
[course site]

Day 3 Lecture 1

# Backpropagation

Elisa Sayrol

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Department of Signal Theory
and Communications
*Image Processing Group*

# Acknowledgements


Kevin McGuinness

kevin.mcguinness@dcu.ie

Research Fellow

Insight Centre for Data Analytics

Dublin City University

…in our last lecture

# Multilayer perceptrons

When each node in each layer is a linear combination of **all inputs from the previous layer** then the network is called a multilayer perceptron (MLP)

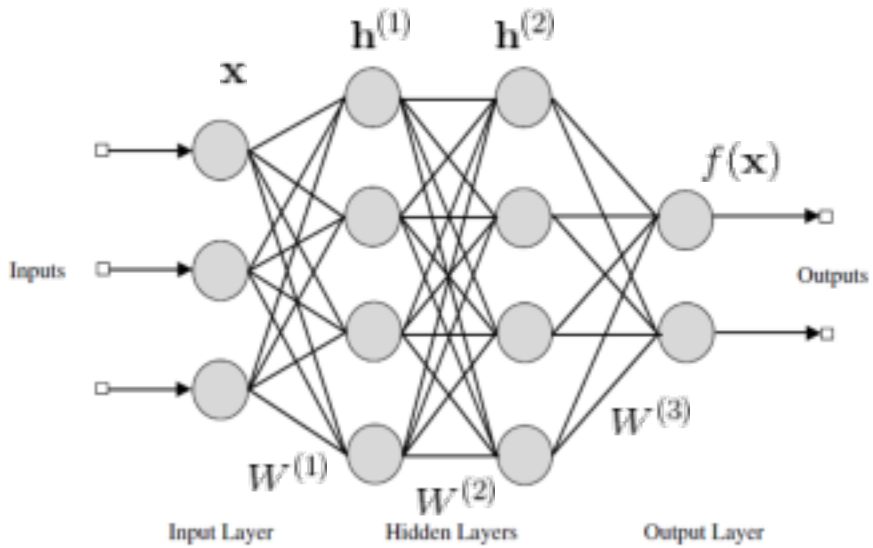
Weights can be organized into matrices.

**Forward pass** computes

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}^{(t)} = g(\overbrace{W^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)}}^{\mathbf{a}^{(t)}})$$

$$f(\mathbf{x}) = \mathbf{h}^{(L)}$$



# Training MLPs

With Multiple layers we need to minimize the **loss function** $\mathcal{L}(\boldsymbol{f_\theta}(\boldsymbol{x}), \boldsymbol{y})$ with respect to all the parameters of the model $\boldsymbol{\theta(W^{(k)}, b^{(k)})}$**:**

$$W^* = argmin_\theta \mathcal{L}(f_\theta(x), y)$$

**Gradient Descent:** Move the parameter $\boldsymbol{\theta_j}$ in small steps in the direction opposite sign of the derivative of the loss with respect j:

$$\boldsymbol{\theta_j}^{(\boldsymbol{n})} = \boldsymbol{\theta_j}^{(\boldsymbol{n-1})} - \boldsymbol{\alpha}^{(\boldsymbol{n-1})} \cdot \boldsymbol{\nabla_{\theta_j} \mathcal{L}(y, f(x))}$$

**Stochastic gradient descent (SGD):** estimate the gradient with one sample, or better, with a **minibatch** of examples.

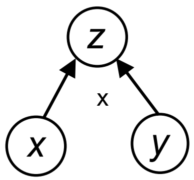
For MLP gradients can be found using the **chain rule** of differentiation.

The calculations reveal that the gradient wrt. the parameters in layer k only depends on the error from the above layer and the output from the layer below. This means that the gradients for each layer can be computed iteratively, starting at the last layer and propagating the error back through the network. This is known as the **backpropagation** algorithm.

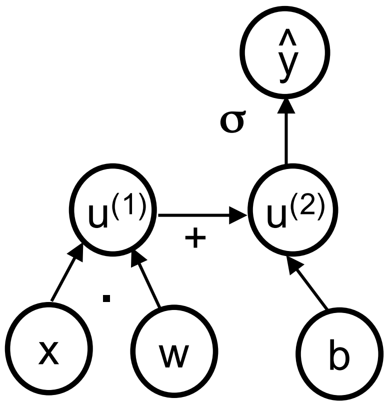
# Backpropagation algorithm

- Computational Graphs
- Examples applying chain of rule in simple graphs
- Backpropagation applied to Multilayer Perceptron
- Issues on Backpropagation and training

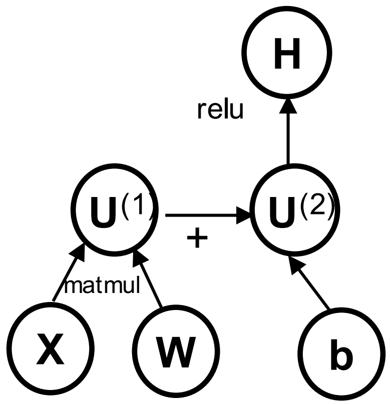
# Computational graphs



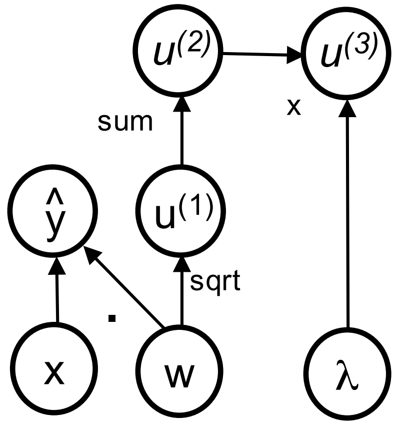

$z = xy$




$\hat{y} = \sigma(\mathrm{x}^T \mathrm{w} + \mathrm{b})$




$\boldsymbol{H} = \max\{0, \boldsymbol{XW} + \boldsymbol{b}\}$




$\hat{y} = \mathrm{x}^T \mathrm{w}$

$$\lambda \sum_i w_i^2$$

*From Deep Learning Book*

# Computational graphs

*Applying the Chain Rule  to Computational Graphs*

$$y = g(x) \quad z = f(y)$$

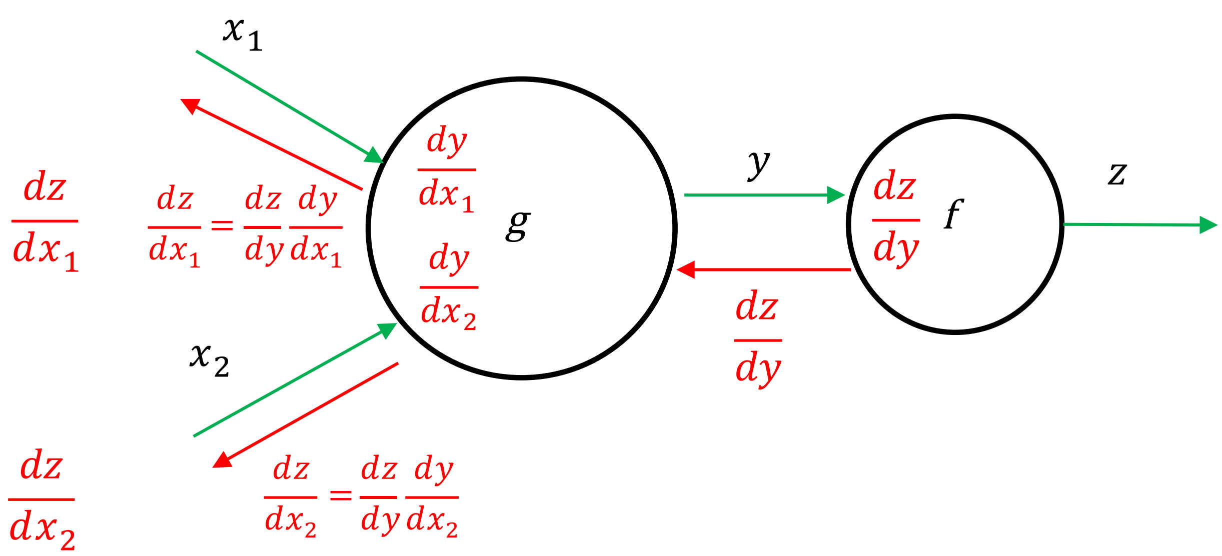
$$z = f\big(g(x)\big)$$

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

*For vectors:*

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$$



# Computational graphs

*Numerical Examples*

$$f(x, y, z) = (x + y)z$$

$$Example\ x = -2, y = 5, z = -4$$

$$q = (x + y) \qquad \frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$$

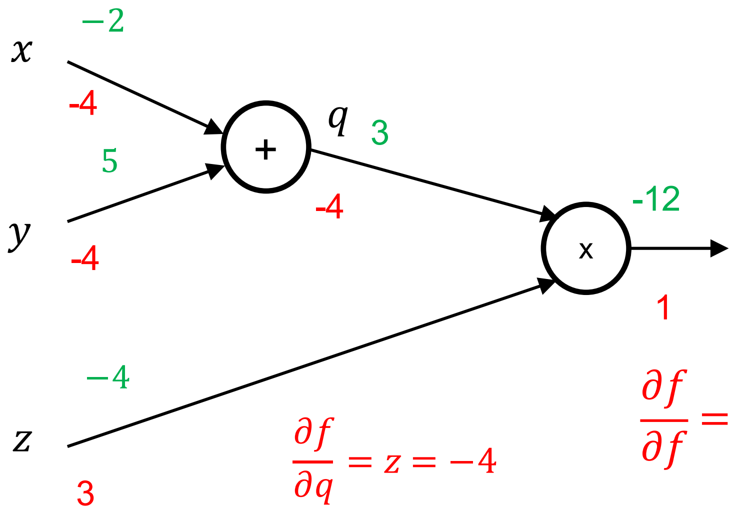
$$f = qz \qquad \frac{\partial f}{\partial q} = z \qquad \frac{\partial f}{\partial z} = q$$

$$\frac{\partial f}{\partial f} = 1$$

$$We\ want\ to\ compute: \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = -4 \cdot 1 = -4$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = -4 \cdot 1 = -4$$




$$\frac{\partial f}{\partial q} = z = -4$$

$$\frac{\partial f}{\partial f} = 1$$

$$\frac{\partial f}{\partial z} = q = 3$$

*From Stanford Course: Convolutional Neural Networks for Visual Recognition 2017*

# Computational graphs

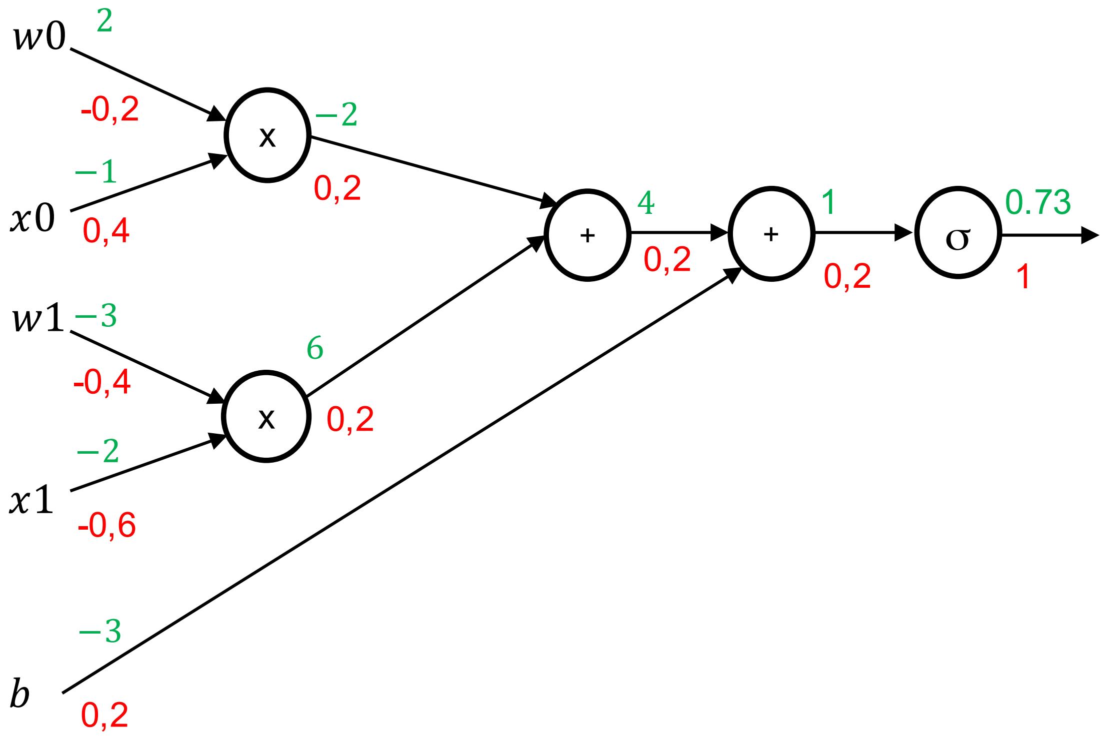
*Numerical Examples*

$$f(x,y,z) = \sigma(w_0 x_0 + w_1 x_1 + b)$$

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

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

$$\frac{d\sigma(x)}{x} = (1-\sigma(x))(\sigma(x))$$




*From Stanford Course: Convolutional Neural Networks for Visual Recognition*

# Computational graphs

*Gates. Backward Pass*

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \frac{d\sigma(x)}{x} = (1-\sigma(x))(\sigma(x))$$

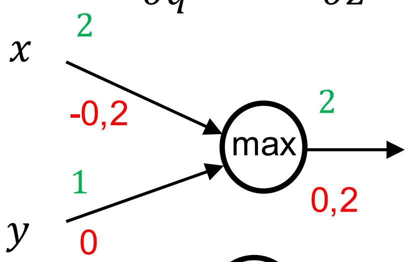
*In general: Derivative of a function*

$$q = (x+y) \qquad \frac{\partial q}{\partial x} = 1 \quad \frac{\partial q}{\partial y} = 1$$

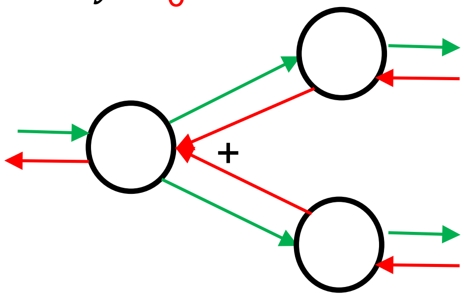
*Sum: Distributes the gradient to both branches*

$$f = qz \qquad \frac{\partial f}{\partial q} = z \quad \frac{\partial f}{\partial z} = q$$

*Product: Switches gradient weigth values*




*Max: Routes the gradient only to the higher input branche (not sensitive to the lower branche)*




*Add branches: Branches that split in the forward pass and merge in the backward pass, add gradients*

# Computational graphs

*Numerical Examples*

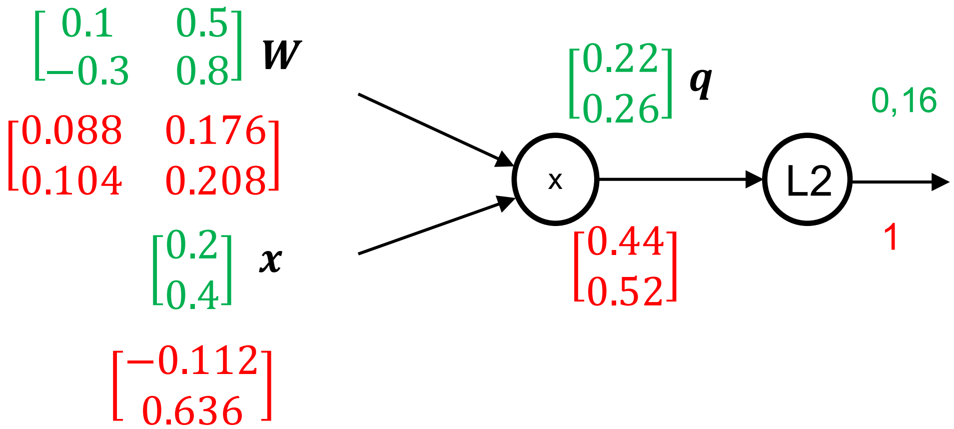
$$f(x, W) = \|\boldsymbol{W} \cdot \boldsymbol{x}\|^2 = \sum_{i=1}^{n} (\boldsymbol{W} \cdot \boldsymbol{x})_i^2 = \sum_{i=1}^{n} (\boldsymbol{q})_i^2$$

$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\nabla_{\boldsymbol{q}} f = 2\boldsymbol{q}$$

$$\nabla_{\boldsymbol{W}} f = 2\boldsymbol{q} \cdot \boldsymbol{x}^T$$

$$\nabla_{\boldsymbol{x}} f = 2\boldsymbol{W}^T \cdot \boldsymbol{q}$$




*From Stanford Course: Convolutional Neural Networks for Visual Recognition 2017*

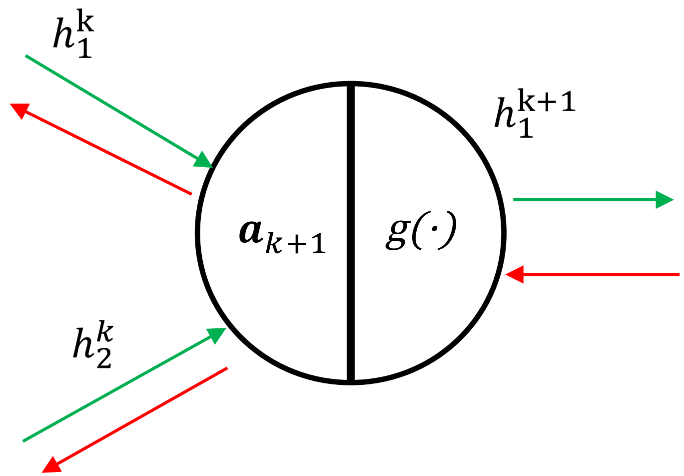
# Backpropagation applied to Multilayer Perceptron

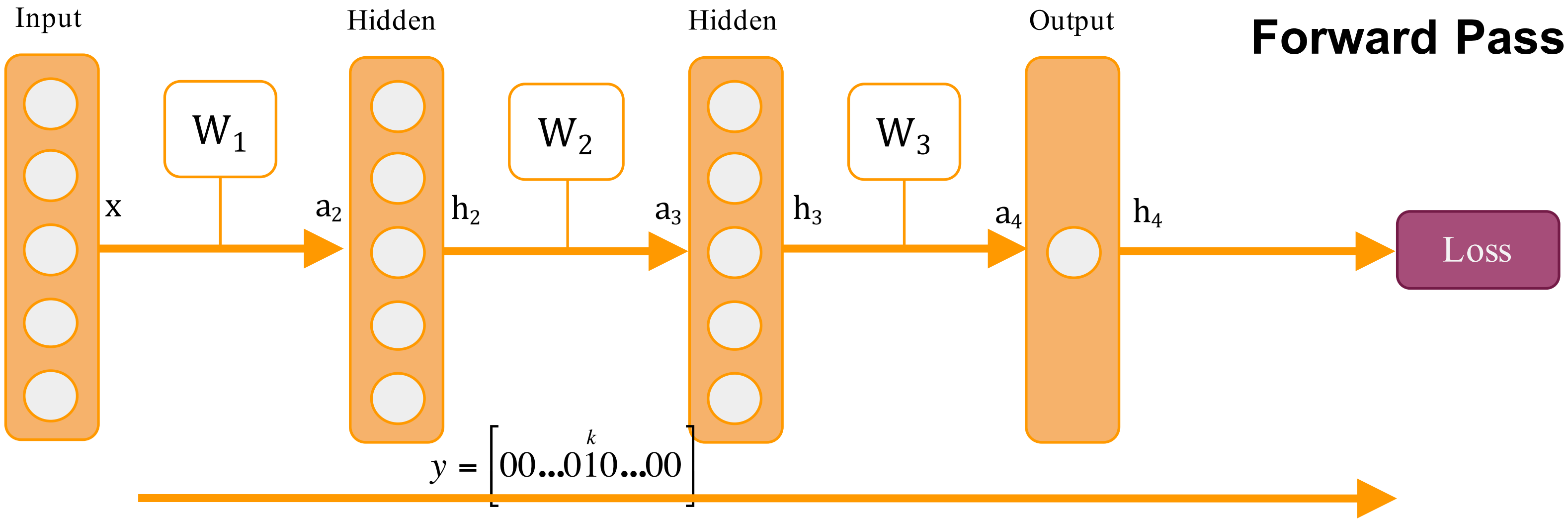
*For a single neuron with its linear and non-linear part*

$$\boldsymbol{h}_{k+1} = g(\boldsymbol{W}_k\boldsymbol{h}_k + \boldsymbol{b}_k) = g(\boldsymbol{a}_{k+1})$$

$$\frac{\partial \boldsymbol{h}_k}{\partial \boldsymbol{a}_k} = g'(\boldsymbol{a}_k)$$

$$\frac{\partial \boldsymbol{ha}_{k+1}}{\partial \boldsymbol{h}_k} = \boldsymbol{W_k}$$



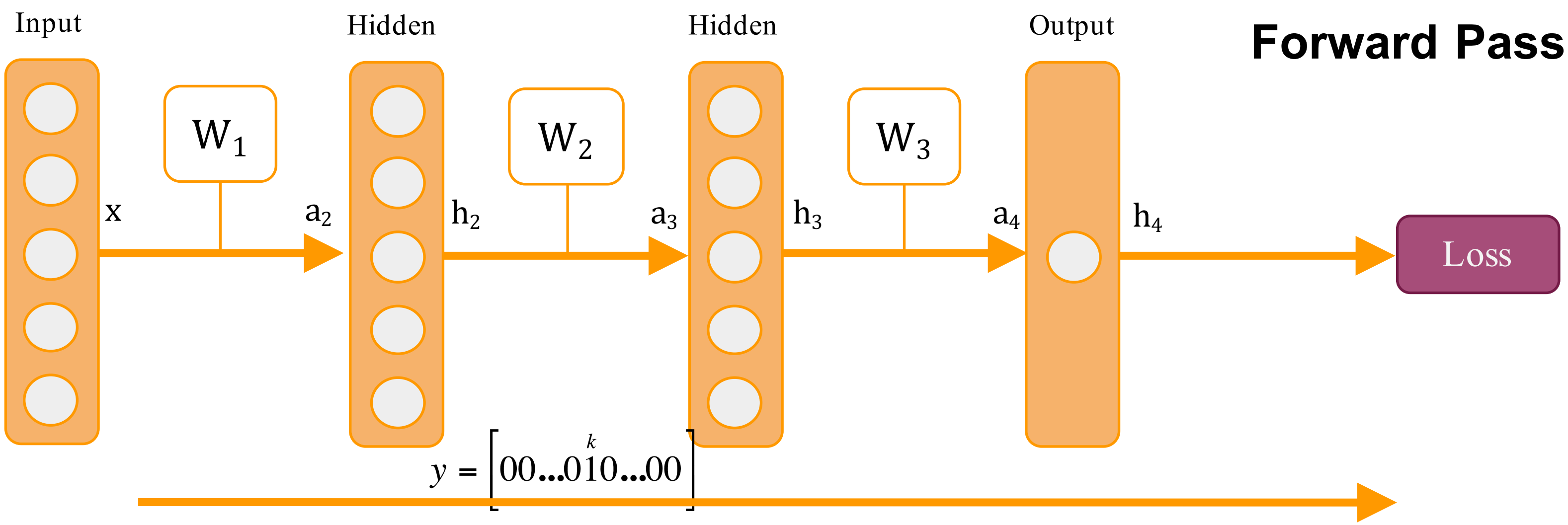



**Probability Class given an input (softmax)**

$$p(c_k = 1|\mathbf{x}) = \frac{\exp(a_k)}{\sum_c \exp(a_c)}$$

*Figure Credit: Kevin McGuiness*

## Forward Pass




**Probability Class given an input (softmax)**

$$p(c_k = 1|\mathbf{x}) = \frac{\exp(a_k)}{\sum_c \exp(a_c)}$$

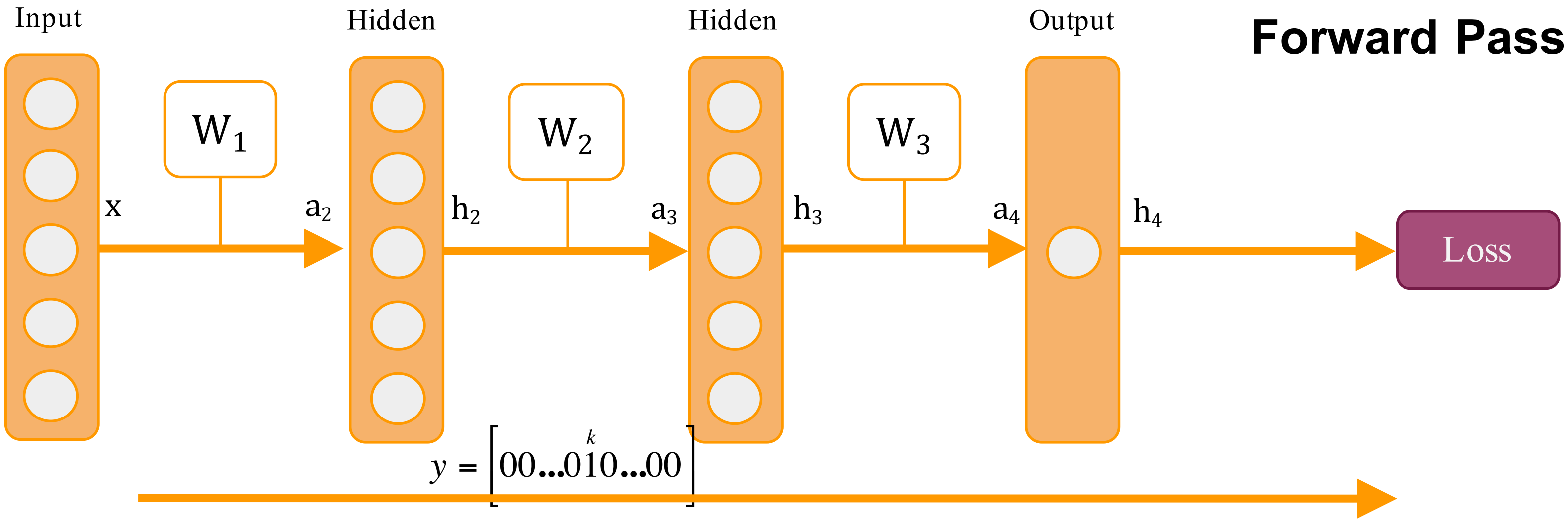
*Figure Credit: Kevin McGuiness*

**Loss function; e.g., negative log-likelihood (good for classification)**

$$L(\mathbf{x}, y; \mathbf{W}) = -\sum_j y_j \log(p(c_j|\mathbf{x}))$$

**Regularization term (L2 Norm) aka as weight decay**

$$L(\mathbf{x}, y; \mathbf{W}) = -\sum_j y_j \log(p(c_j|\mathbf{x})) + \frac{\lambda}{2}\|\mathbf{W}\|_2^2$$




**Probability Class given an input (softmax)**

$$p(c_k = 1|\mathbf{x}) = \frac{\exp(a_k)}{\sum_c \exp(a_c)}$$

*Figure Credit: Kevin McGuiness*

**Loss function; e.g., negative log-likelihood (good for classification)**

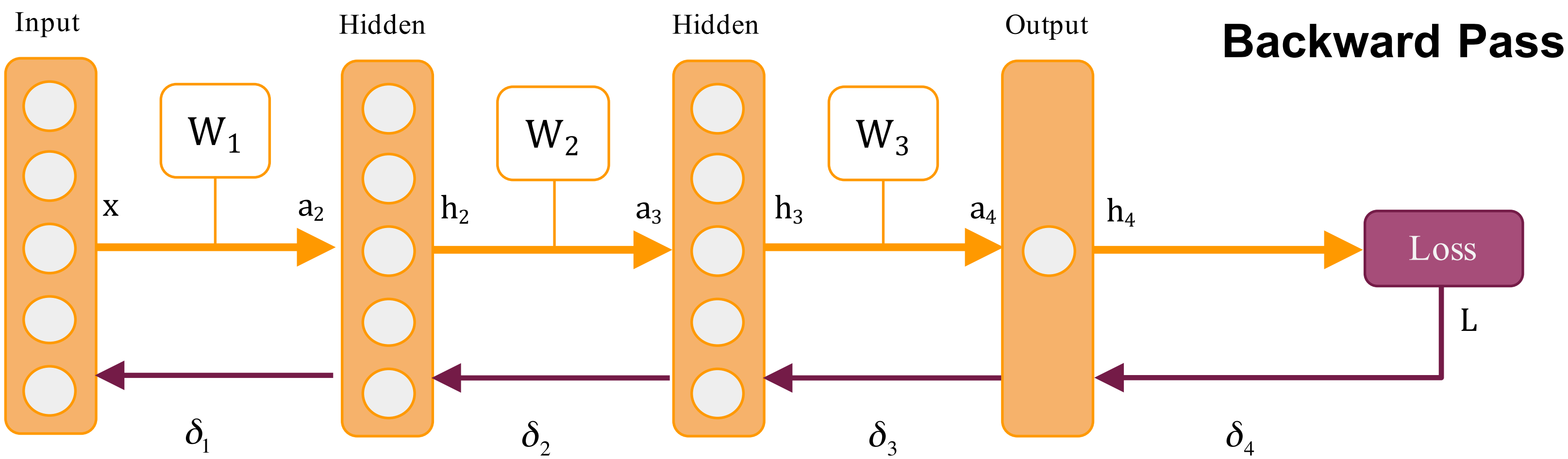
$$L(\mathbf{x}, y; \mathbf{W}) = -\sum_j y_j \log(p(c_j|\mathbf{x}))$$

**Regularization term (L2 Norm) aka as weight decay**

$$L(\mathbf{x}, y; \mathbf{W}) = -\sum_j y_j \log(p(c_j|\mathbf{x})) + \frac{\lambda}{2}\|\mathbf{W}\|_2^2$$

**Minimize the loss (plus some regularization term) w.r.t. Parameters over the whole training set.**

$$\mathbf{W}^* = argmin_\theta \sum_j L(\mathbf{x}^n, y^n; \mathbf{W})$$



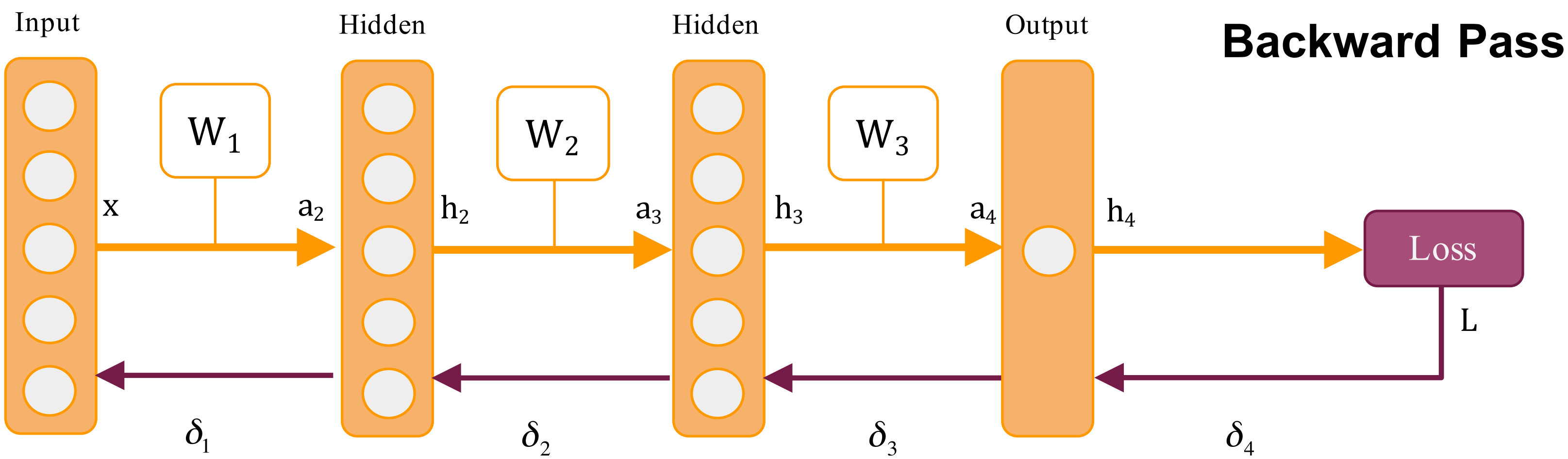

**1. Find the error in the top layer:**

$$\delta_K = \frac{\partial L}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K}\frac{\partial h_K}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K} \bullet g'(a_K)$$

*Figure Credit: Kevin McGuiness*




**1. Find the error in the top layer:**

$$\delta_K = \frac{\partial L}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K}\frac{\partial h_K}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K} \bullet g'(a_K)$$

*Figure Credit: Kevin McGuiness*

**2. Compute weight updates**

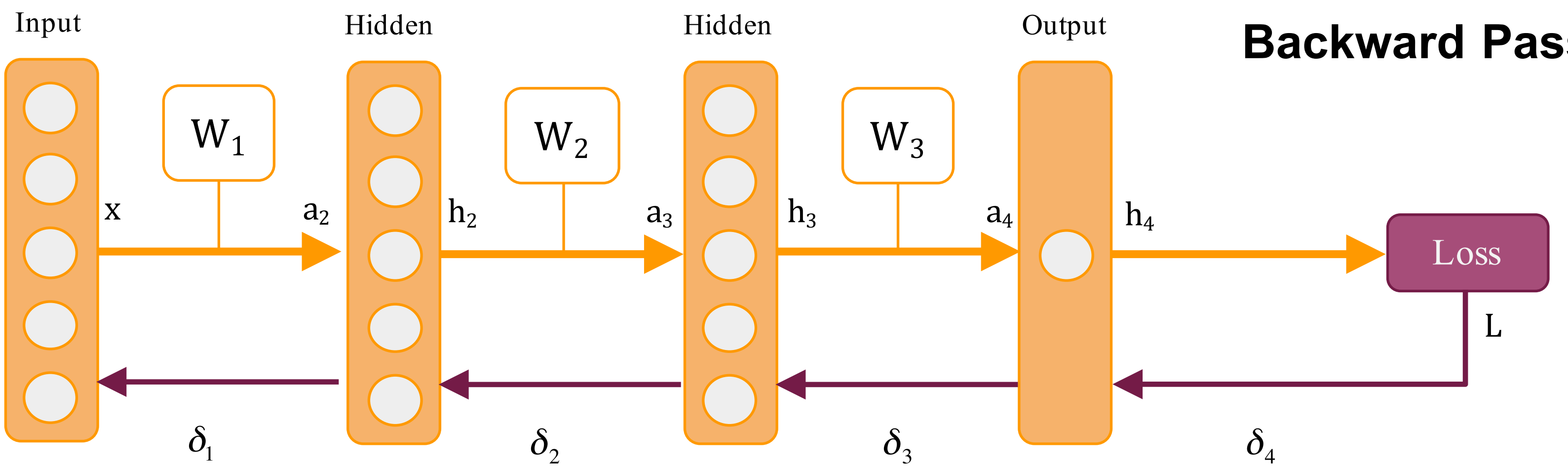
$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial a_{k+1}}\frac{\partial a_{k+1}}{\partial W_k}$$

$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial a_{k+1}} \bullet h_k$$

$$\frac{\partial L}{\partial W_k} = \delta_{k+1} \bullet h_k$$

**To simplify we don't consider the biass**

# Backward Pass




### 1. Find the error in the top layer:

$$\delta_K = \frac{\partial L}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K} \frac{\partial h_K}{\partial a_K}$$

$$\delta_K = \frac{\partial L}{\partial h_K} \bullet g'(a_K)$$

*Figure Credit: Kevin McGuiness*

### 2. Compute weight updates

$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial a_{k+1}} \frac{\partial a_{k+1}}{\partial W_k}$$

$$\frac{\partial L}{\partial W_k} = \frac{\partial L}{\partial a_{k+1}} \bullet h_k$$

$$\frac{\partial L}{\partial W_k} = \delta_{k+1} \bullet h_k$$

**To simplify we don't consider the biass**

### 3. Backpropagate error to layer below

$$\delta_k = \frac{\partial L}{\partial a_k}$$

$$\delta_k = \frac{\partial L}{\partial a_{k+1}} \frac{\partial a_{k+1}}{\partial h_k} \frac{\partial h_k}{\partial a_k}$$

$$\delta_k = W_k^T \frac{\partial L}{\partial a_{k+1}} \bullet g'(a_k)$$

$$\delta_k = W_k^T \delta_{k+1} \bullet g'(a_k)$$

# Issues on Backpropagation and Training

**Gradient Descent:** Move the parameter $\theta_j$ in small steps in the direction opposite sign of the derivative of the loss with respect j.

$$\theta^{(n)} = \theta^{(n-1)} - \alpha^{(n-1)} \cdot \nabla_\theta \mathcal{L}\big(y, f(x)\big) - \lambda\theta^{(n-1)}$$

**Weight Decay:** Penalizes large weights, distributes values among all the parameters

**Stochastic gradient descent (SGD):** estimate the gradient with one sample, or better, with a **minibatch** of examples.

**Momentum:** the movement direction of parameters averages the gradient estimation with previous ones.

Several strategies have been proposed to update the weights: **optimizers**

# Weight initialization

Need to pick a starting point for gradient descent: an initial set of weights

Zero is a very **bad idea**!

- Zero is a **critical point**
- Error signal will not propagate
- Gradients will be zero: no progress

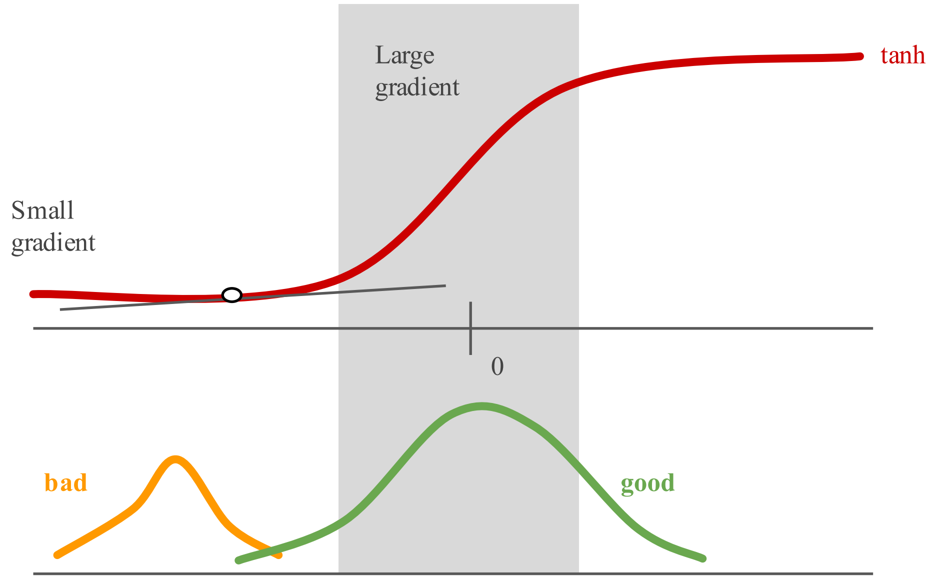
Constant value also bad idea:

- Need to break symmetry

Use **small random values:**

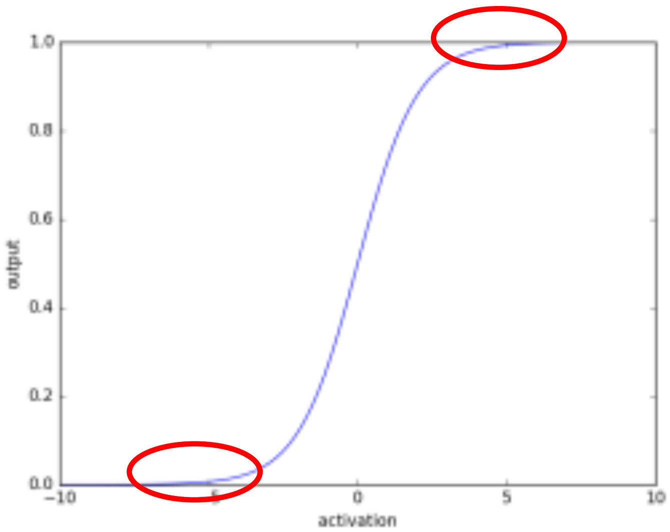
- E.g. zero mean Gaussian noise with constant variance

Ideally we want inputs to activation functions (e.g. sigmoid, tanh, ReLU) to be mostly **in the linear area** to allow larger gradients to propagate and converge faster.



# "Vanishing Gradients"

In the backward pass you might be in the flat part of the sigmoid (or any other activation function like tanh) so derivative tends to zero and your training loss will not go down



# Note on hyperparameters

So far we have lots of **hyperparameters** to choose:

1. Learning rate ($\alpha$)
2. Regularization constant ($\lambda$)
3. Number of epochs
4. Number of hidden layers
5. Nodes in each hidden layer
6. Weight initialization strategy
7. Loss function
8. Activation functions
9. …