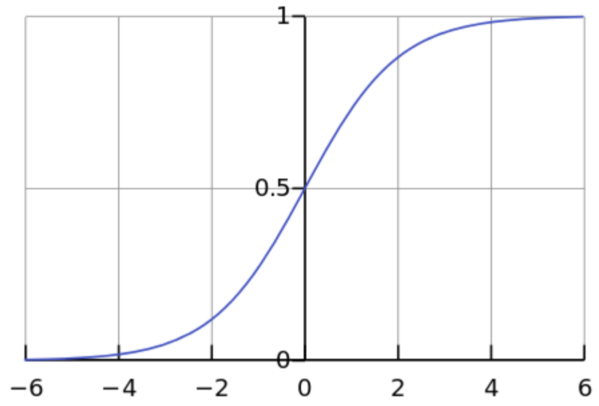
bias

$$\theta^T x = -0.06\times2.7 + 2.5\times8.6 + 1.4\times0.002 + \underset{\uparrow}{4} = 25.34$$

The logistic function translates into values 0 to 1.



$$g(z) = (1 + exp(-z))^{-1}$$

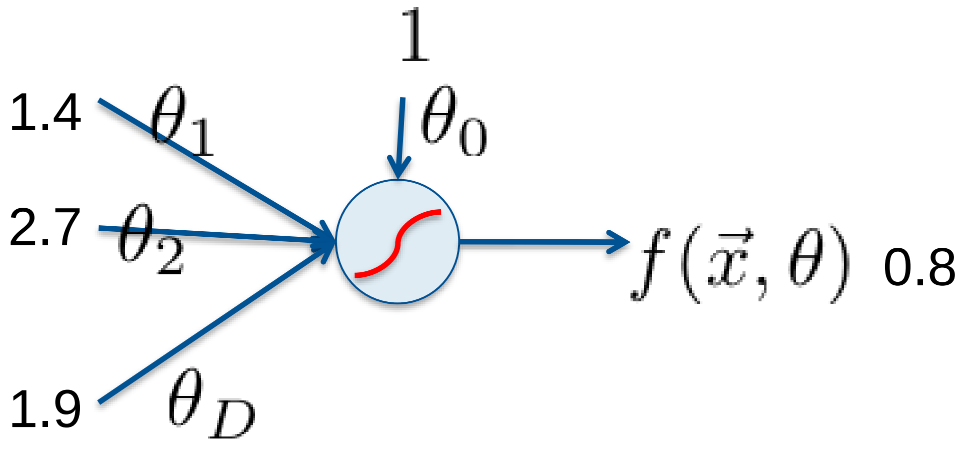


Example

Initialize with random weights
Present a training pattern
Feed it through to get output

Training data set

Fields			*class*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

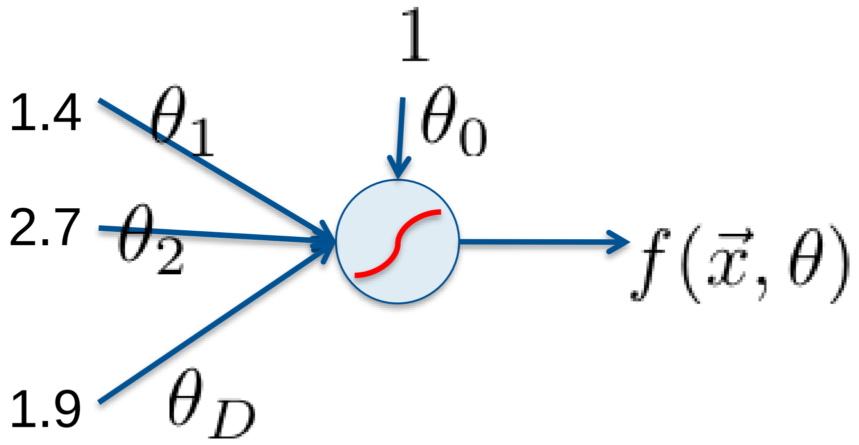


Example

Initialize with random weights
Present a training pattern
Feed it through to get output

Training data set

Fields			*class*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			



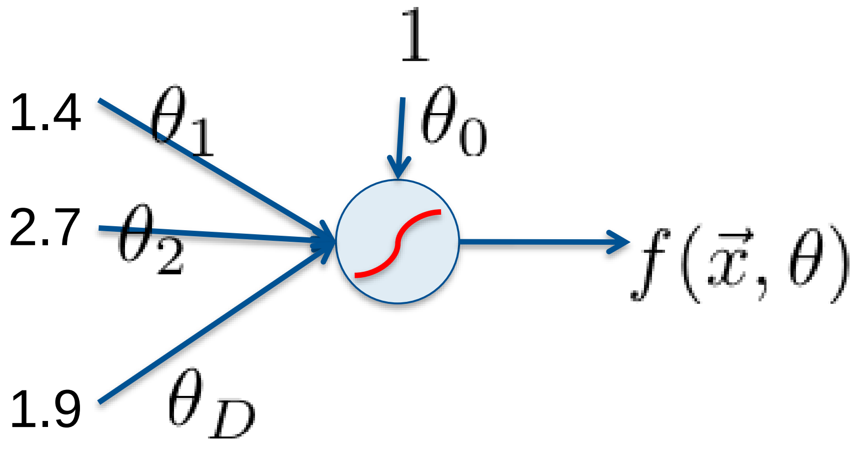
0.8
Correct 0
Error 0.8^2

Example

Initialize with random weights
Present a training pattern
Feed it through to get output
Adjust weights θ via backpropagation

Training data set

Fields			*class*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0
etc …			

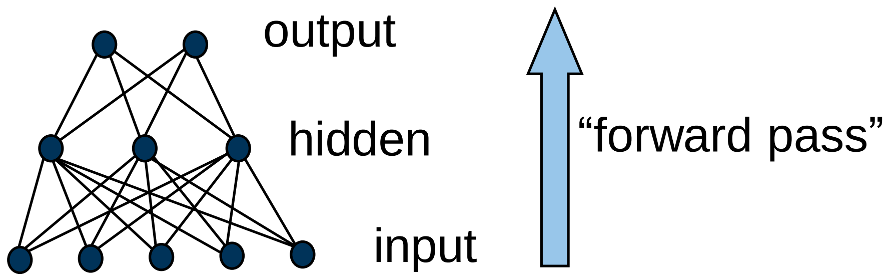


0.8
Correct 0
Error 0.8^2

Multi-Layer Feed-Forward Networks

- Multi-layer networks can represent arbitrary non-linear functions.

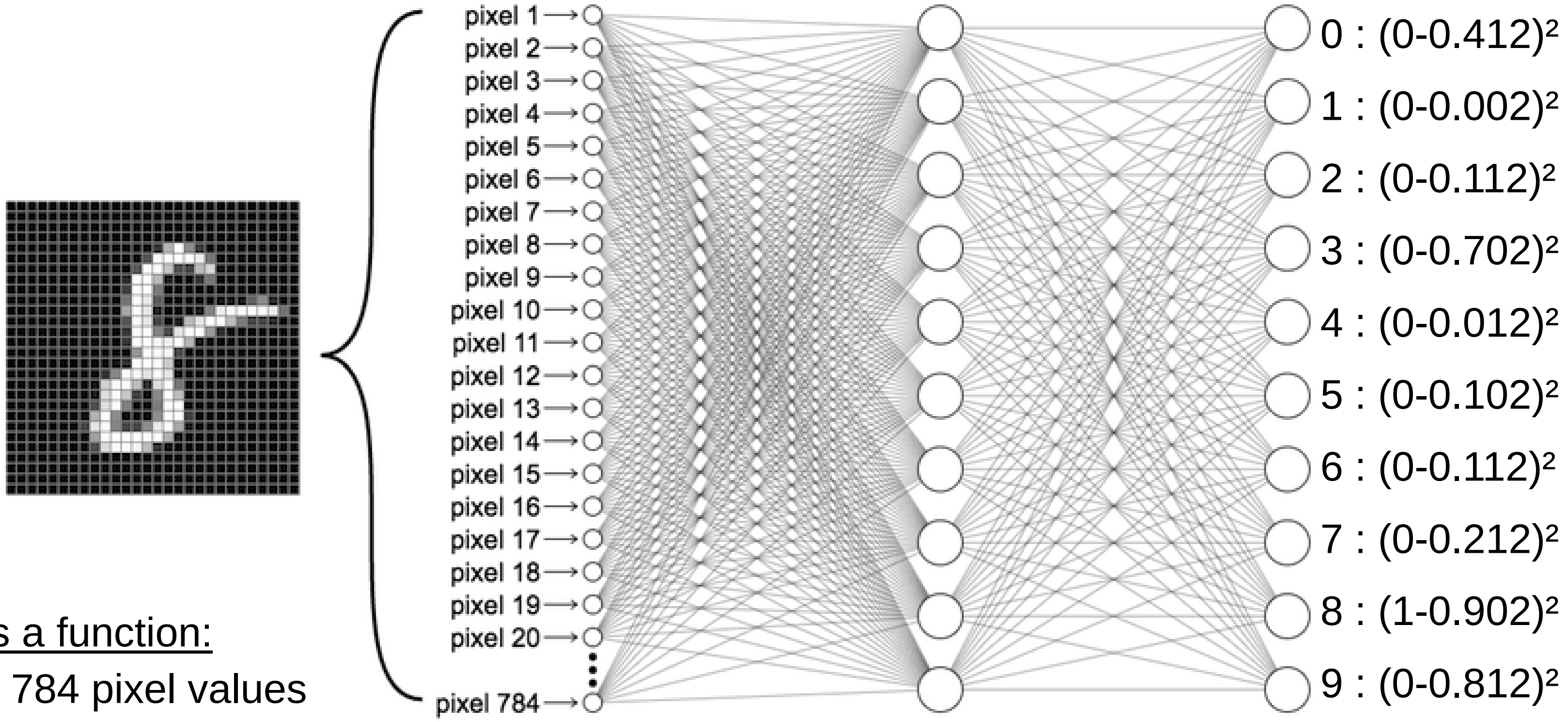
- A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward.



- Multi-layer networks have thousands of weights and biases that need to be adapted. The weights determine the overall function computed.
- Weights are updated via backpropagation (i.e. gradient decent)

Loss Function

Use a *loss function* (aka. cost function) to compute the average (misclassification) cost in the example below adds up the squared error for a single training example. If we average over all training examples, this is referred to as the *empirical risk function (total cost function)*.



0 : $(0-0.412)^2$
1 : $(0-0.002)^2$
2 : $(0-0.112)^2$
3 : $(0-0.702)^2$
4 : $(0-0.012)^2$
5 : $(0-0.102)^2$
6 : $(0-0.112)^2$
7 : $(0-0.212)^2$
8 : $(1-0.902)^2$
9 : $(0-0.812)^2$

<u>NN as a function:</u>

Input: 784 pixel values

Output: 10 numbers

Parameters: 784*10+10*10=7940 weights plus 20 biases

Empirical Risk Minimization

Minimizing the **empirical risk function $\boldsymbol{R}(\boldsymbol{\theta})$**, which is modeling ***expected loss*** (as we don't know the true distribution of data). This means, the empirical risk $\boldsymbol{R}(\boldsymbol{\theta})$ is the average loss over the training data.

$$R(\theta) = \frac{1}{N}\sum_n L(y_n, f(x_n)) = \frac{1}{2N}\sum_n \left(y_n - g(\theta^T x_n)\right)^2$$

derivative of $f(z)^2 \Rightarrow 2f(z)f'(z)$ (chain rule)

$$\nabla_\theta R = \frac{1}{2N}\sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$

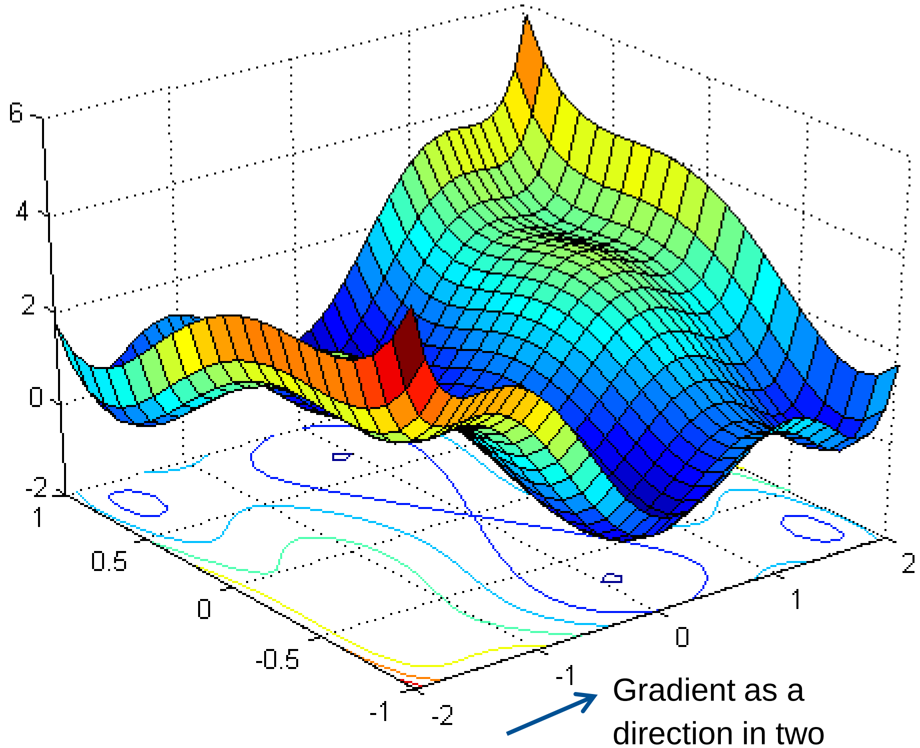
$$g(z) = (1 + exp(-z))^{-1}$$

Unfortunately, there is no "closed-form" solution. We can backpropagate the error for every training example (there are ways to do this more effectively).

To backpropagate a single training example, we use *gradient descent*.

Minimizing a Loss Function in a Nutshell

We use the gradient descent algorithm to adapt the weights such that the loss function is minimized. For this we use the steepest descent of a function.
The negative gradient shows the local shift in weights of the NN that helps most in minimizing the loss function.



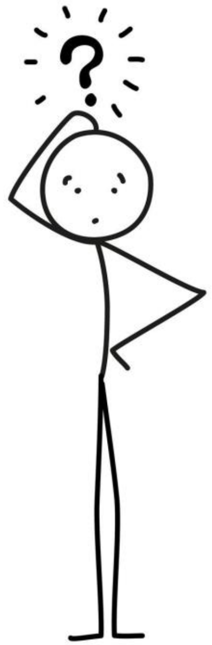
Gradient as a direction in two input dimensions

<u>Loss function:</u>
Input: 7960 parameters (weights + biases)
Output: scalar (cost)

Adapting the weights and biases for all training examples via gradient descent results in the negative gradient of the empirical risk function.

=> *Learning = minimizing the loss function!*

What is the mathematics behind loss minimization?



Multidimensional Optimization Problems

Goal, given any function $f: \mathbb{R}^d \to \mathbb{R}$, find

$$x^* = \underset{x \in \mathbb{R}^d}{\operatorname{argmin}} f(x)$$

- If $f: \mathbb{R}^d \to \mathbb{R}$ is differentiable, then its **gradient** at position x is defined as

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_d} \end{pmatrix}$$

- The vector $\nabla f(x)$ of the partial derivatives $\frac{\partial f(x)}{\partial x_i}$ at x is the **gradient** of f, which points to the steepest ascent.
- Thus $-\nabla f(x)$ is a **descent direction**, i.e. (at least) for small $\alpha > 0$: $f\big(x - \alpha \nabla f(x)\big) \leq f(x)$

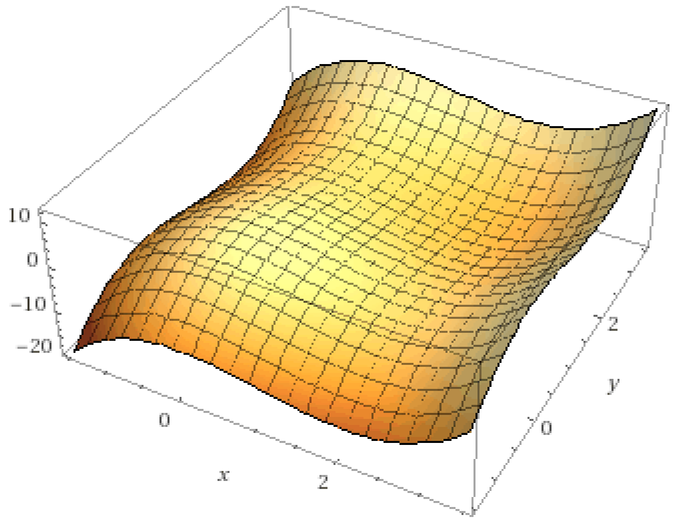
Example Gradient

$f(x,y) = x^3 - 3x^2 + y^3 - 3y^2$

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 3x^2 - 6x \\ 3y^2 - 6y \end{pmatrix}$$

Stationary points are at $3x^2 - 6x = 0$ and $3y^2 - 6y = 0$

Four stationary points: $(0,0), (2,0), (0,2), (2,2)$



(Source: Wolfram Alpha)

Extreme Values

If $X \subseteq \mathbb{R}^d$ is open and $f: X \to \mathbb{R}$ is a twice continuously differentiable function of multiple variables $x = (x_1, \dots, x_d)$, then

- $\nabla f(x') = 0$ is a necessary condition for x' to be a local minimum or maximum
- The symmetric matrix $H(x) = \left(\frac{\partial f^2(x)}{\partial x_i \partial x_j}\right)_{\mathrm{i,j}}$ is the **Hessian**
- $\nabla f(x') = 0$ and - $H(x')$ is positive definite (or $H(x')$ negativ definite) is a sufficient condition that x' is a local maximum.
    - $H(x')$ negative definite => eigenvalues < 0 => maximum
    - $H(x')$ positive definite => eigenvalues > 0 => minimum
    - $H(x')$ indefinite => eigenvalues < 0 and > 0 => saddle point
    - $H(x')$ positive semidefinite => eigenvalues $\geq$ 0 => inconclusive

Principal Minors

Let A be a symmetric $n \times n$ matrix. We can determine the definiteness of A by computing its eigenvalues. Another method is to use the *principal minors*.

Definition:
A minor of A of order k is *principal* if it is obtained by deleting $n - k$ rows and the $n - k$ columns with the same numbers. The *leading principal minor* of A of order k is the minor of order k obtained by deleting the last $n - k$ rows and columns.

Example:
Let $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ be a symmetric 2×2 matrix. Then the leading principal minors are $D_1 = a$ and $D_2 = ac - b^2$.

Rules to determine if a matrix A is positive or negative definite:
(a) A is positive definite $\Leftrightarrow D_k > 0$ for all leading principal minors.
(b) A is negative definite $\Leftrightarrow (-1)^k D_k > 0$ for all leading principal minors
(c) If none of the leading principal minors is zero, and neither (a) nor (b) holds, then the matrix is indefinite.

Example Hessian

$$f(x,y) = x^3 - 3x^2 + y^3 - 3y^2$$

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} = \begin{pmatrix} 3x^2 - 6x \\ 3y^2 - 6y \end{pmatrix}$$

$$H(x,y) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}$$

$$= \begin{pmatrix} 6x - 6 & 0 \\ 0 & 6y - 6 \end{pmatrix}$$

Rules

(a) A is positive definite $\Leftrightarrow D_k > 0$ for all leading principal minors.

(b) A is negative definite $\Leftrightarrow$ $(-1)^k D_k > 0$ for all leading principal minors

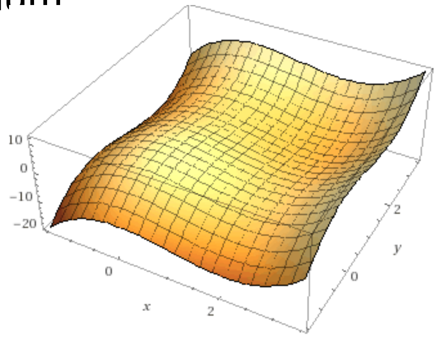
(c) If none of the leading principal minors is zero, and neither (a) nor (b) holds, then the matrix is indefinite.

at (0,0): $\begin{pmatrix} -6 & 0 \\ 0 & -6 \end{pmatrix}$ is negative definite ($D_1 = -6, D_2 = 36$), so a maximum

at (2,0): $\begin{pmatrix} 6 & 0 \\ 0 & -6 \end{pmatrix}$ is indefinite ($D_1 = 6, D_2 = -36$), so a saddle point.

at (0,2): $\begin{pmatrix} 6 & 0 \\ 0 & -6 \end{pmatrix}$ is indefinite ($D_1 = -6, D_2 = -36$), so a saddle point.

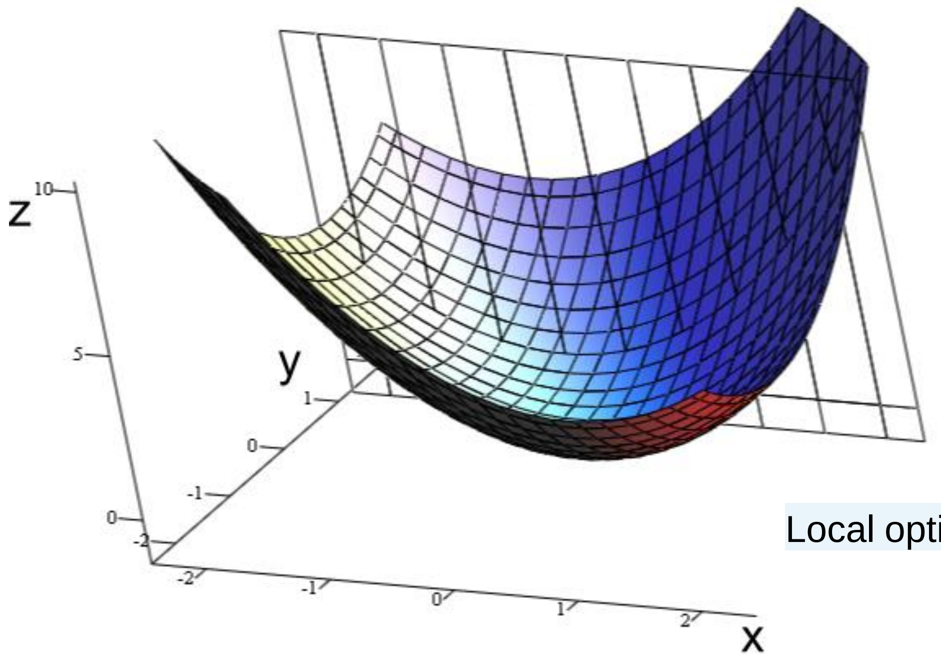
at (2,2): $\begin{pmatrix} 6 & 0 \\ 0 & 6 \end{pmatrix}$ is positive definite ($D_1 = 6, D_2 = 36$), so a minimum.



Convexity in Multiple Dimensions

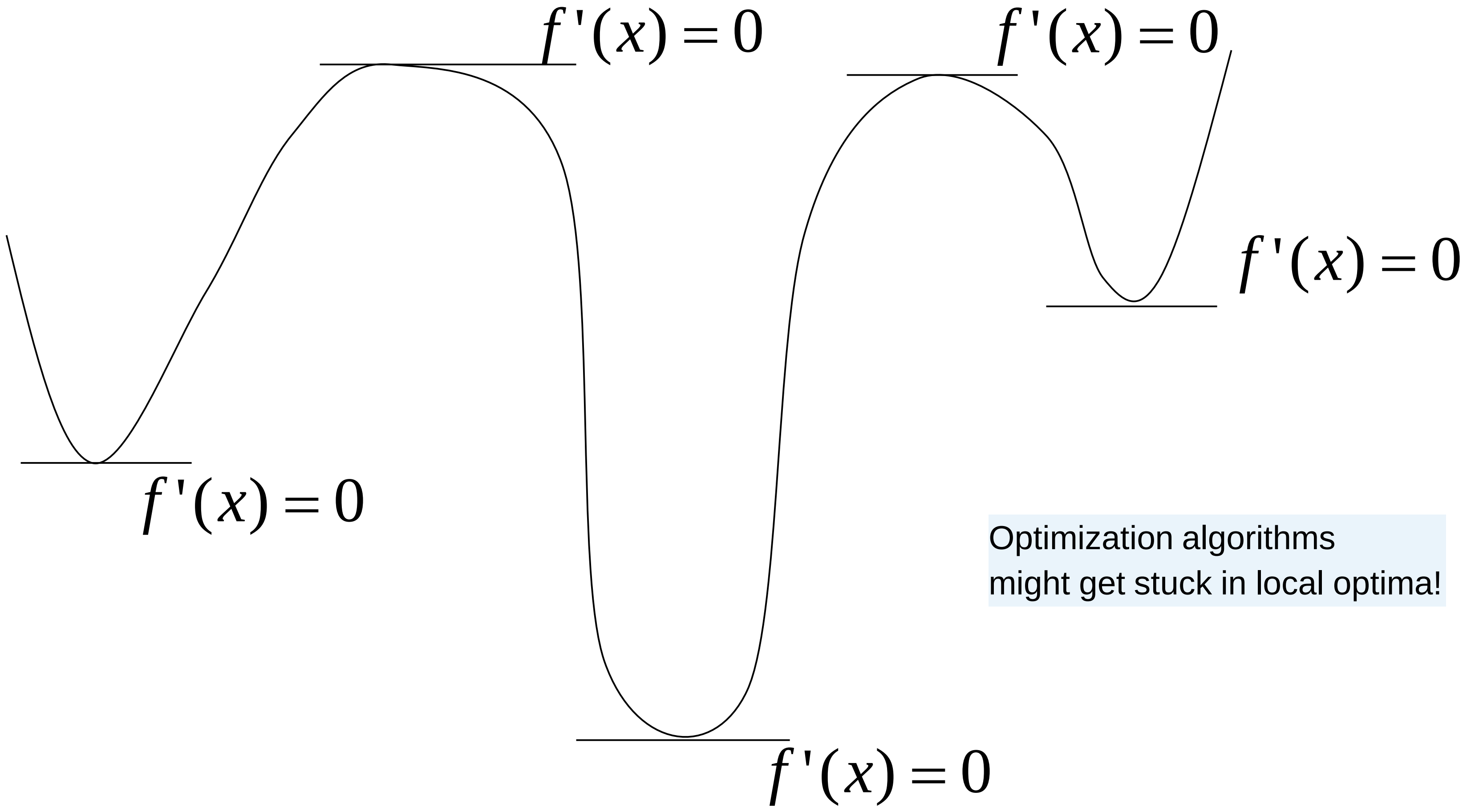
- A function $f: \mathbb{R}^d \to \mathbb{R}$ is called **convex** iff

$$\forall \lambda \in (0,1), x, y \in \mathbb{R}^d:$$
$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$$



Local optima are globally optimal!

Neither Concave nor Convex



Optimization algorithms
might get stuck in local optima!

Local Minima of Risk Functions

- The risk function of a neural network is in general neither convex nor concave. This means that the Hessian is neither positive semidefinite, nor negative semidefinite.
- The probability of finding a "bad" (high value) local minimum is non-zero for small-size neural networks and decreases quickly with network size.
- For large networks, most local minima are equivalent and yield similar performance on a test set (see https://arxiv.org/pdf/1412.0233v3.pdf)
- If there's no hidden layers the logistic neural network is convex, just like logistic regression.
- Neural networks with linear activation functions and square loss yield a convex problem.

Gradient Ascent

Input: Concave, continuously differentiable function $f\colon \mathbb{R}^d \to \mathbb{R}$,
feasible start point $x^{(1)} \in \mathbb{R}^d$, and parameter $\varepsilon > 0$

```
$k = 1$
While($\|\nabla f(x^{(k)})\| \geq \varepsilon$){
    • Choose step size $\alpha > 0$,
      e.g. compute $\alpha^* > 0$, to maximize $f\left(x^{(k)} + \alpha \nabla f\left(x^{(k)}\right)\right)$. (“line search”)
    • Set $x^{(k+1)} = x^{(k)} + \alpha^* \nabla f\left(x^{(k)}\right)$
    • $k++$
}
```

Gradient Ascent

$$f(x_1, x_2) = 2(x_1 + x_2) - x_1^2 - 2x_2^2$$

$$\nabla f(x) = \begin{pmatrix} 2(1 - x_1) \\ 2(1 - 2x_2) \end{pmatrix}$$

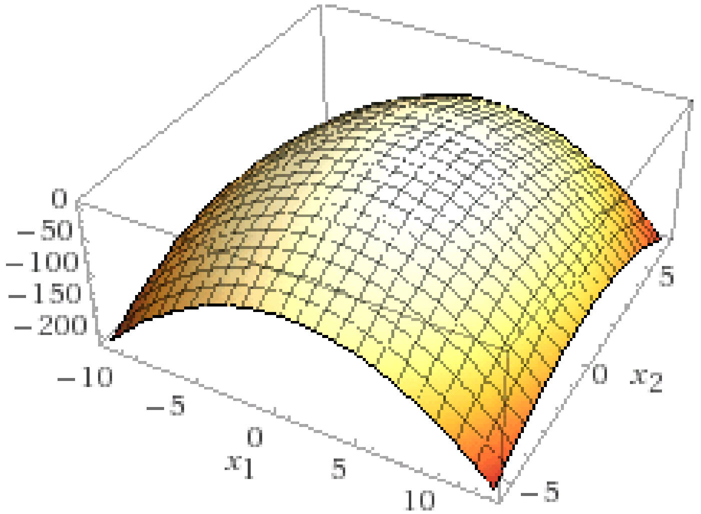
```
$k = 1$
While($\|\nabla f(x^{(k)})\| \geq \varepsilon$){
    • Compute $\alpha^* > 0$, to maximize $f\left(x^{(k)} + \alpha \nabla f(x^{(k)})\right)$.
    • Set $x^{(k+1)} = x^{(k)} + \alpha^* \nabla f\left(x^{(k)}\right)$
    • $k++$
}
```

Start at $x^{(1)} = (0,0), f\left(x^{(1)}\right) = 0, \nabla f\left(x^{(1)}\right) = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$

Maximize $f(0 + 2\alpha, 0 + 2\alpha) = 8\alpha - 12\alpha^2$

→ Maximum is at $\frac{1}{3}$

→ $x^{(2)} = \left(\frac{2}{3}, \frac{2}{3}\right), f\left(x^{(2)}\right) = \frac{4}{3}, \nabla f\left(x^{(2)}\right) = \begin{pmatrix} \frac{2}{3} \\ -\frac{2}{3} \end{pmatrix}$



this is the direction of steepest ascent

Line search is too expensive in NNs and replaced by fixed learning rates.

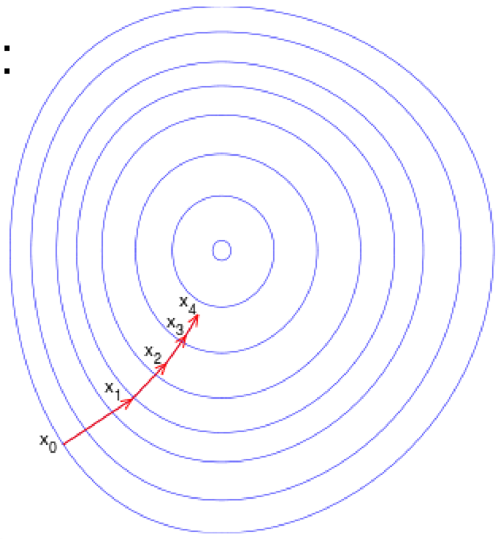
Gradient Descent

Steps:

- Start in x^0, calculate $\nabla f(x^0)$
- Choose a step size / “learning rate” α
- Take a step in the direction opposite of the gradient:

Pseudo code:

```
$k = 1$
While($\|\nabla f(x^{(k)})\| \geq \varepsilon$ ){
    • Compute $\alpha^* > 0$, to minimize $f\left(x^{(k)} - \alpha\nabla f\left(\mathrm{x}^{(\mathrm{k})}\right)\right)$.
    • Set $x^{(k+1)} = x^{(k)} - \alpha^*\nabla f\left(x^{(k)}\right)$
    • $k++$
}
```



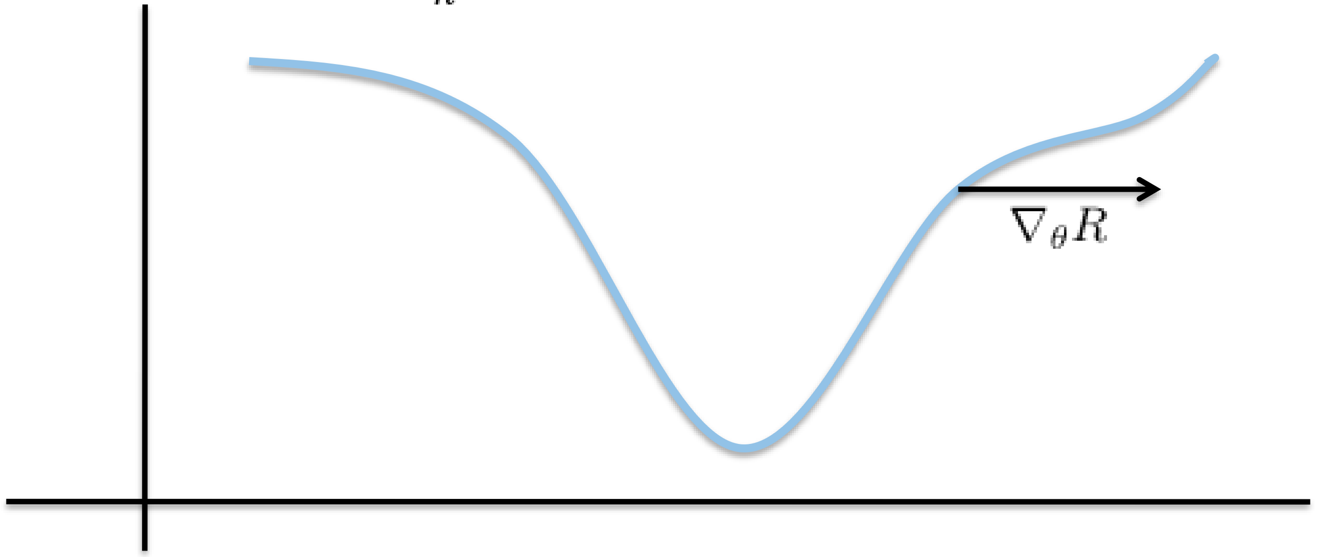
https://en.wikipedia.org/wiki/Gradient_descent

Gradient Descent (1-Dimensional)

Gradient points in the direction of steepest increase

To minimize R, move in the opposite direction

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$

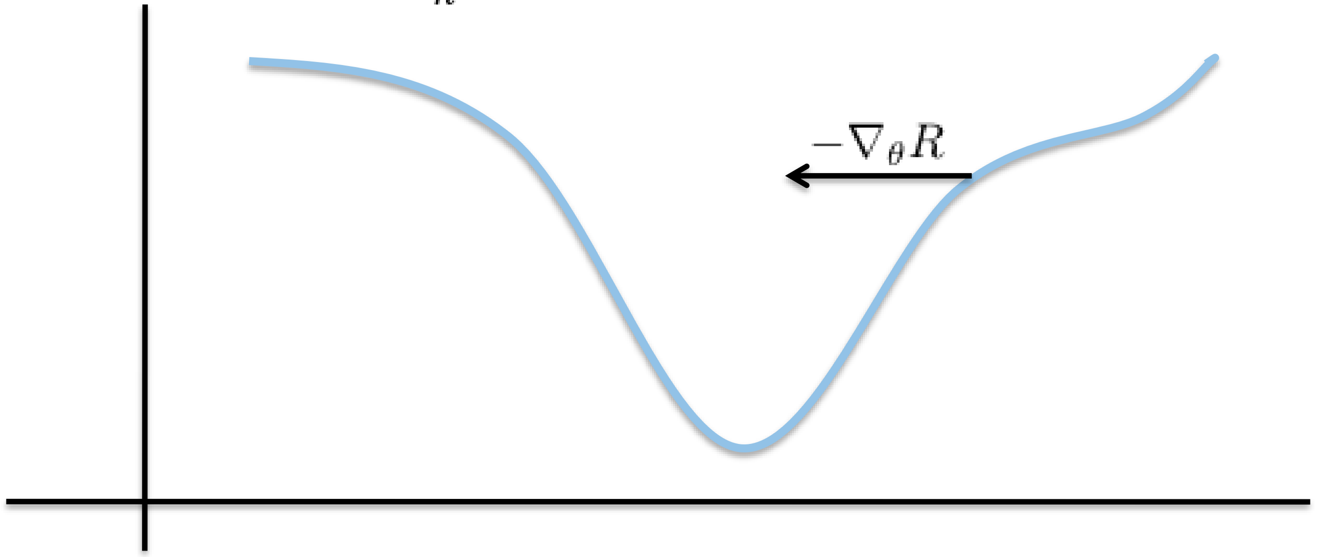


Gradient Descent

Gradient points in the direction of steepest increase

To minimize R, move in the opposite direction

$$\nabla_\theta R = \frac{1}{2N}\sum_n 2(y_n - \mathrm{g}(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

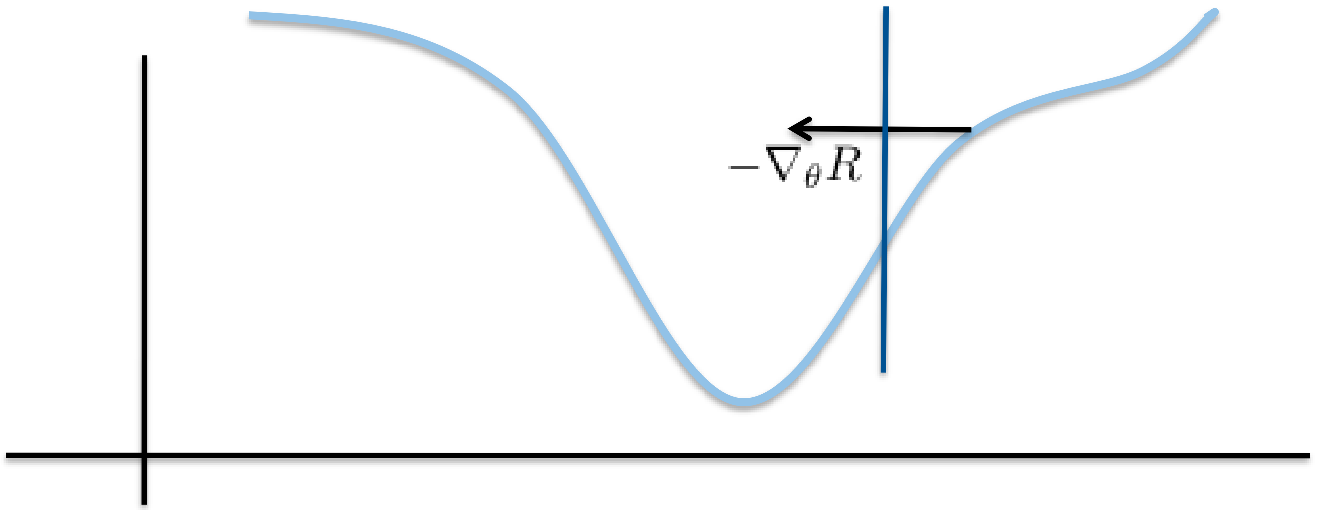
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



Gradient Descent

Initialize randomly

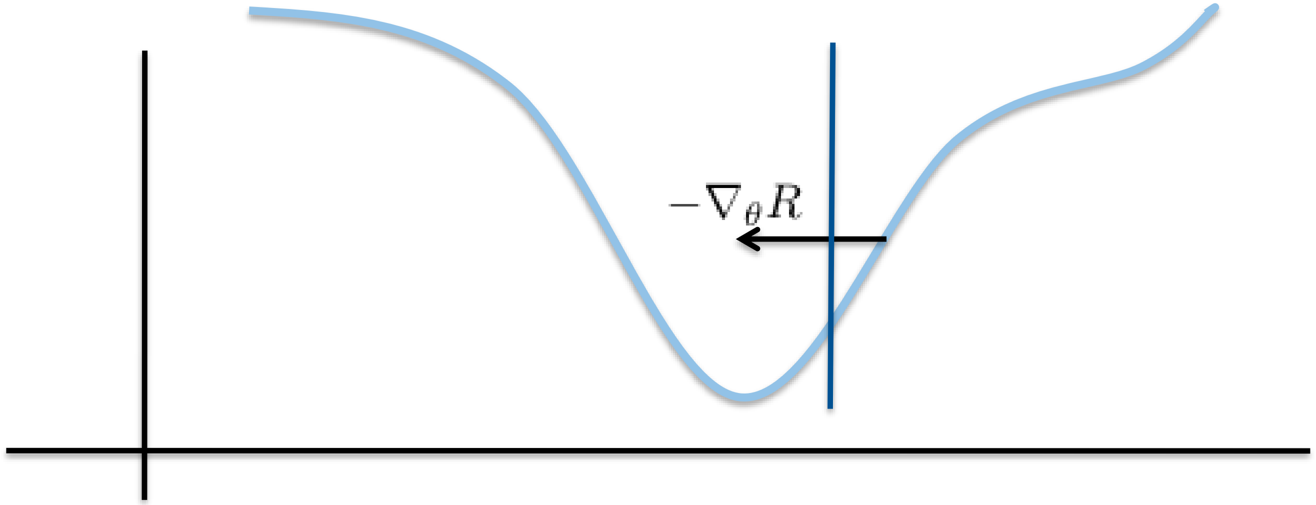
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



Gradient Descent

Initialize randomly

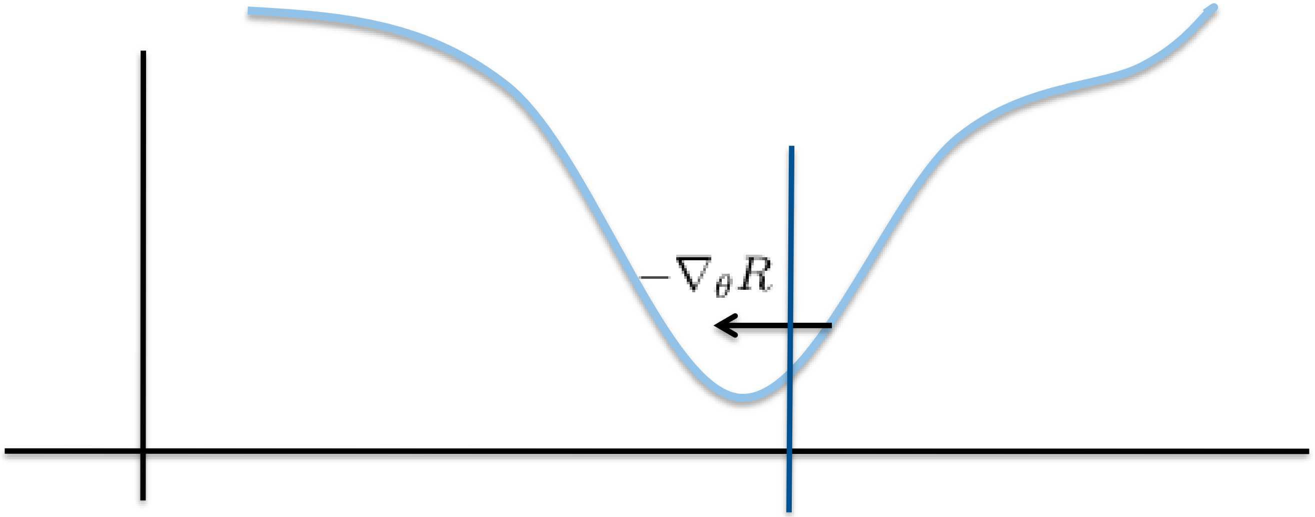
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N}\sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

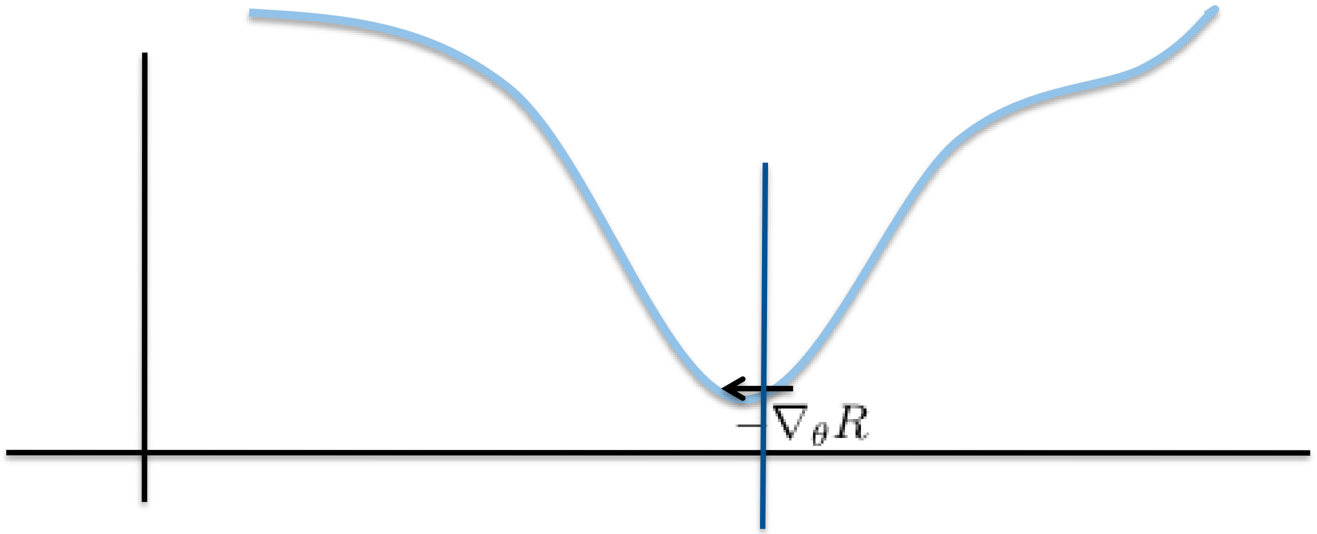
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



Gradient Descent

Initialize randomly

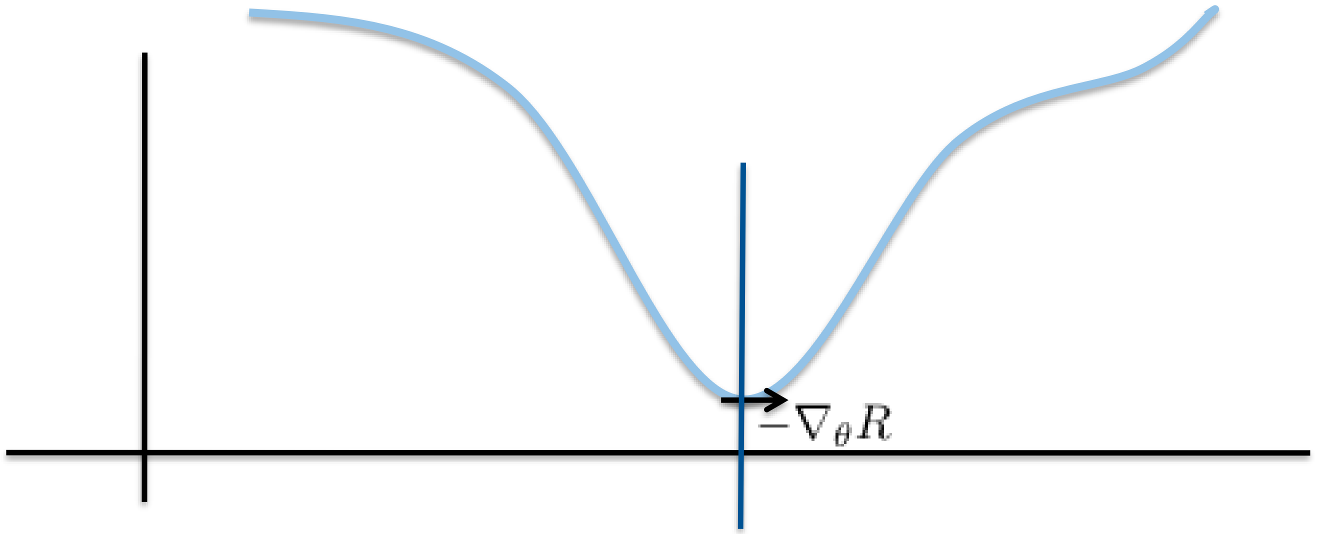
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



Gradient Descent

Initialize randomly

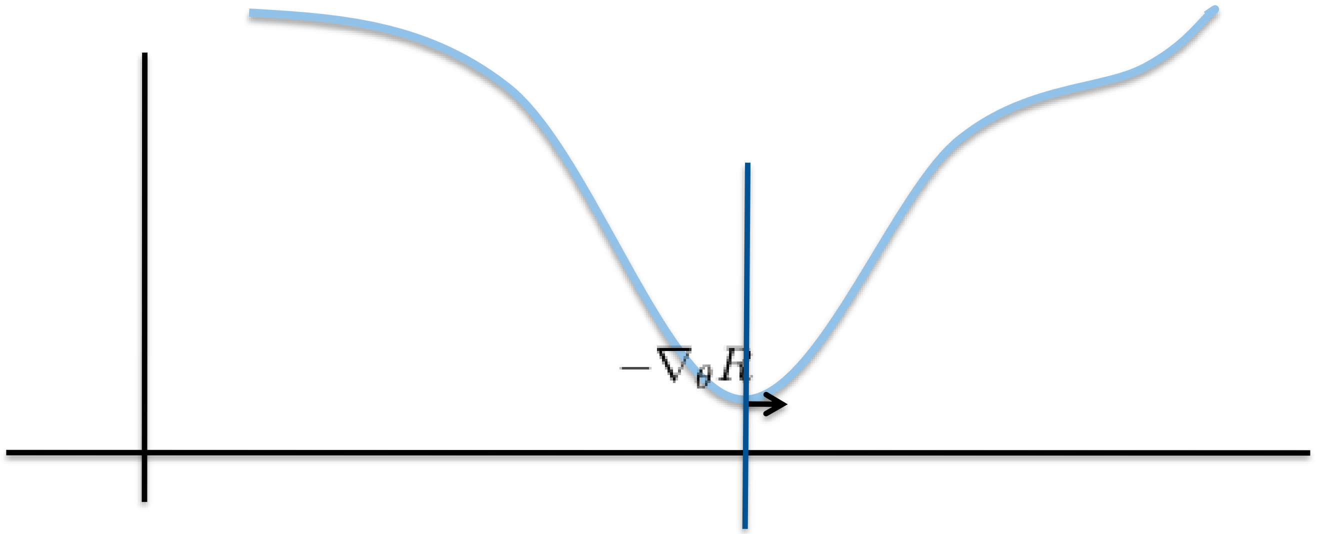
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



Gradient Descent

Initialize randomly

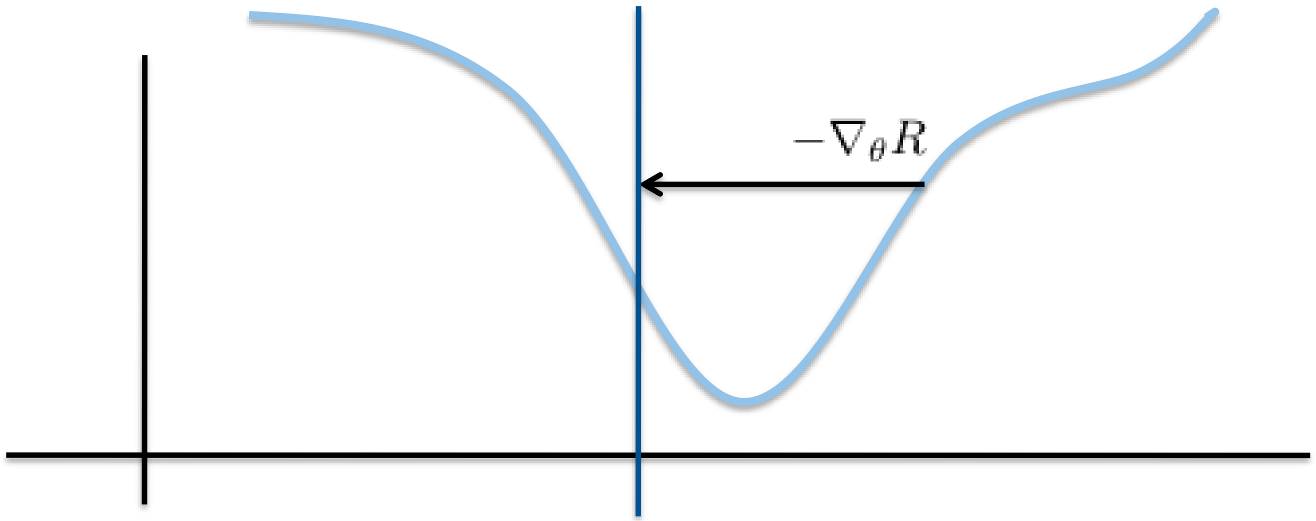
Update with small steps

Can oscillate if α is too large

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



Gradient Descent

Initialize randomly

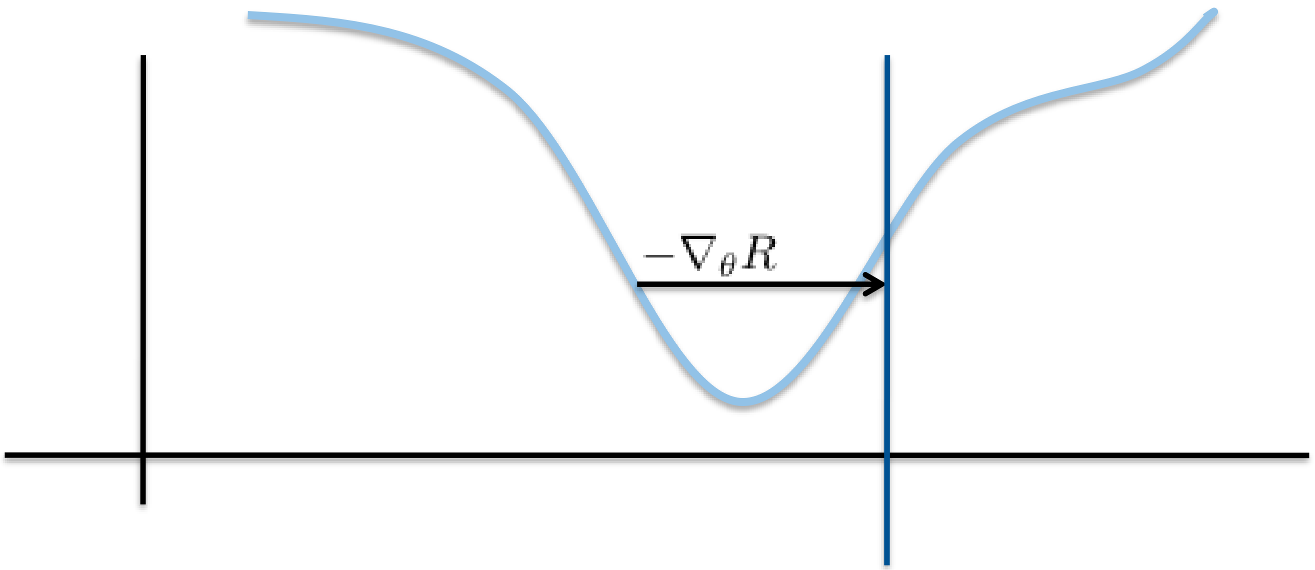
Update with small steps

Can oscillate if α is too large

$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



Gradient Descent

Initialize randomly

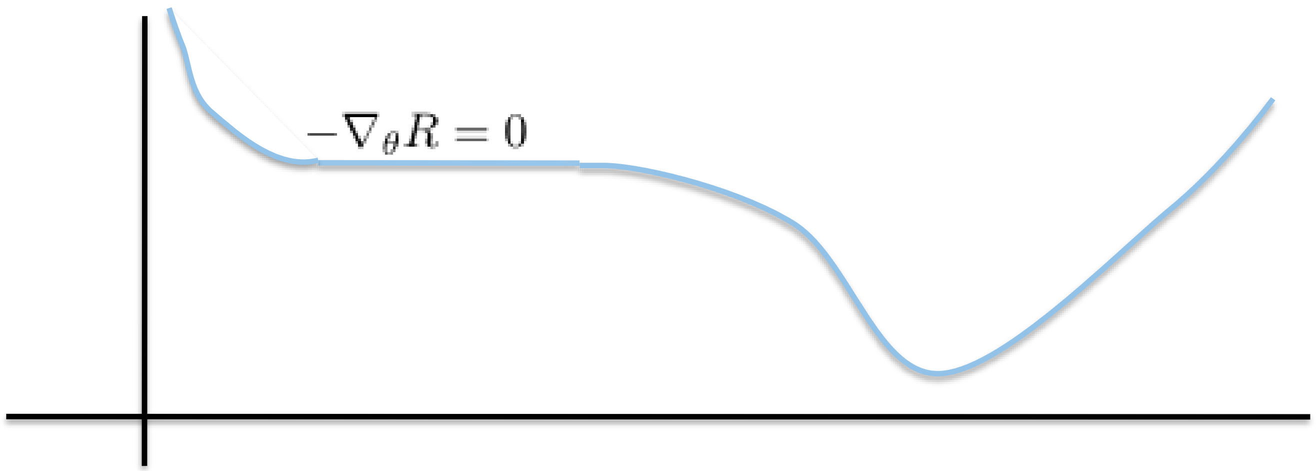
Update with small steps

Can stall if $-\nabla_\theta R$ is ever 0 not at the minimum

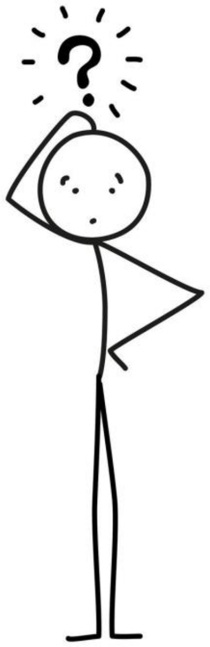
$$\theta_0 = random$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta R \Big|_{\theta_t}$$

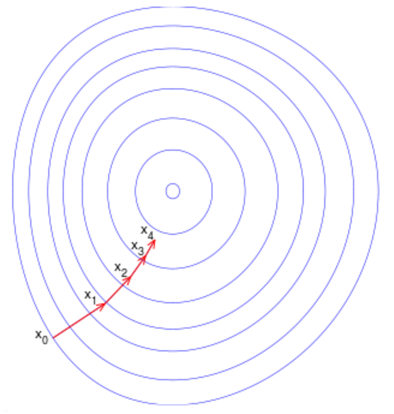
$$\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$$



What is an epoch in the training of a neural network?



Epoch vs. Batch vs. Iteration



- Gradient descent needs to pass through the complete dataset to move one step closer to the optimum value.
  - $\nabla_\theta R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1) g'(\theta^T x_n) x_n = 0$

- one **epoch**: one forward pass and one backward pass of *all* the training examples.

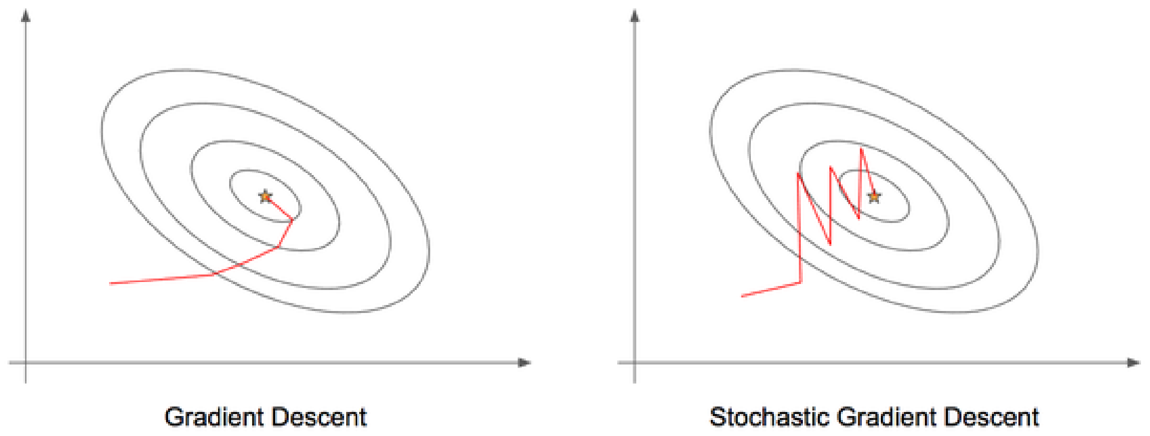
- **batch size**: the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.

- number of **iterations**: number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

- Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Stochastic Gradient Descent

- *Mini-batch Stochastic Gradient Descent* (SGD) replaces the actual gradient by an estimate thereof to reduce the computational cost based on *subsets (batches)* of the entire data set (e.g., split the training set into mini-batches of 1000 samples). Batch size is a hyperparameter when training an NN.



- *Momentum* uses an exponential averaging of gradients to make sudden changes in direction less likely.

Mini-Batches and Momentum Versions

- Typically contain two to several hundred examples
  - For large models the choice may be constrained by resources
  - Batch size often influences the stability and speed of learning; some sizes work particularly well for a given model and data set.

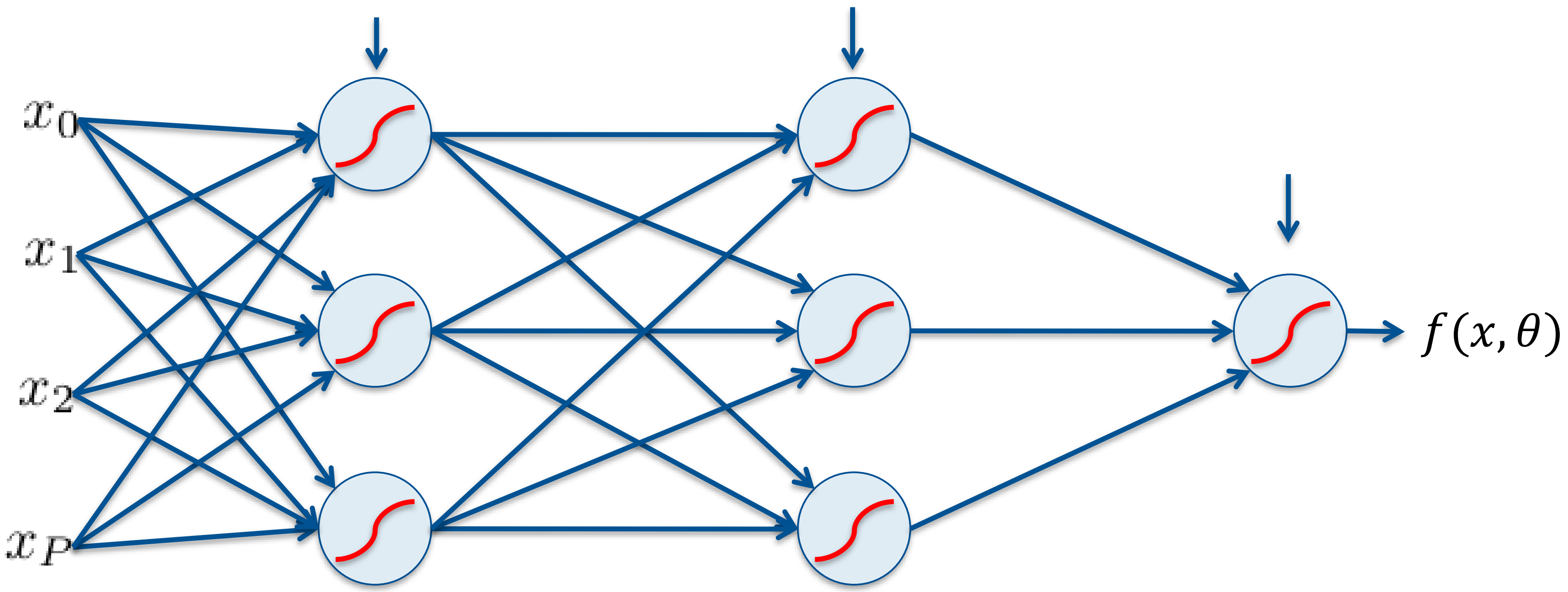
- Sometimes a search is performed over a set of potential batch sizes to find one that works well, before doing a lengthy optimization.

- The mix of class labels in the batches can influence the result
  - For unbalanced data there may be an advantage in pre-training the model using mini-batches in which the labels are balanced, then fine-tuning the upper layer or layers using the unbalanced label statistics.

- In modern Machine Learning, most common optimization algorithms are Stochastic Gradient Descent and (stochastic versions) of momentum methods such as RMSprop, ADAM, AdaDelta …

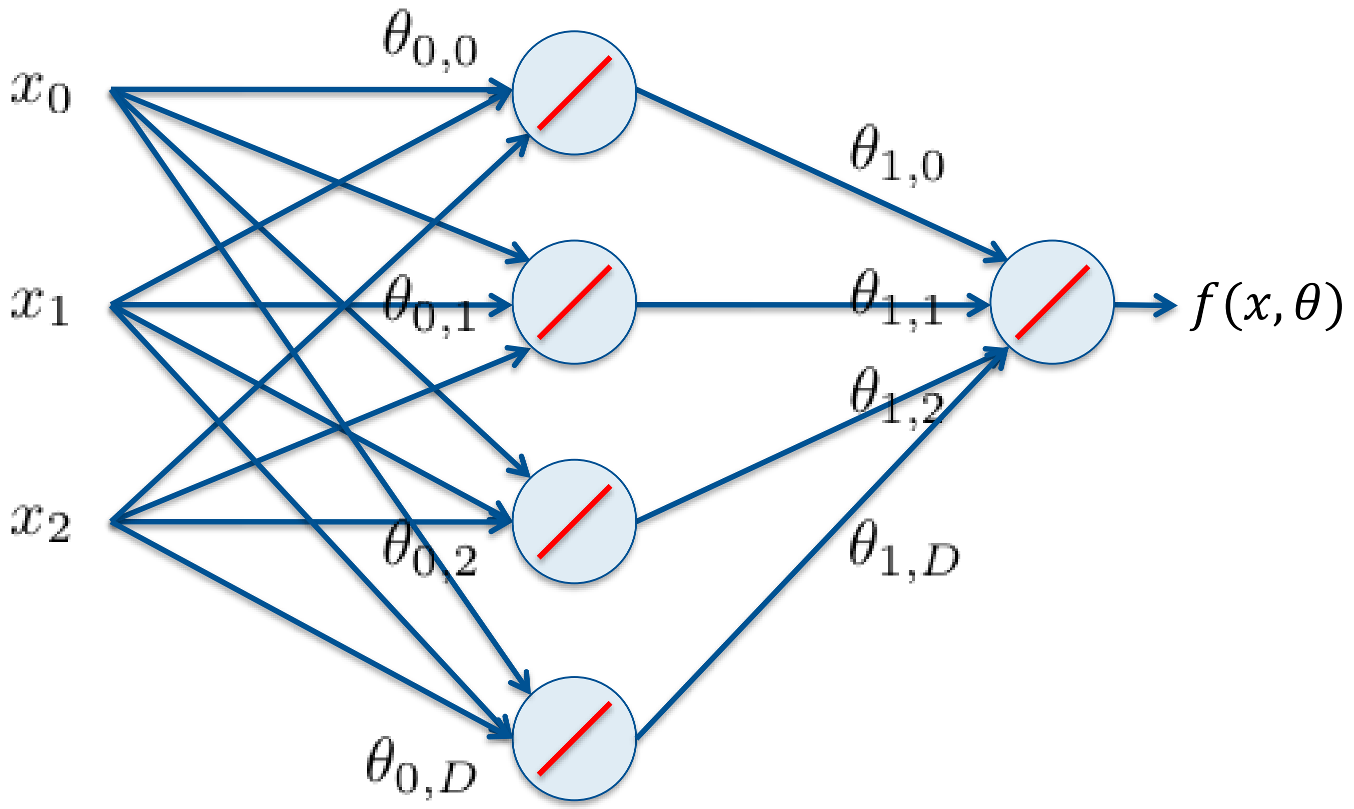
Multilayer Networks

P input variables and N data instances. Cascade neurons together.
The output from one layer is the input to the next.
Each layer has its own sets of weights.



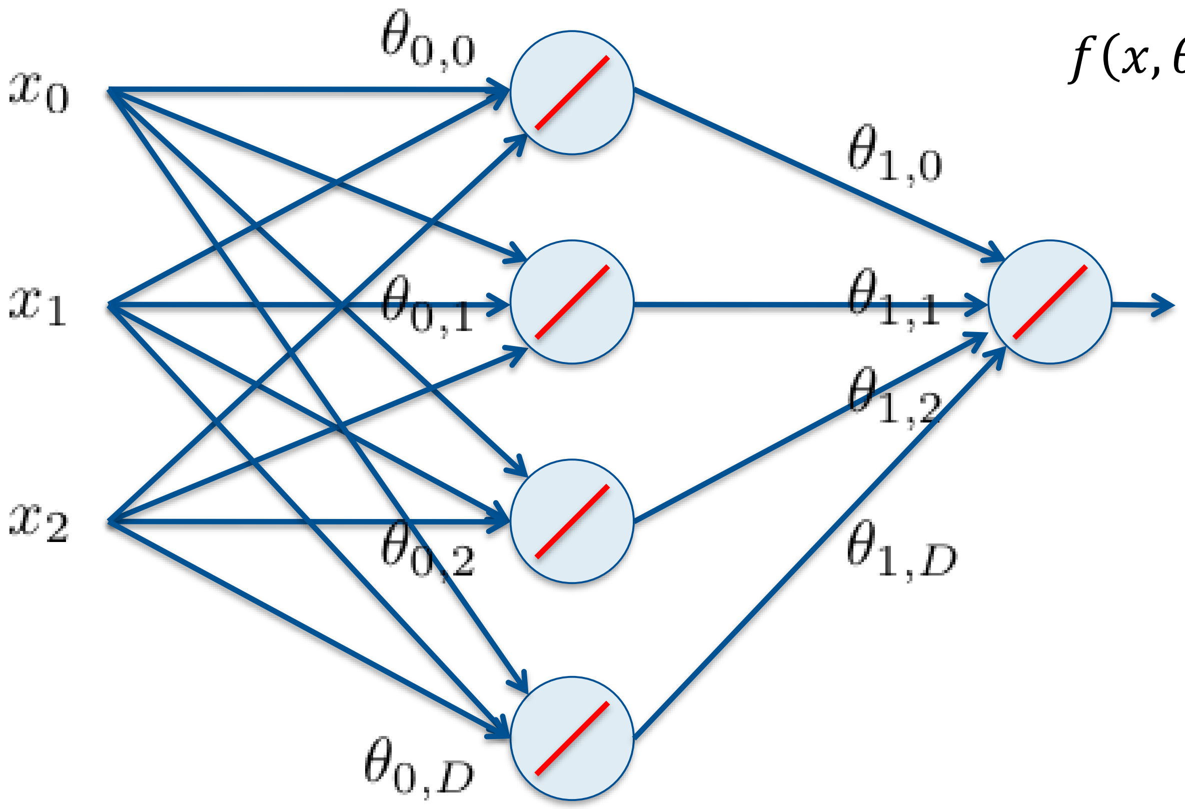
Linear Regression Neural Networks

What happens when we arrange **linear neurons** in a multilayer network?



Linear Regression Neural Networks

The composition of two linear transformations is itself a linear transformation:
A **neural network** with a **linear activation** function is simply a linear regression model.



$$f(x,\theta) = \sum_{i=0}^{D} \theta_{1,i} \sum_{i=n}^{N-1} \theta_{0,i,n} x_n$$

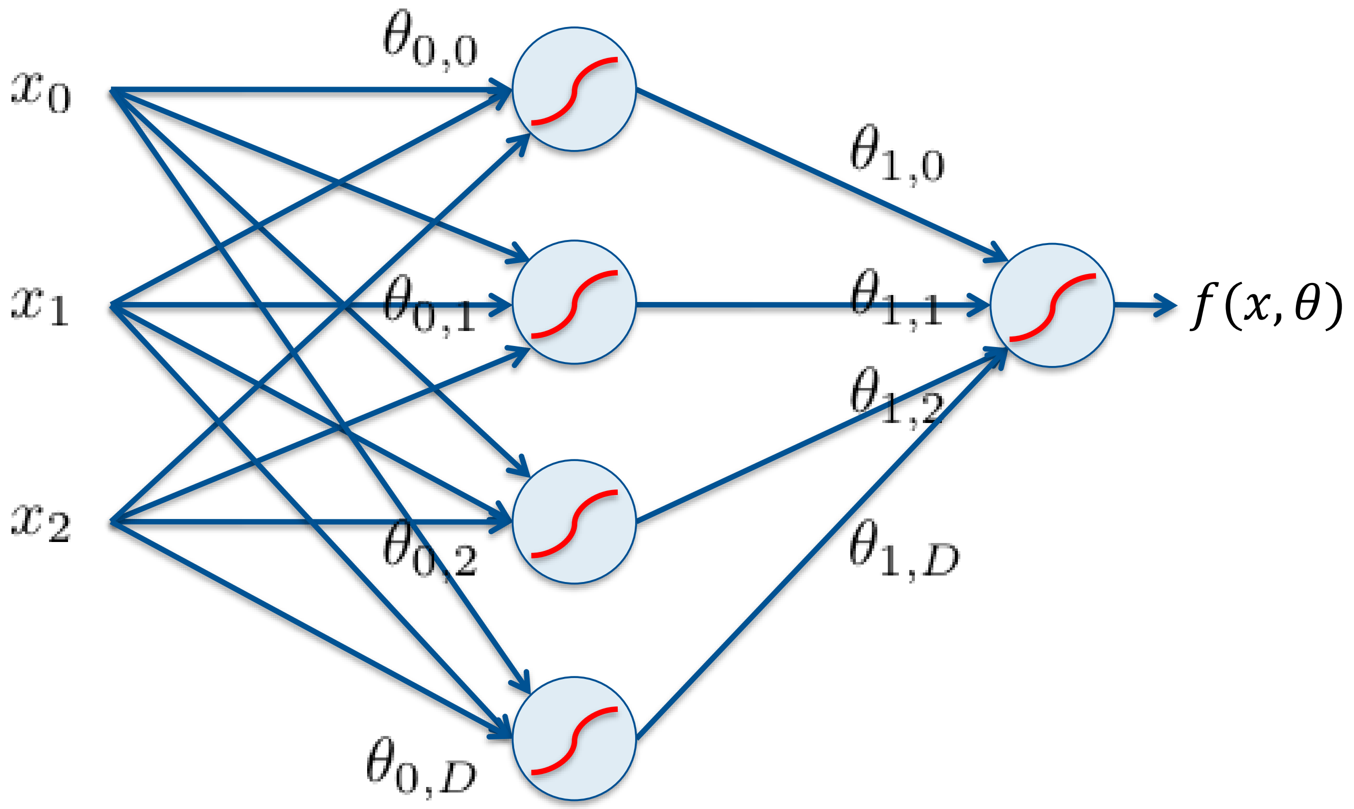
$$f(x,\theta) = \sum_{i=0}^{D} \theta_{1,i} [\theta_{0,i}^T x]$$

$$f(x,\theta) = \sum_{i=0}^{D} [\hat{\theta}_i^T x]$$

Neural Networks

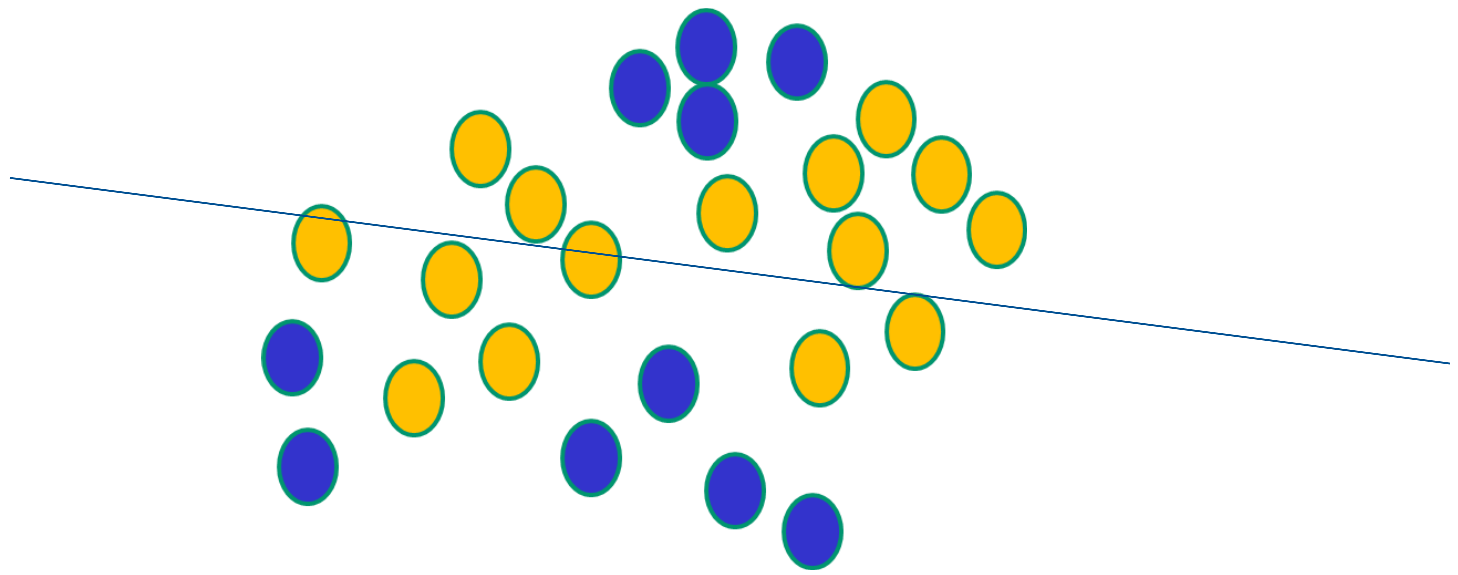
We want to introduce non-linearities to the network.

Non-linearities allow a network to identify complex regions in space

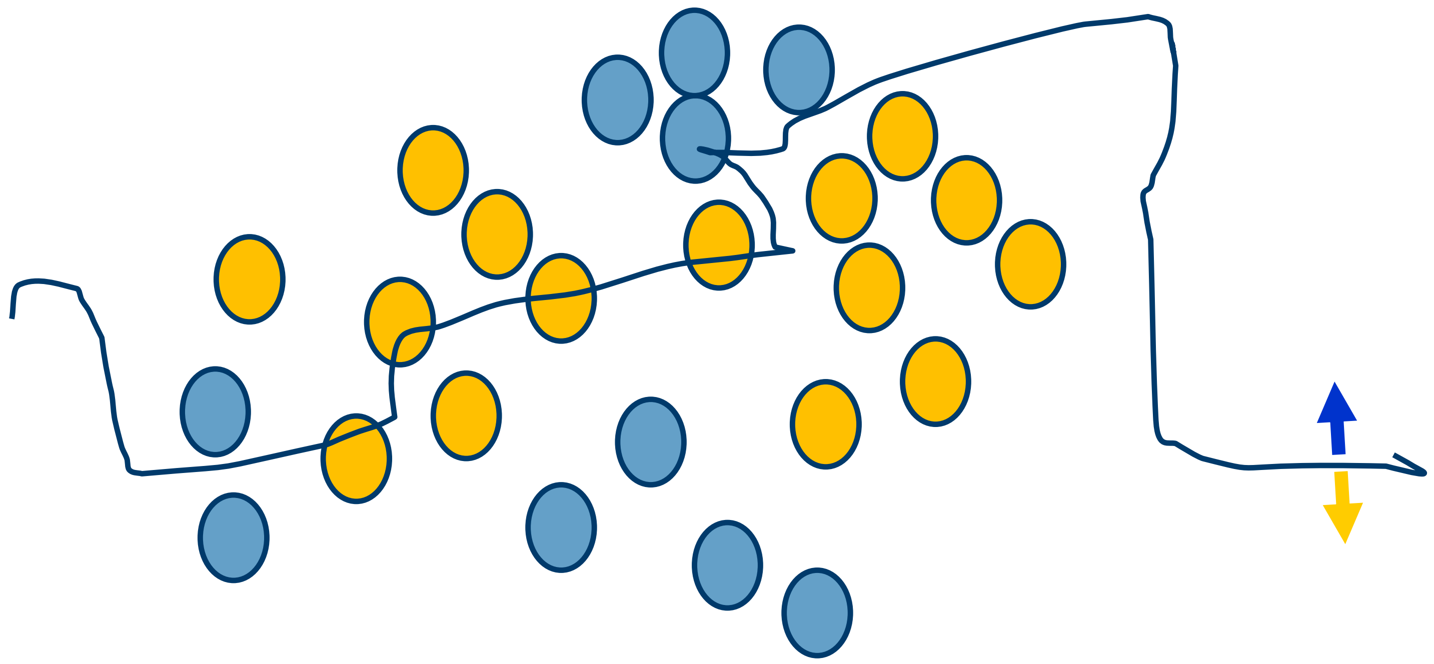


Linearly Separability

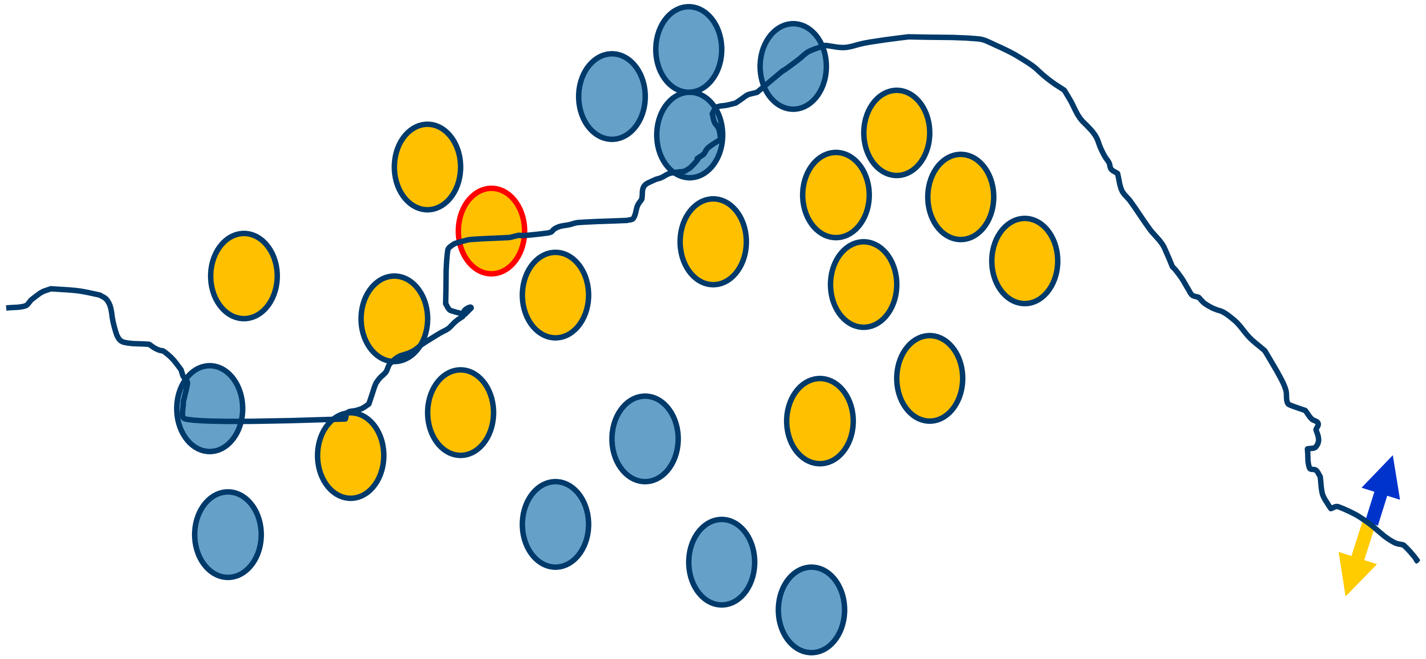
- Two classes of points are **linearly separable**, iff there exists a line such that all the points of one class fall on one side of the line, and all the points of the other class fall on the other side of the line
- If $f(x)$ is linear, the NN can **only** draw straight decision boundaries (even if there are many layers of units)
- Established approach is to use the non-linear, differentiable sigmoidal “logistic” function $f(x)$, that we have seen earlier, which can draw complex boundaries.



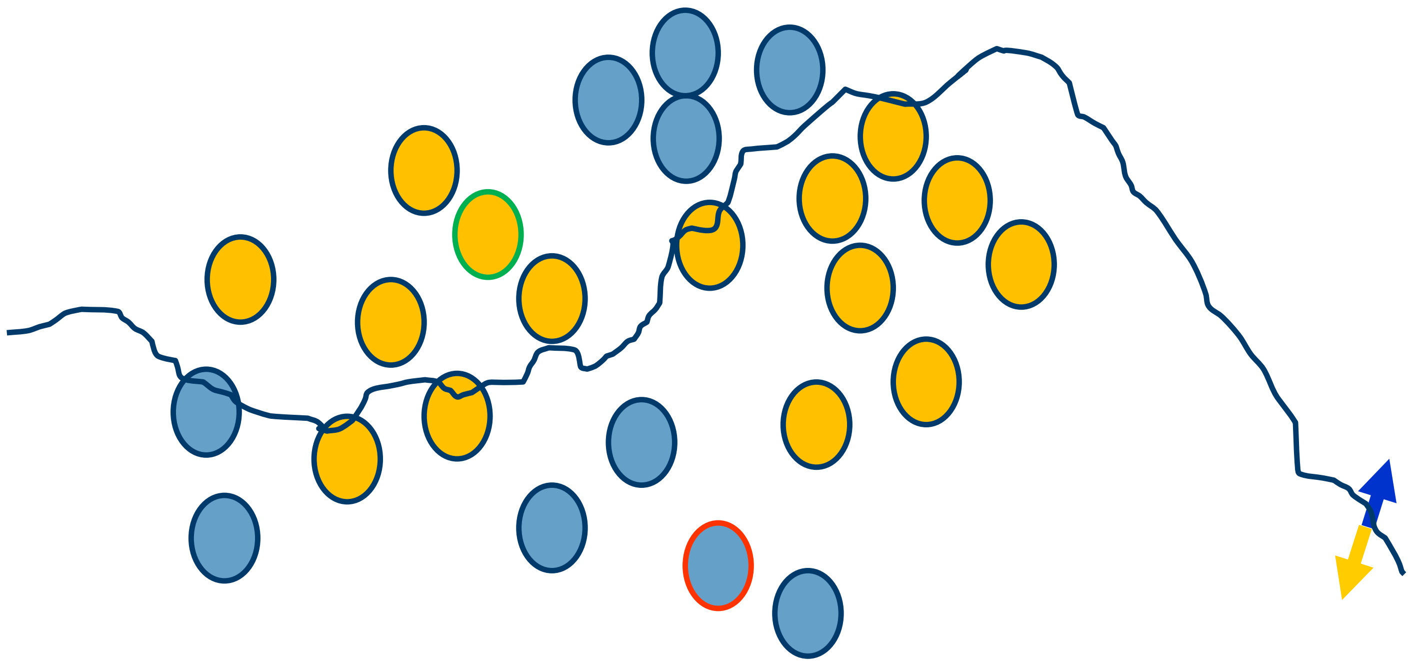
Initial Weights



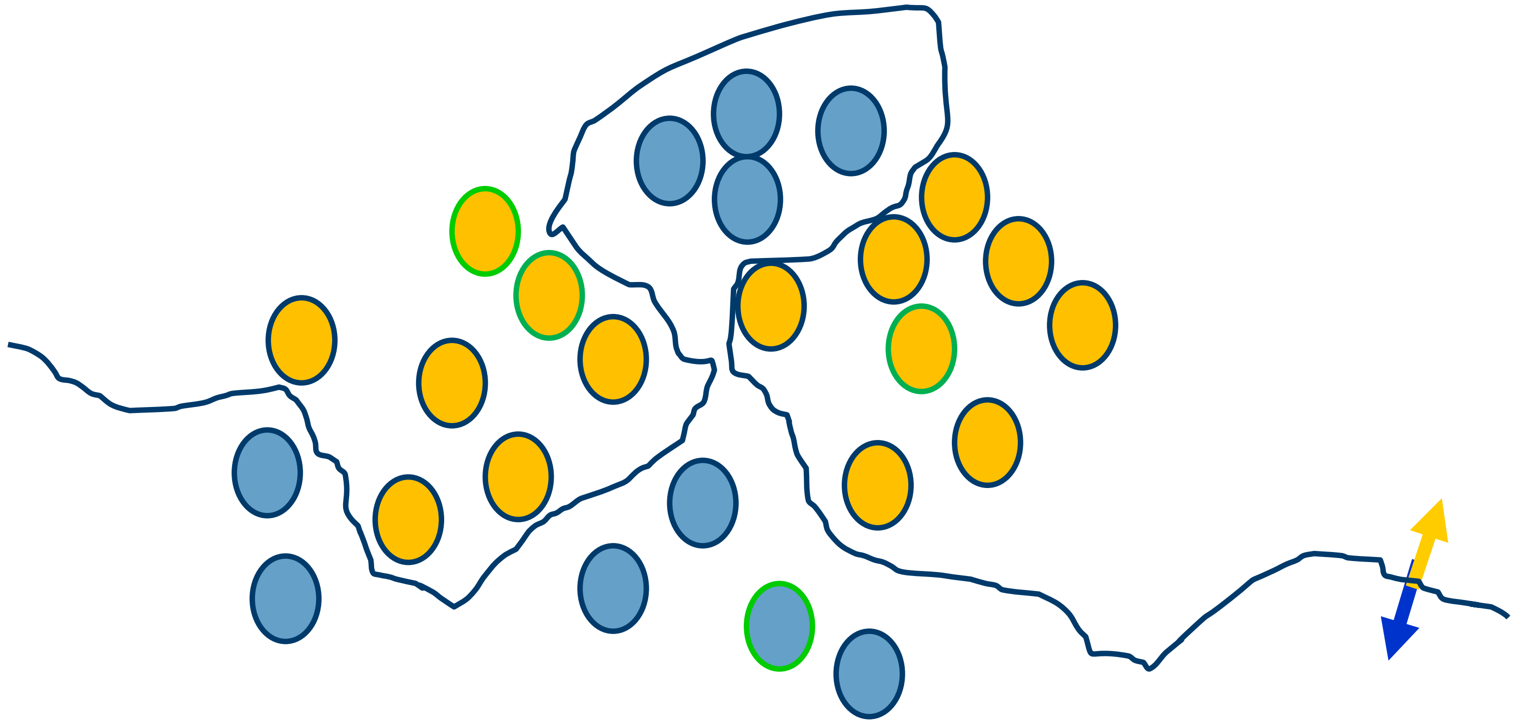
Present Instance and Adjust Weights



Present Instance and Adjust Weights



Eventual Final Classification



NNs are making thousands of tiny adjustments, each making the network do better at the most recent pattern, but perhaps a little worse on many others

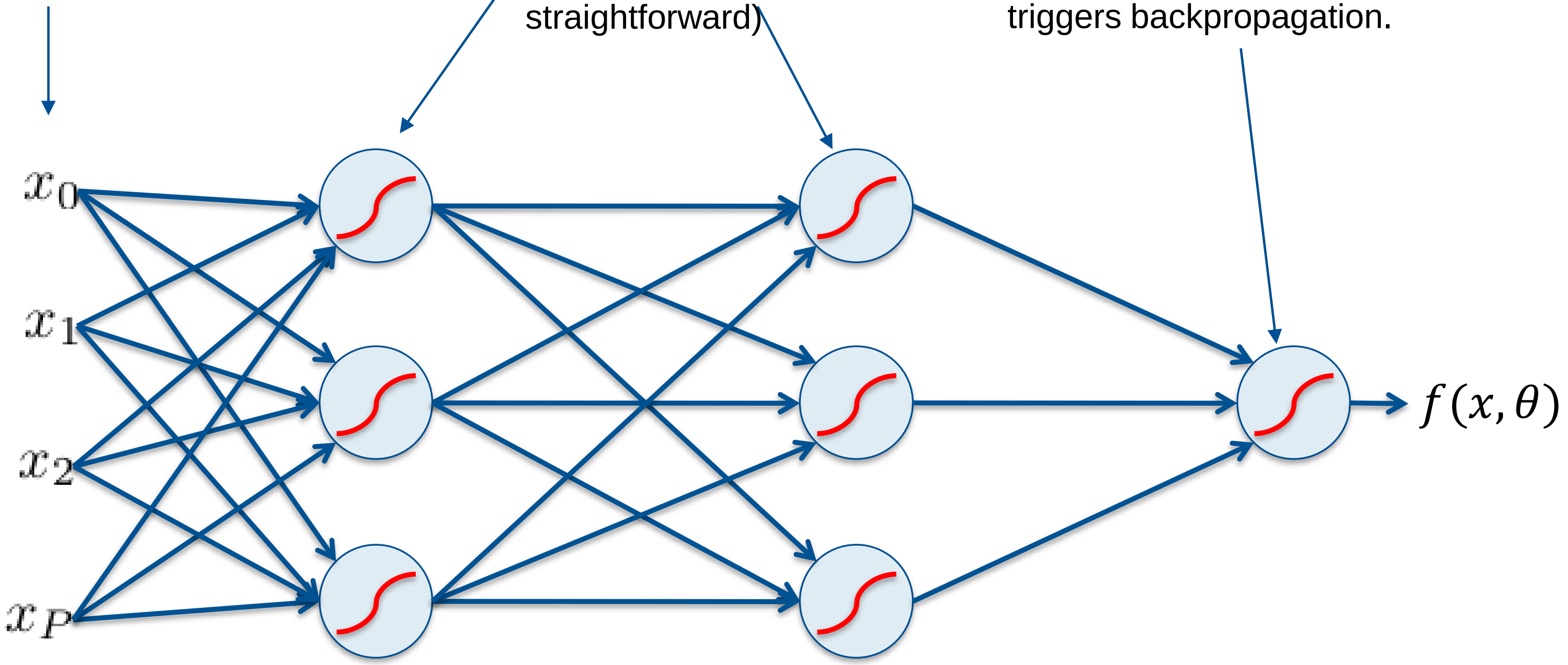
Neural Networks Recap

Number of Nodes per Layer

Input layer:
depending on data dimension

Hidden Layers:
Should be chosen in a way that improves model performance. (NOT straightforward)

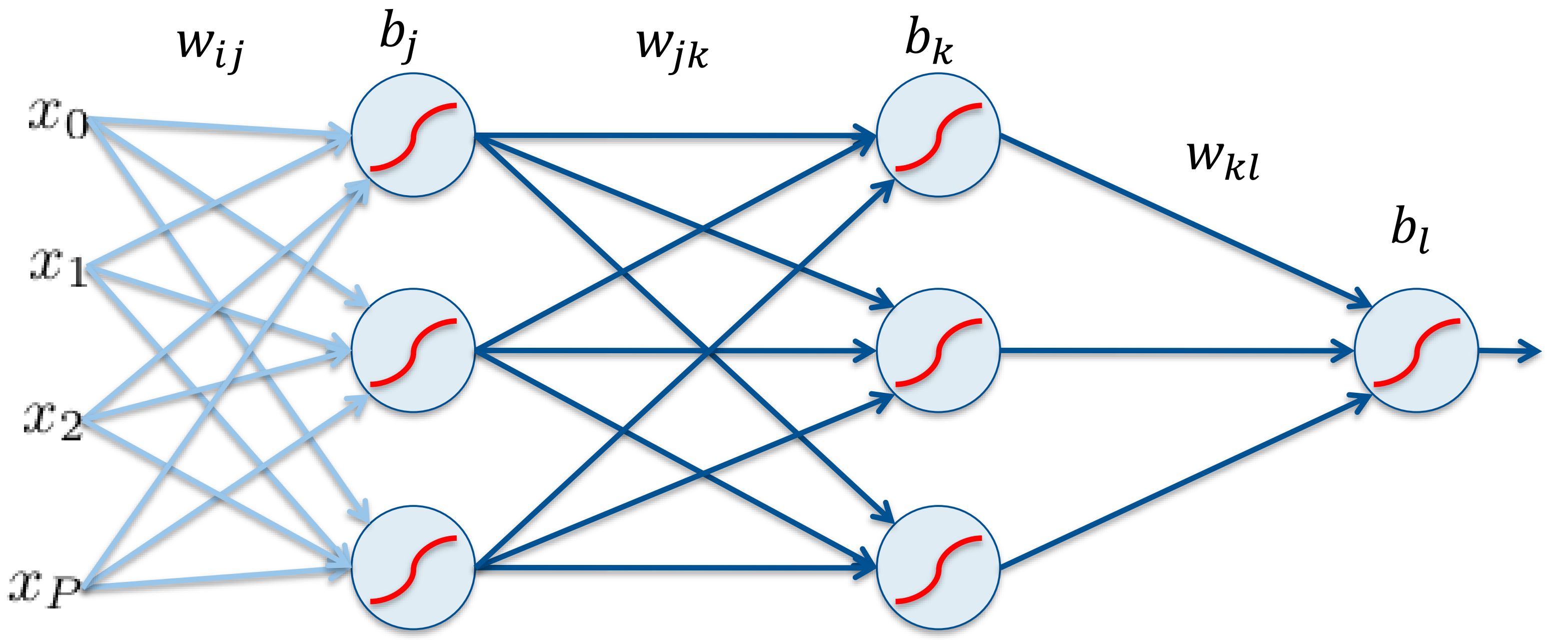
Output layer:
Corresponding to modeling needs, i.e. 1 for binary classification, $k-1$ for multiclass classification, etc. Activation of output layer for each observation becomes the input to the loss function L that triggers backpropagation.



Feed-Forward Networks

Instances are fed forward through the network.

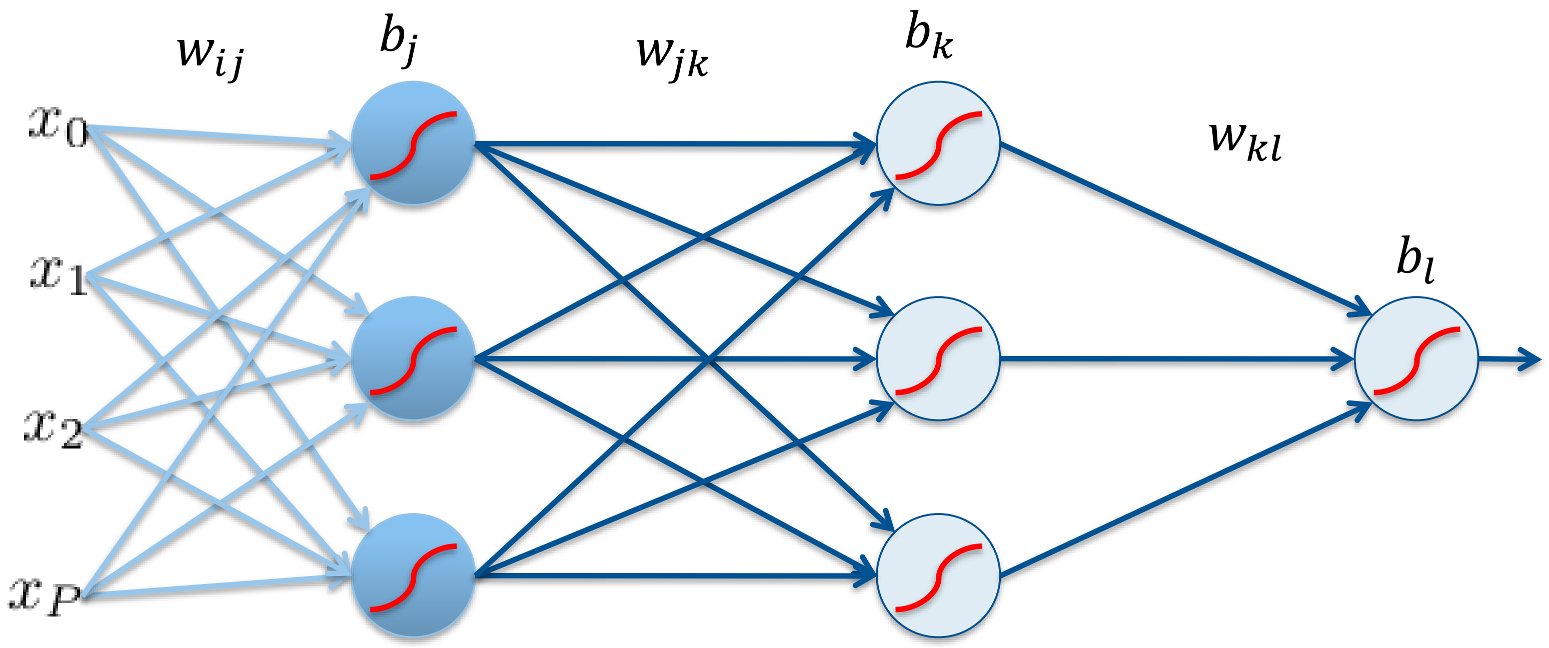
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

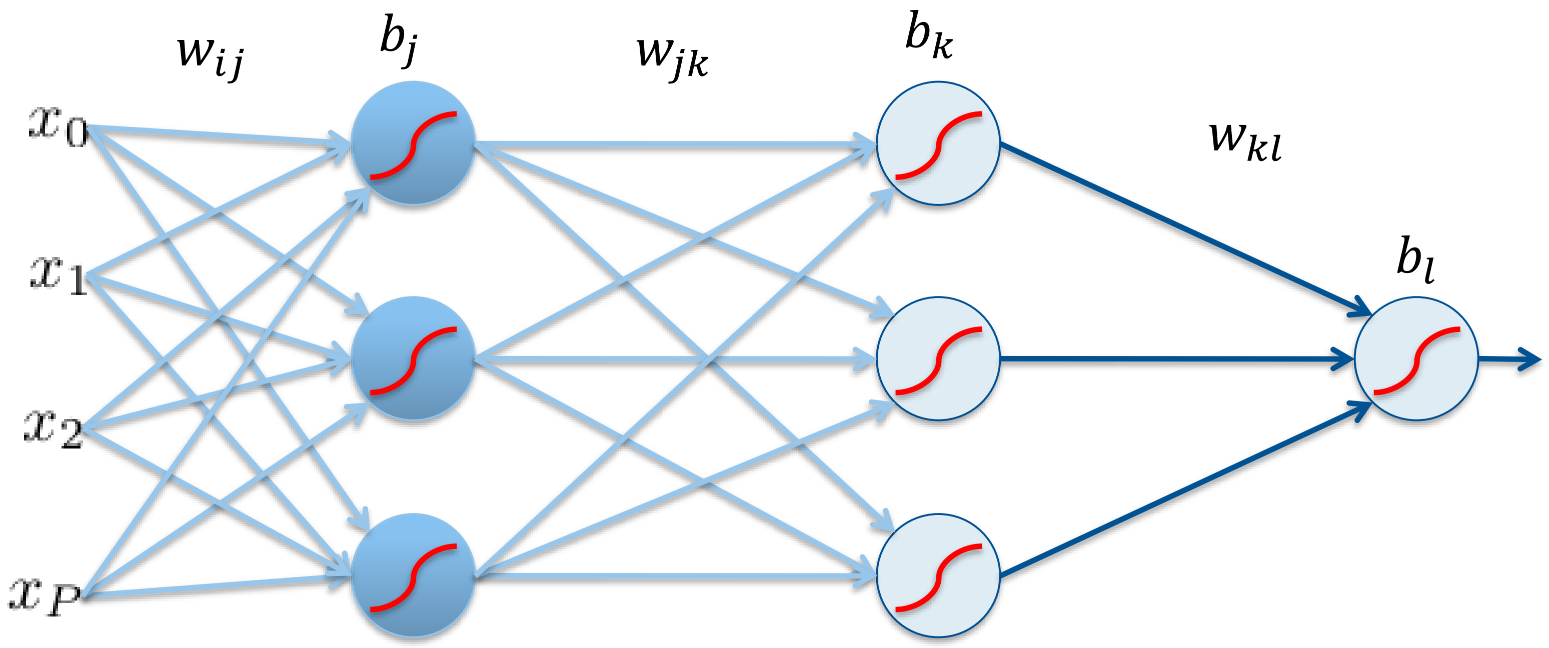
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

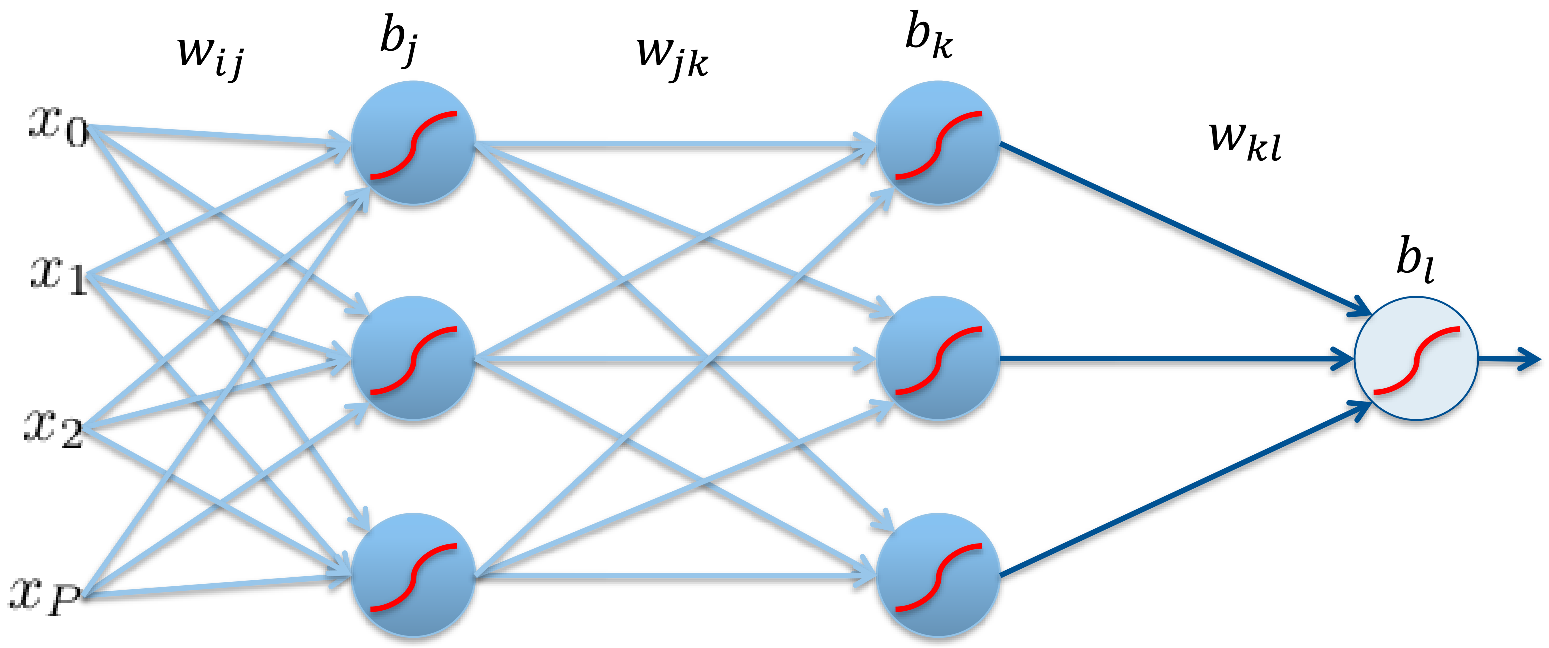
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

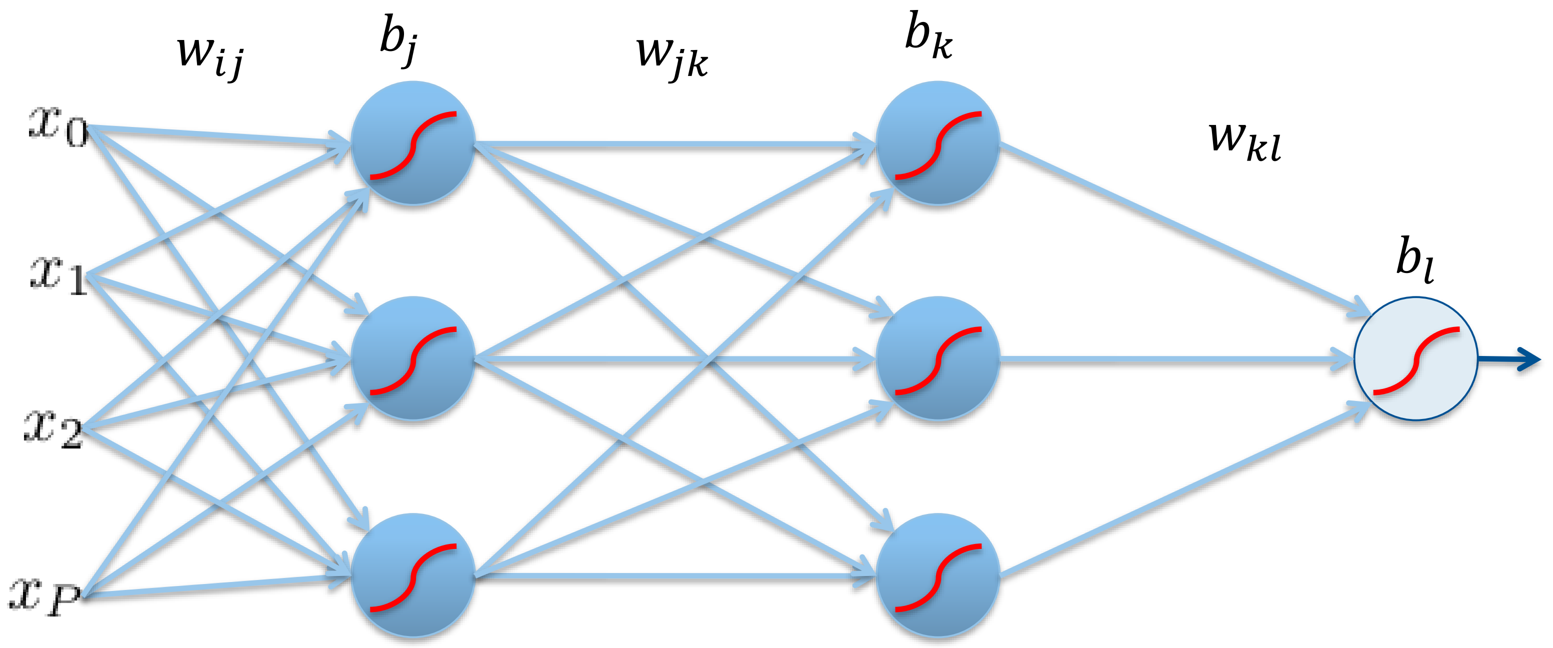
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

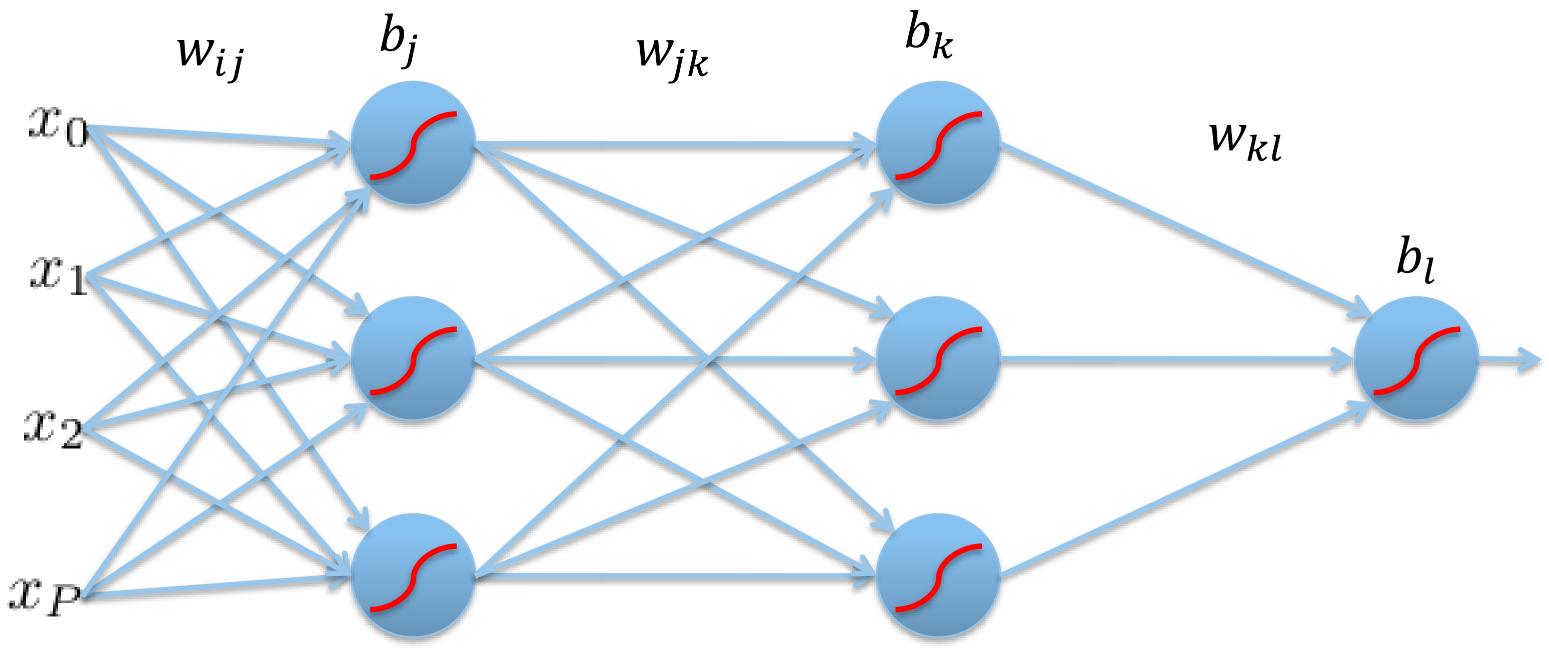
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

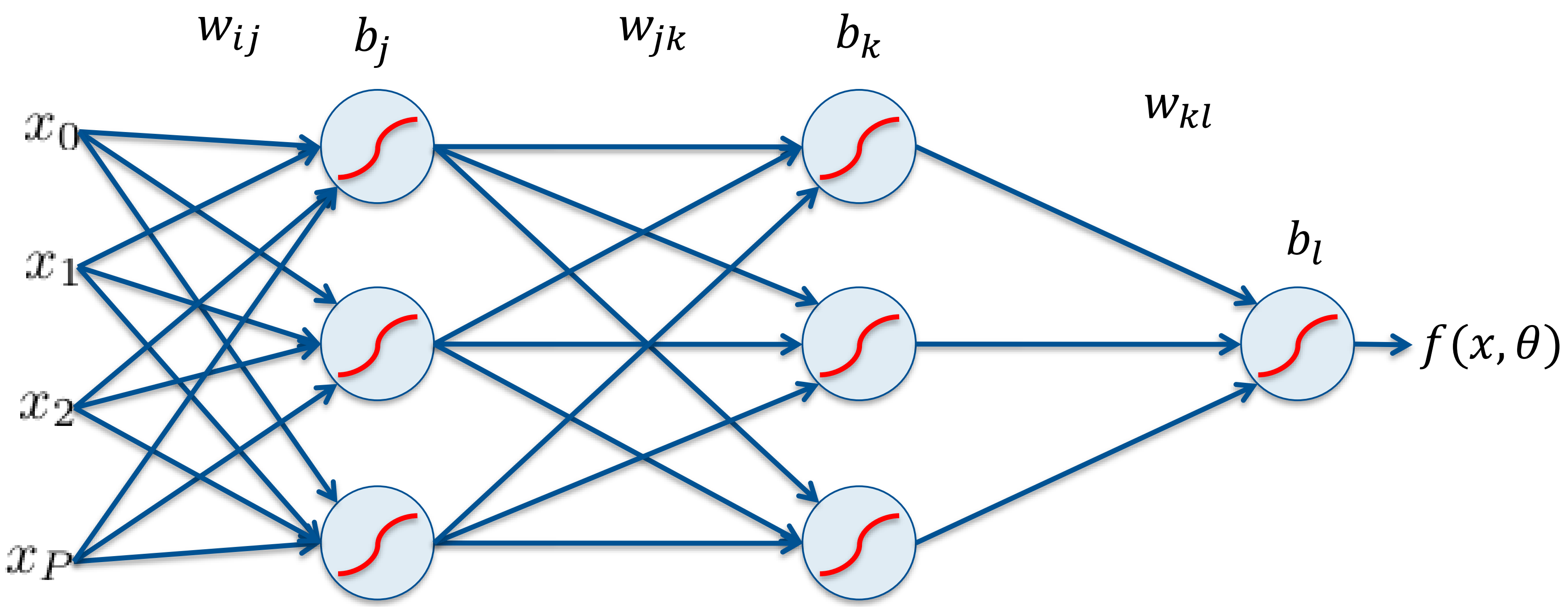
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Error Backpropagation

We will do gradient descent on the whole network.

Training will proceed from the last layer to the first.



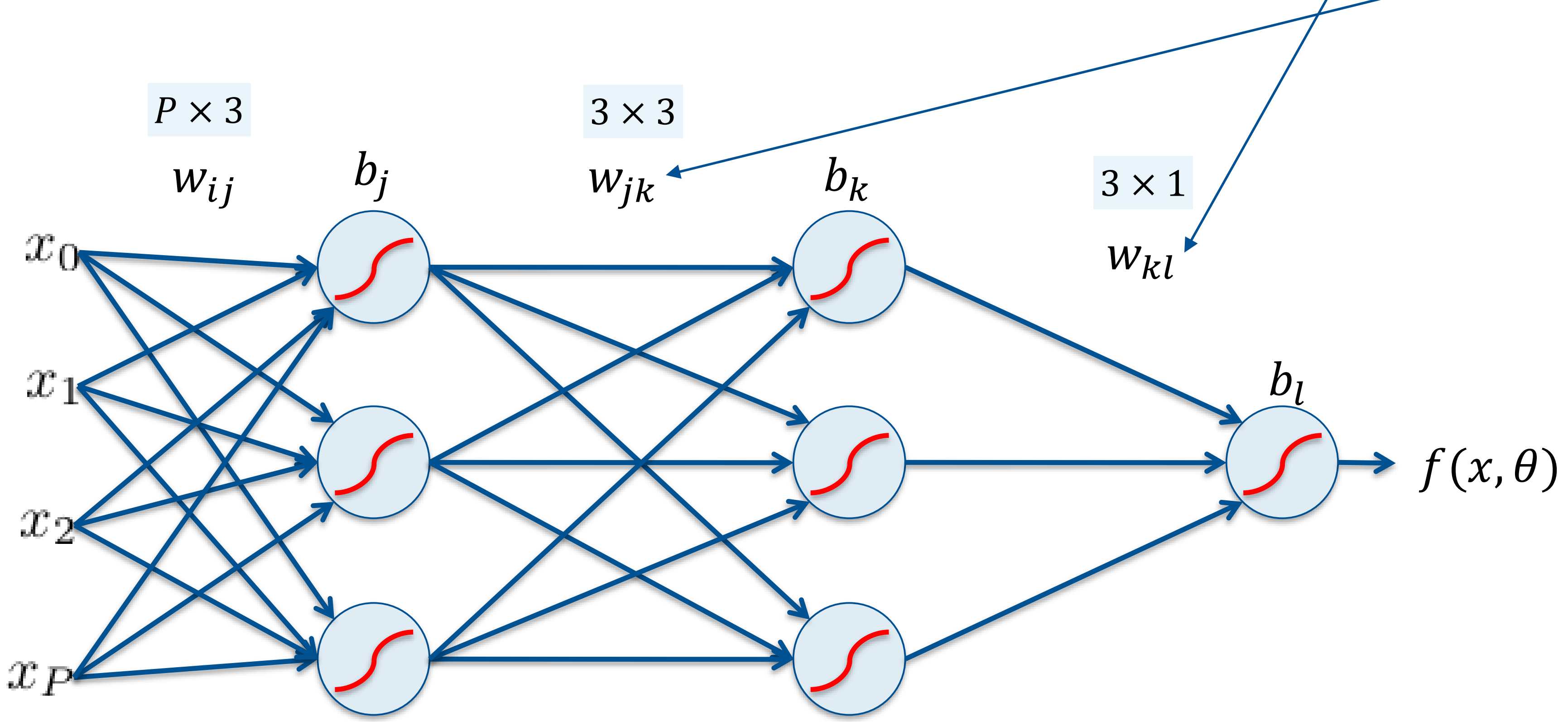
Recap

- *Backpropagation* describes *how a single training example*, starting from the output neurons, dertermines the goal for the neurons on the next layer and steps backwards recursively.
- An *epoch* is one pass through the training set, with an adjustment through backpropagation to the network weights for each training example.
- The average change of weights and biases across all training instances results in changes of the network like the negative gradient of the risk function.
- In contrast, *stochastic gradient descent* trains on mini-batches. Each mini-batch gives an approximation of the neg. gradient, but speeds up the training process.

Error Backpropagation

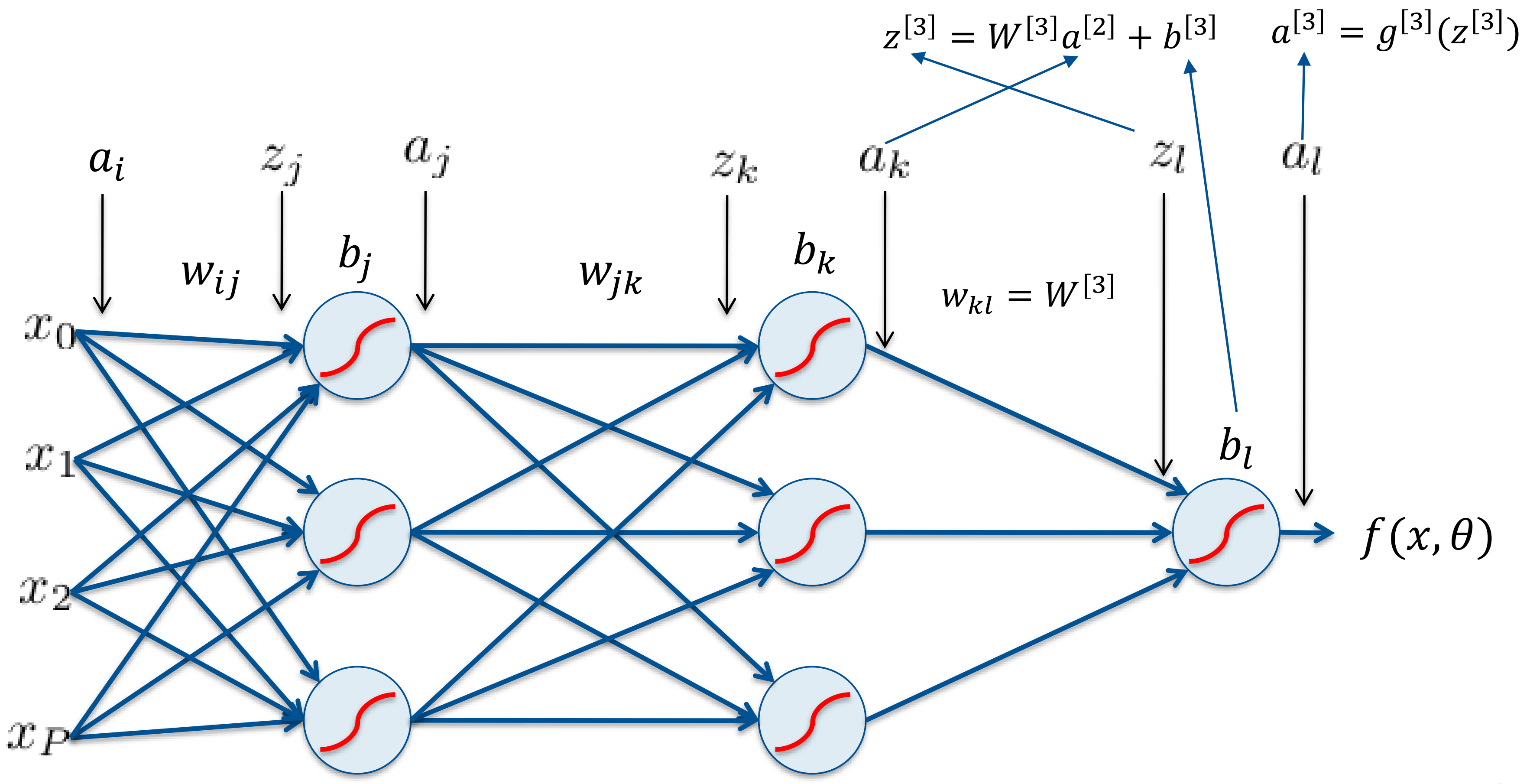
Introduce variables over the neural network.

Note that w_{kl} refers to a 3×1 matrix. It is sometimes more convenient to write this as the weights of the last layer: $W^{[3]}$ or $W^{[2]}$



Error Backpropagation

Introduce variables to distinguish the input and output of each node.

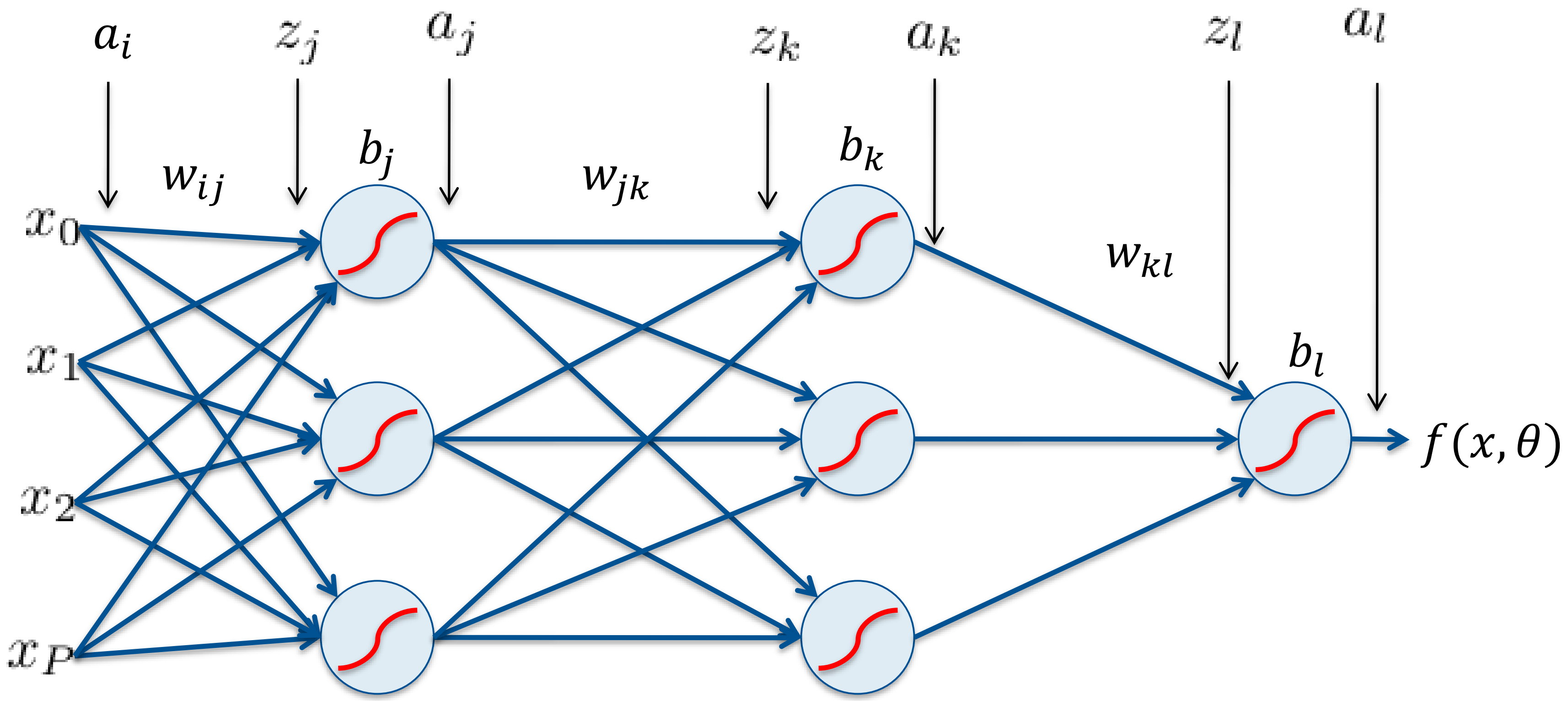


Error Backpropagation

Matrix notation: $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$ $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$

Scalar notation: $z_j = \Sigma_i w_{ij} a_i + b_j$ $z_k = \Sigma_j w_{jk} a_j + b_k$ $z_l = \Sigma_k w_{kl} a_k + b_l$

$a_j = g(z_j)$ $a_k = g(z_k)$ $a_l = g(z_l)$

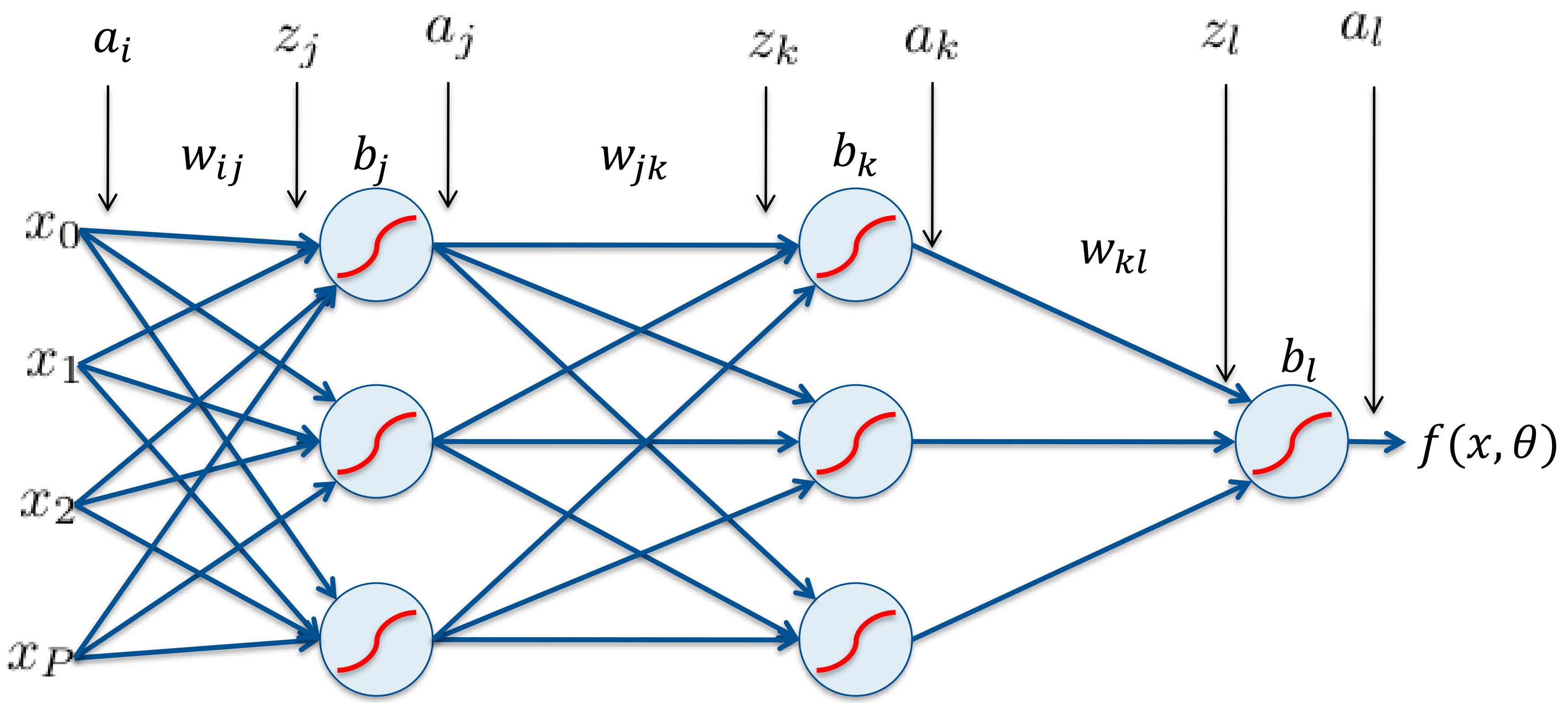


Error Backpropagation

Training: Take the gradient of the last component and iterate backwards

$$z_j = \Sigma_i w_{ij} a_i + b_j \qquad z_k = \Sigma_j w_{jk} a_j + b_k \qquad z_l = \Sigma_k w_{kl} a_k + b_l$$

$$a_j = g(z_j) \qquad a_k = g(z_k) \qquad a_l = g(z_l)$$



Error Backpropagation

class label

$$R(\theta) = \frac{1}{N}\sum_{n=0}^{N} L(y_n - f(x_n))$$

Empirical Risk Function

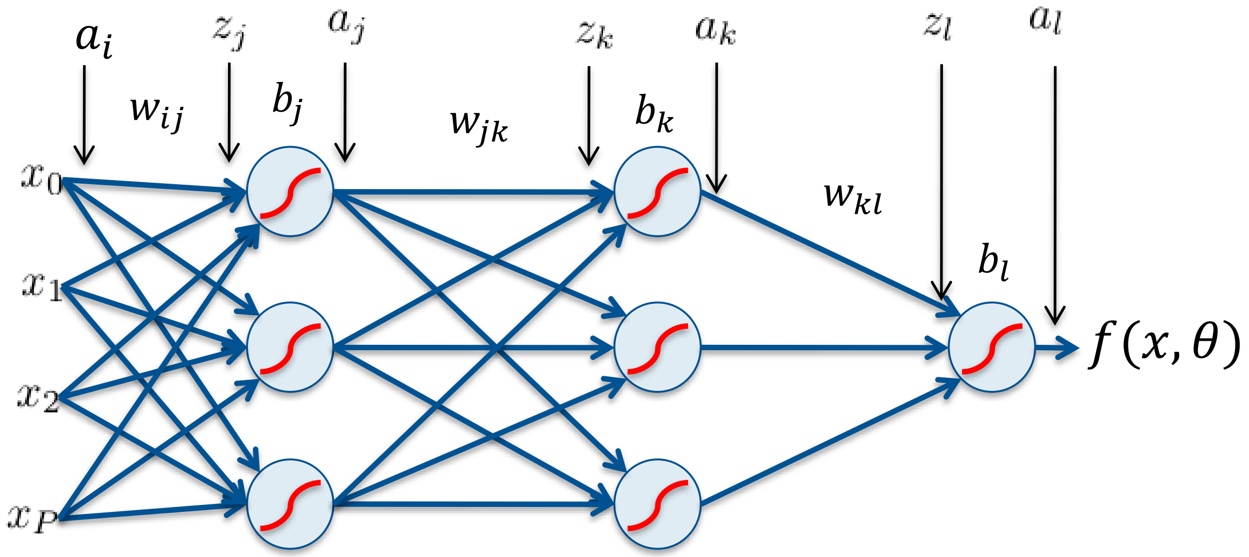
introduced for convenience

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}(y_n - f(x_n))^2$$

NN prediction

$$= \frac{1}{N}\sum_{n=0}^{N} \frac{1}{2}\left(y_n - g\left(\sum_k w_{kl}\, g\left(\sum_j w_{jk}\, g\left(\sum_i w_{ij} x_{n,i}\right)\right)\right)\right)^2$$

(ignoring the bias for brevity)



Reminder: Chain Rule of (Univariate) Calculus

If g is differentiable at x and f is differentiable at $g(x)$, then the composite function $F(x) = f(g(x))$ is differentiable at x and F' is given by the product

$$F'(x) = f'\big(g(x)\big) * g'(x)$$

In Leibniz notation, if $y = f(u)$ and $u = g(x)$ are both differentiable functions, then

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

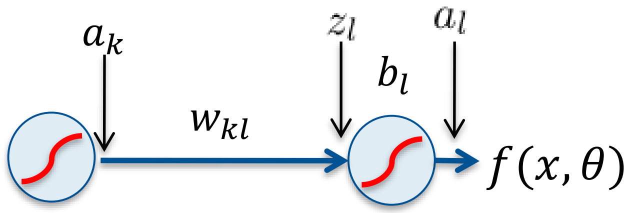
Example:

$$\frac{d}{dx}\left((3x^2 - x + 1)^{\frac{1}{3}}\right) = \frac{1}{3}(3x^2 - x + 1)^{-\frac{2}{3}}(6x - 1)$$

Error Backpropagation

Derive partial derivative wrt. all weights and biases: $\nabla R = \begin{pmatrix} \frac{\partial R}{\partial w_{ij}} \\ \frac{\partial R}{\partial b_j} \\ \vdots \\ \boxed{\frac{\partial R}{\partial w_{kl}}} \end{pmatrix}$

Look at the last layer weights w_{kl} of a simplified network.



$$\boxed{\frac{\partial R}{\partial w_{kl}}} = \frac{1}{N} \sum_n \frac{\partial L_n}{\partial a_{ln}} \frac{\partial a_{ln}}{\partial z_{ln}} \frac{\partial z_{ln}}{\partial w_{kln}}$$

avg. over training instances

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n [-(y_n - z_{ln}) g'(z_{ln})] a_{kn}$$

$$L_n = \frac{1}{2}(y_n - g(w_{kl} a_{kn} + b_l))^2$$

$$L_n = \frac{1}{2}(y_n - a_{ln})^2 \qquad \frac{\partial L_n}{\partial a_{ln}} = -(y_n - a_{ln})$$

$$a_{ln} = g(z_{ln}) \qquad \frac{\partial a_{ln}}{\partial z_{ln}} = g'(z_{ln})$$

$$z_{ln} = w_{kln} a_{kn} + b_{ln} \qquad \frac{\partial z_{ln}}{\partial w_{kln}} = a_{kn}$$

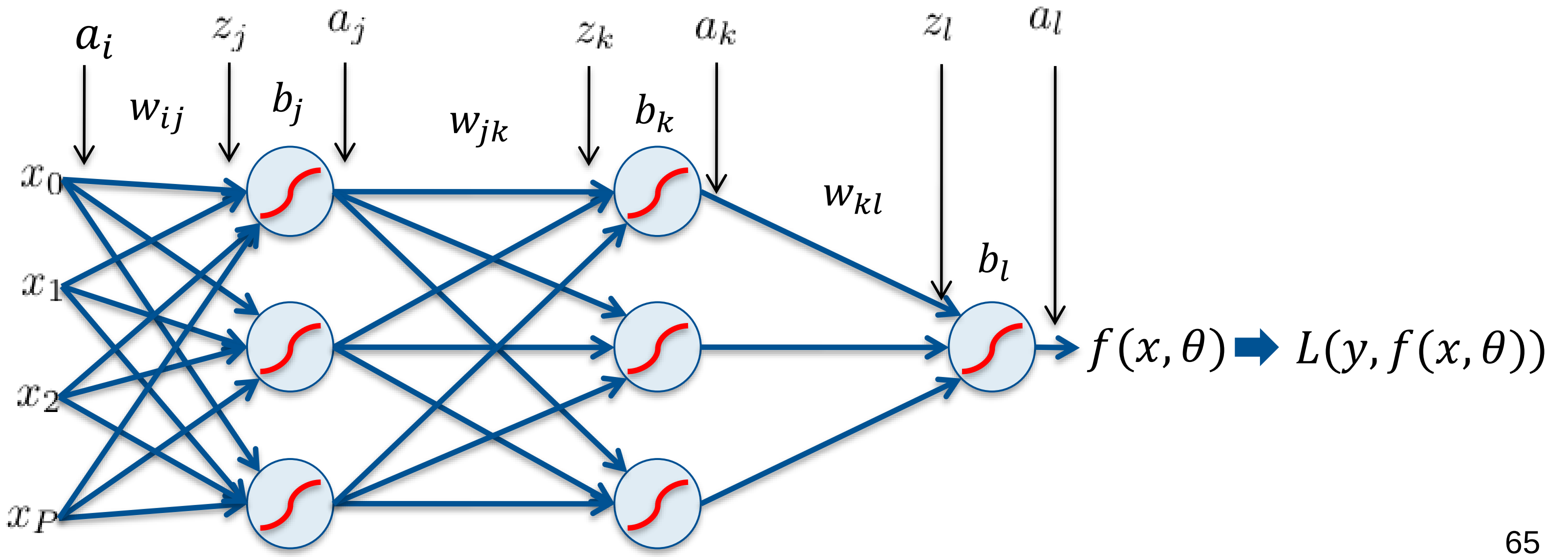
Error Backpropagation

Repeat for all previous layers

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N}\sum_n [-(y_n - z_{ln})g'(z_{ln})]a_{kn} = \frac{1}{N}\sum_n \delta_{ln}a_{kn}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n [\delta_{ln}w_{kl}g'(z_{kn})]a_{jn} = \frac{1}{N}\sum_n \delta_{kn}a_{jn}$$

$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n [\delta_{kn}w_{jk}g'(z_{jn})]a_{in} = \frac{1}{N}\sum_n \delta_{jn}a_{in}$$



Error Backpropagation

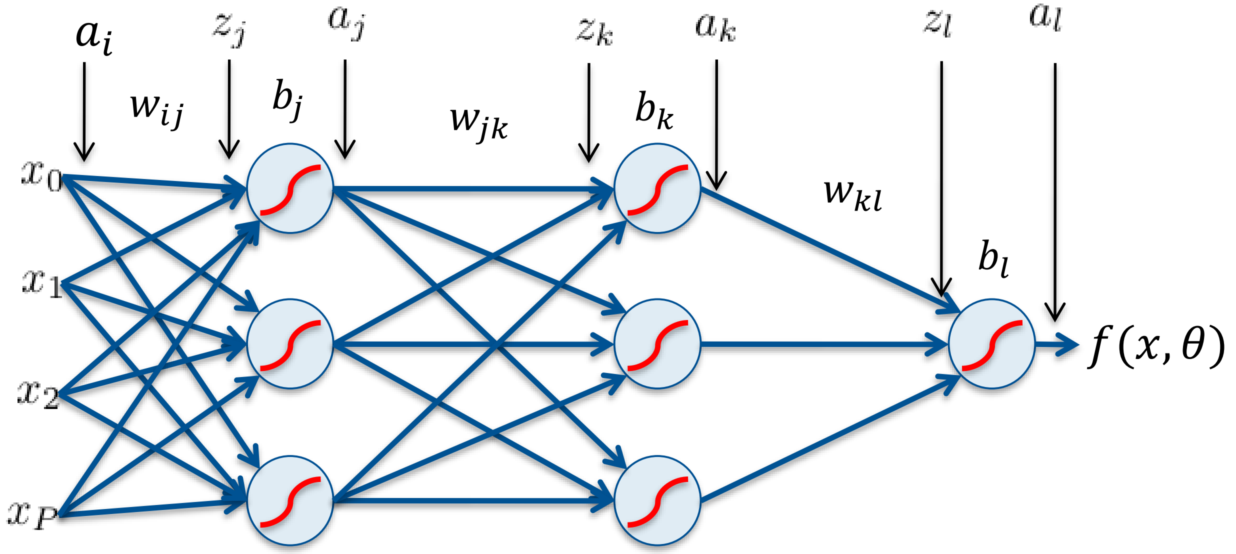
Now that we have well defined gradients for each parameter (weight and bias).
Update using gradient descent:

- Add up the changes for particular weight.
- Multiply by the learning rate.
- Subtract the outcome from the current value of the weight.
- Repeat on previous layers and propagate error backwards.
- Apply the same computation for biases.

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha \frac{\partial R}{\partial w_{ij}}$$

$$w_{jk}^{t+1} = w_{jk}^{t} - \alpha \frac{\partial R}{\partial w_{jk}}$$

$$w_{kl}^{t+1} = w_{kl}^{t} - \alpha \frac{\partial R}{\partial w_{kl}}$$



Summary of Error Backpropagation

Error backprop unravels the multivariate chain rule and solves the gradient for each partial component separately.

The target values for each layer come from the next layer.

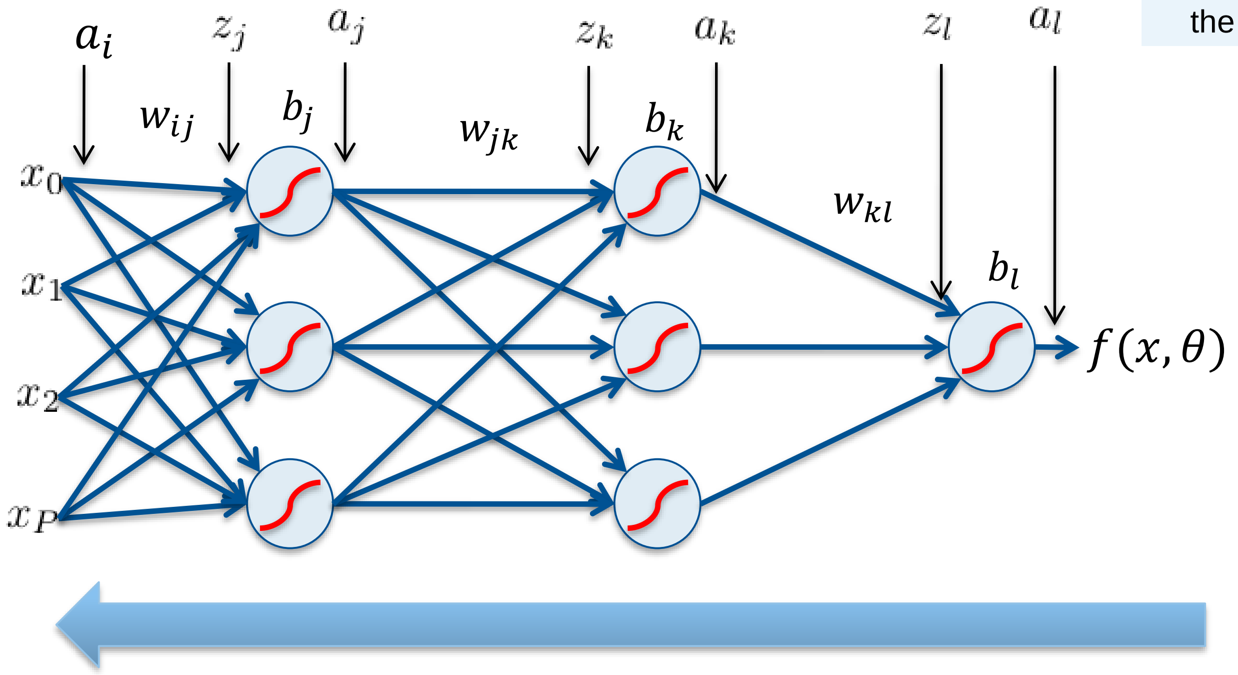
This feeds the errors back along the network.

Matrix notation:

$$W^{[l]} := W^{[l]} - \alpha \cdot dW^{[l]}$$

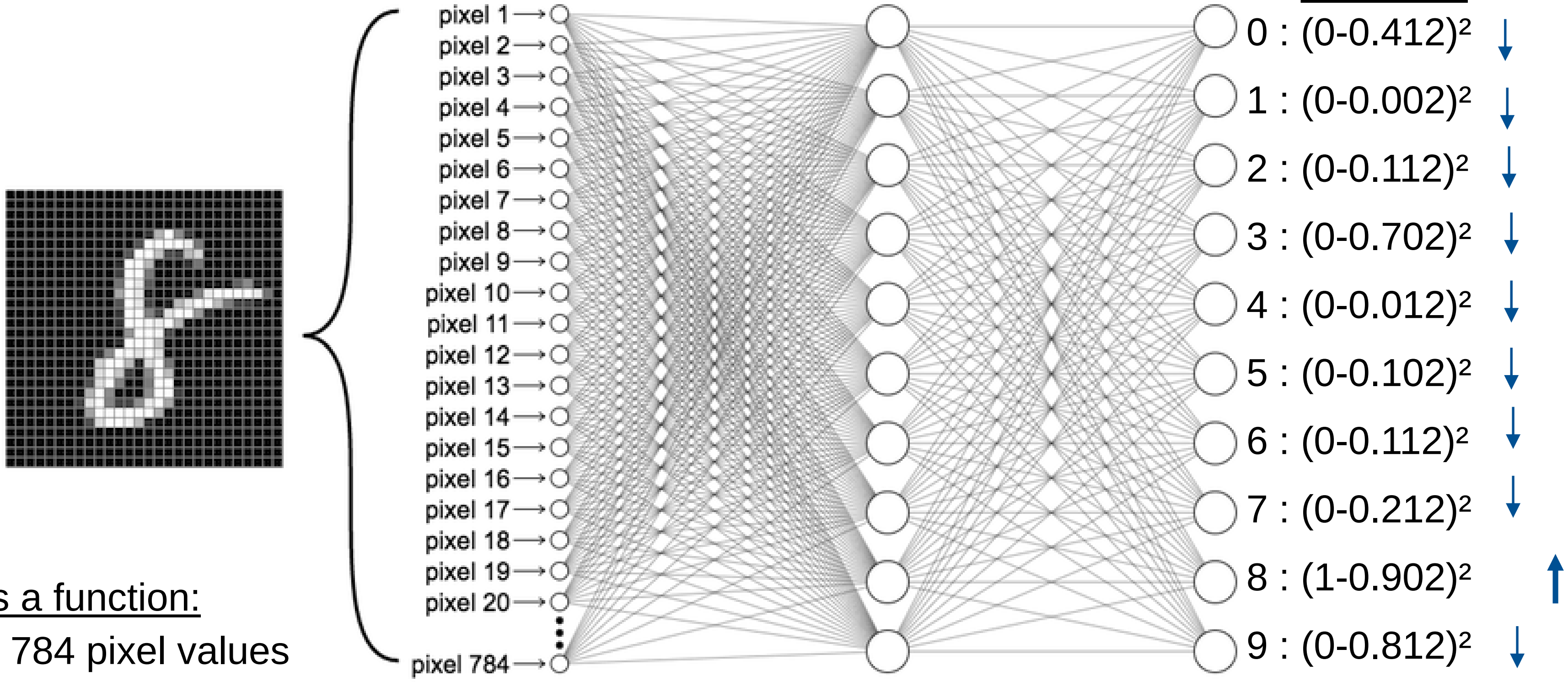
$$b^{[l]} := b^{[l]} - \alpha \cdot db^{[l]}$$

dW and db are the gradient of the loss wrt. W or b resp.



Neural Network with Multiple Outputs

Backpropagation on the following example adapts the weights and biases for this example such that the output value for 8 is increased and that of the other outputs is decreased.



Loss fctn.

$0 : (0-0.412)^2$ ↓

$1 : (0-0.002)^2$ ↓

$2 : (0-0.112)^2$ ↓

$3 : (0-0.702)^2$ ↓

$4 : (0-0.012)^2$ ↓

$5 : (0-0.102)^2$ ↓

$6 : (0-0.112)^2$ ↓

$7 : (0-0.212)^2$ ↓

$8 : (1-0.902)^2$ ↑

$9 : (0-0.812)^2$ ↓

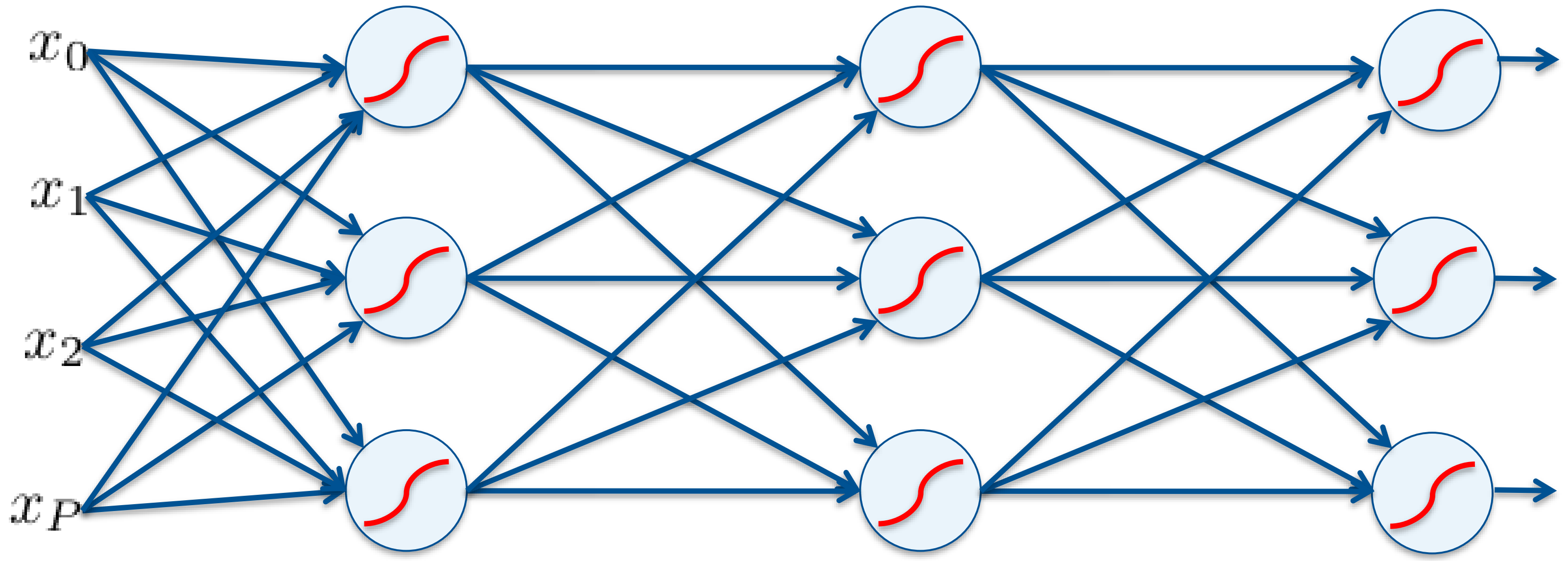
NN as a function:

Input: 784 pixel values

Output: 10 numbers

Parameters: 784*10+10*10=7940 weights plus 20 biases

Multiple Outputs



Used for N-way classification.
Each node in the output layer corresponds to a different class.
No guarantee that the sum of the output vector will equal 1.

Hyperparameters

- Number of hidden layers
- Number of nodes in each hidden layer
- Activation function
  - Sigmoid suffers from vanishing gradient, ReLU is now widespread
- Learning rate
- Iterations and desired error level
- Batch size in SGD
- Other optimization algorithms than SGD such as RMSProp, Adam, etc.
- Regularization (aka. weight decay) in each layer (as in ridge regression) to limit the influence of irrelevant connections with low weight on the network‘s predictions.

$$R(\theta) = \arg\min_\theta \left[\frac{1}{N}\sum_n L\big(y_n, f(x_n,\theta)\big) + \lambda R(\theta)\right]$$

Setting the Number of Nodes in the Hidden Layer

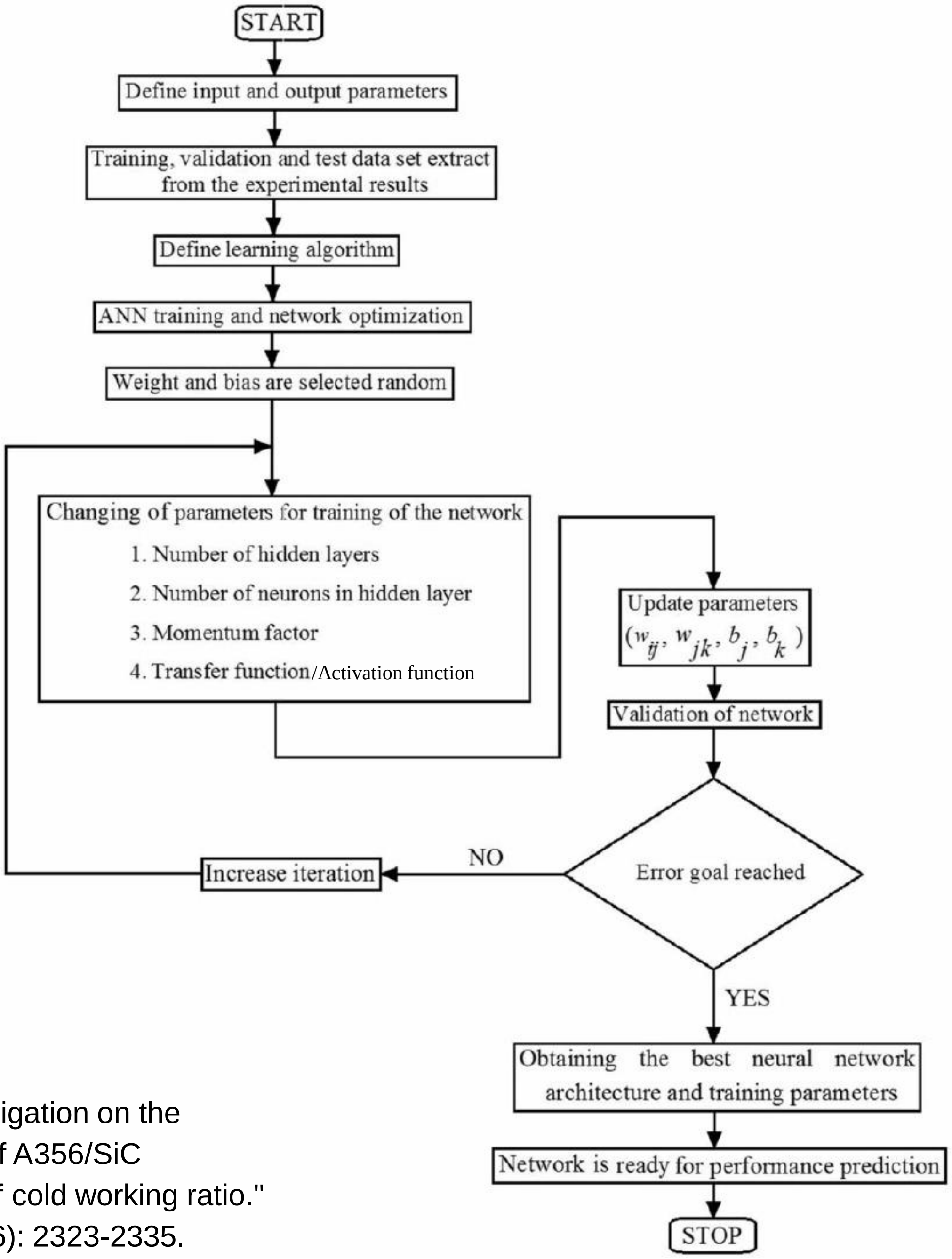
- The number of nodes in the hidden layer affects generality and convergence.
- If too few hidden nodes: convergence may fail.
- Few but not too few nodes: possibly slow convergence but good generalization
- Too many hidden nodes (breadth): Rapid convergence, but “overfitting” happens.

Interpretation of Hidden Layers

- What are the hidden layers doing?!
- Feature extraction
- The non-linearities in the feature extraction can make interpretation of the hidden layers very difficult.
- This leads to Neural Networks being treated as **black boxes**.

A Process Model

So, what do I have to do now?


START
Define input and output parameters
Training, validation and test data set extract from the experimental results
Define learning algorithm
ANN training and network optimization
Weight and bias are selected random
Changing of parameters for training of the network
1. Number of hidden layers
2. Number of neurons in hidden layer
3. Momentum factor
4. Transfer function/Activation function
Update parameters (w_{ij}, w_{jk}, b_j, b_k)
Validation of network
Error goal reached
NO
Increase iteration
YES
Obtaining the best neural network architecture and training parameters
Network is ready for performance prediction
STOP

Tuntas, Remzi, and Burak Dikici. "An investigation on the aging responses and corrosion behaviour of A356/SiC composites by neural network: The effect of cold working ratio." *Journal of Composite Materials* 50.17 (2016): 2323-2335.

Which role does the chain rule play in backpropagation?