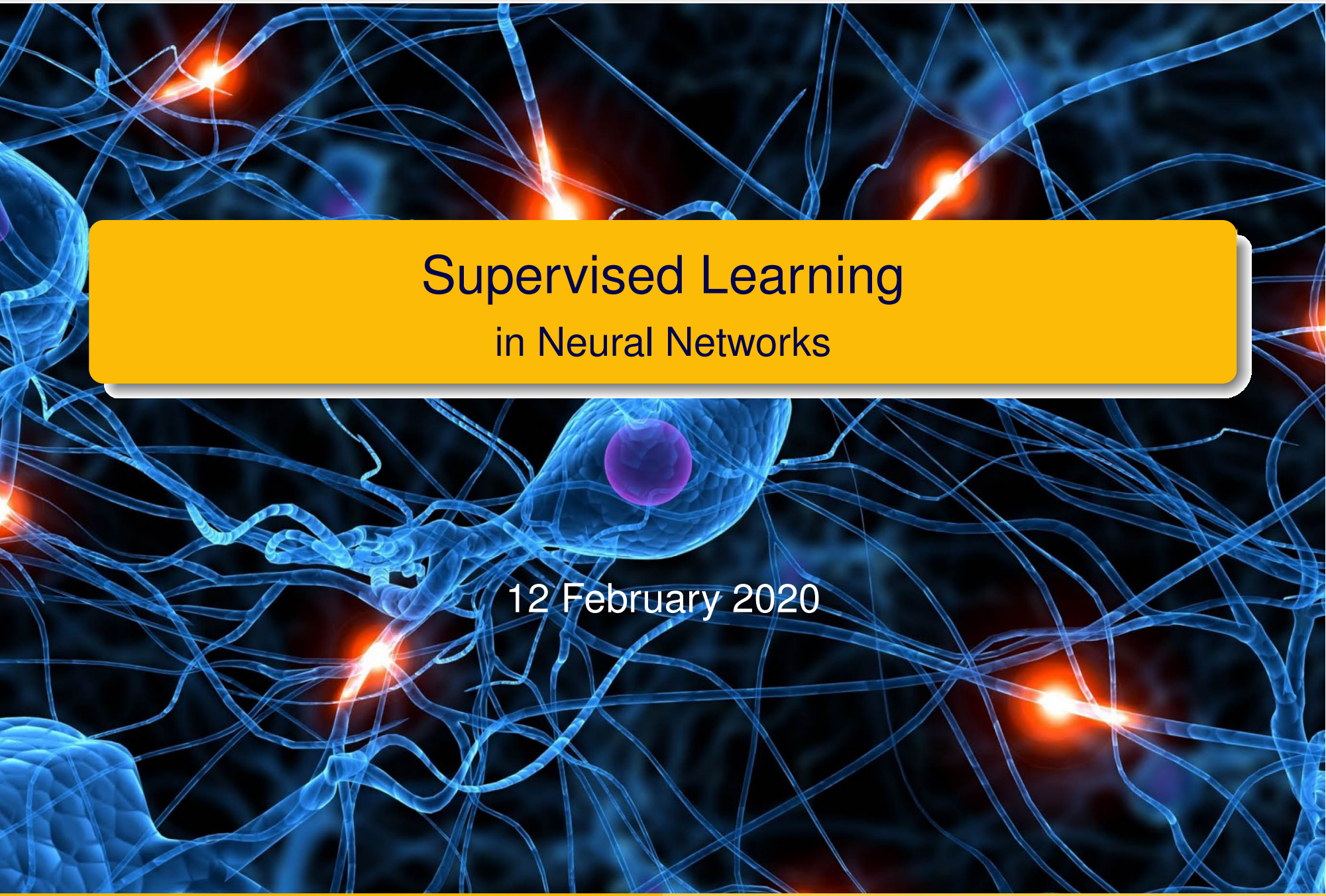


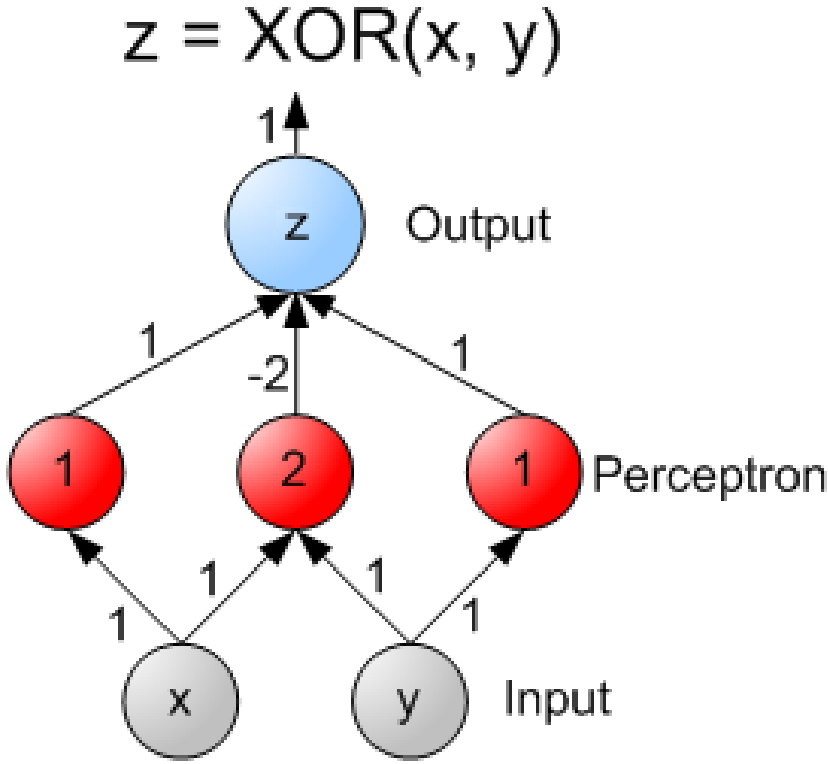
Supervised Learning

in Neural Networks

12 February 2020

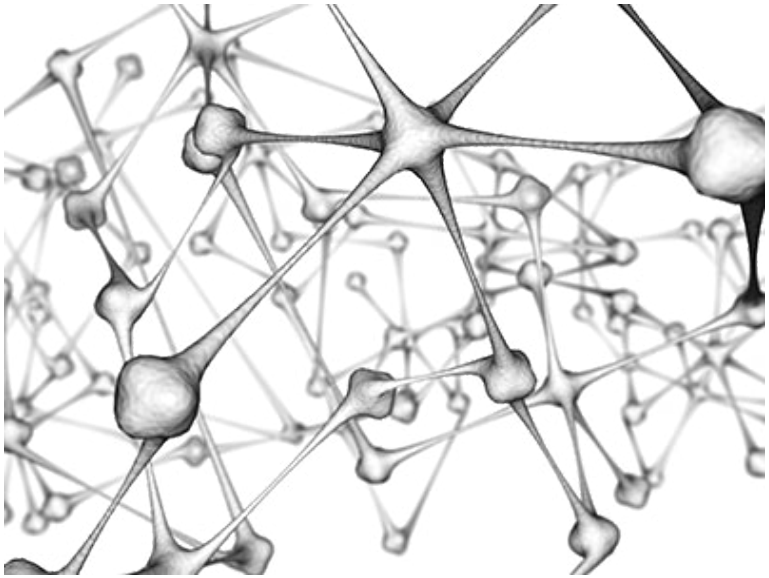
Neural Networks
Universal approximators

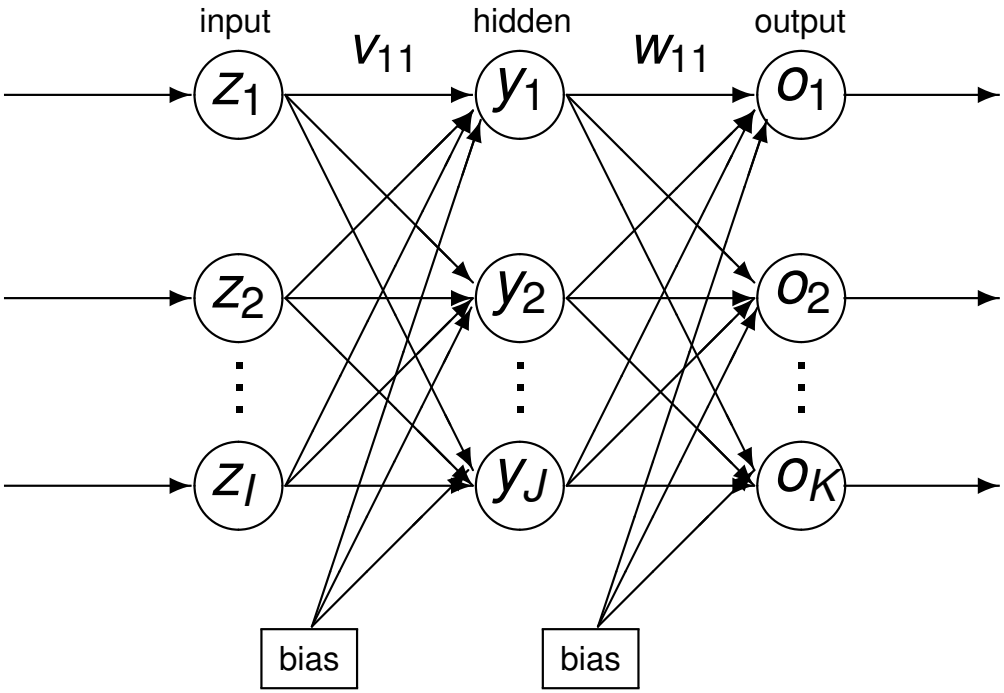
- Linking perceptrons/neurons in successive layers introduces non-linearity to the model
- A neural network with enough hidden neurons can approximate **any** non-linear function
- NNs are “universal approximators”
- NB: This statement only holds true if activation functions are **non-linear**



Feed Forward Neural Network

a.k.a. FFNN





How do you train this “universal approximator”?

Supervised Learning

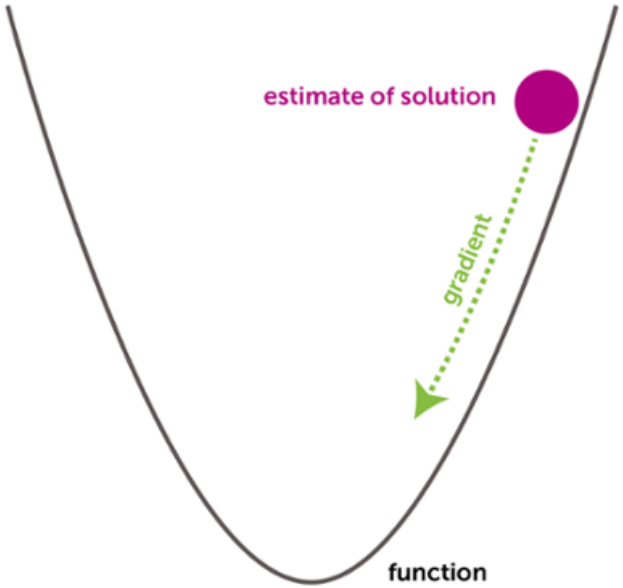
- A data set of inputs and the corresponding targets is given, referred to as the training set
- The training set is iteratively presented to the NN
- Supervised learning algorithm adjusts the weights at each iteration to minimize the difference between target outputs, t, and actual outputs, o

$x_{1,1}$	$x_{1,2}$	$\ldots$				$t_{1,1}$	$t_{1,2}$	$\ldots$	$t_{1,m}$
$x_{2,1}$	$x_{2,2}$	$\ldots$				$t_{2,1}$	$t_{2,2}$	$\ldots$	$t_{2,m}$
$x_{3,1}$	$x_{3,2}$	$\ldots$				$t_{3,1}$	$t_{3,2}$	$\ldots$	$t_{3,m}$
$x_{4,1}$	$x_{4,2}$	$\ldots$				$t_{4,1}$	$t_{4,2}$	$\ldots$	$t_{4,m}$
$x_{5,1}$	$x_{5,2}$	$\ldots$				$t_{5,1}$	$t_{5,2}$	$\ldots$	$t_{5,m}$
$\ldots$	$\ldots$					$\ldots$	$\ldots$		$\ldots$
$x_{n,1}$	$x_{n,2}$					$t_{n,1}$	$t_{n,2}$	$\ldots$	$t_{n,m}$

Gradient descent

- We want to minimize the NN's error - in other words, find the minimum of the error function with respect to the NN's weights
- NN weights are the only parameters, or variables: inputs and targets are pre-defined
- Function minimization algorithm can be applied

- Gradient descent:
  - Calculate the gradient of the error function in the weight space
  - Move the weight vector along the **negative gradient**
  - "Steepest slope descent"



Understanding Gradients

- The gradient is a multi-variable generalization of the derivative:

$$f(x, y) = xy$$

$$\frac{\partial f}{\partial x} = y, \frac{\partial f}{\partial y} = x$$

$$\therefore \nabla f(x, y) = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] = [y, x]$$

- ∇f, or gradient of f, is a vector of partial derivatives of f with respect to the input variables of f
- Like the derivative, the gradient represents the slope of f. I.e., the gradient points in the direction of the greatest rate of increase of the function.
- Where would the negative of the gradient point?

Gradient Descent

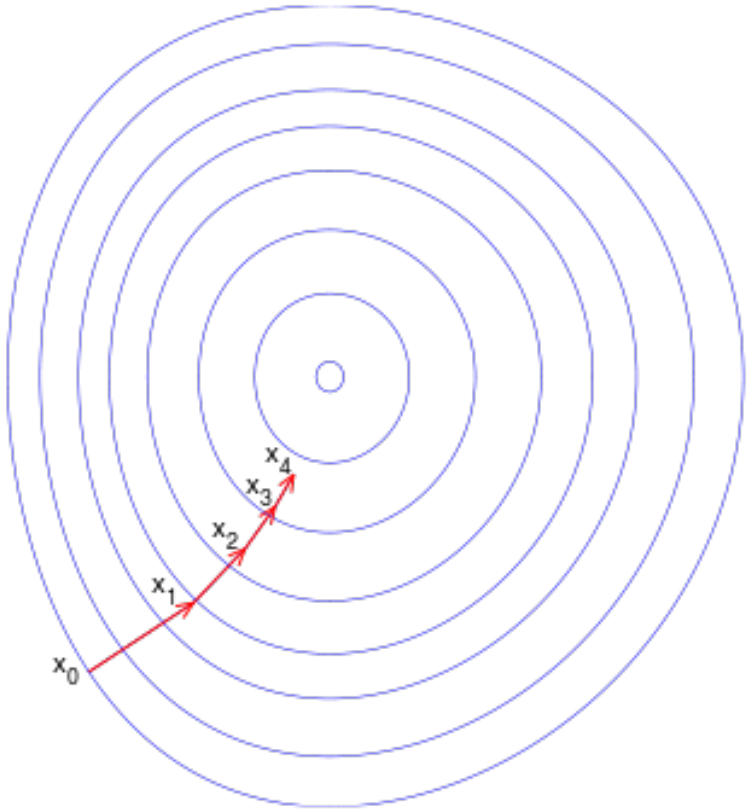
Following the negative gradient

Function minimization

- Suppose you are looking for a minimum of a differentiable function $f(\vec{x})$
- Pick a point in the search space, $\vec{x}_0$
- Calculate $\nabla f(\vec{x}_0)$
- Move from $\vec{x}_0$ to $\vec{x}_1$:

$$\vec{x}_{i+1} = \vec{x}_i - \eta \nabla f(\vec{x}_i)$$

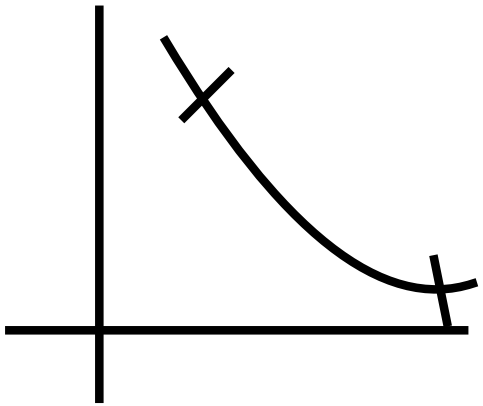
- Repeat till convergence
- η is a configurable parameter, known as the learning rate



Chain Rule

- To apply gradient descent to NNs, we will have to differentiate the error function, $E = \frac{1}{2}(t - y)^2$, where $y = f(net)$
- We have function y nested in function E, and function net nested inside of y
- Differentiation of "nested", or composite functions is performed using the chain rule: If we have $y = f(u)$, and $u = g(x)$, then

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



- Example:
  - $f(x) = (3x + 1)^2$
  - $f(x) = (g(x))^2, g(x) = 3x + 1$
  - $\frac{df}{dg} = 2g(x), \frac{dg}{dx} = 3$
  - $\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx} = 2g(x) * 3 = 2(3x + 1) * 3 = 6(3x + 1)$

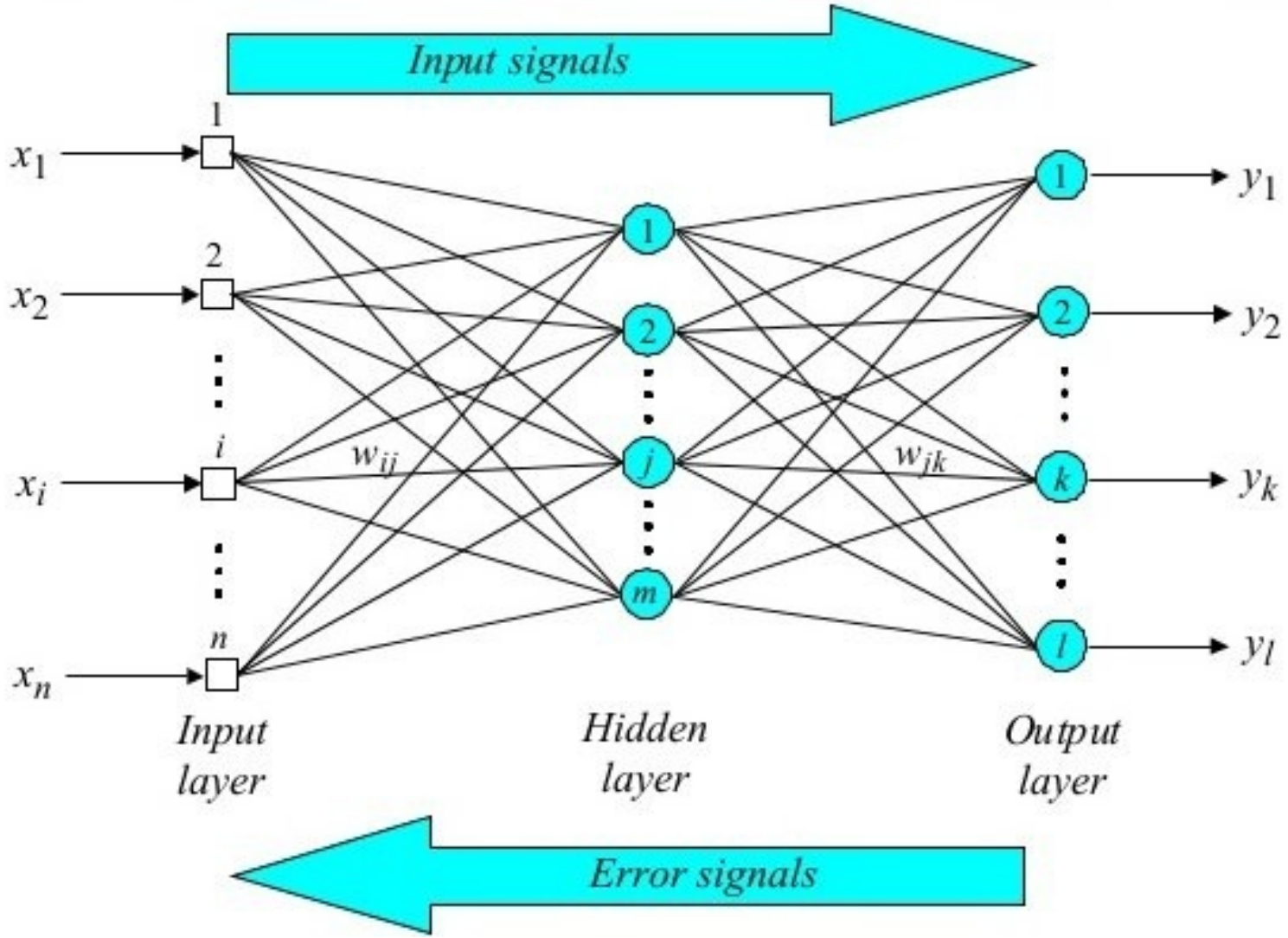
Applying Gradient Descent to Neural Networks

- Minimize the squared error function with respect to weights: $E = \frac{1}{2}(t_j - y_j)^2$, where t is the target and y is the actual output
- For each neuron j, its output is defined as: $y_j = f(net_j) = (1 + e^{-net})^{-1}$, where $f(net_j)$ is an activation function.
- The net input signal is defined as: $net_j = \sum w_{ij} y_i$
- What we need: partial derivative of E with respect to w_{ij}
- (If we know the derivative of E w.r.t. w_{ij}, we can subtract the derivative from w_{ij} and thus update w_{ij})
- Calculate $\frac{\partial E}{\partial w_{ij}}$ using chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

BackPropagation

Forward pass + Backward Pass



BackPropagation

- Start on the outmost set of weights
- Calculate $\frac{\partial E}{\partial w_{ij}}$ using chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

What is the third term, $\frac{\partial net_j}{\partial w_{ij}}$?

$$net_j = \sum w_{ij} y_i = w_{1j} y_1 + \ldots + w_{ij} y_i + \ldots + w_{nj} y_n$$

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} w_{ij} y_i = y_i$$

BackPropagation

- Start on the outmost set of weights
- Calculate $\frac{\partial E}{\partial w_{ij}}$ using chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

What is the second term, $\frac{\partial y_j}{\partial net_j}$?

$$y_j = f(net_j)$$

$$f(net_j) = (1 + e^{-net_j})^{-1}$$

We have to calculate the derivative of the activation function, $f(x)$

Sigmoid derivative

$$f(x) = (1 + e^{-x})^{-1}$$

$$f'(x) = (-1)(1 + e^{-x})^{-2} \frac{d}{dx}(1 + e^{-x})$$

$$= -(1 + e^{-x})^{-2}(-e^{-x})$$

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

A nifty trick to simplify the equation:

$$f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{e^{-x} + 1 - 1}{(1 + e^{-x})^2} = \frac{1 + e^{-x}}{(1 + e^{-x})^2} - \frac{1}{(1 + e^{-x})^2}$$

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

BackPropagation

- Start on the outmost set of weights
- Calculate $\frac{\partial E}{\partial w_{ij}}$ using chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

What is the second term, $\frac{\partial y_j}{\partial net_j}$?

$$y_j = f(net_j)$$

$$f(net_j) = (1 + e^{-net_j})^{-1}$$

$$\frac{\partial y_j}{\partial net_j} = f(net_j)(1 - f(net_j))$$

BackPropagation

- Calculate $\frac{\partial E}{\partial w_{ij}}$ using chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} y_j(1-y_j)y_i$$

What is the first term, $\frac{\partial E}{\partial y_j}$?

Do the outmost set of weights first:

$$E = \frac{1}{2}(t_j - y_j)^2$$

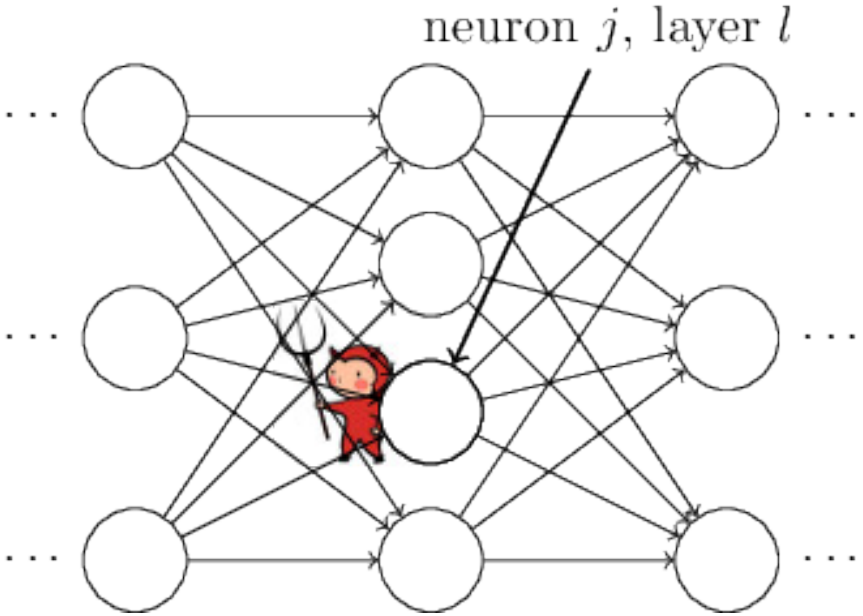
$$\frac{\partial E}{\partial y_j} = \frac{\partial}{\partial y_j}\frac{1}{2}(t_j - y_j)^2 = -1(t_j - y_j) = y_j - t_j$$

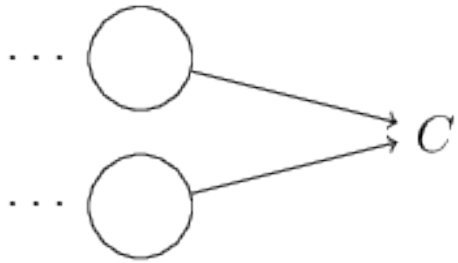
BackPropagation

- **Put it together:** The partial derivative of w_{ij} w.r.t. E for the **outer** set of weights is:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} = (y_j - t_j) y_j (1 - y_j) y_i$$

- What about an arbitrary w_{ij}?





BackPropagation: inner w_{ij}

- What about an arbitrary w_{ij}?

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

- Last term?

$$\frac{\partial net_j}{\partial w_{ij}} = y_i$$

- Second term?

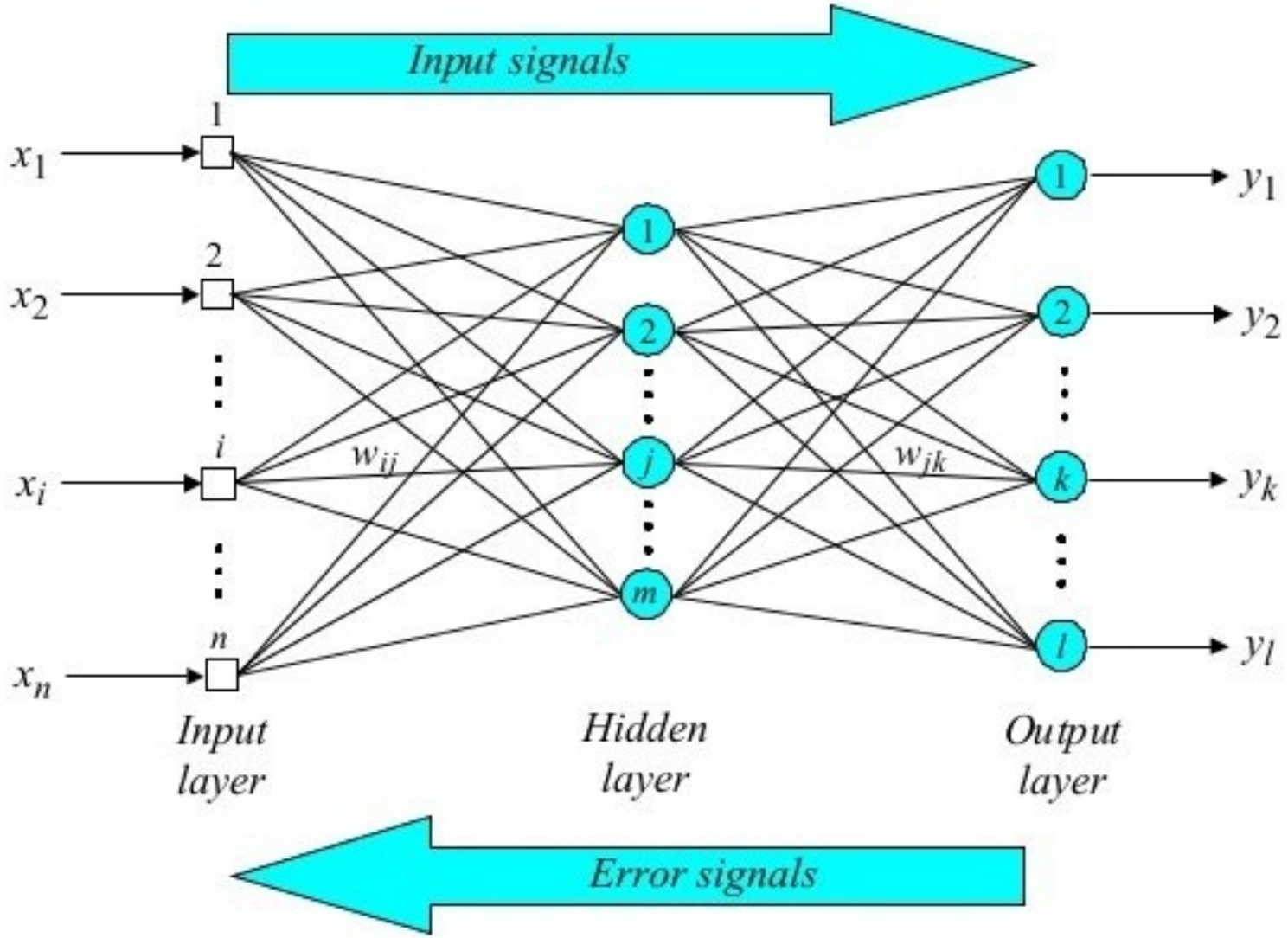
$$\frac{\partial y_j}{\partial net_j} = y_j(1 - y_j)$$

- First term?

$$\frac{\partial E}{\partial y_j} = ???$$

BackPropagation

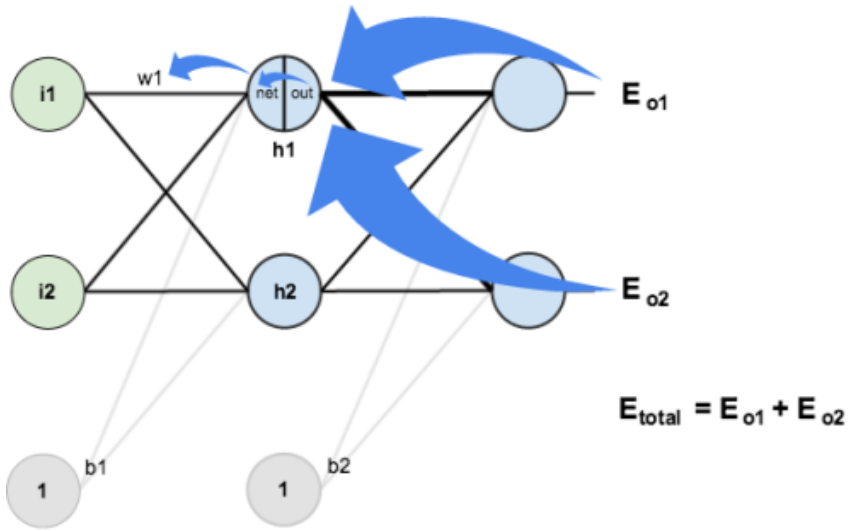
Why are the inner weights difficult?



BackPropagation: inner w_{ij}

First term: Total derivative of the error function

Consider E as a function of the inputs of **all** neurons $L = u, v, \ldots, w$ **receiving** input from neuron j (i.e. all neurons in the successive layer). For an inner weight w_{ij}:



$$\frac{\partial E(y_j)}{\partial y_j} = \frac{\partial E(net_u, net_v, \ldots, net_w)}{\partial y_j} = \sum_{l \in L} \left(\frac{\partial E(net_l)}{\partial y_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial net_l} \frac{\partial net_l}{\partial y_j} \right)$$
$$= \sum_{l \in L} \left(\frac{\partial E}{\partial y_l} \frac{\partial y_l}{\partial net_l} \frac{\partial net_l}{\partial y_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial y_l} \frac{\partial y_l}{\partial net_l} w_{jl} \right)$$

Do you recognise the first two terms?

BackPropagation
Recursive definition

- We can now formulate $\frac{\partial E}{\partial w_{ij}}$ recursively:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}} = \delta_j y_i$$

$$\delta_j = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j} = \begin{cases} (y_j - t_j)y_j(1-y_j) & \text{if } j \text{ is an output neuron,} \\ (\sum_{\ell \in L} \delta_\ell w_{j\ell})y_j(1-y_j) & \text{if } j \text{ is an inner neuron.} \end{cases}$$

- At every iteration, each weight w_{ij} will be updated as follows:

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

$$\Delta w_{ij} = -\eta\frac{\partial E}{\partial w_{ij}} = \begin{cases} -\eta(y_j - t_j)y_j(1-y_j)y_i & \text{if } j \text{ is an out. neuron,} \\ -\eta\left(\sum_{\ell \in L} \delta_\ell w_{j\ell}\right) y_j(1-y_j)y_i & \text{if } j \text{ is an in. neuron.} \end{cases}$$

Stochastic VS Batch Supervised Learning

Stochastic Learning

- "Online" learning
- Calculate the error for each pattern in the training set
- Update the weights after each pattern presentation
- Pros and cons:
  - Slow convergence
  - Fast execution
  - May fluctuate too much on complex data sets

Batch Learning

- "Offline" learning
- Calculate weight updates over the entire training set
- Apply **average** weight updates
- Pros and cons:
  - Faster convergence
  - Slower execution
  - Overfitting?

Stochastic Backprop
Momentum term

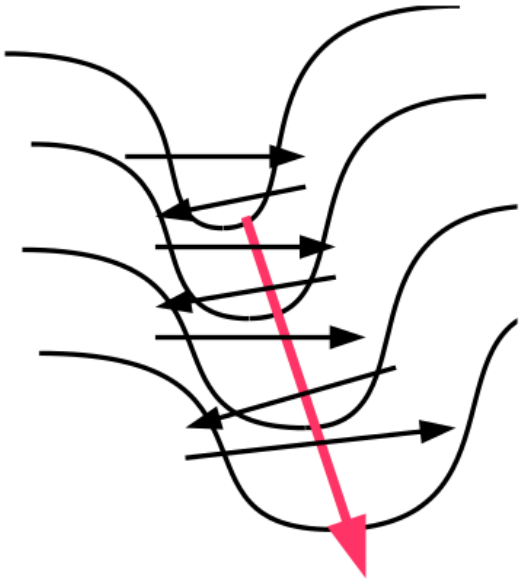
- At every iteration, each weight w_{ij} will be updated as follows:

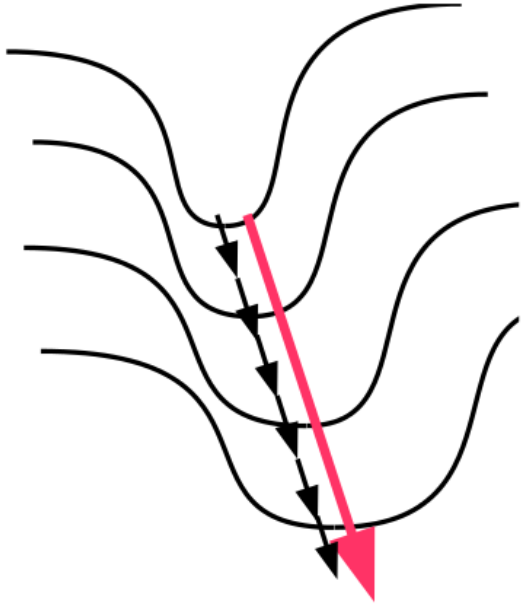
$$w_{ij}(t) = w_{ij}(t) + \Delta w_{ij}(t) + \alpha \Delta w_{ij}(t-1)$$

- α is the momentum coefficient
- Momentum helps maintain the direction of descent
- Do you think momentum is applicable to batch learning?
- We'll discuss more of this later

Pathological curvature of NN error lanscapes

- Is the direction of negative gradient always a good idea?





- Neural networks do not have convex, bowl-shaped error landscapes!
- Left: high curvature direction can lead too far
- Right: low curvature direction -> slow progress

`http://www.cs.toronto.edu/~asamir/cifar/HFO_James.pdf`

Second Order Methods

Newton's method

- Take **curvature** into account when choosing the direction!
- Hessian matrix H of second order partial derivatives encodes curvature information:

$$H(f) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

- You can approximate the error function locally (around $\vec{w}$) as a quadratic function: $E \approx \frac{1}{2}\vec{w}^T H \vec{w} - \theta^T \vec{w}$ (second order Taylor polynomial)
- Now what?

Second Order Methods

1st order VS 2nd order

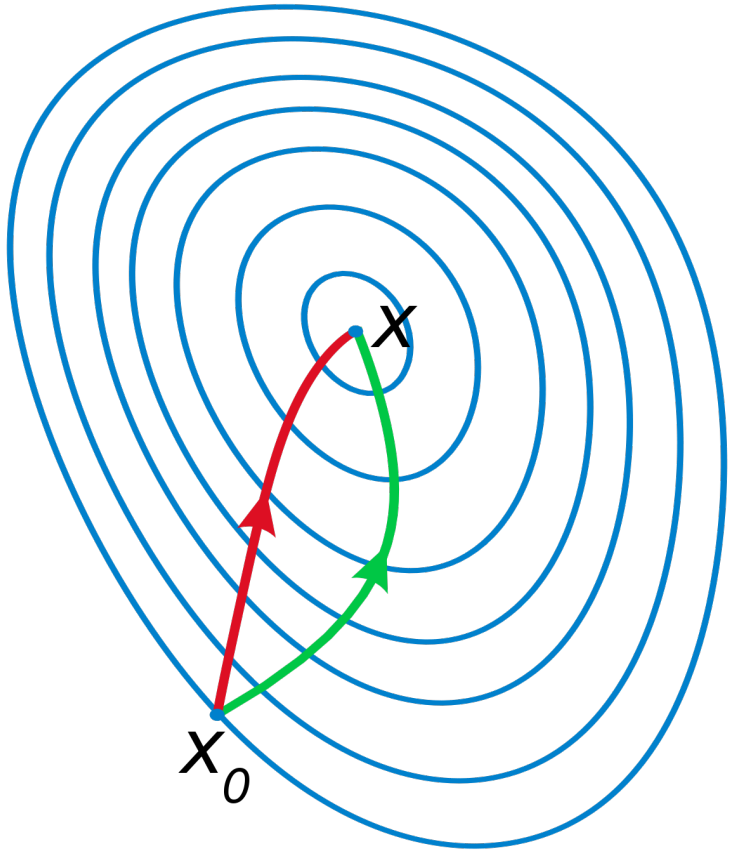
- Gradient descent:

$$\vec{x}_{i+1} = \vec{x}_i - \eta \nabla f(\vec{x}_i)$$

- Newton's method:

$$\vec{x}_{i+1} = \vec{x}_i - \eta [Hf(x_i)^{-1}] \nabla f(\vec{x}_i)$$

- Red line: the route to the minimum is more direct!
- What can possibly go wrong?

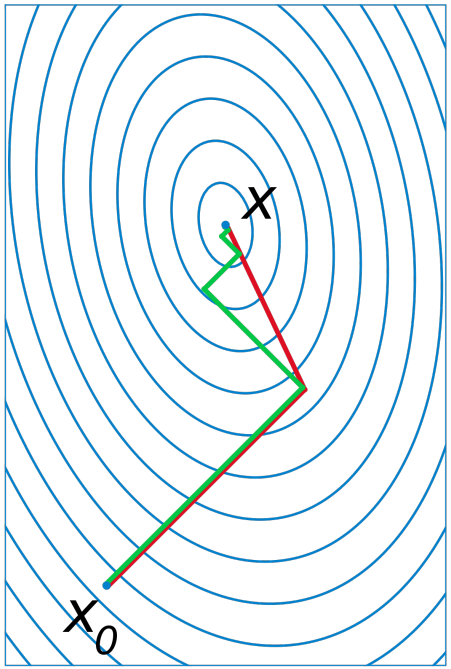


Scalability

Size oh H: w^2. Calculating and inverting H becomes infeasible for high-dimensional problems!

Second Order Methods: Hessian-Free Optimisation

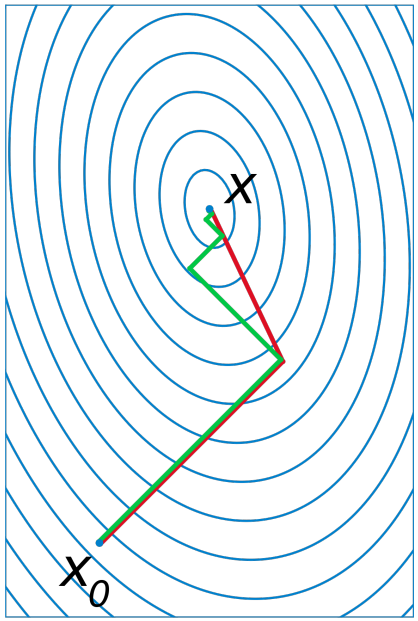
- Option #1: Use a cheap approximation of H, or a subset of H (diagonal only)
- Option #2: Use a minimisation technique designed for quadratic functions that does not use H directly - **conjugate gradient**



- Green line is the gradient
- Red line is better!
- How do we choose a better direction than that of ∇E?
- Rule # 1: Do not mess up what you have already minimized
- Choose **conjugate** [H-orthogonal] gradients: $\vec{u}^T H \vec{v} = 0$ (don't worry, we won't calculate H)

Hessian-Free Optimisation

Scaled Nonlinear Conjugate Gradient

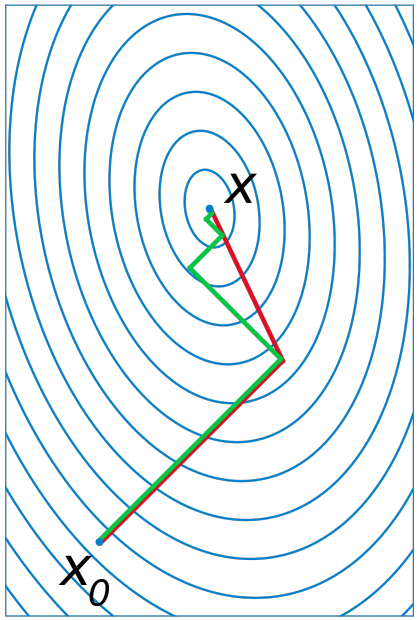


1. Initialisation: $\Delta w_0 = -\nabla E(w_0)$, $d_0 = \Delta w_0$
2. Calculate steepest descent: $\Delta w_n = -\nabla E(w_n)$
3. Compute **direction adjustment**, β_n (How?)
4. Update conjugate direction: $d_n = \Delta w_n + \beta_n d_{n-1}$
5. Perform a **line search** along d_n: $\alpha_n = \arg\min_\alpha E(w_n + \alpha d_n)$
6. Update weights: $w_{n+1} = w_n + \alpha_n d_n$
7. Repeat from (2) until convergence.

Hessian-Free Optimisation

Scaled Nonlinear Conjugate Gradient

- Computing direction adjustment, β_n:



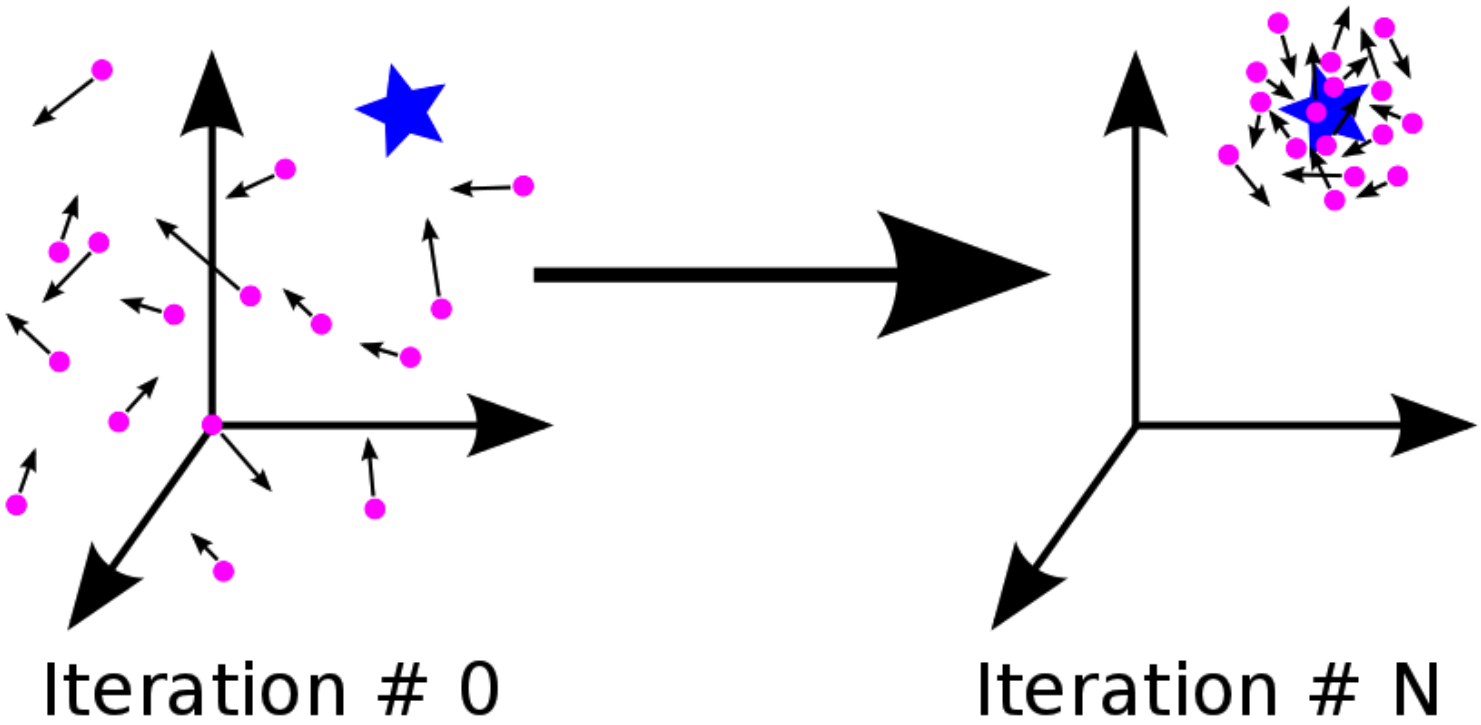
- Fletcher-Reeves: $\beta_n^{FR} = \frac{\Delta w_n^T \Delta w_n}{\Delta w_{n-1}^T \Delta w_{n-1}}$ (ratio of squared gradient norms)
- Polak-Ribieŕe: $\beta_n^{PR} = \frac{\Delta w_n^T (\Delta w_n - \Delta x_{n-1})}{\Delta w_{n-1}^T \Delta w_{n-1}}$
- Hestenes-Stiefel: $\beta_n^{HS} = \frac{\Delta w_n^T (\Delta w_n - \Delta x_{n-1})}{d_{n-1}^T (\Delta w_n - \Delta x_{n-1})}$
- Dai-Yuan: $\beta_n^{DY} = \frac{\Delta w_n^T \Delta w_n}{d_{n-1}^T (\Delta w_n - \Delta x_{n-1})}$

Do you foresee any problems with the 2nd order methods?

Zero-Order Methods

Who needs gradients?

- Numerous population-based optimisation algorithms exist that don't really care about the gradient
- PSO, GA, DE...



Particle Swarm Optimization

- Nature-inspired population-based optimisation technique that models social behaviour of a bird flock
- Operates on a swarm of particles, where each particle is a candidate solution to the problem (i.e., a set of NN weights)
- The swarm "flies" through the search space and converges on an optimal solution:
  - Every particle has a position vector, $\vec{x}(t)$, and a velocity vector, $\vec{v}(t)$. Velocity controls the next step the particle will take:

$$\vec{x}_i(t) = \vec{x}_i(t-1) + \vec{v}_i(t)$$

$$\begin{aligned}\vec{v}_i(t) = \omega\vec{v}_i(t-1) &+ c_1\vec{r}_1 \otimes (\vec{x}_{pbest,i}(t-1) - \vec{x}_i(t-1)) \\ &+ c_2\vec{r}_2 \otimes (\vec{x}_{nbest,i}(t-1) - \vec{x}_i(t-1))\end{aligned}$$

  - Every particle is attracted to the best solution found by its neighbours as well as the best solution encountered by itself

Zero-Order Methods

Why?

- Really easy to understand, visualise, implement
- Less dependent on the starting point
- Activation functions don't have to be differentiable
- Global optimum convergence?..

Problems

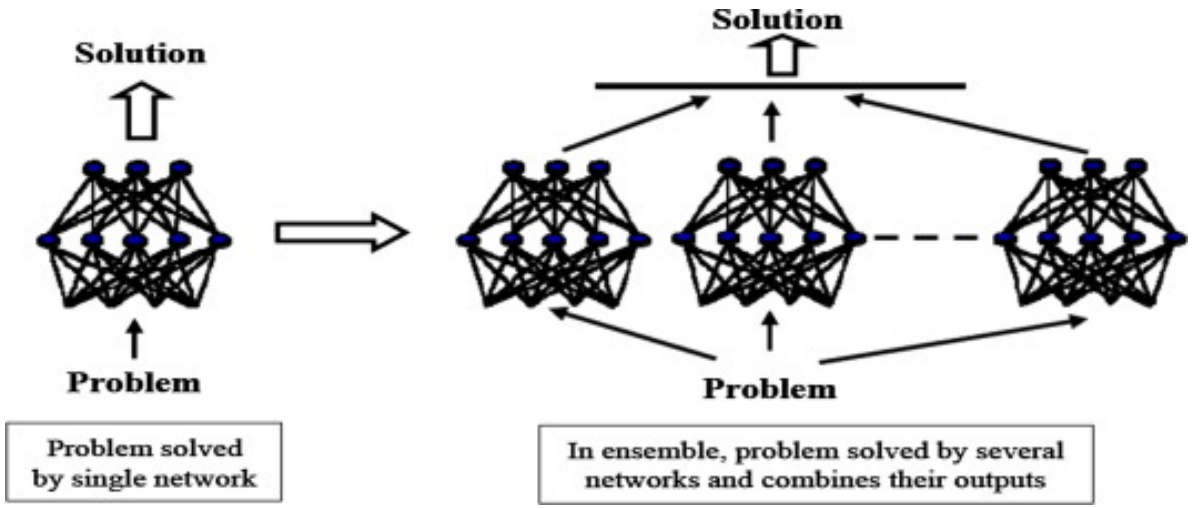
- Scalability and speed
- Distance metrics may not be representative in higher dimensions
- Future research:
  - Cooperative approaches
  - Hybrid approaches: gradient information with social/evolutionary behaviour

Zero-Order, First-Order, Second-Order?

- Zero-Order: too uninformed (and too slow)
- Second-Order: too complex and too slow?
- First-order: used with mind-blowing success

Ensemble Learning

- Generate multiple models from the training data
- Every model makes a prediction on x
- Majority decides what y to output (you can also take the average)
- Question: How do you generate multiple models from the same data set?



The End

- Questions?
- 1st assignment to be released next week
- Lecture next week: NN performance issues