# Deep Learning

*Lecture 2 – Softmax regression and feedforward neural networks*

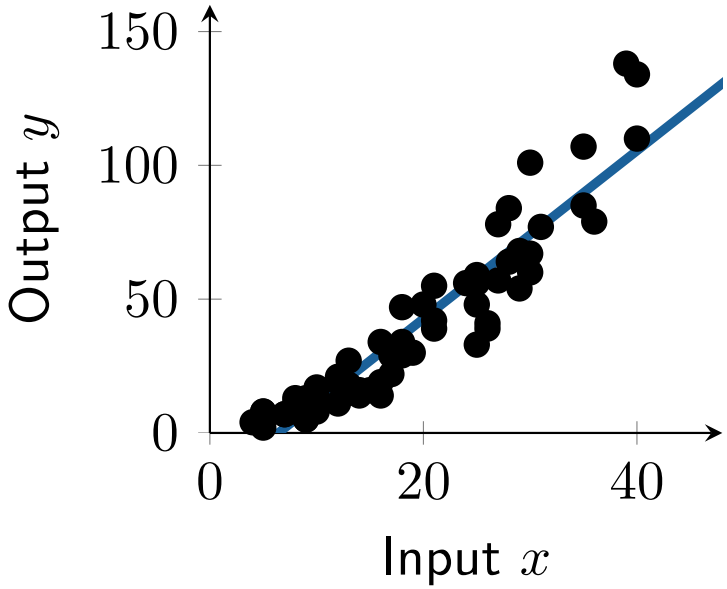
UPPSALA
UNIVERSITET

**Niklas Wahlström**
Division of Systems and Control
Department of Information Technology
Uppsala University

Email: niklas.wahlstrom@it.uu.se

# Summary of lecture 1 (I/III)
# One-dim input - One-dim output

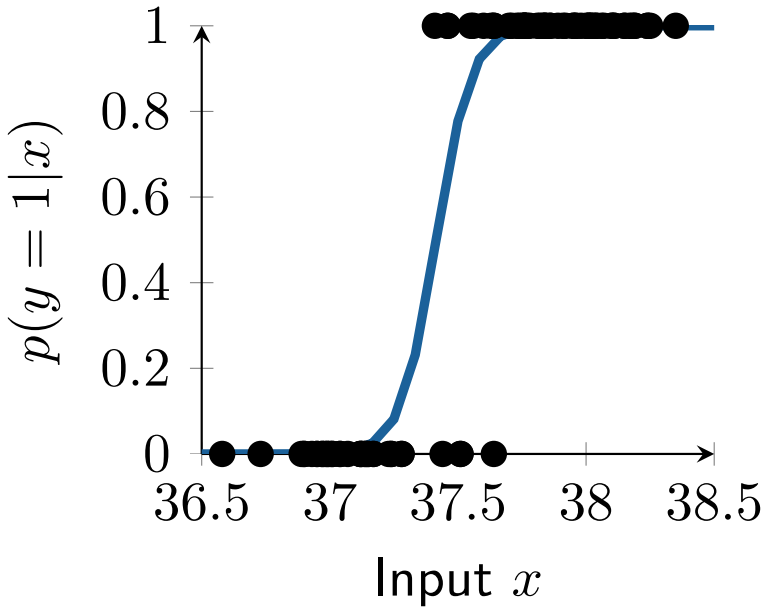
## Linear regression




**Model**

$\hat{y}_i = wx_i + b$

## Logistic regression




**Model**

$p(y_i = 1|x_i) = \frac{e^{z_i}}{1+e^{z_i}}$, where

$z_i = wx_i + b$

# Summary of lecture 1 (II/III)
# Multi-dim input - One-dim output

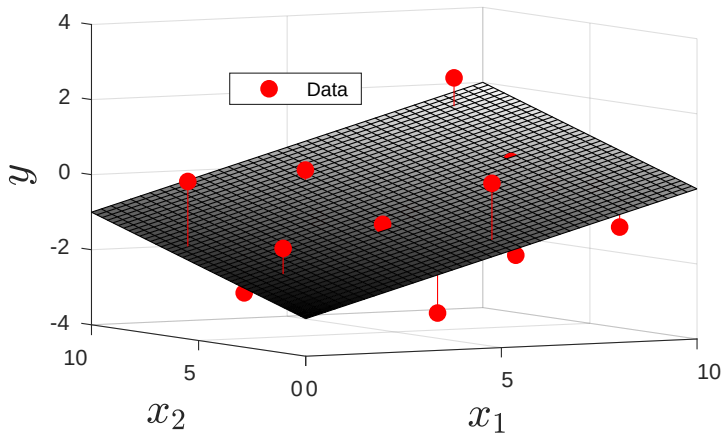
Linear and logistic regression also work for multidimensional inputs

$$\mathbf{x}_i = [x_{i1}, \ldots, x_{ip}]^\mathsf{T}, \qquad i = 1, \ldots, n$$

We assign one parameter for each input dimension

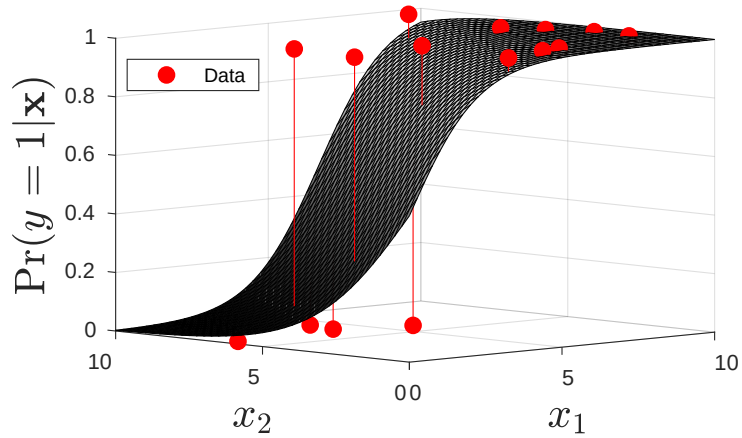
$$\mathbf{w} = [w_1, \ldots, w_p]^\mathsf{T}$$

## Linear regression




$$\hat{y}_i = \mathbf{w}^\mathsf{T}\mathbf{x}_i + b$$

## Logistic regression




$$p(y_i = 1|x_i) = \frac{e^{\mathbf{w}^\mathsf{T}\mathbf{x}_i + b}}{1 + e^{\mathbf{w}^\mathsf{T}\mathbf{x}_i + b}}$$

# Summary of lecture 1 (III/III)
# Multi-dim input - One-dim output

## Linear regression

**Output**

$y_i \in \mathbb{R}$

**Model**

$$\hat{y}_i = \mathbf{w}^\mathsf{T}\mathbf{x}_i + b$$

**Loss**

$$L_i = (y_i - \hat{y}_i)^2$$

## Logistic regression

**Output**

$y_i \in \{0, 1\}$

**Model**

$$p_i = p(y_i = 1|\mathbf{x}_i) = \frac{e^{\mathbf{w}^\mathsf{T}\mathbf{x}_i + b}}{1 + e^{\mathbf{w}^\mathsf{T}\mathbf{x}_i + b}}$$

**Loss**

$$L_i = -y_i \ln(p_i) - (1 - y_i) \ln(1 - p_i)$$

We find $\mathbf{w}$ and $b$ by minimizing sum of the losses

$$\widehat{\mathbf{w}}, \widehat{b} = \underset{\mathbf{w},b}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L_i$$

- For linear regression the problem can be solved analytically.
- For logistic regression we need to use numerical optimization.

# Contents – Lecture 2

- Summary of lecture 1
- Softmax regression
- Feedforward neural network
- Optimization + training/test data
- Why do neural networks work so well?

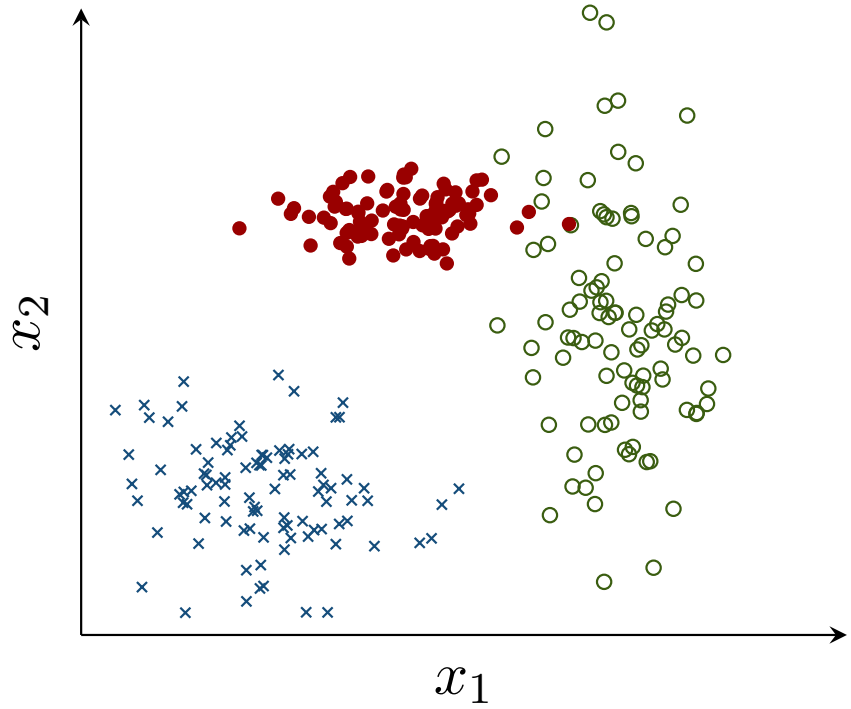
# Multinomial logistic regression (softmax regression)

# A multi-class problem

Consider a classification problem with 3 classes

**Input**: $\mathbf{x} = [x_1, x_2] \in \mathbb{R}^2$

**Output**: $y = \{$"red", "blue", "green"$\}$




Can we extend logistic regression to handle more than 2 classes?

# Softmax regression

Consider a multi-class classification problem $y \in \{1, \ldots, M\}$.

$$p_{im} = p(y_i = m | \mathbf{x}_i)$$

**Softmax regression** Assume a linear model for each "log odds"

$$\ln \frac{p_{im}}{Z} = \mathbf{w}_m^\mathsf{T} \mathbf{x}_i + b_m, \qquad m = 1, \ldots, M$$

where $Z$ is a **normalization factor**. $\Rightarrow$

$$p_{im} = Z e^{z_{im}}, \quad z_{im} = \mathbf{w}_m^\mathsf{T} \mathbf{x}_i + b_m, \qquad m = 1, \ldots, M$$

The normalization factor $Z$ can now be computed as

$$1 = \sum_{m=1}^{M} p_{im} = Z \sum_{m=1}^{M} e^{z_{im}} \quad \Rightarrow \quad Z = \frac{1}{\sum_{m=1}^{M} e^{z_{im}}}$$

Softmax regression model is then

$$p_{im} = \frac{e^{z_{im}}}{\sum_{l=1}^{M} e^{z_{il}}}, \qquad z_{im} = \mathbf{w}_m^\mathsf{T} \mathbf{x}_i + b_m, \qquad m = 1, \ldots, M$$

# Softmax regression using maximum likelihood

Pick $\mathbf{w}_m$ and $b_m$ which make data as likely as possible

$$\widehat{\mathbf{w}}_{1:M}, \widehat{b}_{1:M} = \underset{\mathbf{w}_{1:M}, b_{1:M}}{\text{argmax}} \ \ln p(y_{1:n} | \mathbf{x}_{1:n}, \mathbf{w}_{1:M}, b_{1:M})$$

Assume all $y_i$ to be independent

$$\ln p(y_{1:n} | \mathbf{x}_{1:n}, \mathbf{w}_{1:M}, b_{1:M}) = \sum_{i=1}^{n} \ln p(y_i | \mathbf{x}_i, \mathbf{w}_{1:M}, b_{1:M})$$

$$= \sum_{m=1}^{M} \sum_{\substack{i=1 \\ \text{where} \\ y_i = m}}^{n} \ln \underbrace{p(y_i = m | \mathbf{x}_i, \mathbf{w}_{1:M}, b_{1:M})}_{= p_{im}}$$

This leads to the following optimization problem

$$\widehat{\mathbf{w}}_{1:M}, \widehat{b}_{1:M} = \underset{\mathbf{w}_{1:M}, b_{1:M}}{\text{argmin}} \ -\sum_{i=1}^{n} \sum_{m=1}^{M} \tilde{y}_{im} \ln(p_{im}),$$

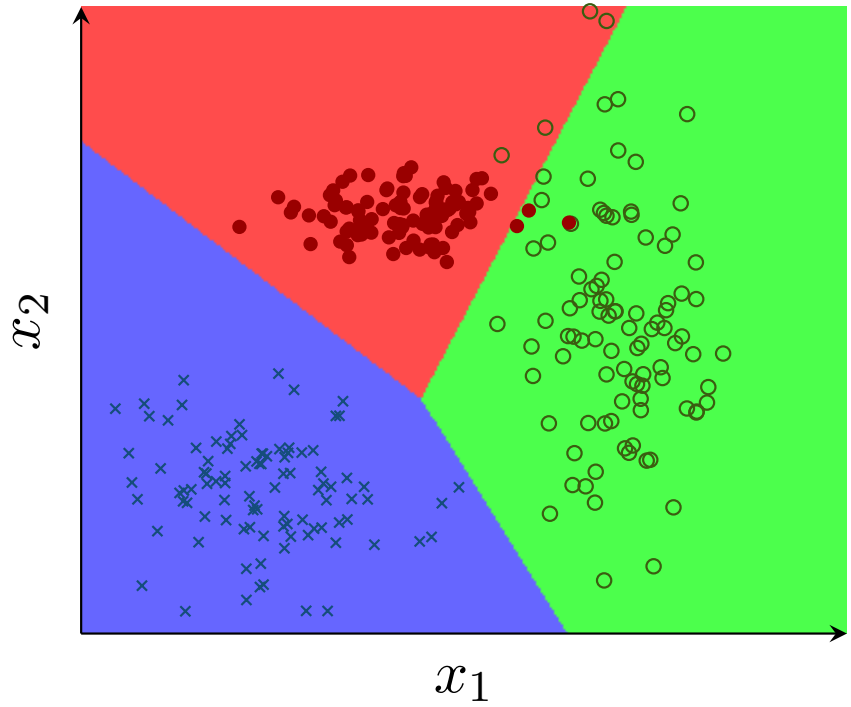
One-hot encoding of output

$$\tilde{y}_{im} = \begin{cases} 1, & \text{if} \quad y_i = m \\ 0, & \text{if} \quad y_i \neq m \end{cases}$$


niklas.wahlstrom@it.uu.se

# Softmax regression on multi-class problem

Consider again the classification problem with 3 classes.




Softmax computes the probability for each of the three classes.

Here the result is visualized with **decision boundaries** for the class which has the highest probability.

# Softmax regression: numerical issue 1, $z \gg 0$

- **Problem** If $z_{im}$ is large positive, then $e^{z_{im}} = \texttt{inf}$.
- **Solution**: Normalize $z_{im}$

$$z_{im} \leftarrow z_{im} - \gamma_i, \quad \gamma_i = \max(z_{i1}, \ldots, z_{iM})$$

Note, subtracting the same value to all $z_{im} \quad \forall m = 1, \ldots, M$ will not change the softmax output.

$$p_{im} = \frac{e^{z_{im}-\gamma_i}}{\sum_{l=1}^{M} e^{z_{il}-\gamma_i}} = \frac{e^{z_{im}} e^{-\gamma_i}}{\sum_{l=1}^{M} e^{z_{il}} e^{-\gamma_i}} = \frac{e^{z_{im}}}{\sum_{l=1}^{M} e^{z_{il}}}$$

# Softmax regression: numerical issue 2, $z \ll 0$

- **Problem** If $z_{im}$ is large negative, then $e^{z_{im}} = \texttt{0.0}$.
- **Solution** Compute softmax and loss in one step

$$
\begin{aligned}
L_i &= -\sum_{m=1}^{M} \tilde{y}_{im} \ln(p_{im}) \\
&= -\sum_{m=1}^{M} \tilde{y}_{im} \ln\left(\frac{e^{z_{im}}}{\sum_{l=1}^{M} e^{z_{il}}}\right) \\
&= -\sum_{m=1}^{M} \tilde{y}_{im} \left(z_{im} - \ln\left(\sum_{l=1}^{M} e^{z_{il}}\right)\right)
\end{aligned}
$$

# Linear and logistic regression
# Multi-dim input - Multi-dim output

## Linear regression

**Output**

$$\mathbf{y}_i = [y_{i1}, \ldots, y_{iM}]^\mathsf{T} \in \mathbb{R}^M$$

**Model**

$$z_{im} = \mathbf{w}_m^\mathsf{T}\mathbf{x}_i + b_m$$

**Loss**

$$L_i = \sum_{m=1}^{M} (y_{im} - z_{im})^2$$

## Logistic regression

**Output**

$$\tilde{\mathbf{y}}_i = [\tilde{y}_{i1}, \ldots, \tilde{y}_{iM}]^\mathsf{T}, \quad \tilde{y}_{im} = \mathbb{I}_{[y_i = m]}$$

**Model**

$$z_{im} = \mathbf{w}_m^\mathsf{T}\mathbf{x}_i + b_m$$

**Loss**

$$L_i = \sum_{m=1}^{M} \tilde{y}_{im}\big(\ln(\textstyle\sum_{l=1}^{M} e^{z_{il}}) - z_{im}\big)$$

We find $\mathbf{w}_{1:M}$ and $b_{1:M}$ by minimizing sum of the losses

$$\widehat{\mathbf{w}}_{1:M}, \widehat{b}_{1:M} = \underset{\mathbf{w}_{1:M}, b_{1:M}}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} L_i$$

# Linear and logistic regression
# Multi-dim input - Multi-dim output

## Linear regression

**Output**

$$\mathbf{y}_i = [y_{i1}, \ldots, y_{iM}]^\mathsf{T} \in \mathbb{R}^M$$

**Model**

$$\mathbf{z}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

**Loss**

$$L_i = \|\mathbf{y}_i - \mathbf{z}_i\|^2$$

## Logistic regression

**Output**

$$\tilde{\mathbf{y}}_i = [\tilde{y}_{i1}, \ldots, \tilde{y}_{iM}]^\mathsf{T}, \quad \tilde{y}_{im} = \mathbb{I}_{[y_i = m]}$$

**Model**

$$\mathbf{z}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

**Loss**

$$L_i = \sum_{m=1}^{M} \tilde{y}_{im} \big( \ln(\textstyle\sum_{l=1}^{M} e^{z_{il}}) - z_{im} \big)$$

We find $\mathbf{W}$ and $\mathbf{b}$ by minimizing sum of the losses

$$\widehat{\mathbf{W}}, \widehat{\mathbf{b}} = \underset{\mathbf{W},\mathbf{b}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L_i$$

# Hand-in assignment 1

- The first step of the hand in assignment is implement the softmax regression.
- Similar to the pre-course exercise we have to work out the derivatives

$$\frac{dJ}{db_m} = \sum_{i=1}^{n} \frac{dJ}{dz_{im}} \frac{dz_{im}}{db_m}, \qquad m = 1, \ldots, M$$

$$\frac{dJ}{dw_{mj}} = \sum_{i=1}^{n} \frac{dJ}{dz_{im}} \frac{dz_{im}}{dw_{mj}} \qquad m = 1, \ldots, M, \quad j = 1, \ldots, p$$

What is $\frac{dJ}{dz_{im}}$, $\frac{dz_{im}}{db_m}$ and $\frac{dz_{im}}{dw_{mj}}$ for the softmax regression model?

# Feedforward neural networks

# Constructing neural networks for regression

A **neural network (NN)** is a nonlinear function $\hat{y} = f(\mathbf{x}; \boldsymbol{\theta})$ from an input $\mathbf{x}$ to an output $\hat{y}$ parameterized by parameters $\boldsymbol{\theta}$.

---

**Linear regression** models the relationship between a continuous output $y$ and a continuous input $\mathbf{x}$,

$$\hat{y} = \sum_{j=1}^{p} W_j x_j + b = \mathbf{W}\mathbf{x} + b,$$

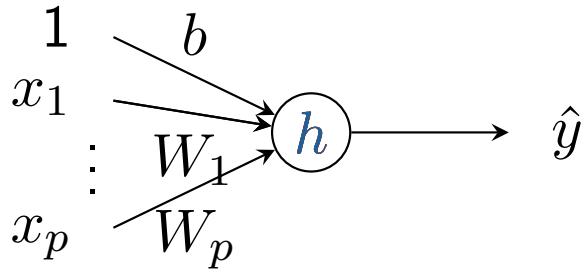
where the parameters $\boldsymbol{\theta}$ are the **weights** $W_j$, and the **offset** (aka bias/intercept) term $b$,

$$\boldsymbol{\theta} = \begin{bmatrix} b & \mathbf{W} \end{bmatrix}^{\mathsf{T}}, \qquad \mathbf{W} = \begin{bmatrix} W_1 & W_2 & \cdots & W_p \end{bmatrix}$$

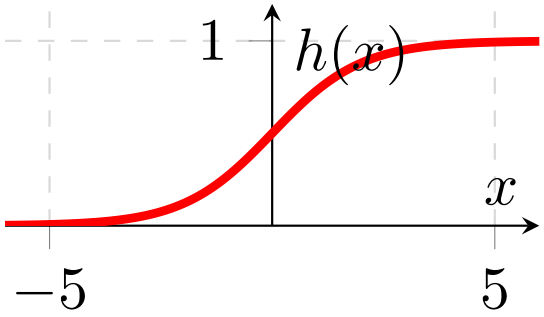
# Generalized linear regression

We can generalize this by introducing nonlinear transformations of the predictor $\mathbf{W}\mathbf{x} + b$,

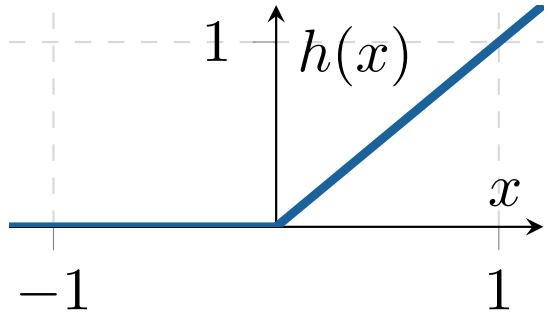
$$\hat{y} = h(\mathbf{W}\mathbf{x} + b),$$




We call $h(x)$ the **activation function**. Two common choices are:




Sigmoid: $h(x) = \frac{e^x}{1+e^x}$



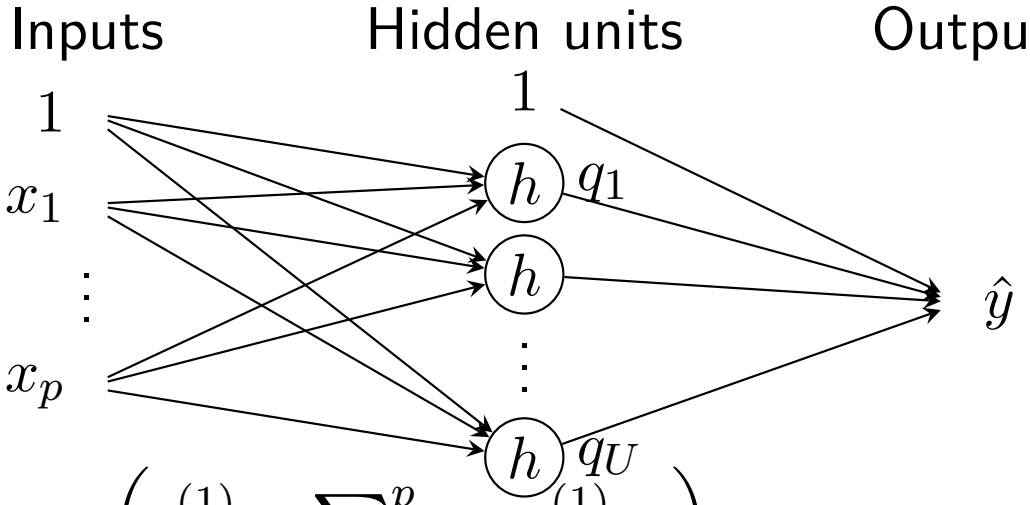

ReLU: $h(x) = \max(0, x)$

---

Let us consider an example of a **neural network**.

# Neural network - construction

A neural network is a sequential construction of **several** generalized linear regression models.




$$q_1 = h\left(b_1^{(1)} + \sum_{j=1}^{p} W_{1j}^{(1)} x_j\right)$$

$$q_2 = h\left(b_2^{(1)} + \sum_{j=1}^{p} W_{2j}^{(1)} x_j\right)$$

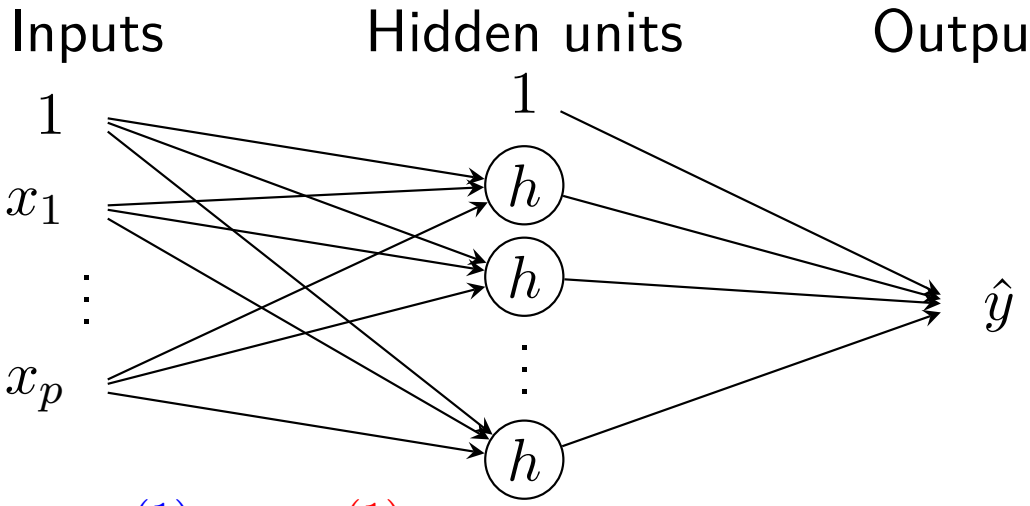
$$\vdots$$

$$q_U = h\left(b_U^{(1)} + \sum_{j=1}^{p} W_{Uj}^{(1)} x_j\right)$$

$$\hat{y} = b^{(2)} + \sum_{i=1}^{U} W_i^{(2)} q_i$$

# Neural network - construction

A neural network is a sequential construction of **several** generalized linear regression models.




$$\mathbf{q} = h(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\hat{y} = \mathbf{W}^{(2)}\mathbf{q} + \mathbf{b}^{(2)}$$

$$\mathbf{W}^{(1)} = \begin{bmatrix} W_{11}^{(1)} & \dots & W_{1p}^{(1)} \\ \vdots & & \vdots \\ W_{U1}^{(1)} & \dots & W_{Up}^{(1)} \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ \vdots \\ b_U^{(1)} \end{bmatrix}, \quad \mathbf{q} = \begin{bmatrix} q_1 \\ \vdots \\ q_U \end{bmatrix}$$

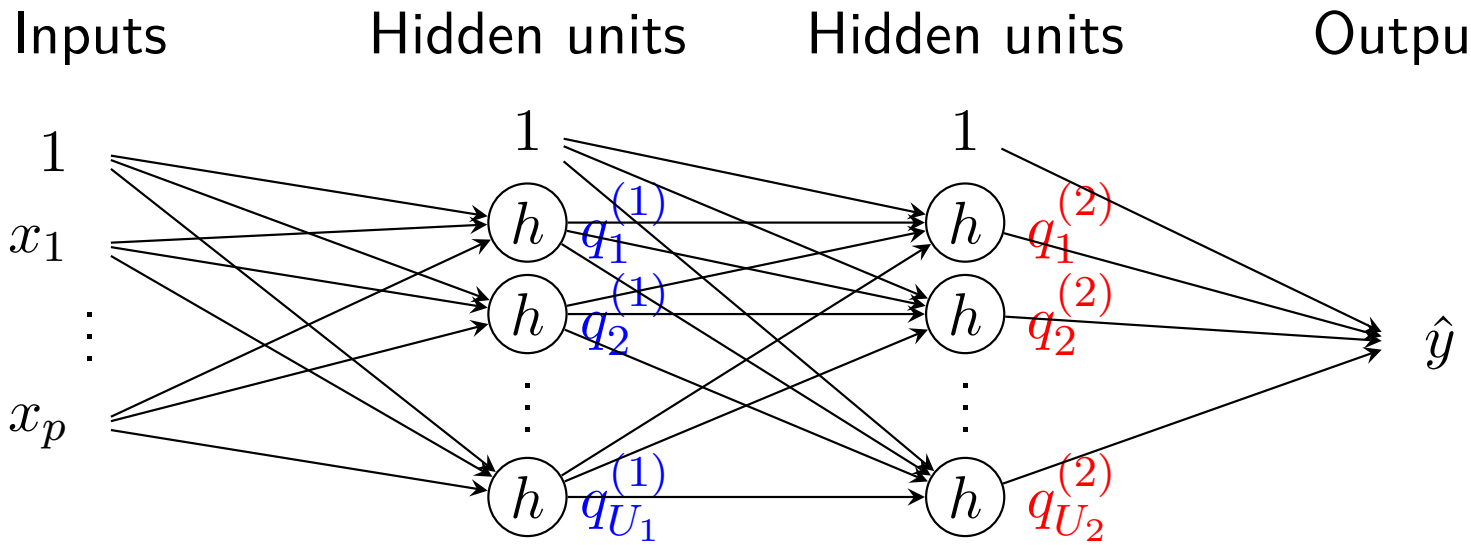
Weight matrix Offset vector Hidden units

$$\mathbf{b}^{(2)} = \begin{bmatrix} b^{(2)} \end{bmatrix}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} W_1^{(2)} & \dots & W_U^{(2)} \end{bmatrix}$$

# Neural network - construction

A neural network is a **sequential** construction of several generalized linear regression models.




$$\mathbf{q}^{(1)} = h(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{q}^{(2)} = h(\mathbf{W}^{(2)}\mathbf{q}^{(1)} + \mathbf{b}^{(2)})$$
$$\hat{y} = \mathbf{W}^{(3)}\mathbf{q}^{(2)} + \mathbf{b}^{(3)}$$

The model learns better using a deep network (several layers) instead of a wide and shallow network.

# Deep learning

A neural network with $L$ layers can be written as

$$\begin{aligned}\mathbf{h}^{(0)} &= \mathbf{x} \\ \mathbf{h}^{(l)} &= \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right), \quad l = 1, \ldots, L-1 \\ \hat{y} &= \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}\end{aligned}$$

All weight matrices and offset vectors in all layers combined are the parameters of the network

$$\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\},$$

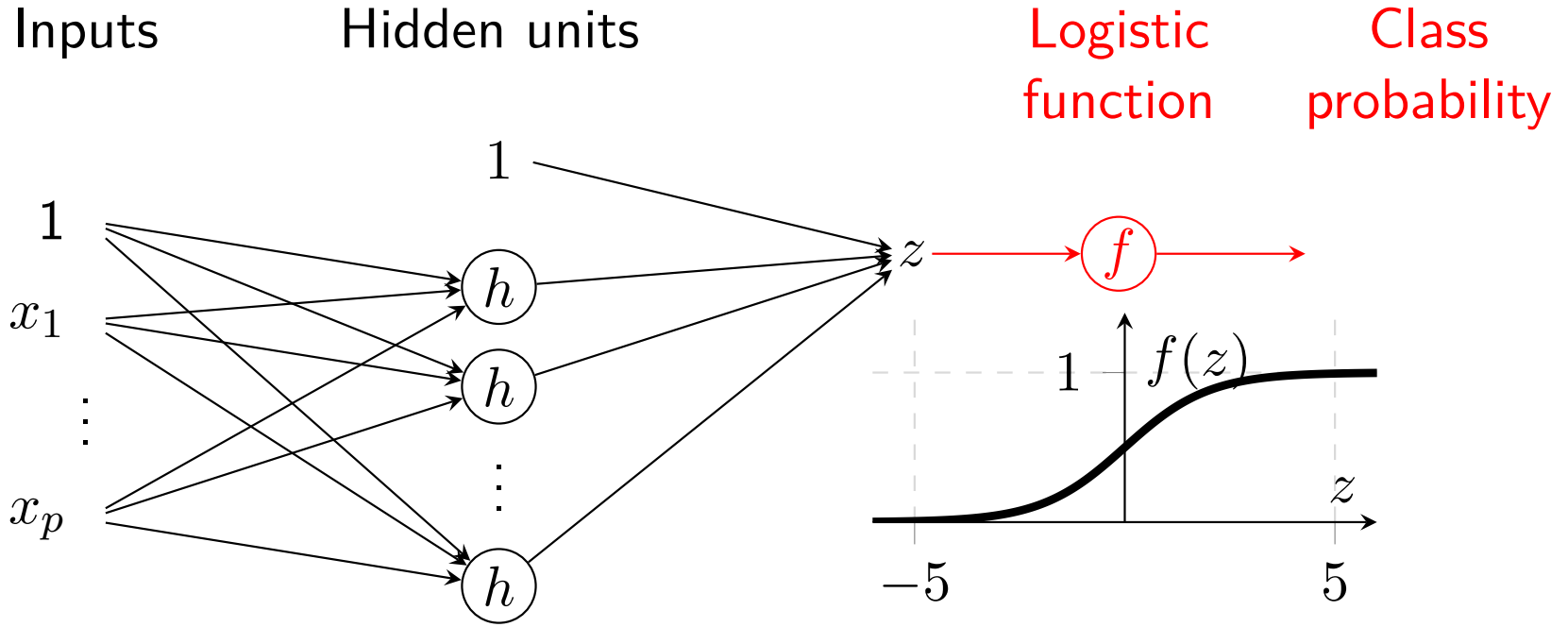
which constitutes the parametric model $\hat{y} = f(\mathbf{x}; \boldsymbol{\theta})$. If $L$ is large we call it a deep neural network.

**Deep learning** is a class of machine learning models and algorithms that use a cascade of multiple layers, each of which is a nonlinear transformation.

# NN for classification ($M = 2$ classes)

We can also use neural networks for classification. With $M = 2$ classes we squash the output through the logistic function to get a **class probability** $f(z) \in [0, 1]$

$$f(z) = p(y = 1|\mathbf{x}, \boldsymbol{\theta}) \quad \text{where} \quad f(z) = \frac{e^z}{1 + e^z}$$

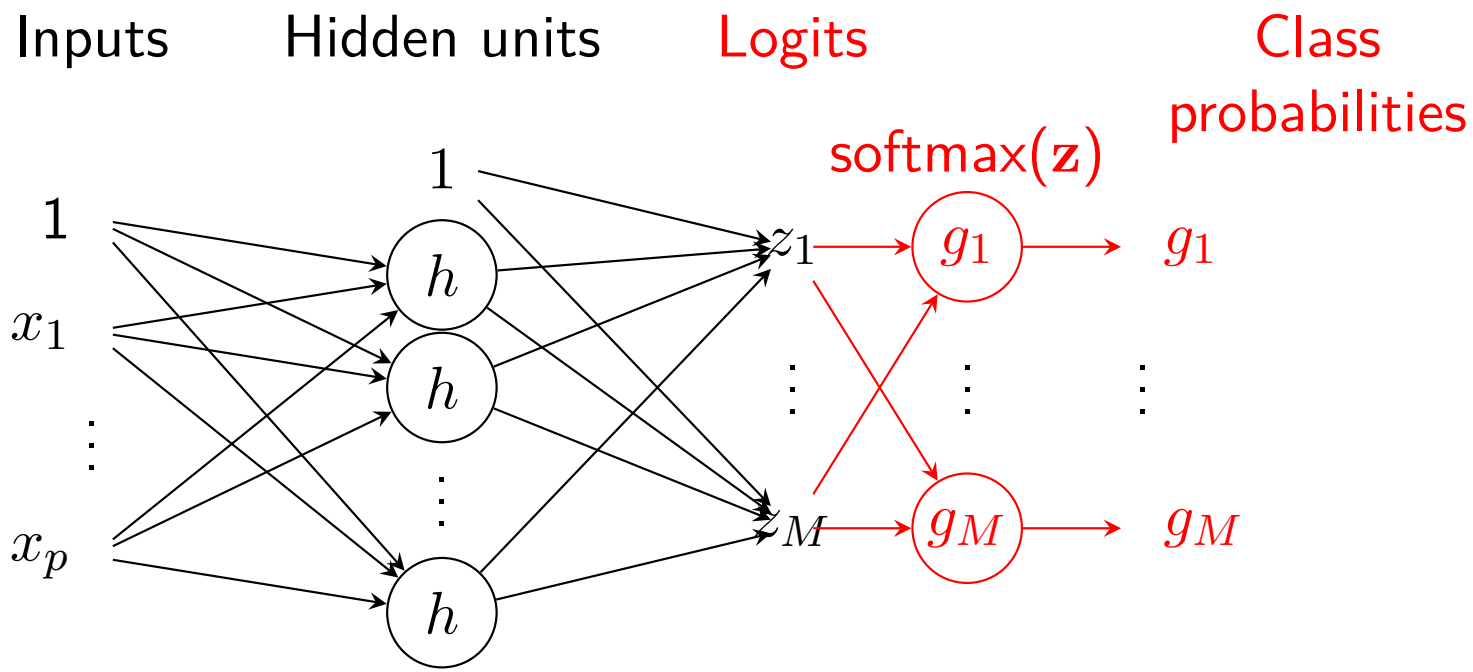



**What if $M > 2$ ?**

# NN for classification ($M > 2$ classes)

For $M > 2$ classes we want to predict the class probability for all $M$ classes $g_m(\mathbf{z}) = p(y = m|\mathbf{x}, \boldsymbol{\theta})$. We extend the logistic function to the **softmax function**

$$g_m(\mathbf{z}) = \frac{e^{z_m}}{\sum_{l=1}^{M} e^{z_l}}, \qquad m = 1, \ldots, M.$$



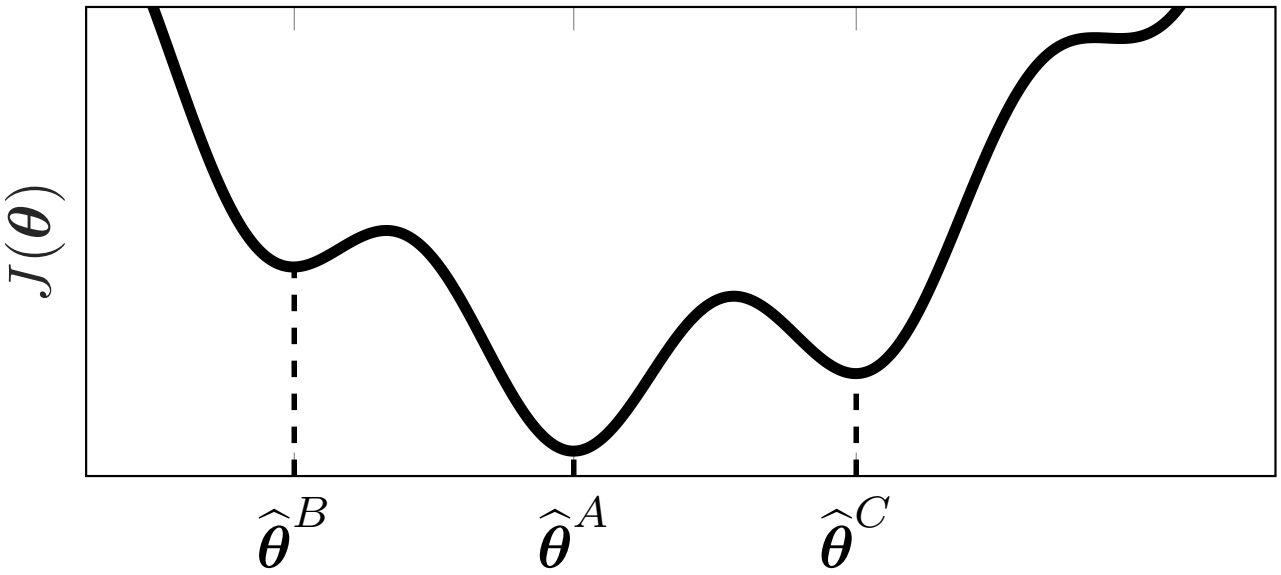

$$\text{softmax}(\mathbf{z}) = [g_1(\mathbf{z}), \ldots, g_M(\mathbf{z})]^\mathsf{T}$$

# Optimization + training/test data

# Unconstrained numerical optimization

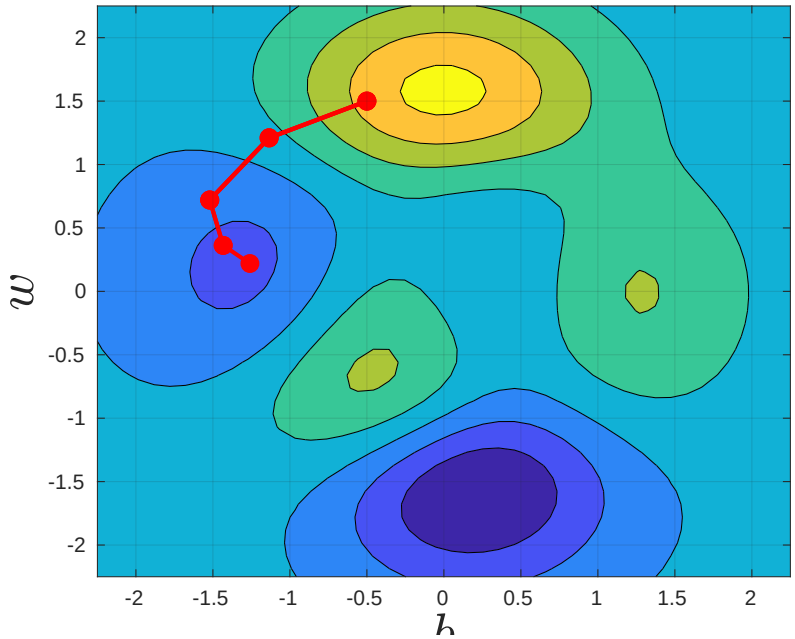
We train logistic regression (and later neural networks) by considering an optimization problem

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \qquad J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\mathbf{x}_i, \mathbf{y}_i, \boldsymbol{\theta})$$




- The best possible solution $\widehat{\boldsymbol{\theta}}$ is the **global minimizer** ($\widehat{\boldsymbol{\theta}}^A$)
- The global minimizer is typically very hard to find, and we have to settle for a **local minimizer** ($\widehat{\boldsymbol{\theta}}^A$, $\widehat{\boldsymbol{\theta}}^B$, $\widehat{\boldsymbol{\theta}}^C$)

# Iterative solution (gradient descent) - Example 2D




$$\boldsymbol{\theta} = [b,\ w]^{\mathsf{T}} \in \mathbb{R}^2$$

1. Pick a $\boldsymbol{\theta}_0$
2. while(*not converged*)
   - Update $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma \mathbf{g}_t,$ where $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
   - Update $t := t + 1$

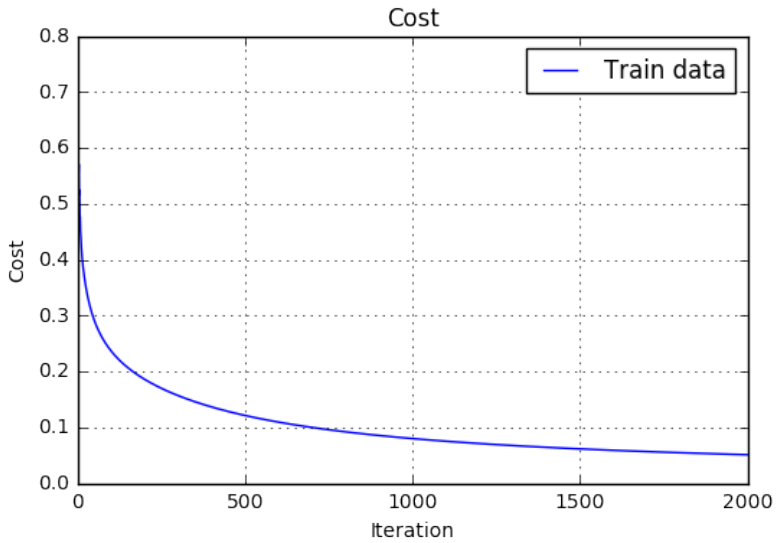
We call $\gamma \in \mathbb{R}^+$ the **step length** or **learning rate**.

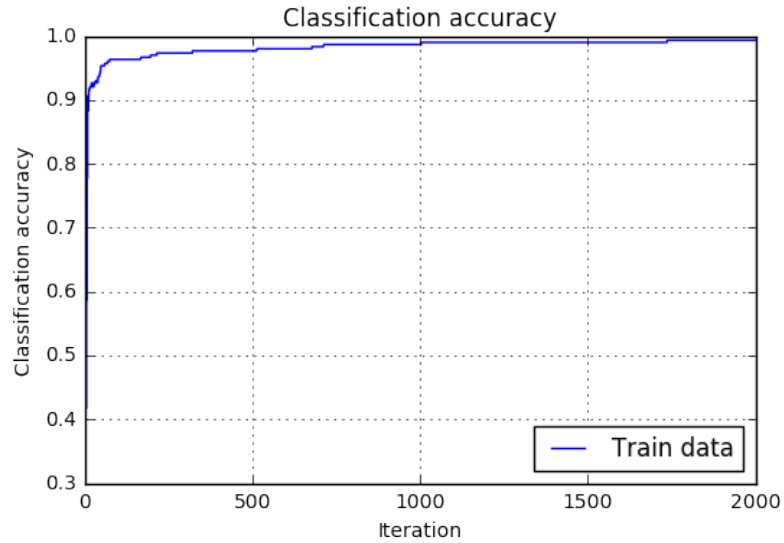
- More about picking $\gamma$ (+ more stuff optimization) in Thomas' lecture next week.

# Performance on training data

Consider a logistic regression problem

Plot the cost and the classification accuracy vs. iteration number




Cost:

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x}_i, y_i, \boldsymbol{\theta})$$

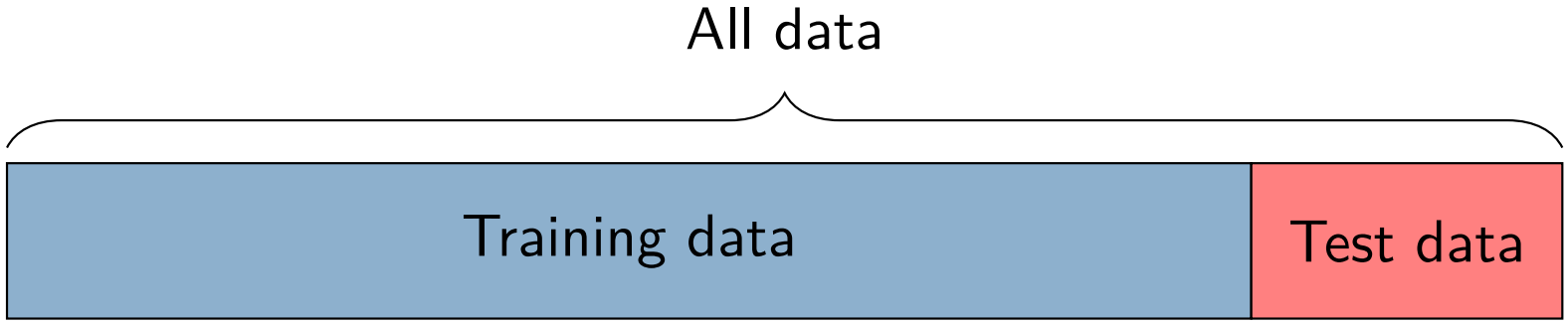
Classification accuracy:

$$\frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\hat{y}_i = y_i)$$

Is the classification accuracy on training data a fair estimate of the actual performance of the model?

# Training/test data split

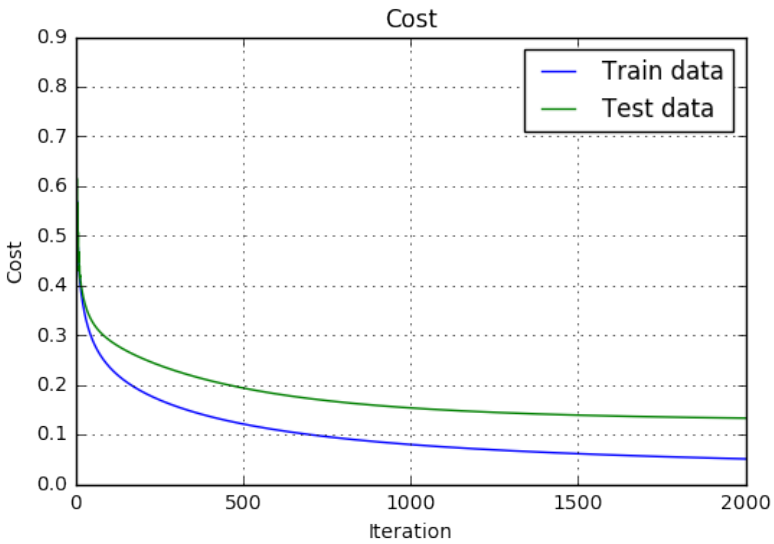
We want to evaluate the model's performance on data which has **not** been used during training.



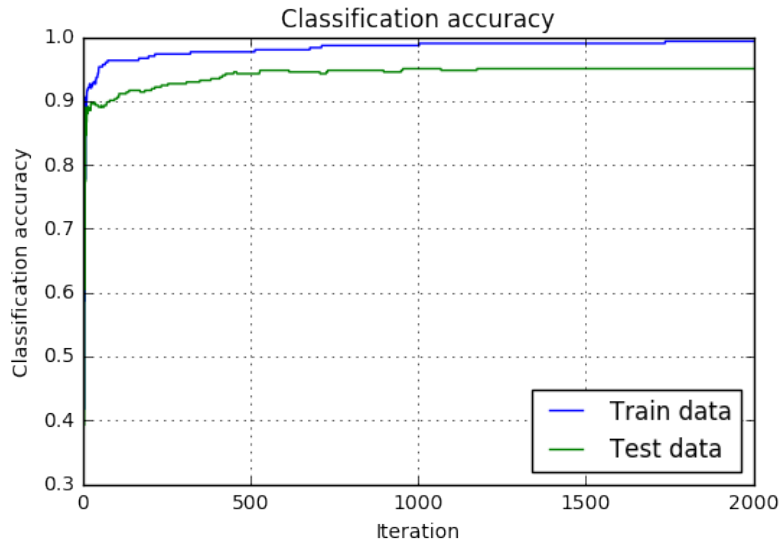

Split your available data into **training data** and **test data**

1. Use the training data to **learn** the model
2. Use the test data to **evaluate** the model's performance

# Performance on training and test data




Cost: $J(\boldsymbol{\theta}) = \frac{1}{n} L(\mathbf{x}_i, y_i, \boldsymbol{\theta})$




Classification accuracy: $\frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\hat{y}_i = y_i)$

We still only train on training data but in these plots we evaluate cost and classification accuracy on both training and test data.

Difference between performance on training and test data is called the **generalization error**. More about this in lecture 6.

# Why do deep neural networks work so well?

# Why do deep neural networks work so well?

Example: Image classification

**Input:** pixels of an **image**
**Output:** **object identity**

- 1 megapixel (black/white) $\Rightarrow$ $2^{1'000'000}$ possible images!
- A **deep neural network** can solve this with a few million parameters!

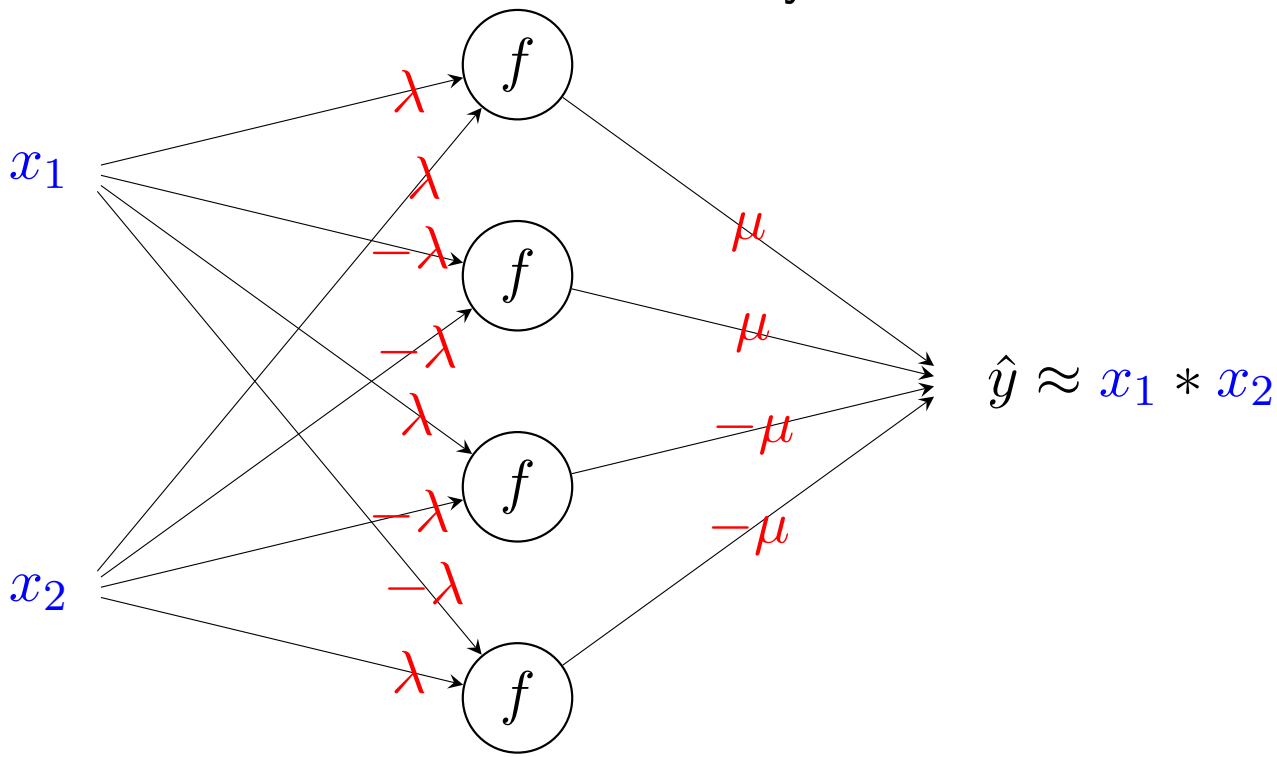
Image removed in handout version due to copyright ©Springer International Publishing

**How can deep neural networks work so well?**

# Why neural networks?

## Continuous multiplication gate

A neural network with only four hidden units can model multiplication of two numbers arbitrarily well.




If we choose $\mu = \frac{1}{4\lambda^2 f''(0)}$ then $\hat{y} \to x_1 * x_2$ when $\lambda \to 0$.

Henry W. Lin and Max Tegmark. (2016) **Why does deep and cheap learning work so well?**, *arXiv*

# A regression example

Input: $\mathbf{x} \in \mathbb{R}^{1000}$
Output: $y \in \mathbb{R}$
Task: Model a quadratic relationship between $y$ and $\mathbf{x}$

## Linear regression

$$\hat{y} = w_{1,1}x_1x_1 + w_{1,2}x_1x_2 + \cdots + w_{1000,1000}x_{1000}x_{1000} = \mathbf{w}^\mathsf{T}\tilde{\mathbf{x}}$$

where

$$\tilde{\mathbf{x}} = \begin{bmatrix} x_1x_1 & x_1x_2 & \dots & x_{1000}x_{1000} \end{bmatrix}^\mathsf{T}$$

$$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1000,1000} \end{bmatrix}^\mathsf{T}$$

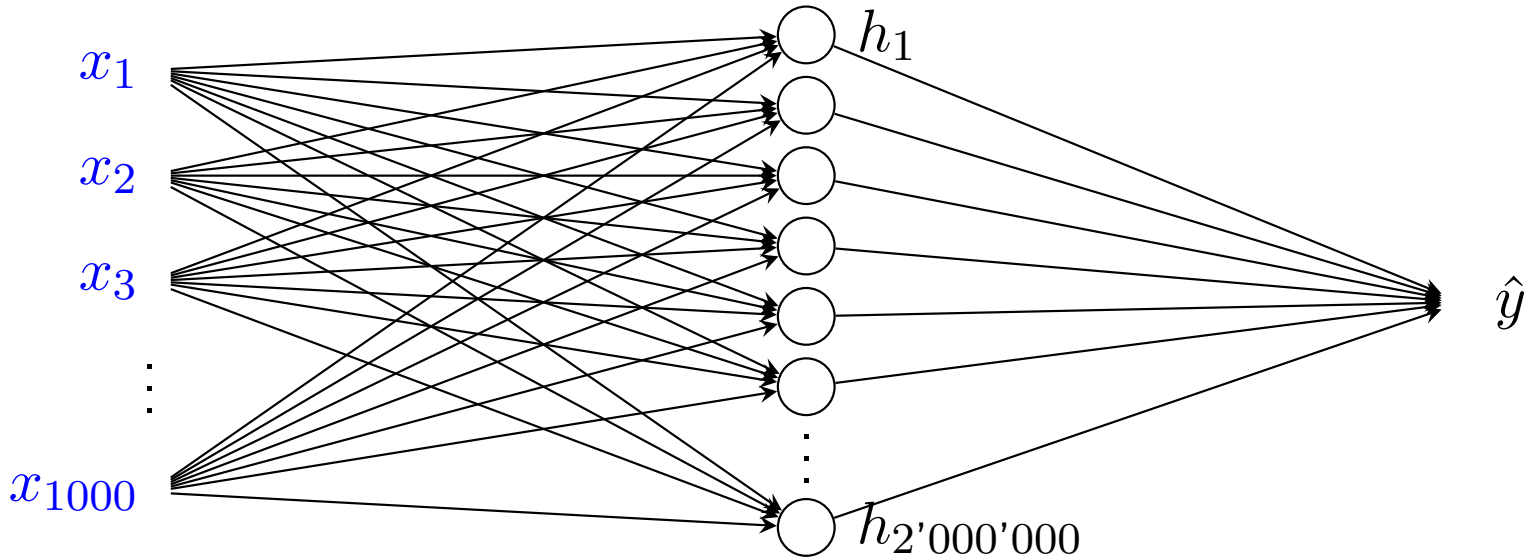
Requires $\approx \frac{1'000*1'000}{2} = 500'000$ parameters!

# A regression example

Input: $\mathbf{x} \in \mathbb{R}^{1000}$
Output: $y \in \mathbb{R}$
Task: Model a quadratic relationship between $y$ and $\mathbf{x}$

## Neural network

To model all products with a neural network we would need $4 * 500'000 = 2 * 10^6$ hidden units and hence 2 billion parameters...




$1000 * (2 * 10^6)$ + $2 * 10^6 \approx 2 * 10^9$ param.

# A regression example (cont.)

Input: $\mathbf{x} \in \mathbb{R}^{1000}$
Output: $y \in \mathbb{R}$
Task: Model a quadratic relationship between $y$ and $\mathbf{x}$

**Assume that only 10 of the regressors $x_i x_j$ are of importance**

**Linear regression**

$$\hat{y} = w_{1,1}x_1x_1 + w_{1,2}x_1x_2 + \cdots + w_{1000,1000}x_{1000}x_{1000} = \mathbf{w}^\mathsf{T}\tilde{\mathbf{x}}$$

where

$$\tilde{\mathbf{x}} = \begin{bmatrix} x_1x_1 & x_1x_2 & \dots & x_{1000}x_{1000} \end{bmatrix}^\mathsf{T}$$

$$\mathbf{w} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1000,1000} \end{bmatrix}^\mathsf{T}$$

500'000 parameters are still required!

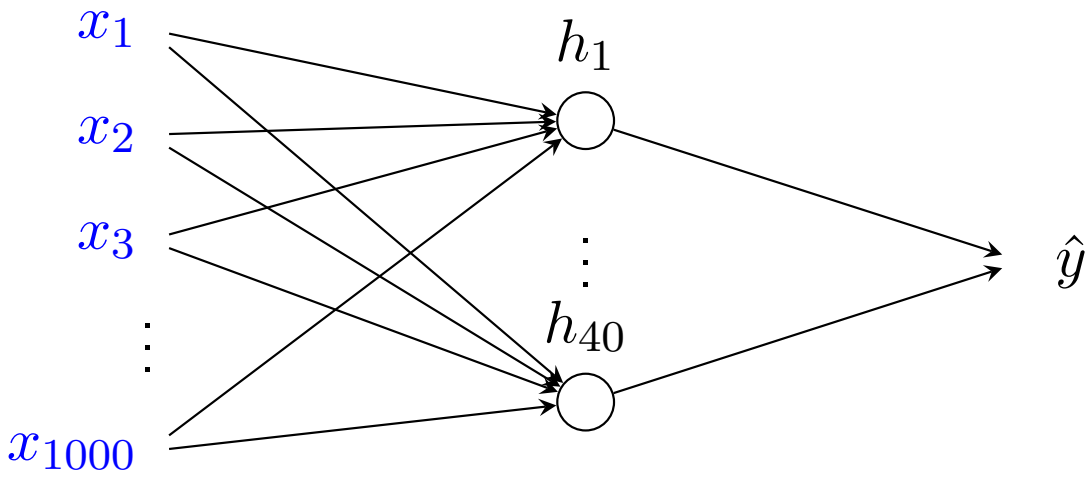
# A regression example (cont.)

Input: $\mathbf{x} \in \mathbb{R}^{1000}$
Output: $y \in \mathbb{R}$
Task: Model a quadratic relationship between $y$ and $\mathbf{x}$

**Assume that only 10 of the regressors $x_i x_j$ are of importance**

**Neural network**

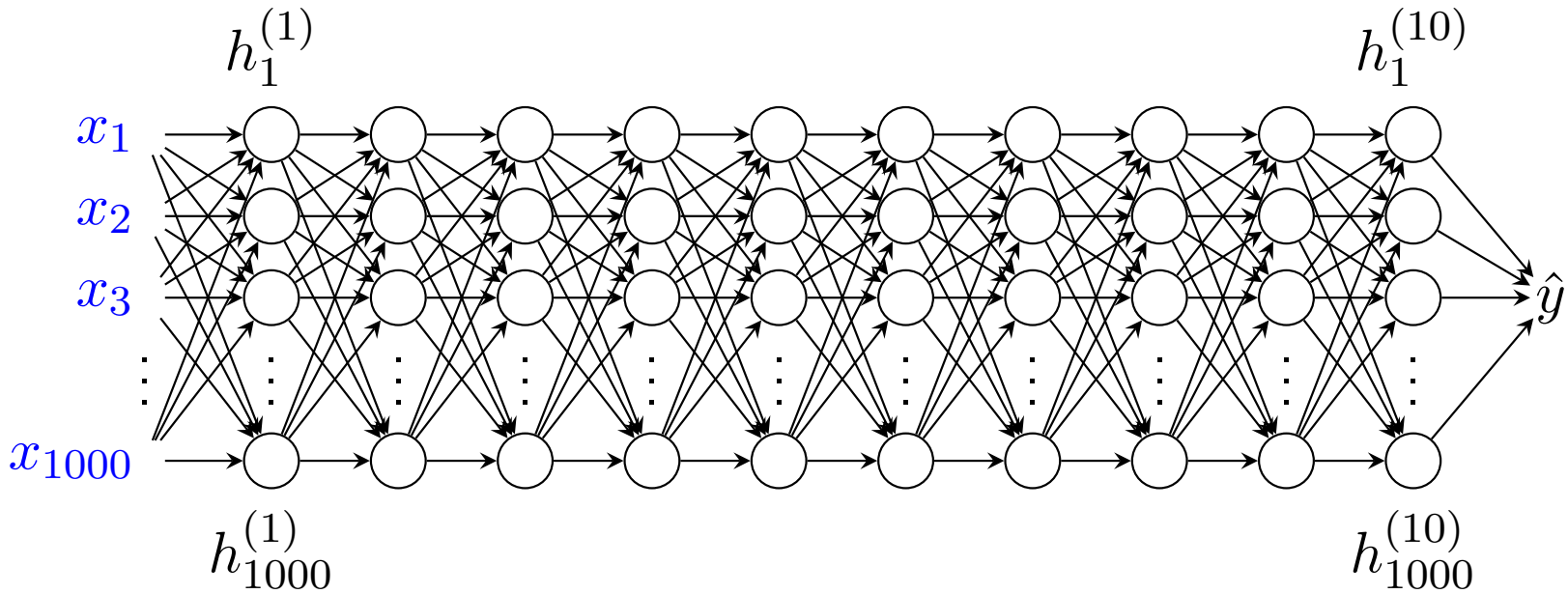
To model 10 products with a neural network we would need 4*10 hidden units, i.e. leading to only $\approx$ 40'000 parameters!




1000*40 + 40 = 40'040

# Why deep? - A regression example

- Consider the same example. Now we want a model with complexity corresponding to polynomials of degree 1'000.
- Keep 250 products in each layer $\Rightarrow 250{*}4{=}1'000$ hidden units.




Linear regression would require $\approx \frac{1000^{1000}}{1000!}$ parameters to model such a relationship...

# Why do deep neural networks work so well?

Conclusions:

- Deep neural networks are very **parameter efficient** in modeling complex nonlinear functions.
- In the ideal case the descriptive power increases **exponentially** with the number of layers.
- This requires some **structure** in the data which can be exploited by the model.

# Some comments - Why now?

Neural networks have been around for more than fifty years. Why have they become so popular now (again)?

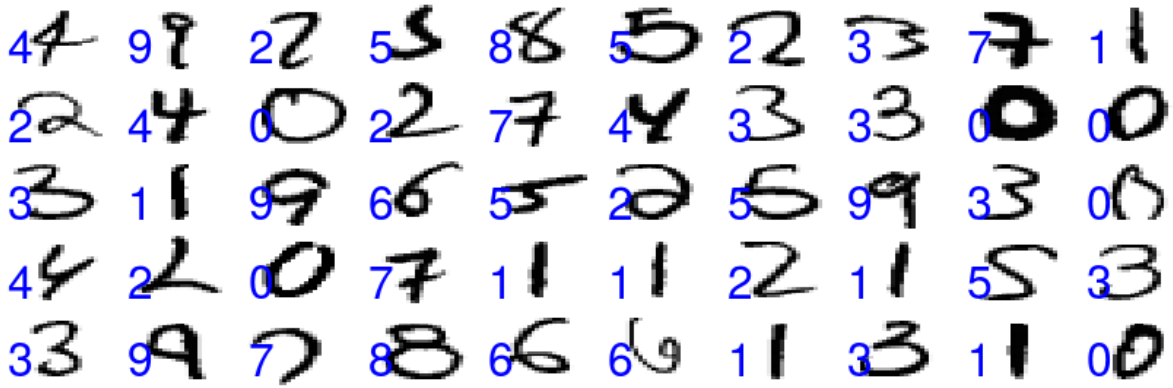
To solve really interesting problems you need:

1. Efficient learning **algorithms**
2. Efficient computational **hardware**
3. A lot of labeled **data**!

These three factors have not been fulfilled to a satisfactory level until the last 5-10 years.

# Hand-in assignment 1 - Classifying hand-written digits with neural networks

**Input** $\mathbf{x}$: Images of hand-written digits with $p = 784$ pixels
**Output** $y$: The digit that the image depicts: 0,1,2,3,4,5,6,7,8 or 9



Extend your code from pre-course assignment in three aspects:

1. Adding **softmax** function on the output to handle a classification problem with $M > 2$ classes.
2. Train with **minibatch** instead of batch gradient descent.
3. Extend the model to include **multiple layers**.

# A few concepts to summarize lecture 2

**Neural network (NN):** A nonlinear parametric model constructed by stacking several linear models with intermediate nonlinear activation functions.

**Activation function:** (a.k.a squashing function) A nonlinear scalar function applied to each output element of the linear models in a NN.

**Hidden units:** Intermediate variables in the NN which are not observed, i.e. belongs neither to the input nor output data.

**Softmax function:** Generalization of the logistic function to mutliple dimensions. Often used as the last activation function in a NN for multi-class classification problems.

**Gradient descent:** Iterative optimization algorithm for finding the minimum of a function by following its gradient.

**Training data:** The dataset that is used to train the model.

**Test data:** The dataset that is used to evaluate the model.