10-601
Machine Learning

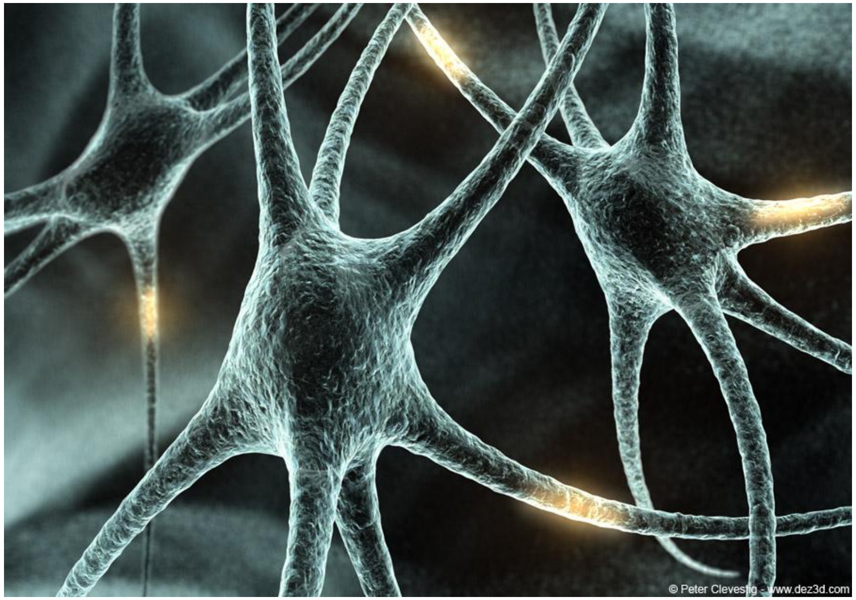
Neural Networks (NN)

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



What can we do with NN?

- Classification

  - We already mentioned many useful applications

- Regression

  Input: Real valued variables

  Output: One or more real values

- Examples:

  - Predict the price of Google's stock from Microsoft's stock price

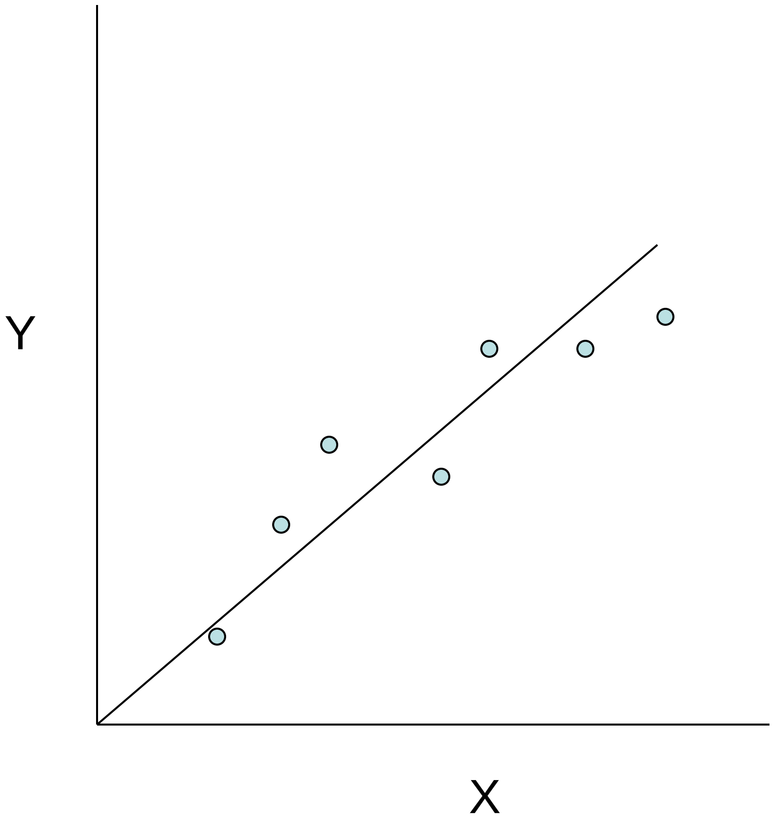
  - Predict distance to obstacle from various sensors

Linear regression

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

$$y = wx+\varepsilon$$

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



Multivariate regression: Least squares

- We already presented a solution for determining the parameters of a linear regression problem.

Define:

$$\Phi = \begin{pmatrix} \phi_0(x^1) & \phi_1(x^1) & \cdots & \phi_m(x^1) \\ \phi_0(x^2) & \phi_1(x^2) & \cdots & \phi_m(x^2) \\ \vdots & \vdots & \cdots & \vdots \\ \phi_0(x^n) & \phi_1(x^n) & \cdots & \phi_m(x^n) \end{pmatrix}$$

Then deriving w we get: $\mathbf{w} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{y}$

Multivariate regression: Least squares

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

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

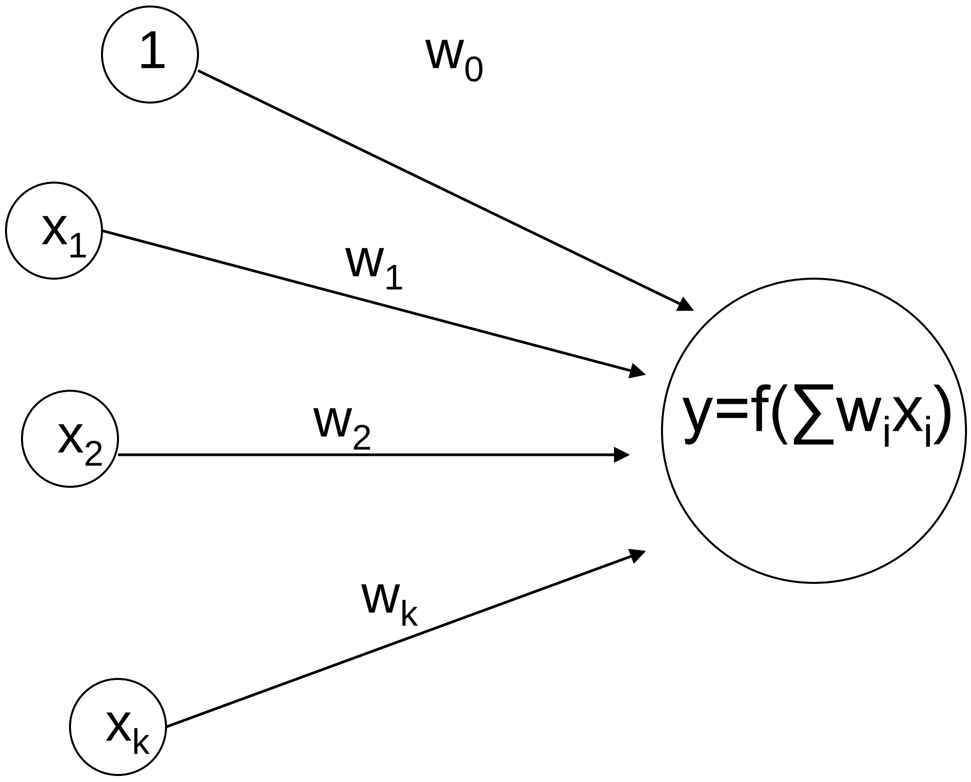
- Solution may be slow

- Does not address general regression problems of the form

$$\mathbf{y} = f(\mathbf{X}\mathbf{w})$$

Back to NN: Preceptron

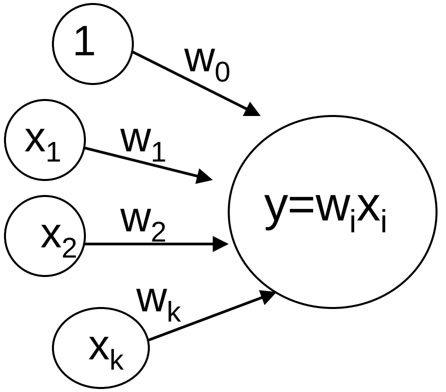
- The basic processing unit of a neural net



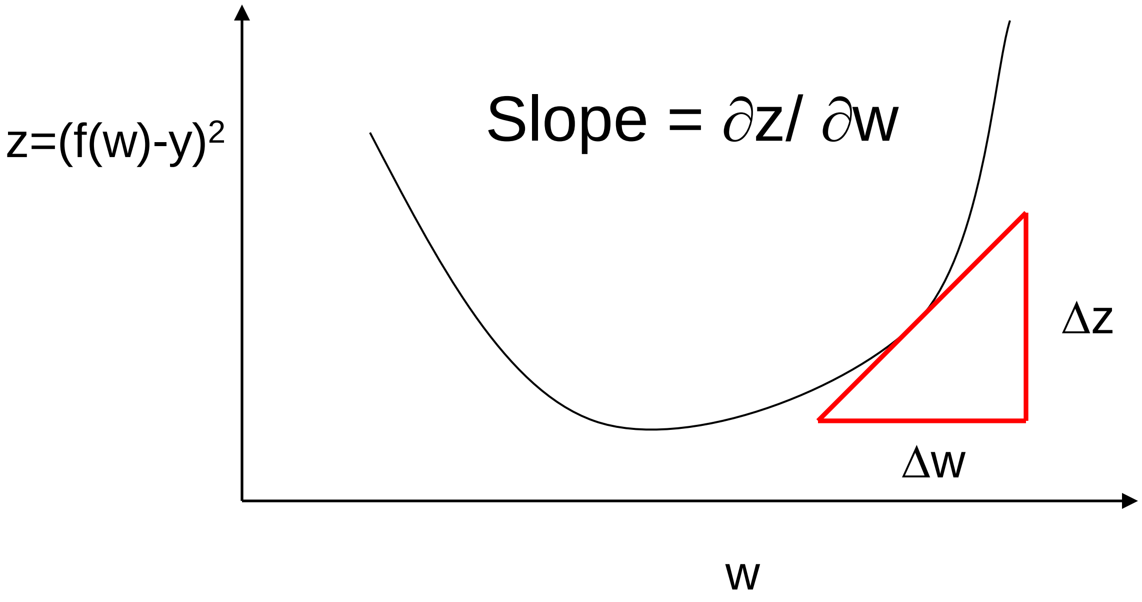
Linear regression

- Lets start by setting $f(\sum w_i x_i)=\sum w_i x_i$
- 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

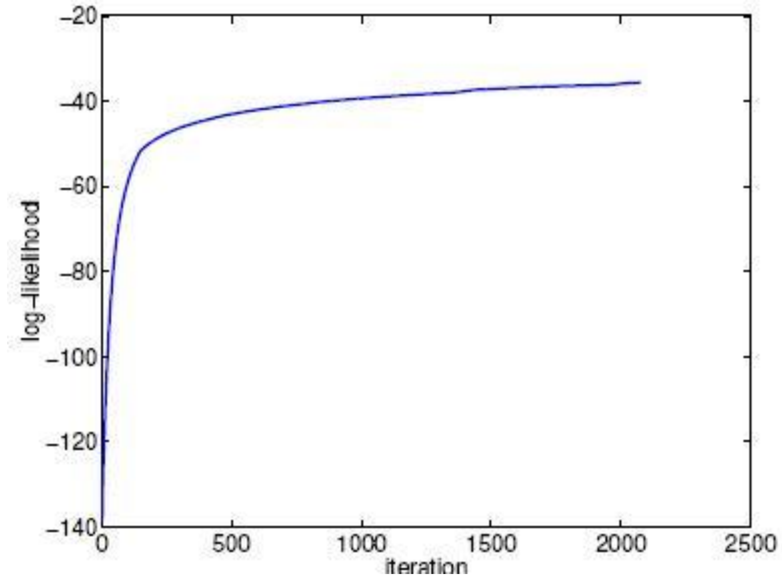
- We thus update the weights by setting:

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

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

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

- 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

Example

- $W = 2$

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?

As with logistic vs. linear regression we use the sigmoid function as part of the perception when using it for classification

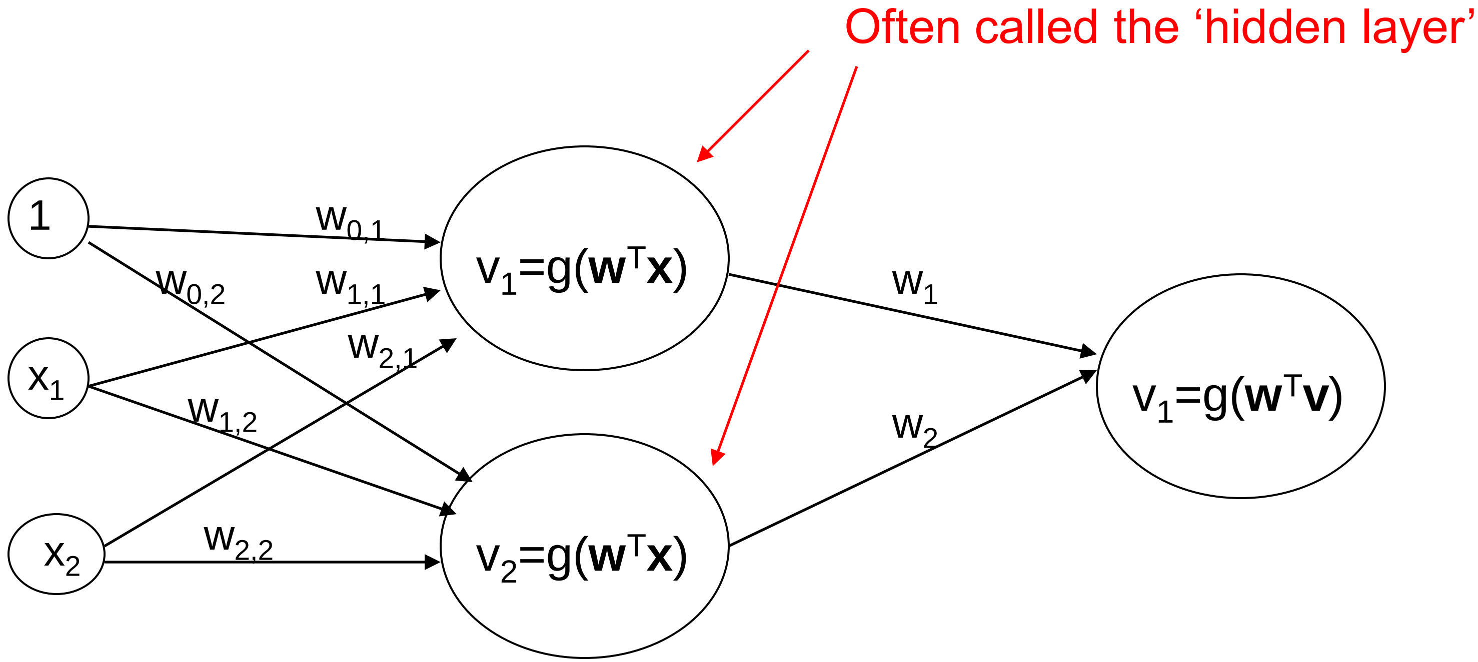
Revised algorithm for sigmoid regression

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

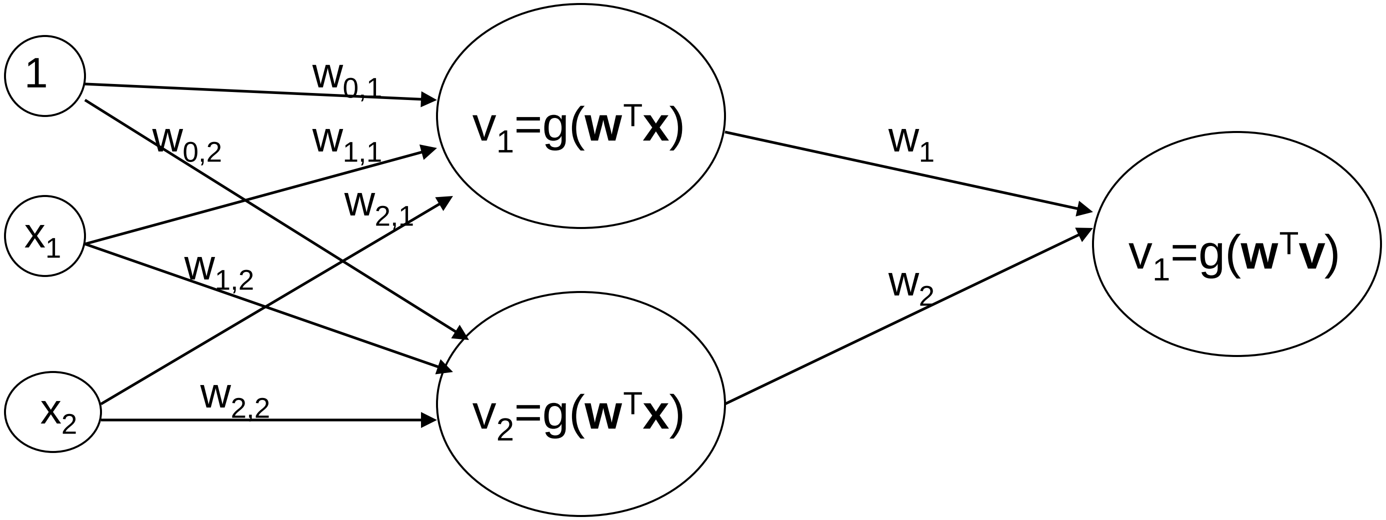
Multilayer neural networks

- 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



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

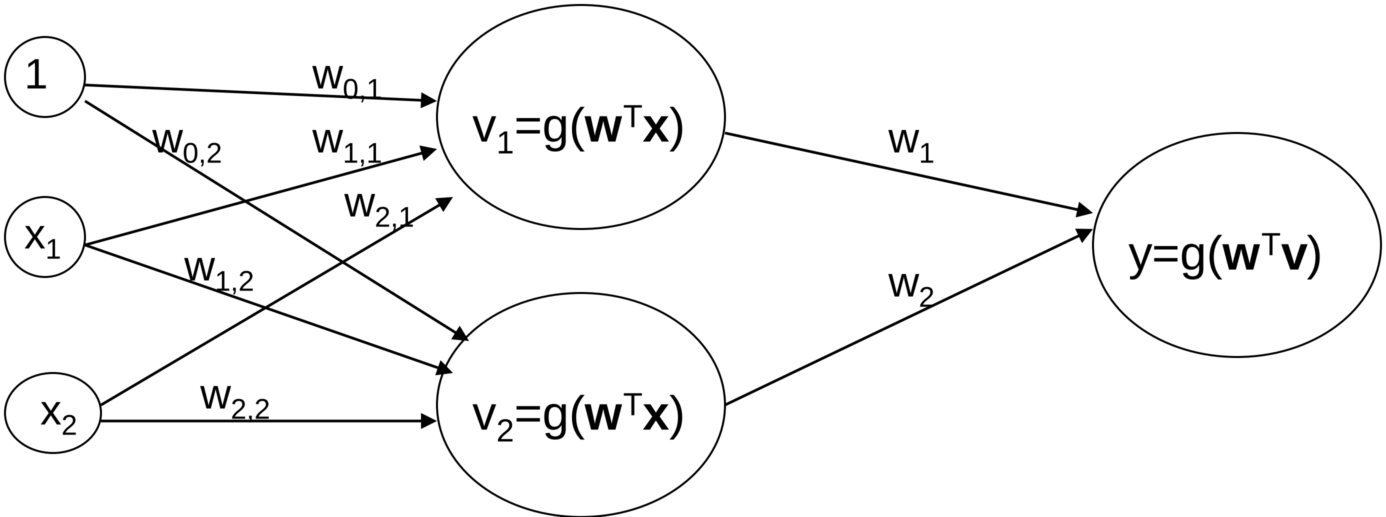


Learning the parameters for multilayer networks

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



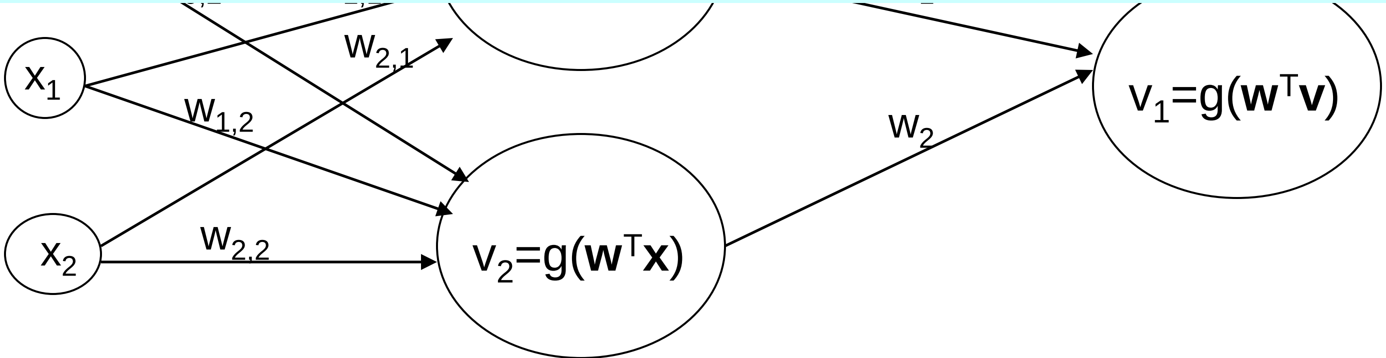
Learning the parameters for multilayer networks

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

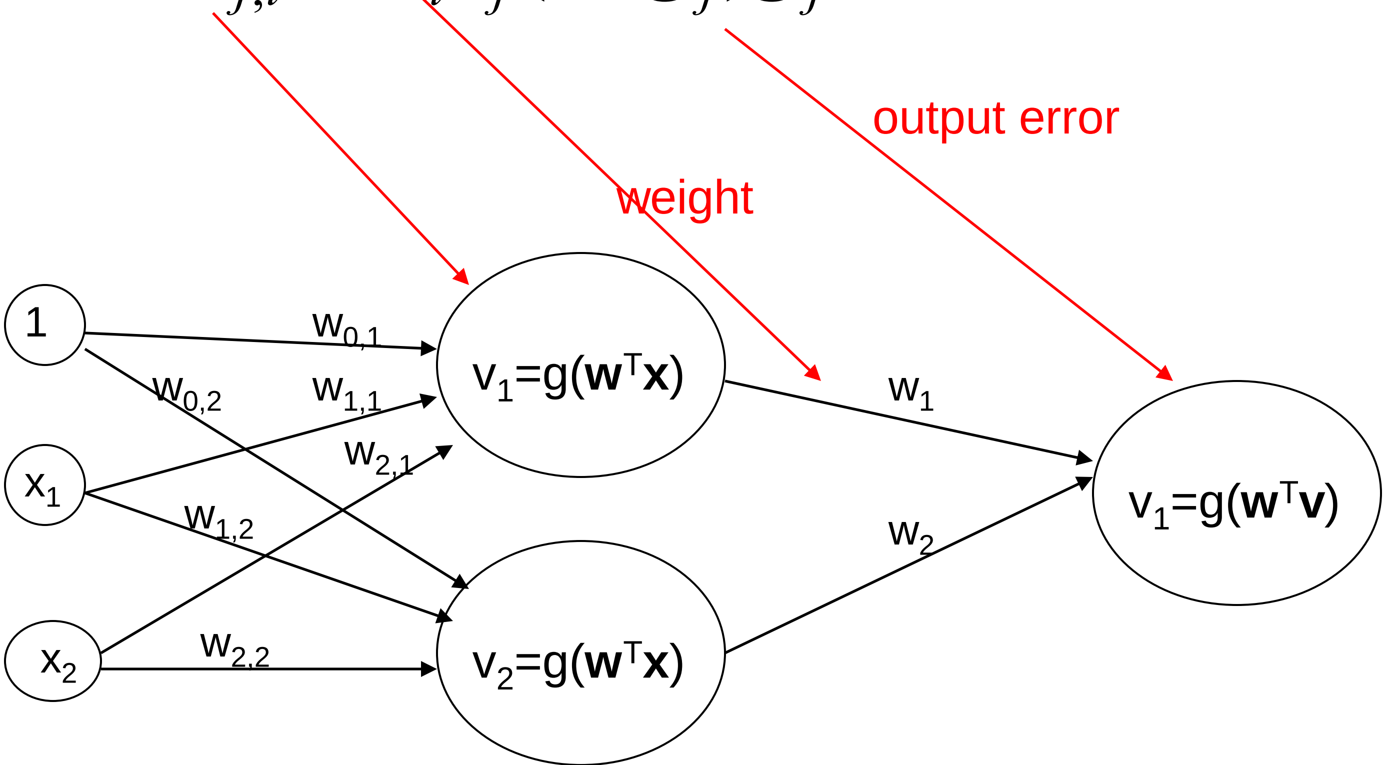
But what is the error associated with each of the hidden layer states?



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

$$\Delta_{j,i} = w_i \delta_j (1 - g_j) g_j$$



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

$$\Delta_{j,i} = w_i \delta_j (1 - g_j) g_j$$

- 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

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

Examples

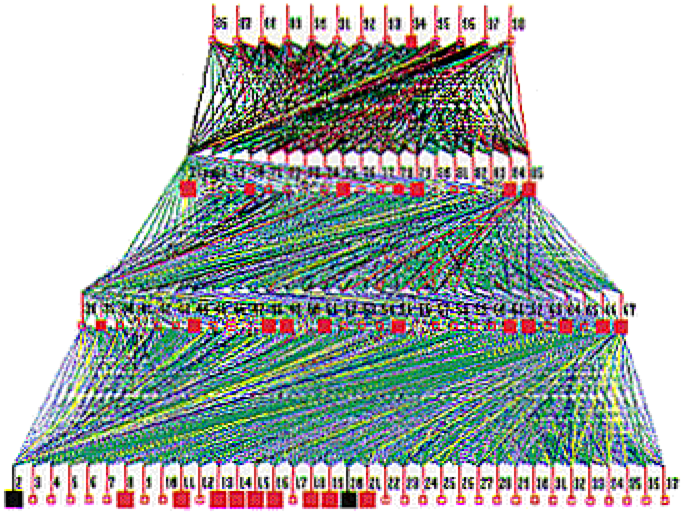


Figure 1: Feedforward ANN designed and tested for prediction of tactical air combat maneuvers.

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

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