15-780: Graduate Artificial Intelligence

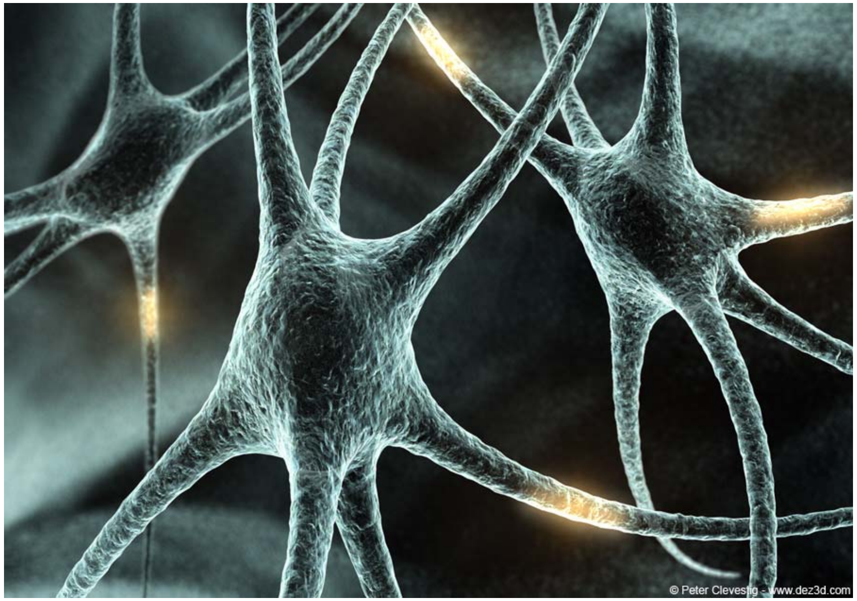
Neural networks

Mimicking the brain

- In the early days of AI there was a lot of interest in developing models that can mimic human thinking.
- While no one knew exactly how the brain works (and, even though there was a lot of progress since, there is still little known), some of the basic computational units were known
- A key component of these units is the neuron.

The Neuron

- A cell in the brain
- Highly connected to other neurons
- Thought to perform computations by integrating signals from other neurons
- Outputs of these computation may be transmitted to one or more neurons



© Peter Clevestig - www.dez3d.com

What can we do with NN?

- Classification

  - We already mentioned many useful applications

- Regression

  - A new concept:

    Input: Real valued variables

    Output: One or more real values

- Examples:

  - Predict the price of Googles stock from Microsofts stock

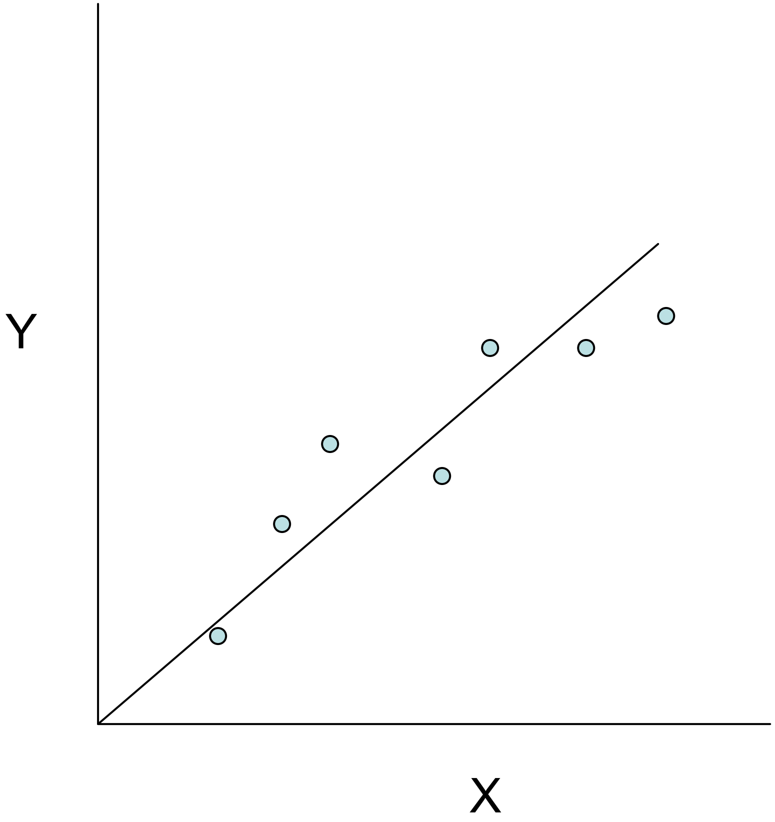
  - Predict distance to obstacle from various sensors

Linear regression

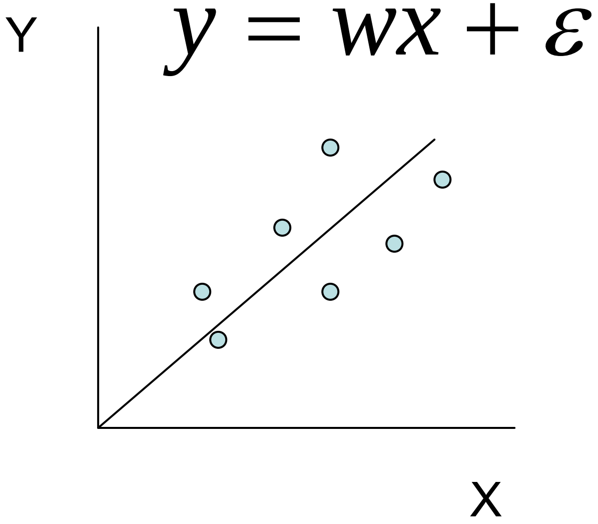
- Given an input x we would like to compute an output y
- In linear regression we assume that y in x are related with the following equation:

$$y = wx+\varepsilon$$

where w is a parameter and ε represents measurement or other noise



Linear regression



$$y = wx + \varepsilon$$

- Our goal is to estimate w from a training data of $<x_i,y_i>$ pairs

- This could be done using a least squares approach

$$\arg\min_w \sum_i (y_i - wx_i)^2$$

- Why least squares?

  - minimizes squared distance between measurements and predicted line

  - has a nice probabilistic interpretation

  - easy to compute

If the noise is Gaussian with mean 0 then least squares is also the maximum likelihood estimate of w

Solving linear regression

- You should be familiar with this by now …
- We just take the derivative w.r.t. to w and set to 0:

$$\frac{\partial}{\partial w}\sum_i (y_i - wx_i)^2 = 2\sum_i - x_i(y_i - wx_i) \Rightarrow$$

$$2\sum_i x_i(y_i - wx_i) = 0 \Rightarrow$$

$$\sum_i x_i y_i = \sum_i wx_i^2 \Rightarrow$$

$$w = \frac{\sum_i x_i y_i}{\sum_i x_i^2}$$

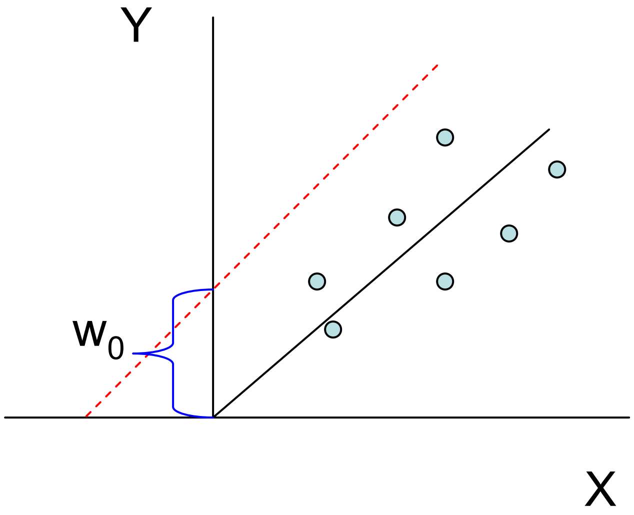
Regression example

Affine regression

- So far we assumed that the line passes through the origin
- What if the line does not?
- No problem, simply change the model to

$$y = w_0 + w_1 x + \varepsilon$$

- Can use least squares to determine w_0 , w_1



$$w_0 = \frac{\sum_i y_i - w_1 x_i}{n} \qquad w_1 = \frac{\sum_i x_i (y_i - w_0)}{\sum_i x_i^2}$$

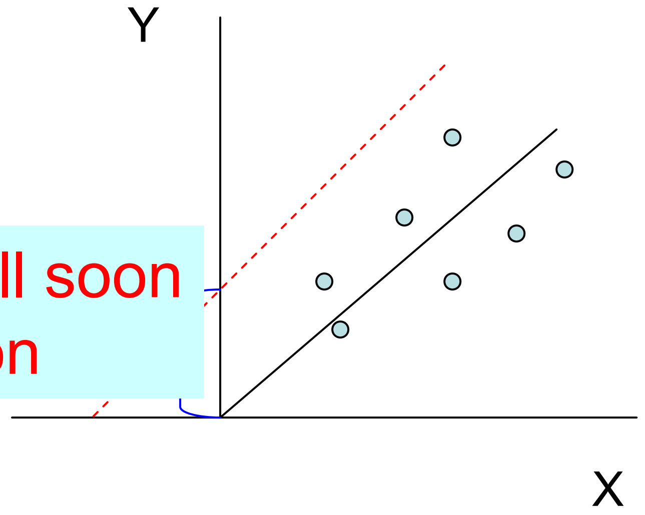
Affine regression

- So far we assumed that the line passes through the origin
- What if the line does not?
- No problem, simply change the model to

y = w

Just a second, we will soon give a simpler solution

- Can use least squares to determine w_0 , w_1



$$w_0 = \frac{\sum_i y_i - w_1 x_i}{n} \qquad w_1 = \frac{\sum_i x_i (y_i - w_0)}{\sum_i x_i^2}$$

Multivariate regression

- What if we have several inputs?

  - Stock prices for Yahoo, Microsoft and Ebay for the Google prediction task

- This becomes a multivariate regression problem
- Again, its easy to model:

$$y = w_0 + w_1x_1 + \ldots + w_kx_k + \varepsilon$$

Notations:

Lower case: variable or parameter (w_0)

Lower case bold: vector (**w**)

Upper case bold: matrix (**X**)

Multivariate regression: Least squares

- We are now interested in a vector **w**^{T} = [w$_0$, w$_1$,… , w$_k$]
- It would be useful to represent this in matrix notations:

$$\mathbf{X} = \begin{bmatrix} \vdots & \vdots & \vdots \\ X_1 & \cdots & X_n \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} 1 & x_{11} \cdots x_{n1} \\ 1 & x_{12} \cdots x_{n2} \\ & \vdots \\ 1 & x_{1k} \cdots x_{nk} \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

- We can thus re-write our model as $\mathbf{y} = \mathbf{w}^T\mathbf{X}+\varepsilon$

- The solution turns out to be: $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

- The is an instance of a larger set of computational solutions which are usually referred to as ‘generalized least squares’

Multivariate regression: Least squares

- We can re-write our model as $\mathbf{y} = \mathbf{w}^T\mathbf{X}$
- The solution turns out to be: $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$
- The is an instance of a larger set of computational solutions which are usually referred to as 'generalized least squares'

- $\mathbf{X}^T\mathbf{X}$ is a k by k matrix
- $\mathbf{X}^T\mathbf{y}$ is a vector with k entries

Why is $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$ the right solution?

Hint: Multiply both sides by $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$

Multivariate regression: Least squares

- We can re-write our model as $\mathbf{y} = \mathbf{w}^T\mathbf{X}$
- The solution turns out to be: $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

We need to invert a k by k matrix

- This takes $O(k^3)$
- Depending on k this can be rather slow

Where we are

- Linear regression – solved!
- But

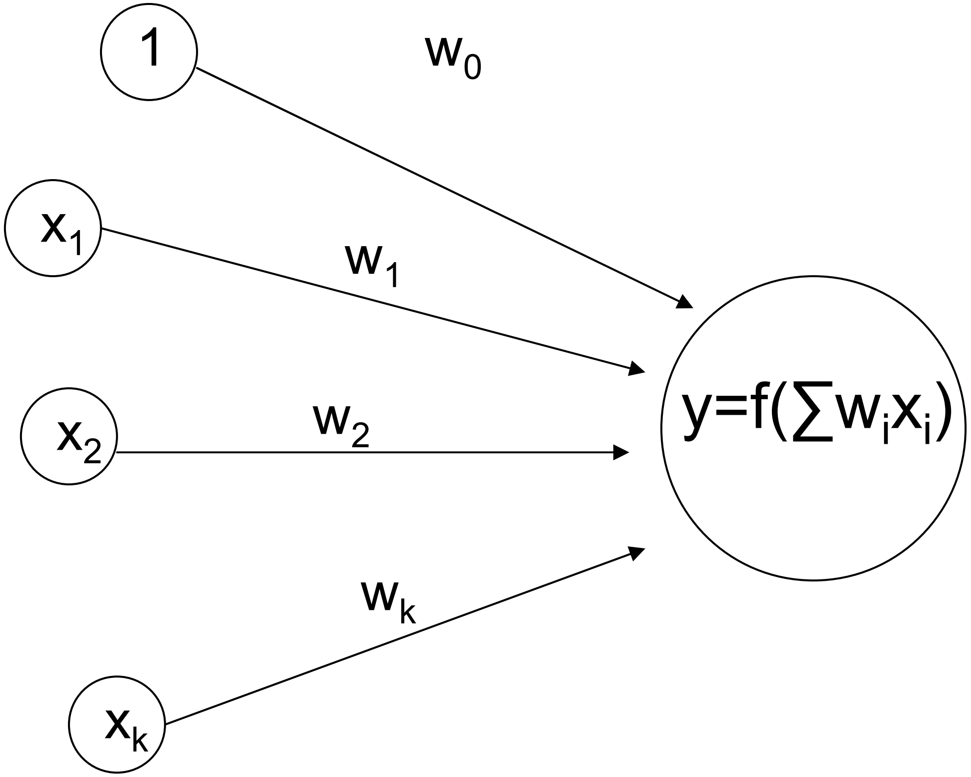
- Solution may be slow

- Does not address general regression problems of the form

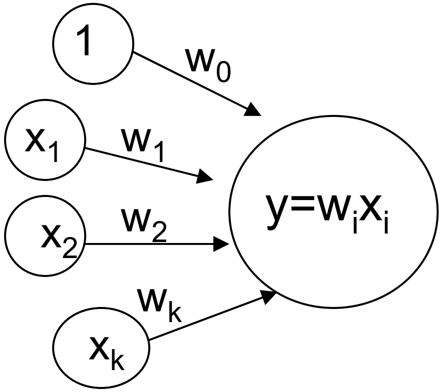
$$\mathbf{y} = f(\mathbf{w}^T\mathbf{x})$$

Back to NN: Preceptron

- The basic processing unit of a neural net



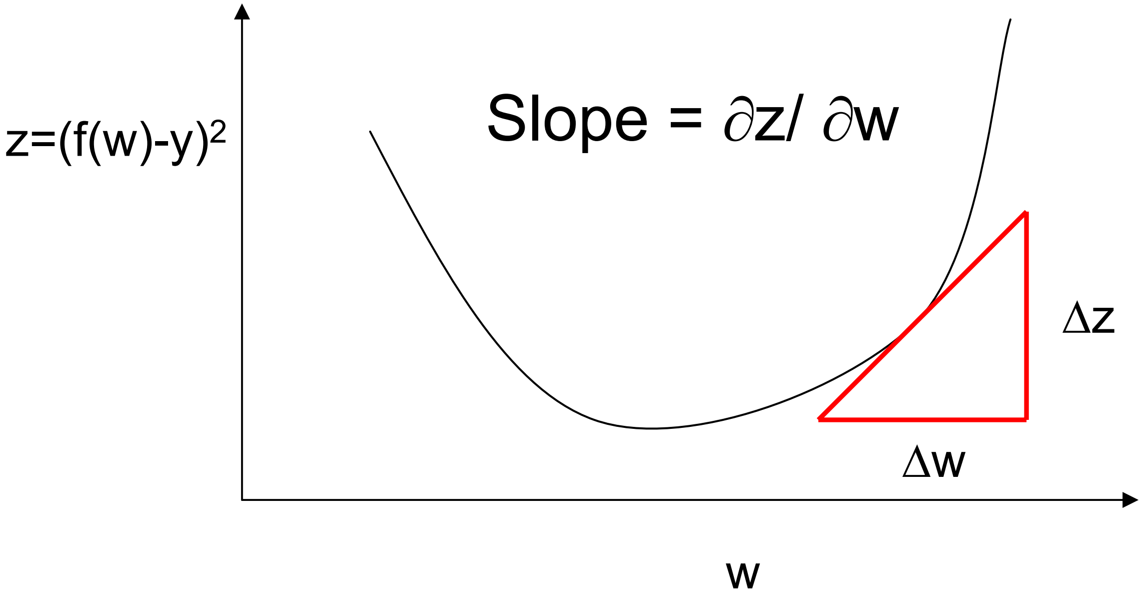
Linear regression



- Lets start by setting $f(\sum w_ix_i)=\sum w_ix_i$
- We are back to linear regression
- Unlike our original linear regression solution, for perceptrons we will use a different strategy
- Why?

  - We will discuss this later, for now lets focus on the solution …

Gradient descent



- Going in the *opposite* direction to the slope will lead to a smaller z

- But not too much, otherwise we would go beyond the optimal w

Gradient descent

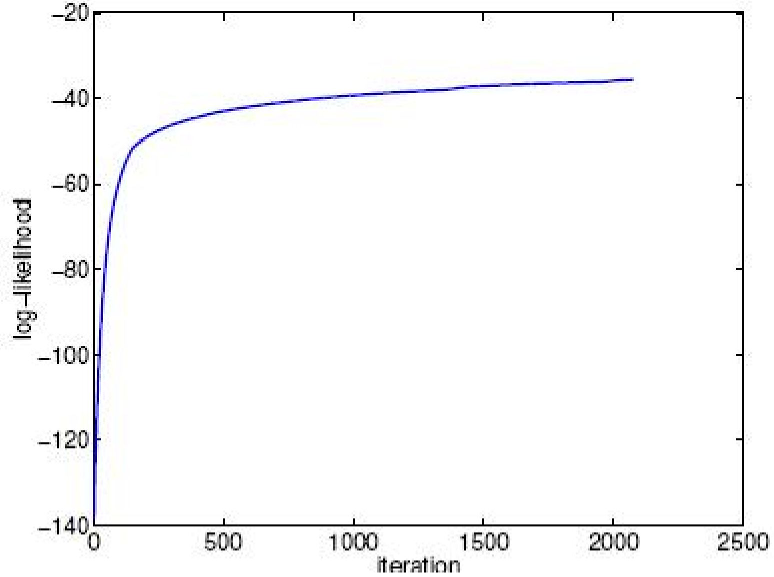
- Going in the *opposite* direction to the slope will lead to a smaller z
- But not too much, otherwise we would go beyond the optimal w
- We thus update the weights by setting:

$$w \leftarrow w - \lambda \frac{\partial z}{\partial w}$$

where λ is small constant which is intended to prevent us from passing the optimal w

Example when choosing the 'right' λ

- We get a monotonically decreasing error as we perform more updates



Gradient descent for linear regression

- We compute the gradient w.r.t. to each w_i

$$\frac{\partial}{\partial w_i}\left(\sum_k y - w_k x_k\right)^2 = -2x_i(\sum_k y - w_k x_k)$$

- And if we have n measurements then

$$\frac{\partial}{\partial w_i}\sum_{j=1}^{n}(y_j - \mathbf{w}^T\mathbf{x}_j)^2 = -2\sum_{j=1}^{n} x_{j,i}(y_j - \mathbf{w}^T\mathbf{x}_j)$$

where $x_{j,i}$ is the i’th value of the j’th input vector

Gradient descent for linear regression

- If we have n measurements then

$$\frac{\partial}{\partial w_i}\sum_{j=1}^{n}(y_j - \mathbf{w}^T\mathbf{x}_j)^2 = -2\sum_{j=1}^{n}x_{j,i}(y_j - \mathbf{w}^T\mathbf{x}_j)$$

- Set $\delta_j = (y_j - \mathbf{w}^T\mathbf{x}_j)$

- Then our update rule can be written as

$$w_i \leftarrow w_i + \lambda 2\sum_{j=1}^{n}x_{j,i}\delta_j$$

Gradient descent algorithm for linear regression

1. Chose λ
2. Start with a guess for **w**
3. Compute δ_j for all j
4. For all i set $$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^{n} x_{j,i} \delta_j$$
5. If no improvement for $$\sum_{j=1}^{n} (y_j - \mathbf{w}^T \mathbf{x}_j)^2$$

   stop. Otherwise go to step 3

Gradient descent vs. matrix inversion

- Advantages of matrix inversion

  - No iterations

  - No need to specify parameters

  - Closed form solution in a predictable time

- Advantages of gradient descent

  - Applicable regardless of the number of parameters

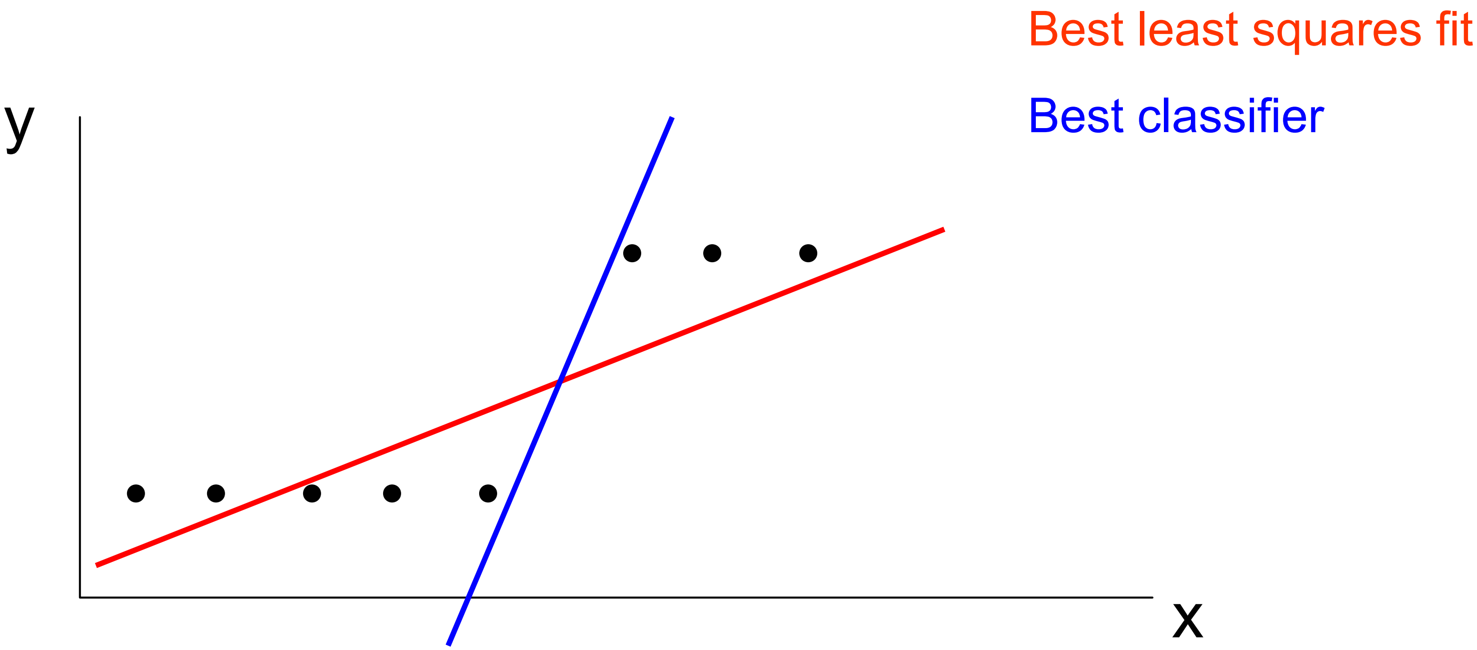
  - General, applies to other forms of regression

Perceptrons for classification

- So far we discussed regression
- However, perceptrons can also be used for classification
- For example, output 1 is $\mathbf{w}^T\mathbf{x} > 0$ and -1 otherwise
- Problem?

Perceptrons for classification

- So far we discussed regression
- However, perceptrons can also be used for classification
- For example, output 1 is $\mathbf{w}^T\mathbf{x} > 1/2$ and 0 otherwise
- Problem?



The sigmoid function

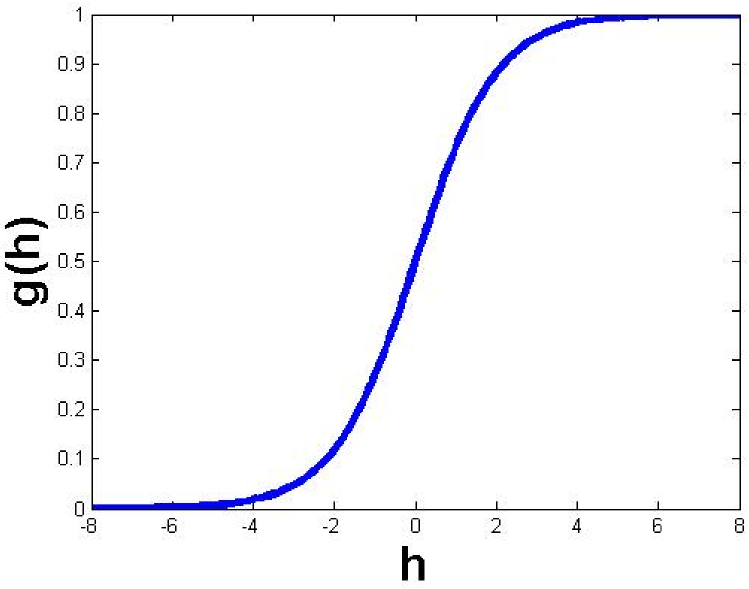
- To classify using a perceptron we replace the linear function with the sigmoid function:

$$g(h) = \frac{1}{1+e^{-h}}$$

- Using the sigmoid we would minimize

$$\sum_{j=1}^{n} (y_j - g(\mathbf{w}^T \mathbf{x}_j))^2$$

- Where y_j is either 0 or 1 depending on the class



Gradient descent with sigmoid

- Once we defined our target function, we can minimize it using gradient descent
- This involves some math, and relies on the following derivation*:

$$g'(h) = g(h)(1 - g(h))$$

- So,

$$\frac{\partial}{\partial w_i}\sum_{j=1}^{n}(y_j - g(\mathbf{w}^T\mathbf{x}_j))^2 = 2\sum_{j=1}^{n}(y_j - g(\mathbf{w}^T\mathbf{x}_j))\frac{\partial}{\partial w_i}(y_j - g(\mathbf{w}^T\mathbf{x}_j))$$

$$= -2\sum_{j=1}^{n}(y_j - g(\mathbf{w}^T\mathbf{x}_j))g'(\mathbf{w}^T\mathbf{x}_j)\frac{\partial}{\partial w_i}\mathbf{w}^T\mathbf{x}_j$$

$$= -2\sum_{j=1}^{n}(y_j - g(\mathbf{w}^T\mathbf{x}_j))g(\mathbf{w}^T\mathbf{x}_j)(1 - g(\mathbf{w}^T\mathbf{x}_j))x_{j,i}$$

*I have included a derivation of this at the end of the lecture notes

Gradient descent with sigmoid

$$\frac{\partial}{\partial w_i}\sum_{j=1}^{n}(y_j - g(\mathbf{w}^T\mathbf{x}_j))^2 = -2\sum_{j=1}^{n}(y_j - g(\mathbf{w}^T\mathbf{x}_j))g(\mathbf{w}^T\mathbf{x}_j)(1-g(\mathbf{w}^T\mathbf{x}_j))x_{j,i}$$

Set $\delta_j = y_j - g(\mathbf{w}^T\mathbf{x}_j)$ $g_j = g(\mathbf{w}^T\mathbf{x}_j)$

$$\frac{\partial}{\partial w_i}\sum_{j=1}^{n}(y_j - g(\mathbf{w}^T\mathbf{x}_j))^2 = -2\sum_{j=1}^{n}\delta_j g_j(1-g_j)x_{j,i}$$

So our update rule is:

$$w_i \leftarrow w_i + \lambda 2\sum_{j=1}^{n}\delta_j g_j(1-g_j)x_{j,i}$$

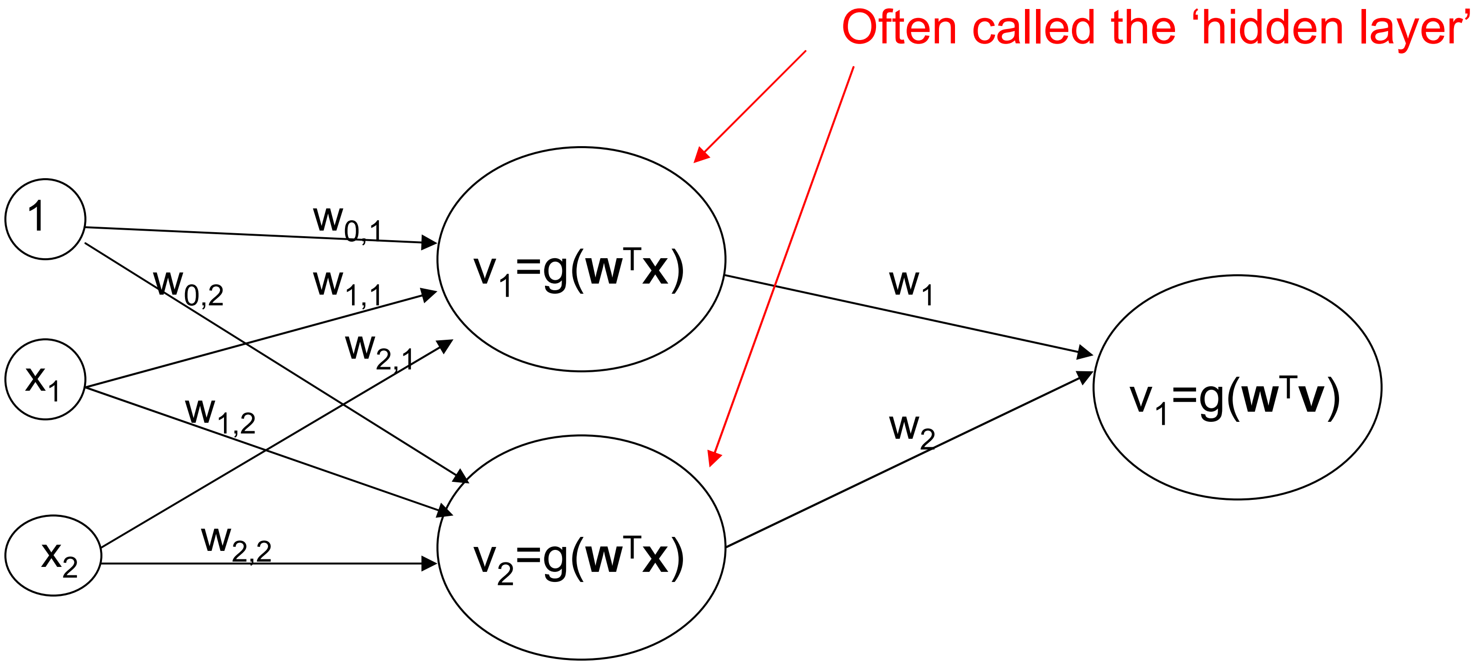
Revised algorithm for sigmoid regression

1. Chose λ
2. Start with a guess for **w**
3. Compute δ_j for all j
4. For all i set $w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^{n} \delta_j g_j (1 - g_j) x_{j,i}$
5. If no improvement for $\sum_{j=1}^{n} (y_j - g(\mathbf{w}^T \mathbf{x}_j))^2$

   stop. Otherwise go to step 3

Multilayer neural networks

- So far we discussed networks with one layer.
- But these networks can be extended to combine several layers, increasing the set of functions that can be represented using a NN

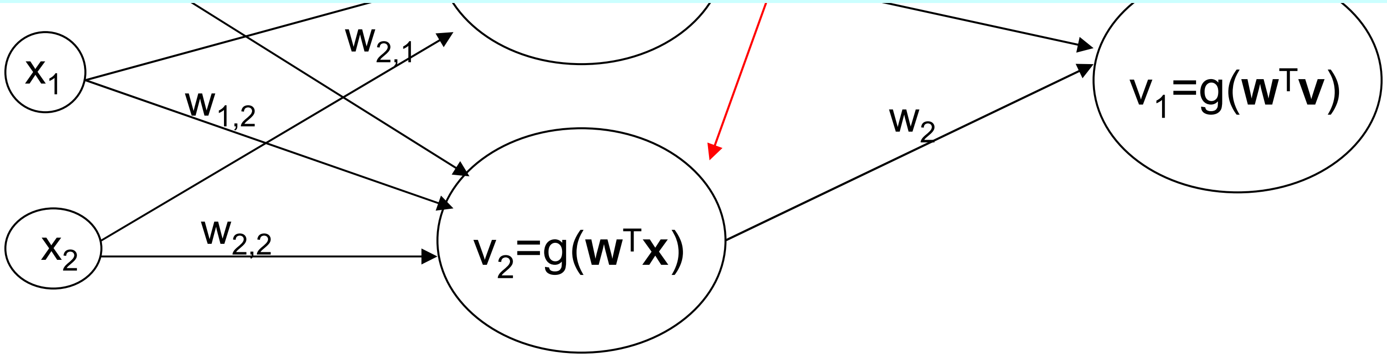

Often called the 'hidden layer'
1
x_1
x_2
$w_{0,1}$
$w_{0,2}$
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$v_1=g(\mathbf{w}^T\mathbf{x})$
$v_2=g(\mathbf{w}^T\mathbf{x})$
w_1
w_2
$v_1=g(\mathbf{w}^T\mathbf{v})$

Multilayer neural networks

- So far we discussed networks with one layer.
- But these networks can be extended to combine several layers, increasing the set of functions that can be r

The book contains an interesting discussion about the types of Boolean functions that can be computed using various multilayer neural networks. We won’t cover it in class so you should look it up yourself.

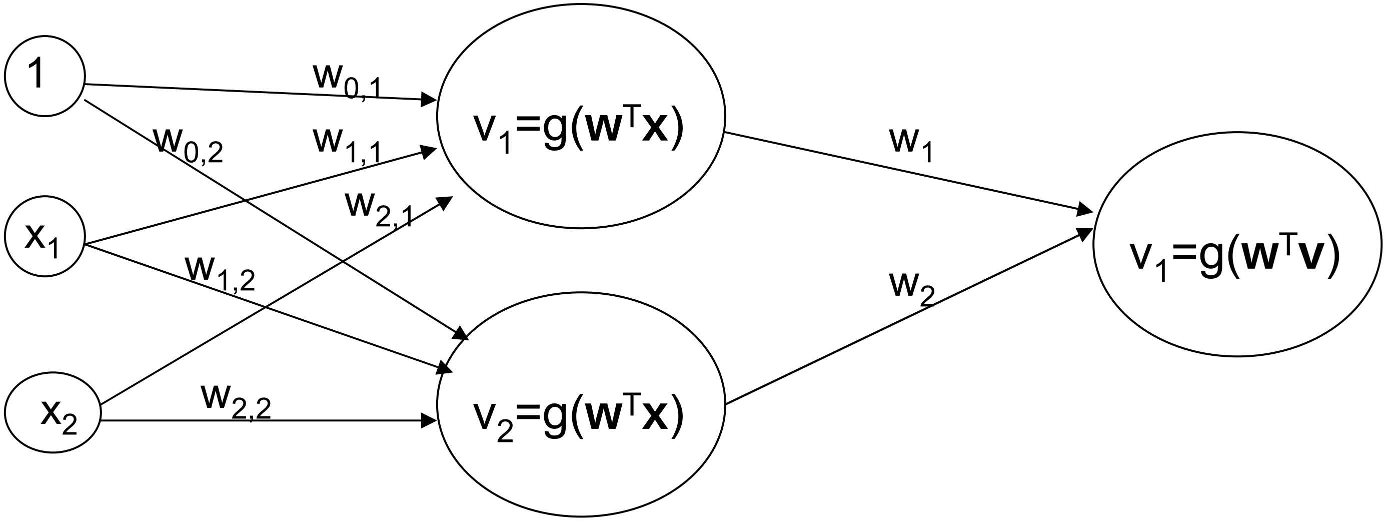
r’



$w_{2,1}$
x_1
$w_{1,2}$
w_2
$v_1=g(\mathbf{w}^T\mathbf{v})$
x_2
$w_{2,2}$
$v_2=g(\mathbf{w}^T\mathbf{x})$

Learning the parameters for multilayer networks

- Gradient descent works by connecting the output to the inputs.
- But how do we use it for a multilayer network?
- We need to account for both, the output weights and the hidden layer weights

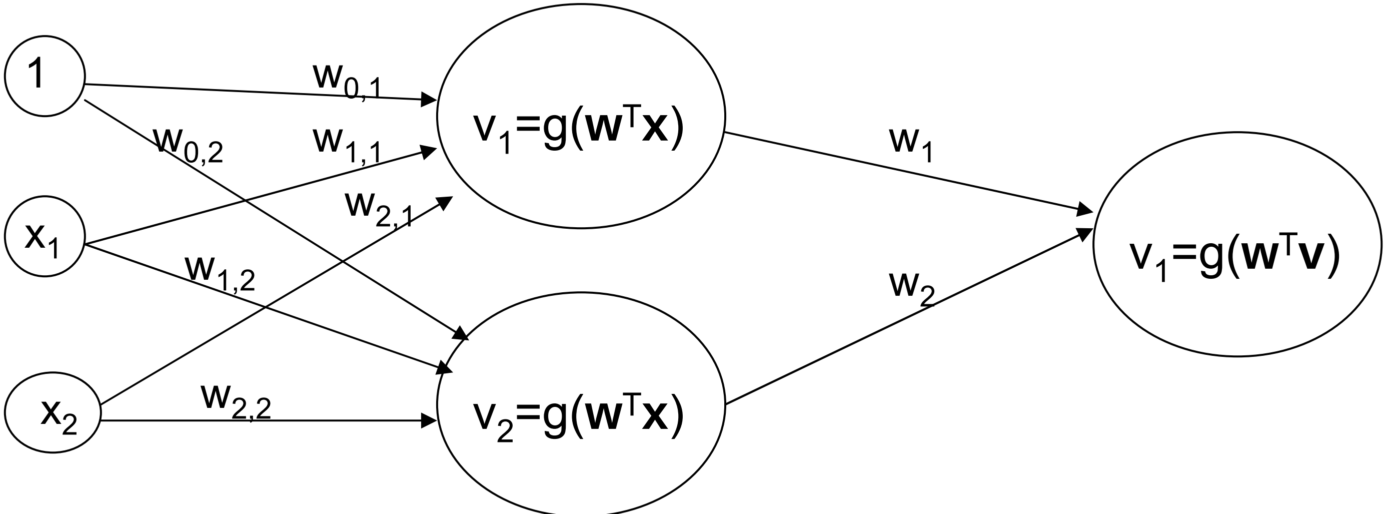

1
x_1
x_2
$w_{0,1}$
$w_{0,2}$
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$v_1=g(\mathbf{w}^T\mathbf{x})$
$v_2=g(\mathbf{w}^T\mathbf{x})$
w_1
w_2
$v_1=g(\mathbf{w}^T\mathbf{v})$

Learning the parameters for multilayer networks

- Its easy to compute the update rule for the output weights w_1 and w_2:

$$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^{n} \delta_j g_j (1 - g_j) v_{j,i}$$

where $\delta_j = y_j - g(\mathbf{w}^T \mathbf{v}_j)$


1
x_1
x_2
$w_{0,1}$
$w_{0,2}$
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$v_1=g(\mathbf{w}^T\mathbf{x})$
$v_2=g(\mathbf{w}^T\mathbf{x})$
w_1
w_2
$v_1=g(\mathbf{w}^T\mathbf{v})$

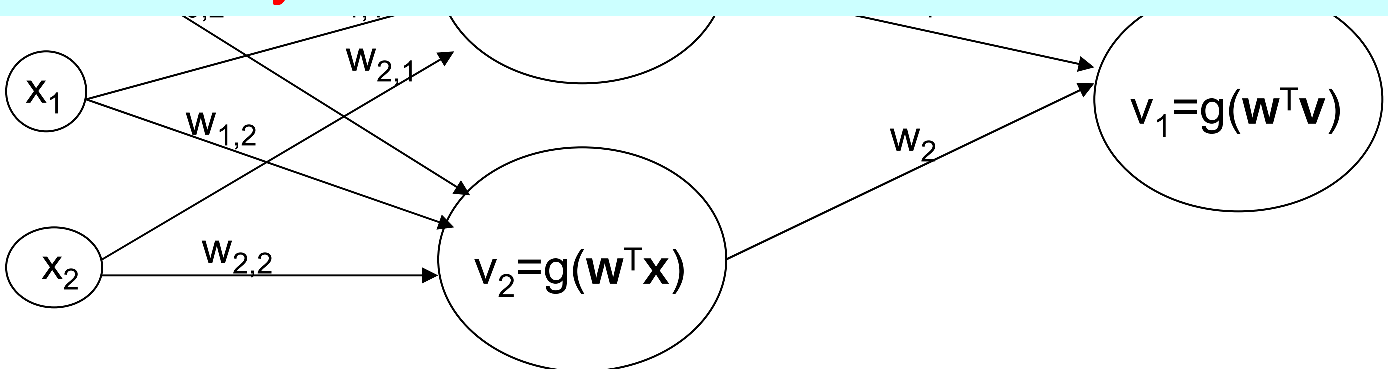
Learning the parameters for multilayer networks

- Its easy to compute the update rule for the output weights w_1 and w_2:

$$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^{n} \delta_j g_j (1 - g_j) v_{j,i}$$

where $\delta_j = y_j - g(\mathbf{w}^T \mathbf{v}_j)$

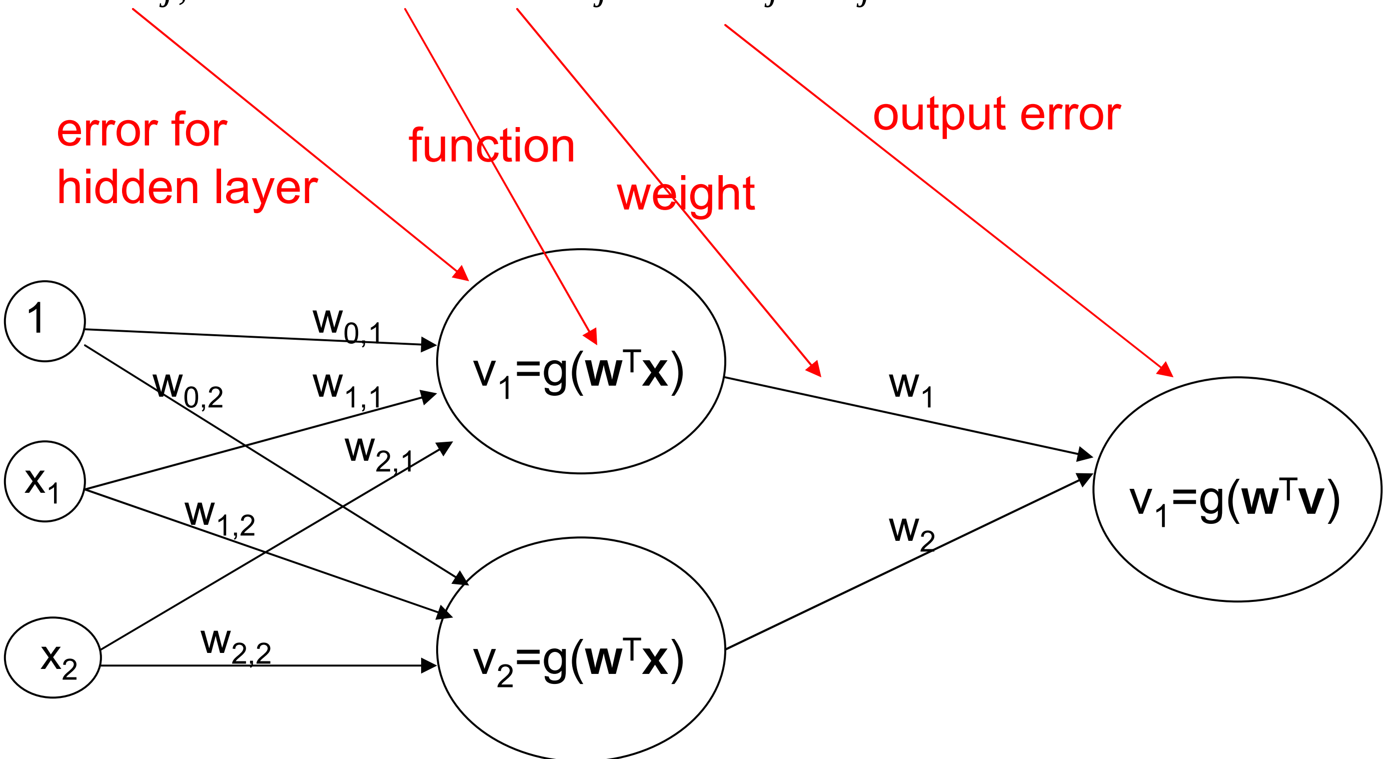
But what is the error associated with each of the hidden layer states?


x_1
x_2
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
w_2
$v_1=g(\mathbf{w}^T\mathbf{v})$
$v_2=g(\mathbf{w}^T\mathbf{x})$

Backpropagation

- A method for distributing the error among hidden layer states
- Using the error for each of these states we can employ gradient descent to update them
- Set

$$\Delta_{j,i} = g'(\mathbf{w}^T\mathbf{x})w_i\delta_j(1-g_j)g_j$$



error for hidden layer

function

weight

output error

1
x_1
x_2
$w_{0,1}$
$w_{0,2}$
$w_{1,1}$
$w_{2,1}$
$w_{1,2}$
$w_{2,2}$
$v_1=g(\mathbf{w}^T\mathbf{x})$
$v_2=g(\mathbf{w}^T\mathbf{x})$
w_1
w_2
$v_1=g(\mathbf{w}^T\mathbf{v})$

Backpropagation

- A method for distributing the error among hidden layer states
- Using the error for each of these states we can employ gradient descent to update them
- Set

$$\Delta_{j,i} = g'(\mathbf{w}^T \mathbf{x}) w_i \delta_j (1 - g_j) g_j$$

- Our update rule changes to:

$$w_{k,i} \leftarrow w_{k,i} + \lambda 2 \sum_{j=1}^{n} \Delta_{j,i} g_{j,i} (1 - g_{j,i}) x_{j,k}$$

Backpropagation

- A method for distributing the error among hidden layer states
- Using the error for each of these states we can employ gradient descent to update them
- Set

$$\Delta_{j,i} = g'(\mathbf{w}^T\mathbf{x}) w_i \delta_j (1-g_j) g_j$$

The correct error term for each hidden state can be determined by taking the partial derivative for each of the weight parameters of the hidden layer w.r.t. the global error function*:

$$Err_j = (y_j - g(\mathbf{w}^T g(\mathbf{w}_i^T \mathbf{x}))^2$$

*See RN book for details (pages 746-747)

Revised algorithm for multilayered neural network

1. Chose λ
2. Start with a guess for **w, w**$_i$
3. Compute values $v_{i,j}$ for all hidden layer states i and inputs j
4. Compute δ_j for all j: $\delta_j = y_j - g(\mathbf{w}^T \mathbf{v}_j)$
5. Compute $\Delta_{j,l}$
6. For all i set $$w_i \leftarrow w_i + \lambda 2 \sum_{j=1}^{n} \delta_j g_j (1 - g_j) v_{j,i}$$
7. For all k and i set $$w_{k,i} \leftarrow w_{k,i} + \lambda 2 \sum_{j=1}^{n} \Delta_{j,i} g_{j,i} (1 - g_{j,i}) x_{j,k}$$
8. If no improvement for $\sum_{j=1}^{n} \delta_j^2 + \sum_{i=1}^{s} \Delta_{j,i}^2$ stop. Otherwise go to step 3

What you should know

- Linear regression

  - Solving a linear regression problem
- Gradient descent
- Perceptrons

  - Sigmoid functions for classification
- Multilayered neural networks

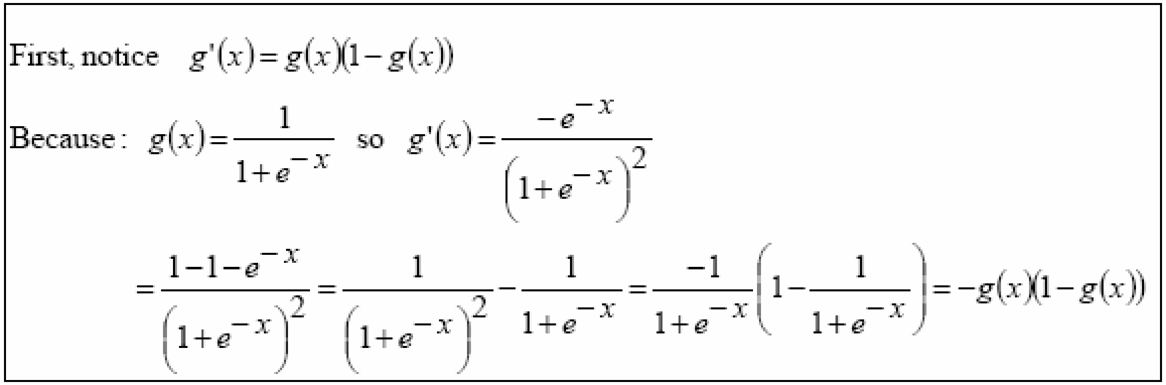
  - Backpropagation

Deriving g’(x)

- Recall that g(x) is the sigmoid function so

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

- The derivation of g’(x) is below



First, notice $g'(x) = g(x)(1-g(x))$

Because: $g(x) = \frac{1}{1+e^{-x}}$ so $g'(x) = \frac{-e^{-x}}{\left(1+e^{-x}\right)^2}$

$$= \frac{1-1-e^{-x}}{\left(1+e^{-x}\right)^2} = \frac{1}{\left(1+e^{-x}\right)^2} - \frac{1}{1+e^{-x}} = \frac{-1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) = -g(x)(1-g(x))$$