

Machine Learning – COMS3**007**

# Neural Networks

(Learning)

Benjamin Rosman

[Benjamin.Rosman1@wits.ac.za](mailto:Benjamin.Rosman1@wits.ac.za) / [benjros@gmail.com](mailto:benjros@gmail.com)

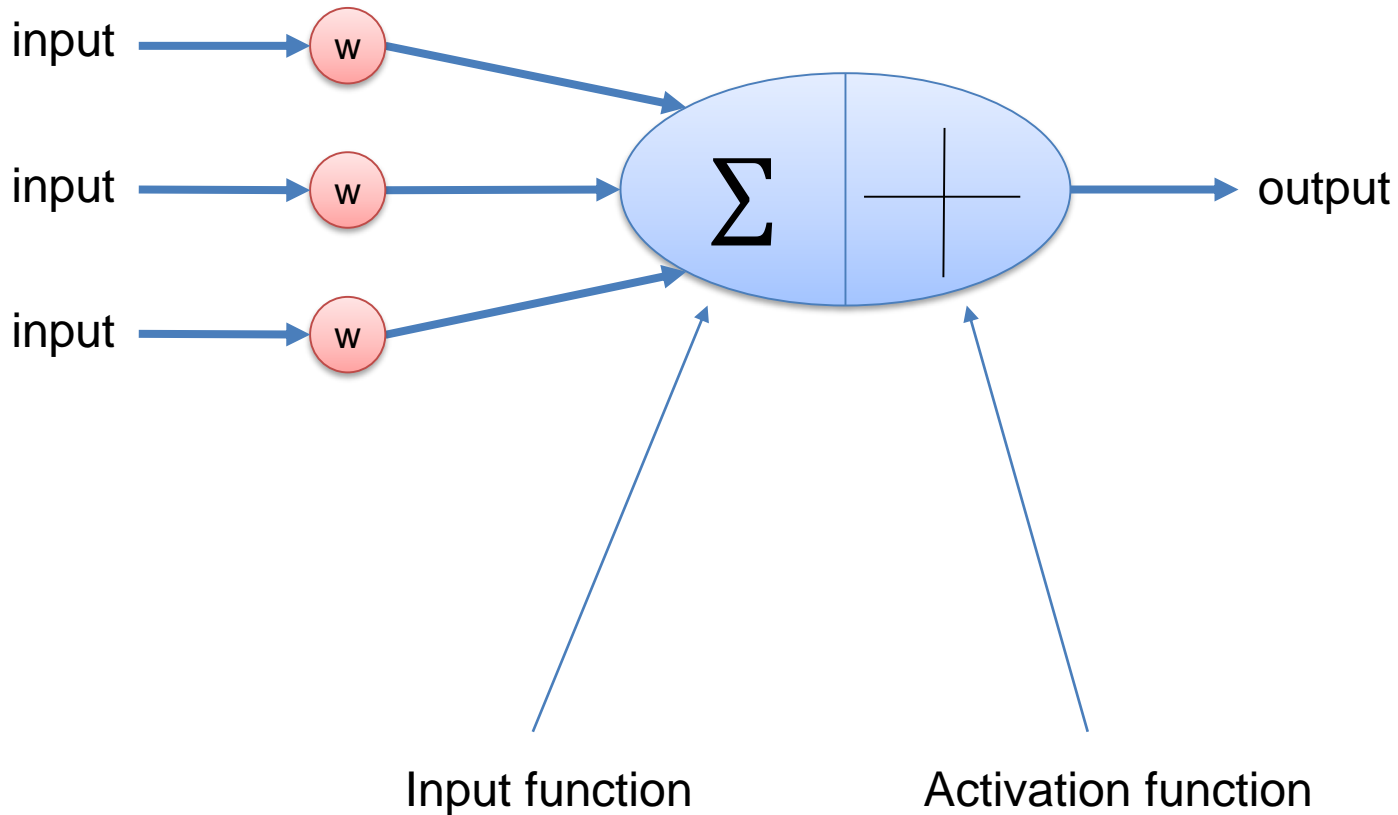
April 2018

Based heavily on course notes by  
Geoffrey Hinton, Chris Williams and  
Victor Lavrenko, Amos Storkey, Eric  
Eaton, and Clint van Alten

# Logistics

- Assignment:
  - How to do well?
    - What was the best performance you could get on your dataset? How?
    - Discussion: why were some choices good/bad? Do your results make sense?
    - Be curious! You are a scientist trying to uncover the “physics” of the dataset

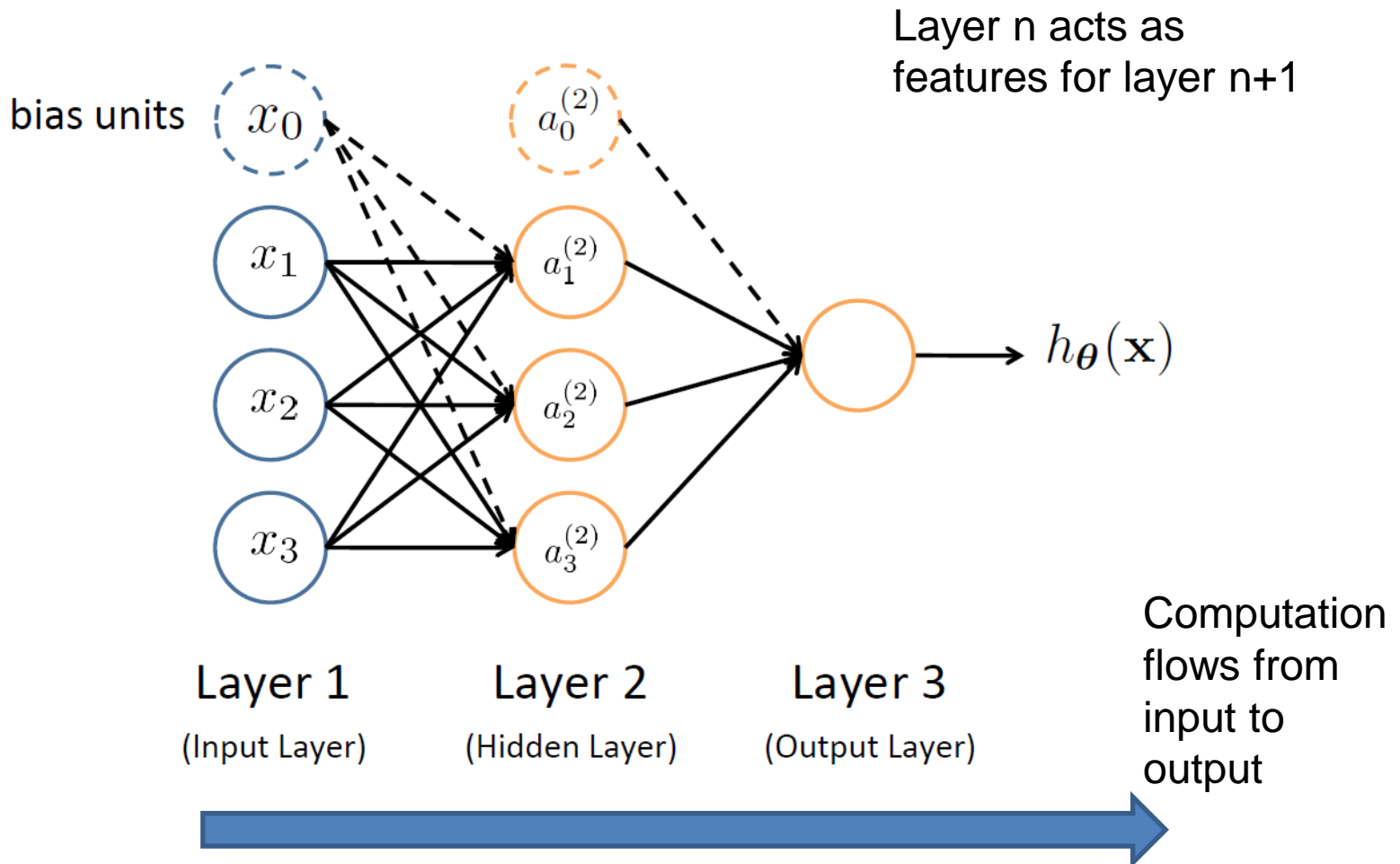
# Neuron anatomy



- Intuition: stack these neurons to learn features!

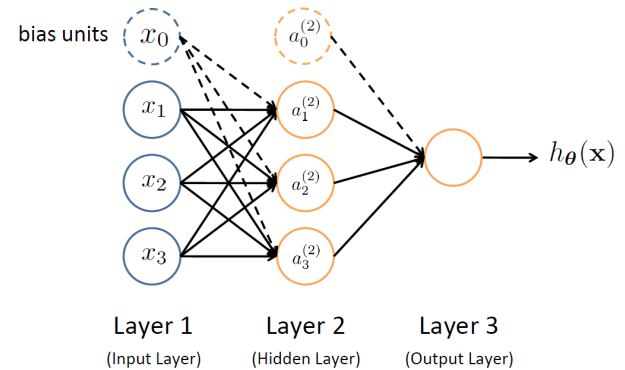
# Stacking neurons

- Neurons arranged in layers



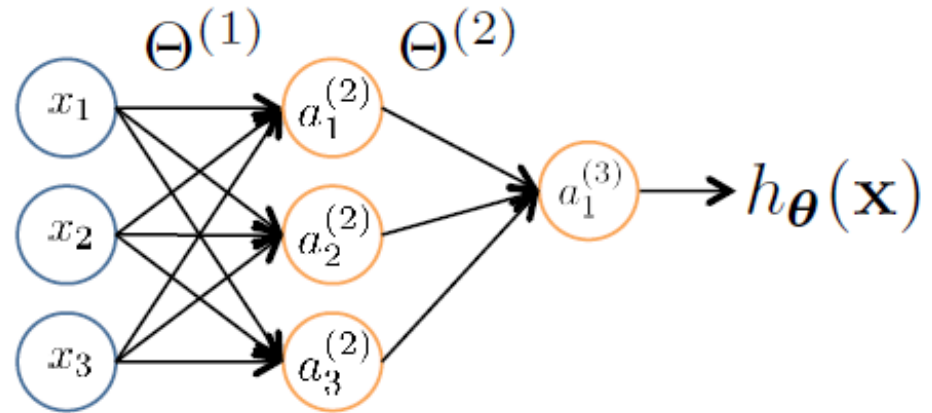
# Feed-forward networks

Sometimes called multilayer perceptrons (MLPs)



- Input layers
  - Raw data
  - As provided by sensor measurements
- Feed-forward networks (most common)
  - **Outputs from one layer become inputs to the next**
- Working forward through the network:
  - Apply **input function** to compute total input
    - Usually just the sum of inputs
  - **Activation function** transforms input to final value
    - Usually nonlinear function
- Output layer: computation target

# Vectorization



- $a_1^{(2)} = g \left( \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \right) = g(z_1^{(2)})$
- $a_2^{(2)} = g \left( \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \right) = g(z_2^{(2)})$
- $a_3^{(2)} = g \left( \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \right) = g(z_3^{(2)})$
- $h_{\Theta}(x) = g \left( \Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)} \right) = g(z_1^{(3)})$

Vectorized steps:

- $\mathbf{z}^{(2)} = \Theta^{(1)} \mathbf{x}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$
- Add  $a_0^{(2)} = 1$
- $\mathbf{z}^{(3)} = \Theta^{(2)} \mathbf{a}^{(2)}$
- $h_{\Theta}(\mathbf{x}) = \mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$

# Perceptron Learning Rule

- $\theta \leftarrow \theta + \alpha(y - h(x))x$
- Intuition:
  - If output is correct ( $y = h(x)$ ):
    - Don't change weights
  - If output too low ( $h(x) = 0, y = 1$ ):
    - Increment weights
  - If output too high ( $h(x) = 1, y = 0$ ):
    - Decrement weights
- If the data is linearly separable (set of consistent weights exists): guaranteed to converge.

# Learning in a NN

- Similar to perceptron learning:
  - Cycle through training examples
  - If network output is correct, no changes are made
  - If there is an error, adjust weights to reduce error
- We are just performing (stochastic) gradient descent
- Challenge:
  - It's easy to talk about error in the output layer, but what about the hidden layers?
    - We need to assign “blame” to the weights that need to change



# Cost functions

- Logistic regression:

- $J(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^n [y_i \log h_{\boldsymbol{\theta}}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_i))] + \frac{\lambda}{2n} \sum_{j=1}^d \theta_j^2$

- Neural network:

- $h_{\boldsymbol{\theta}} \in \mathbb{R}^K$
- $(h_{\boldsymbol{\theta}}(\mathbf{x}))_i = i^{\text{th}}$  output

- $J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^n \sum_{k=1}^K y_{ik} \log(h_{\Theta}(\mathbf{x}_i))_k + (1 - y_{ik}) \log(1 - (h_{\Theta}(\mathbf{x}_i))_k) \right] + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} \left( \Theta_{ji}^{(l)} \right)^2$

- This is for classification. The error changes if the task changes.

- E.g. regression:  $J(\Theta) = \frac{1}{2n} \sum_{i=1}^n (h_{\Theta}(\mathbf{x}_i) - y_i)^2$

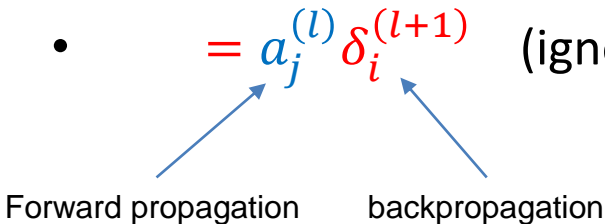
# Optimising the NN

- $J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^n \sum_{k=1}^K y_{ik} \log(h_{\Theta}(\mathbf{x}_i))_k + (1 - y_{ik}) \log(1 - (h_{\Theta}(\mathbf{x}_i))_k) \right] + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} (\theta_{ji}^{(l)})^2$
- Solve as  $\min_{\Theta} J(\Theta)$ 
  - **No closed-form solution in general**
    - Use iterative solution (GD)
  - Also: this is not convex, so GD on a neural net will give us a **local optimum**
- Gradient descent:
  - For each parameter  $\Theta_{ji}^{(l)}$ :
    - $\Theta_{ji}^{(l)} \leftarrow \Theta_{ji}^{(l)} - \alpha \frac{\partial J(\Theta)}{\partial \Theta_{ji}^{(l)}}$

Learning rate



# Optimising the NN

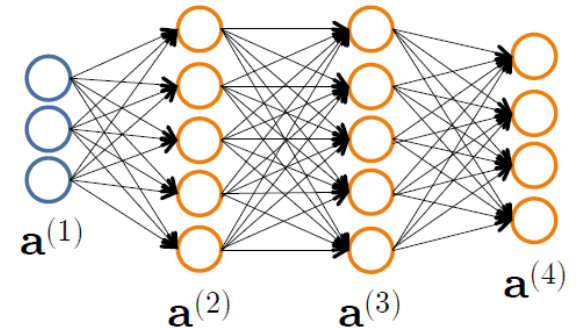
- So, need to be able to compute:
  - $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$  - the gradient of the error wrt **all** the parameters
    - Use **the chain rule** to compute: called **backpropagation** in ANNs
    - **Compute backwards by layer from output layer to input layer**
  - How to compute this?
    - $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = (\text{influence of connection}) \times (\text{error at next layer})$
    - $= a_j^{(l)} \delta_i^{(l+1)}$  (ignoring regularization)

# Forward propagation

- Given one labelled training instance  $(\mathbf{x}, y)$ :
- Compute activations  $\mathbf{y}^{(i)}$

- Forward propagation:

- $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{z}^{(2)} = \Theta^{(1)} \mathbf{a}^{(1)}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$
- $\mathbf{z}^{(3)} = \Theta^{(2)} \mathbf{a}^{(2)}$
- $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$
- $\mathbf{z}^{(4)} = \Theta^{(3)} \mathbf{a}^{(3)}$
- $\mathbf{a}^{(4)} = h_{\Theta}(\mathbf{x}) = g(\mathbf{z}^{(4)})$



[add  $a_0^{(2)}$ ]

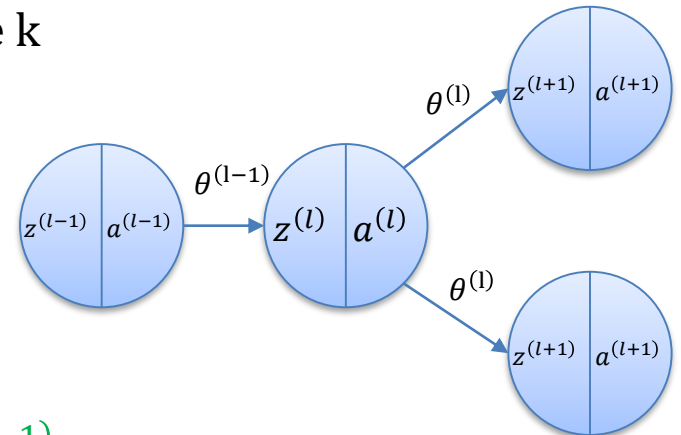
[add  $a_0^{(3)}$ ]

# Backpropagation intuition

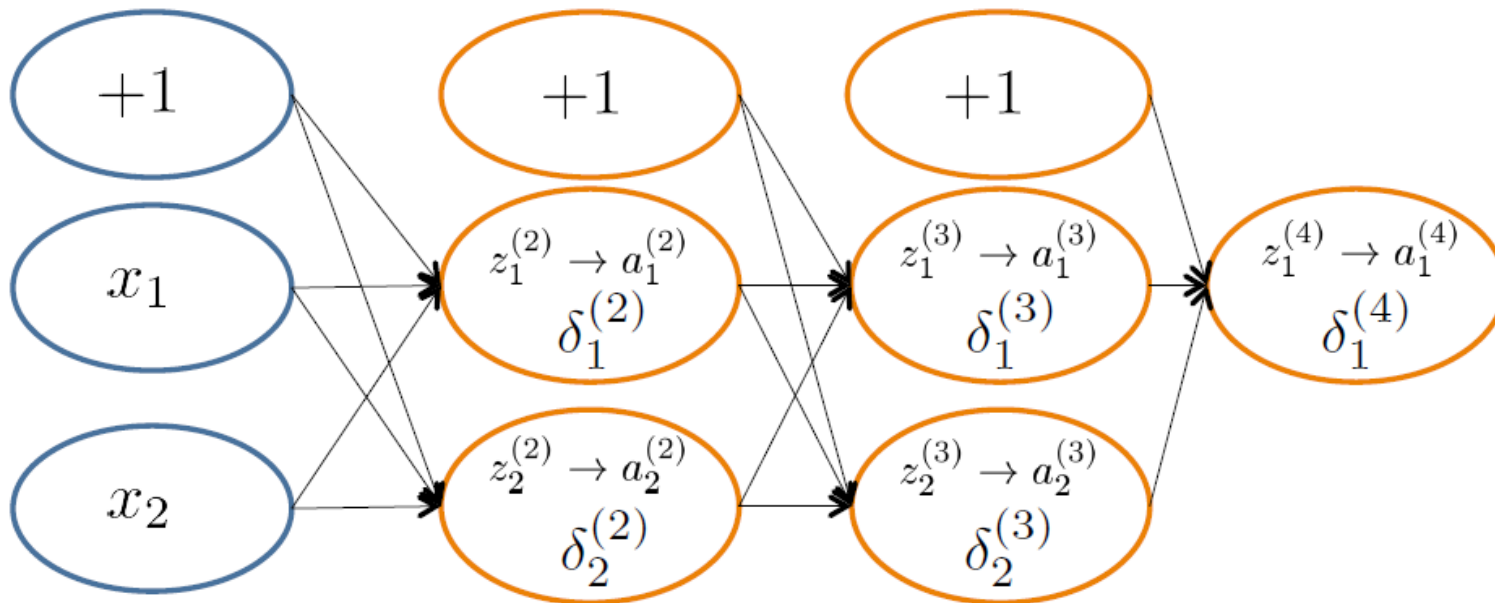
- Forward prop gave us the activations  $\mathbf{a}$
- Each hidden node  $j$  is “responsible” for some fraction of the error  $\delta_j^{(l)}$  in each of the output nodes to which it connects
- $\delta_j^{(l)}$  is divided according to the strength of the connection between hidden node and output node
- Then, the “blame” is propagated back to provide error values for the hidden layer

# Backpropagation derivation

- We need to be able to compute the derivative:  $\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l-1)}}$
- $\frac{\partial J}{\partial \theta_{ij}^{(l-1)}} = \frac{\partial J}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial \theta_{ij}^{(l-1)}}$  (chain rule)
- $\frac{\partial J}{\partial a_i^{(l)}} = \sum_m \frac{\partial J}{\partial z_m^{(l+1)}} \frac{\partial z_m^{(l+1)}}{\partial a_i^{(l)}} = \sum_m \frac{\partial J}{\partial z_m^{(l+1)}} \theta_{mi}^{(l)}$  (sum over next neurons)
- Let  $\delta_k^{(l)} = \frac{\partial J}{\partial z_k^{(l)}}$  be the change in error from node k
- $\frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = g'(z_i^{(l)})$
- $\frac{\partial z_i^{(l)}}{\partial \theta_{ij}^{(l-1)}} = a_j^{(l-1)}$
- Therefore:  $\frac{\partial J}{\partial \theta_{ij}^{(l-1)}} = \left( \sum_m \delta_m^{(l+1)} \theta_{mi}^{(l)} \right) g'(z_i^{(l)}) a_j^{(l-1)}$



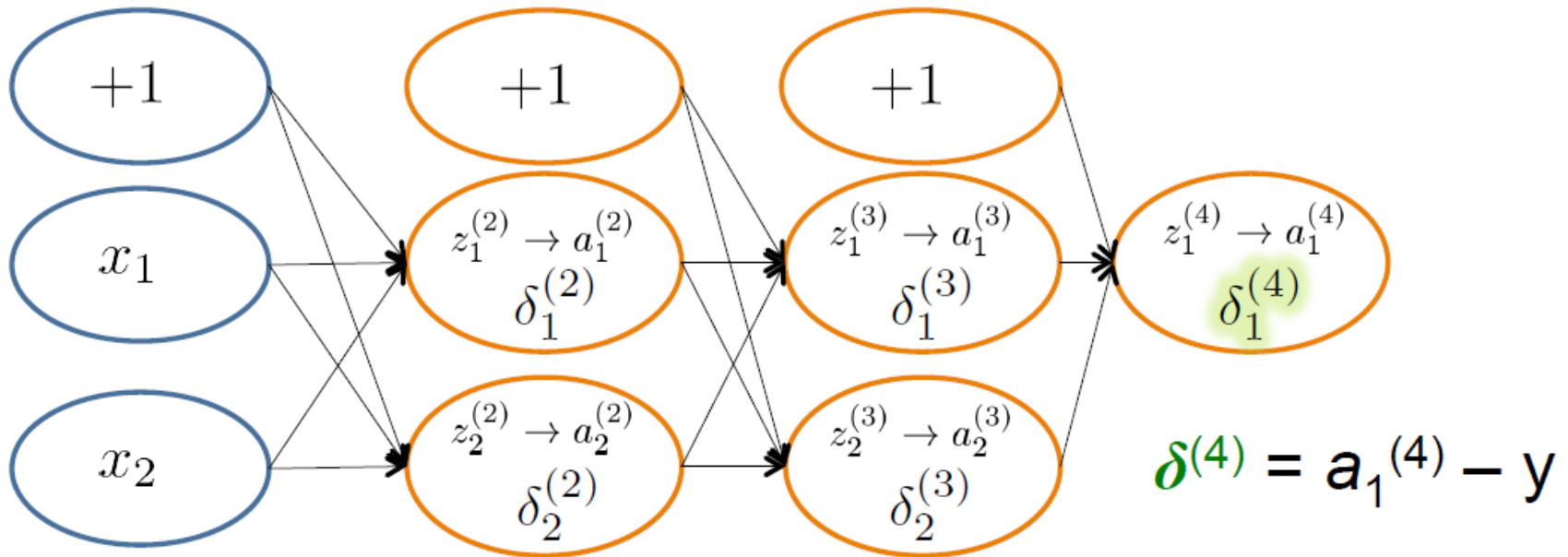
# Backpropagation intuition



- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} E(\mathbf{x}_i)$

Different notation for  
the same thing:  
 $E = C = J = \text{cost}$

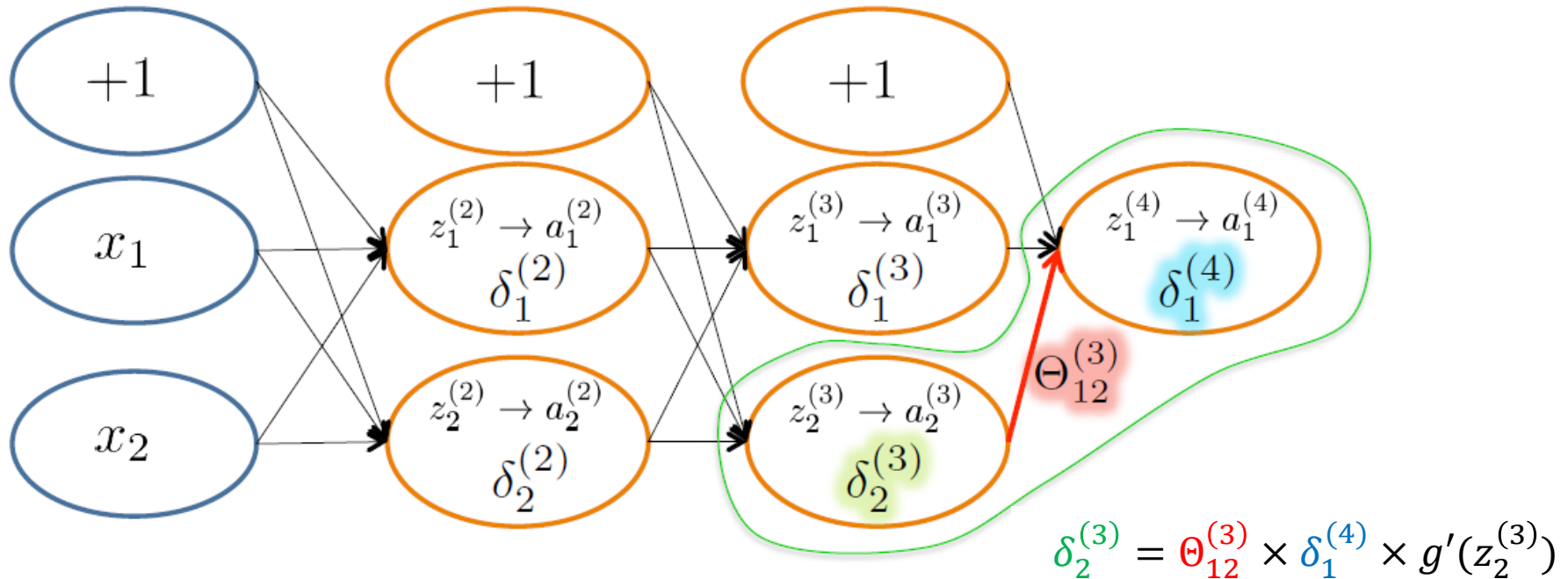
# Backpropagation intuition



- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} E(\mathbf{x}_i)$

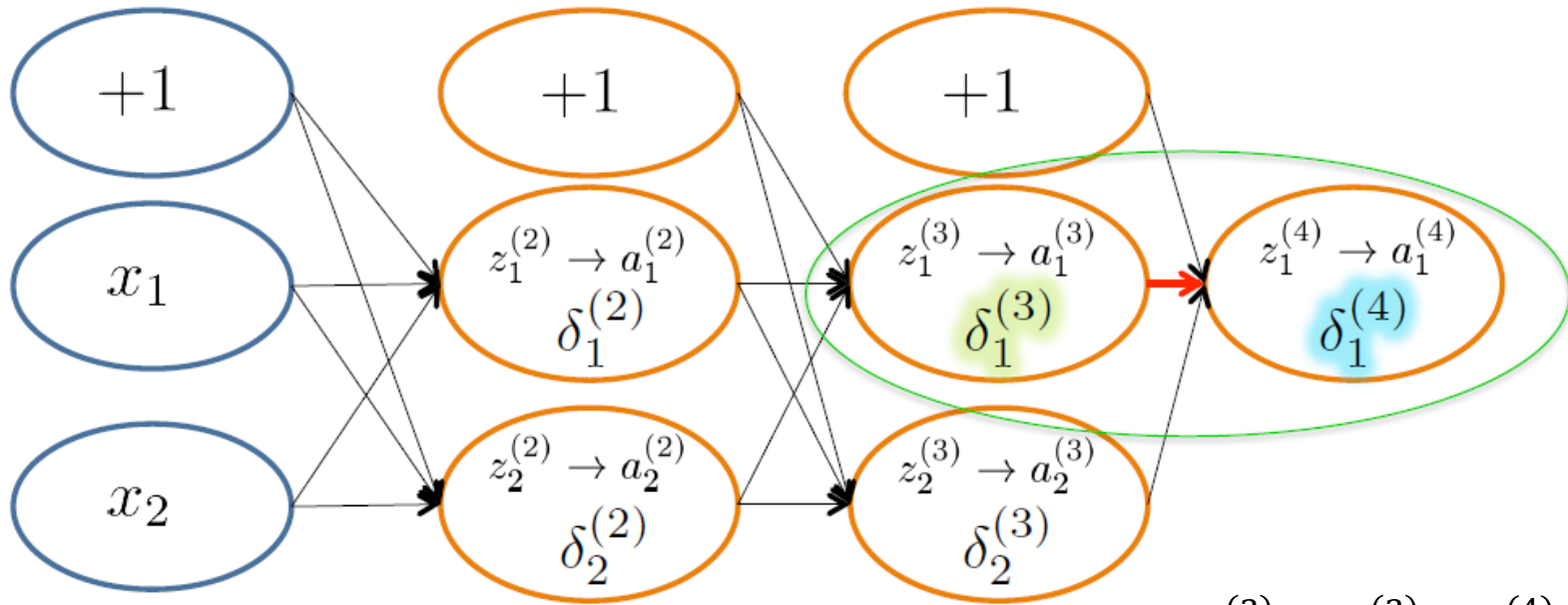


# Backpropagation intuition



- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} E(\mathbf{x}_i)$

# Backpropagation intuition

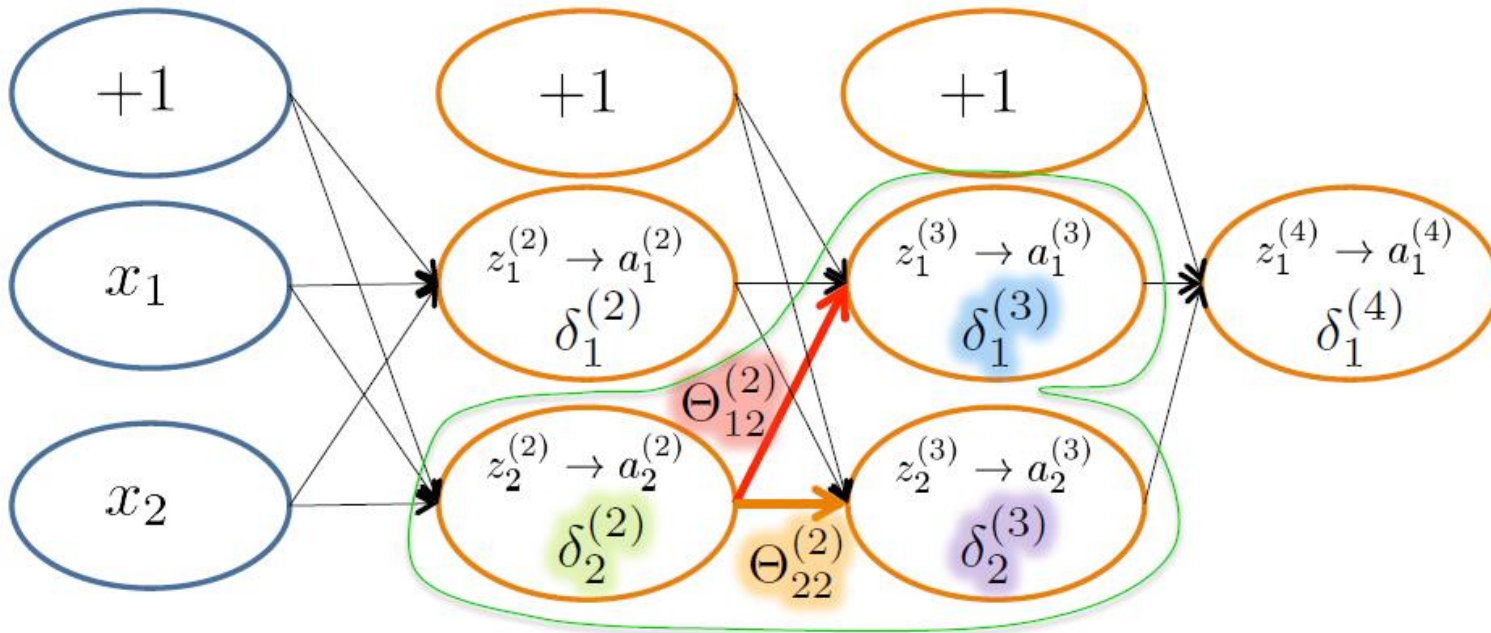


$$\delta_2^{(3)} = \Theta_{12}^{(3)} \times \delta_1^{(4)} \times g'(z_2^{(3)})$$

$$\delta_1^{(3)} = \Theta_{11}^{(3)} \times \delta_1^{(4)} \times g'(z_1^{(3)})$$

- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} E(\mathbf{x}_i)$

# Backpropagation intuition

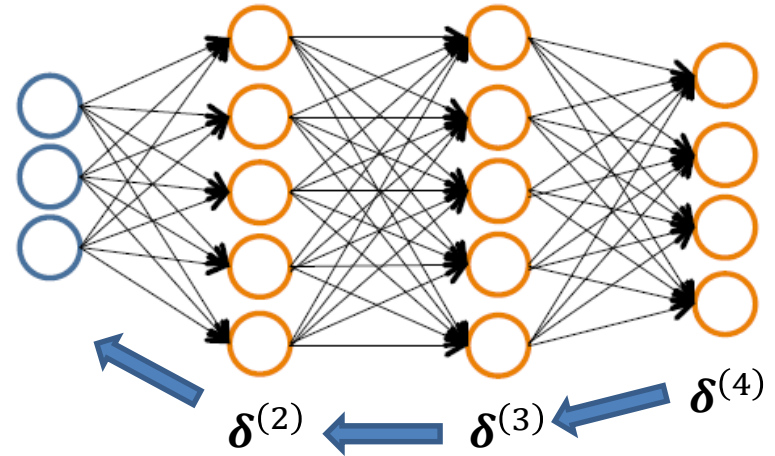


$$\delta_2^{(2)} = \Theta_{12}^{(2)} \times \delta_1^{(3)} \times g'(z_2^{(2)}) + \Theta_{22}^{(2)} \times \delta_2^{(3)} \times g'(z_2^{(2)})$$

- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} E(\mathbf{x}_i)$

# Backpropagation intuition

- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$



- Backprop:

- $\delta^{(4)} = \mathbf{a}^{(4)} - \mathbf{y}$

- $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \times g'(\mathbf{z}^{(3)})$

- $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \times g'(\mathbf{z}^{(2)})$

- No  $\delta^{(1)}$  - no error in inputs

If  $g$  is a sigmoid:

$$g'(\mathbf{z}^{(3)}) = \mathbf{a}^{(3)}(1 - \mathbf{a}^{(3)})$$

$$g'(\mathbf{z}^{(2)}) = \mathbf{a}^{(2)}(1 - \mathbf{a}^{(2)})$$

- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$

# Backpropagation algorithm

Set  $\Delta_{ij}^{(l)} = 0, \forall l, i, j$

For each training instance  $(\mathbf{x}_i, y_i)$ :

Set  $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute  $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$  with forward propagation

Compute  $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors  $\{\boldsymbol{\delta}^{(L-1)}, \dots, \boldsymbol{\delta}^{(2)}\}$

Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Don't  
regularise  
the bias

Compute average regularised gradient  $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

- $\mathbf{D}^{(l)}$  is the matrix of partial derivatives of  $J(\Theta)$
- Note: can vectorise  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$  as  $\boldsymbol{\Delta}^{(l)} = \boldsymbol{\Delta}^{(l)} + \boldsymbol{\delta}^{(l+1)} \mathbf{a}^{(l)T}$

# Training a NN with GD and Backprop

Given: training data  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$

Initialise all  $\Theta^{(l)}$  randomly (NOT to 0)

Loop (for each epoch):

Set  $\Delta_{ij}^{(l)} = 0, \forall l, i, j$

For each training instance  $(\mathbf{x}_i, y_i)$ :

Set  $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute  $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$  with forward propagation

Compute  $\delta^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors  $\{\delta^{(L-1)}, \dots, \delta^{(2)}\}$

Compute gradients  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute average regularised gradient  $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

Backpropagation

Update weights via gradient step  $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$

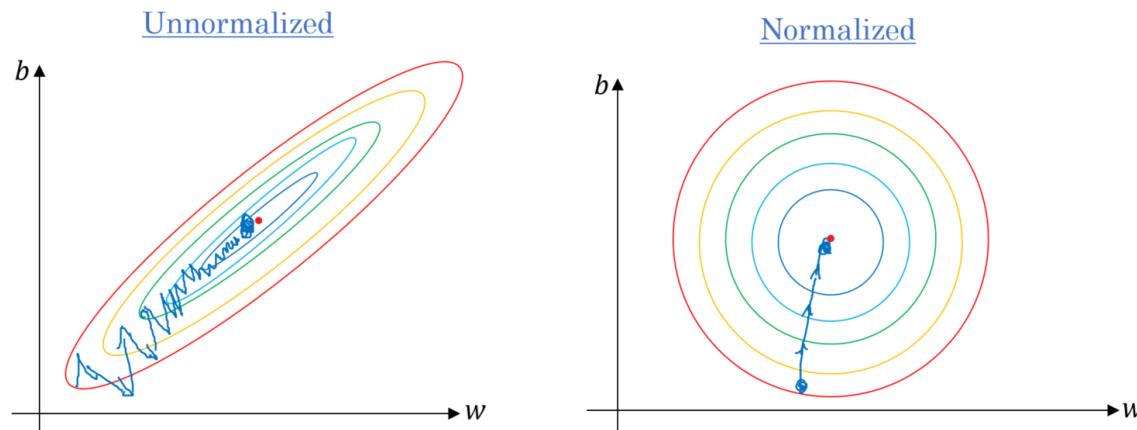
Until weights converge or max epochs reached

# Implementation: initialisation

- If two hidden units have:
  - The same bias
  - The same incoming and outgoing weights
- Then they **will always get the exact same gradient!**
  - Can never learn to be different features
  - **Break symmetry by initialising weights to small random values**
    - And specifically Gaussian random numbers

# Implementation: normalisation

- When training data features have different scales, learning can be slower



- **Scale (normalise) features to lie in the same intervals**

- Either: set means to 0, variances to 1

$$x \leftarrow \frac{x - \text{mean}(x)}{\text{variance}(x)}$$

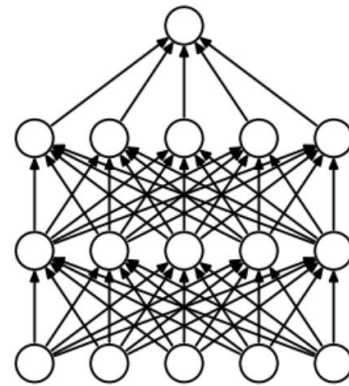
- Or: min to 0, max to 1

$$x \leftarrow \frac{x - \min(x)}{\max(x) - \min(x)}$$

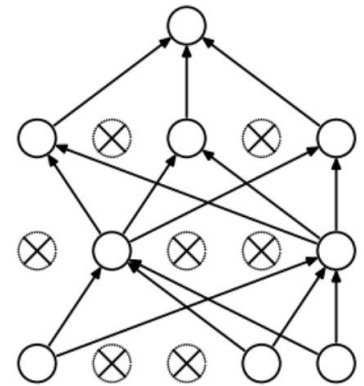


# Implementation: dropout

- Neural networks often have a large number of parameters
  - Leads to overfitting



(a) Standard Neural Net

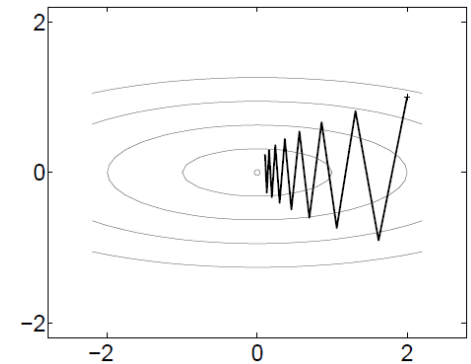


(b) After applying dropout.

- Dropout:
  - While training, at each stage, with probability  $p$  remove a node and all its connections
    - Another hyperparameter!
    - Prevents network from becoming too dependent on any one node
  - While testing, use all nodes

# Implementation: momentum

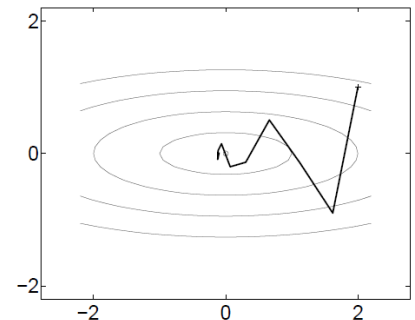
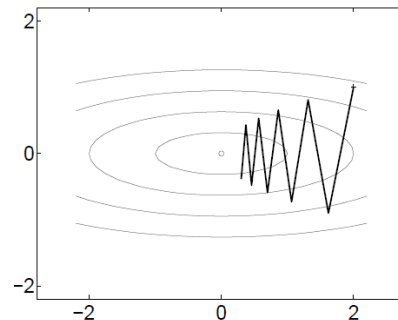
- GD often ends up “zig-zagging” in ravines
- Use momentum:
  - “slow changes in direction”
  - A ball rolling down a hill
  - Another hyperparameter



- Let change in weight  $\theta$  at time  $t$  be  $\delta\theta(t)$
- Then  $\delta\theta(t + 1) = -\alpha \frac{\partial J}{\partial \theta} + \beta \delta\theta(t)$

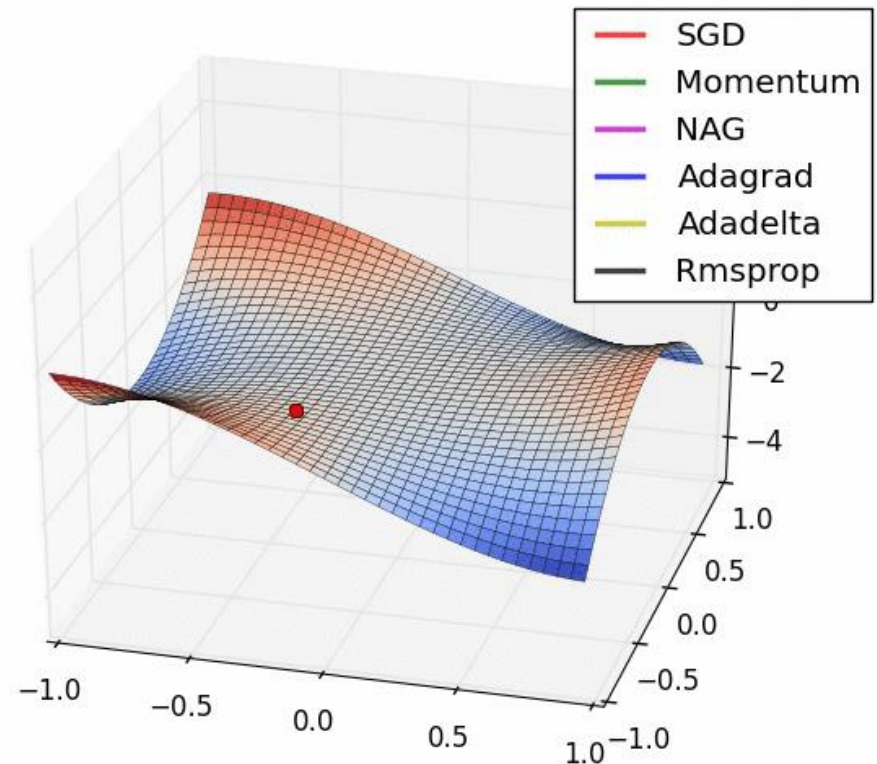
$\beta$  is a hyperparameter controlling amount of momentum

$\delta\theta(t)$  is the previous weight change



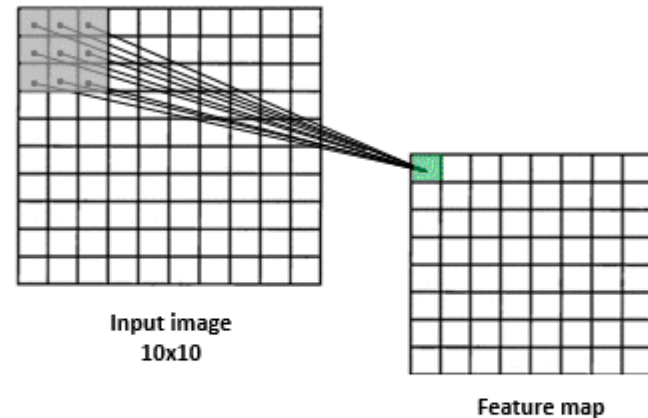
# Implementation: optimisers

- There are many different optimisers that can be used instead of GD and momentum
  - Often **adapting learning rates**



# Images: shared weights

- Consider an image:
  - 1M pixels
  - 10k neurons on first hidden layer
  - = 10,000,000,000 weights to first layer!!!!
- Instead of connecting everything, **connect small local patches (kernels)**
  - **Share weights** between all patches
  - Far fewer weights!
  - Asking hidden neuron to activate wherever this pattern is seen



- Define many of these patches
- Called: **convolutional neural network**
  - Main architecture for images

# Recap

- Recap on neural networks
- Revisited the perceptron learning rule
- General idea of learning
- Forward propagation
- Backpropagation
- Incorporating gradient descent
- Tips for improving training
  - Initialisation, normalisation, dropout, momentum, optimisers, shared weights