

# Lecture 6: Artificial neural networks

Introduction to machine learning

---

Kevin Webster

Department of Mathematics  
Imperial College London

# Outline

Regression and classification review

Artificial neural network

Deep learning background

Neural network basics

Activation functions

Output layers

Stochastic gradient descent

Parameter estimation

Maximum likelihood estimation

Multiple target dimensions

Backpropagation

# Outline

Regression and classification review

Artificial neural network

Deep learning background

Neural network basics

Activation functions

Output layers

Stochastic gradient descent

Parameter estimation

Maximum likelihood estimation

Multiple target dimensions

Backpropagation

# Regression and classification: setting

We once again review the setting for regression and classification.

We are given a dataset consisting of  $N$  input points

$$\mathbf{x} = (x_1, x_2, \dots, x_N), \quad x_i \in \mathbb{R}^d \quad \forall i$$

and  $N$  corresponding output values

$$\mathbf{y} = (y_1, y_2, \dots, y_N)$$

In the case of regression, the outputs take real values  $y_i \in \mathbb{R} \quad \forall i$ .

In the case of classification, the outputs take one of finitely many values,  $y_i \in \mathcal{C} \quad \forall i$ , where  $\mathcal{C}$  is a finite set of classes.

In binary classification, typically we assume each  $y_i \in \{-1, +1\}$  or  $y_i \in \{0, 1\}$ .

# Regression and classification models: review

- We have studied models for both regression and classification that consist of linear combinations of basis functions
- The linear regression problem can be solved in closed form using the normal equation
- Logistic regression models need to be fitted using gradient-based methods
- In both cases, we need to carefully choose basis functions for the particular dataset
- Both linear regression and logistic regression are parametric models, with the number of parameters determined by the choice of basis functions
- Nonparametric methods also exist, where the capacity of the model grows with the size of the dataset (e.g. support vector machine (SVM), Gaussian processes)

# Outline

Regression and classification review

Artificial neural network

Deep learning background

Neural network basics

Activation functions

Output layers

Stochastic gradient descent

Parameter estimation

Maximum likelihood estimation

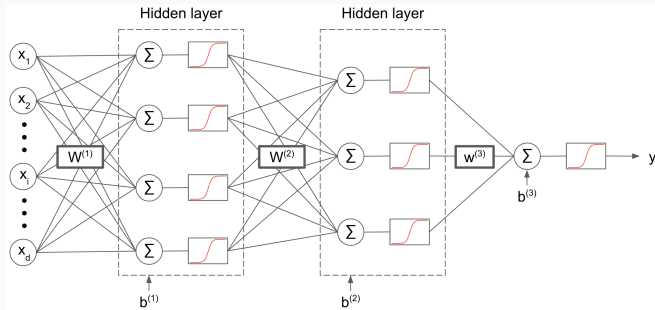
Multiple target dimensions

Backpropagation

# Artificial neural network

- Artificial neural networks (ANNs) can be seen as an alternative approach where the number of basis functions are fixed, but are themselves parameterised
- The parameters are adapted during the training phase
- A common type of neural network is called a **feedforward network** or a **multilayer perceptron** (MLP)
- The MLP is constructed by stacking hidden layers of logistic regression functions, terminating in an output prediction layer
- The term *multilayer perceptron* is a misleading name, since it uses continuous nonlinearities like the sigmoid function (as in logistic regression) instead of a discontinuous step function activation (as in the perceptron)
- The field of ANNs has evolved into what is now known as **deep learning**

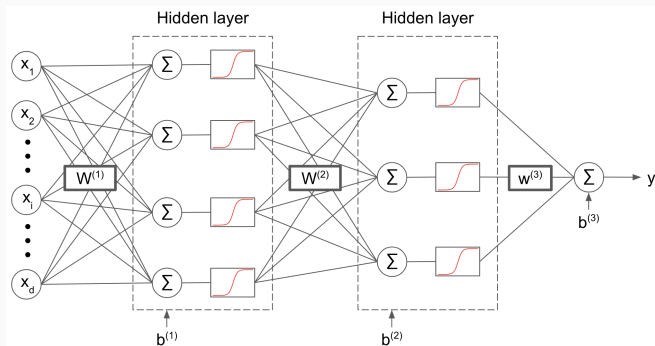
# Feedforward network / MLP: binary classification



- The MLP constructed above consists of two hidden layers of neurons, each of which has the form of a logistic regression model
- The output  $y$  can be thought of as a logistic regression model where the basis functions are determined by the neurons in the last hidden layer



# Feedforward network / MLP: regression

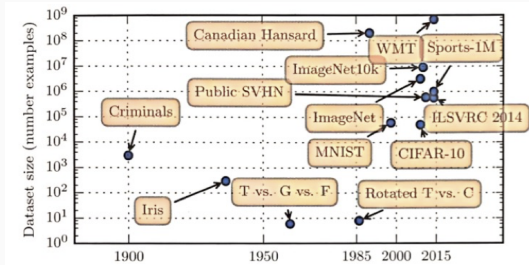


- An MLP can also be used to solve regression problems. The hidden layers are constructed in the same way
- The only difference is in the output layer, which can be thought of as a linear combination of basis functions given by the neurons in the last hidden layer

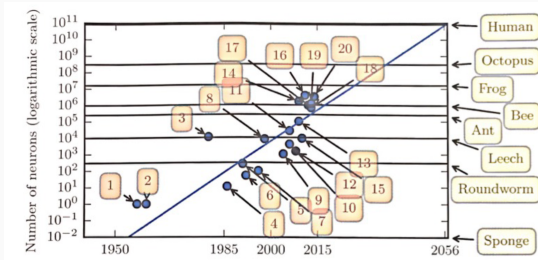
# Deep learning

- The 'deep' in deep learning refers to the number of hidden layers in the network
- In general, greater depth in networks allow for a richer class of approximating functions
- Intuition is that deep networks allow the model to build hierarchies of concepts
- Concepts are built on top of each other in layers
- This reduces the need for hand-engineered features (basis functions)
- Deep learning can be seen as part of *representation learning*, which aims to discover the best representations or features of the data

# Historical trends in Deep Learning



Increasing datasets over time



Increasing network size over time

1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart et al., 1986)
5. RNN for speech recognition (Robinson and Fallside, 1991)
6. MLP for speech recognition (Bengio et al., 1991)
7. Mean field SBN (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep Belief Network (Hinton et al., 2006)
11. GPU ConvNet (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009)
13. GPU DBN (Raina et al., 2009)
14. Unsupervised ConvNet (Jarrett et al., 2009)
15. GPU MLP (Ciresan et al., 2010)
16. OMP-1 network (Coates and NG, 2011)
17. Distributed autoencoder (Le et al., 2012)
18. Multi-GPU ConvNet (Krizhevsky et al., 2012)
19. COTS HPC unsupervised ConvNet (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014)

(Images source: Deep Learning book)

# Deep learning frameworks

Caffe

DL4J  
DEEPLEARNING4J

The Microsoft Cognitive Toolkit

A free, easy-to-use, open source, commercial grade toolkit that trains deep learning algorithms to learn like the human brain.

mxnet



TensorFlow

PYTORCH



BigDL

torch



Chainer



Caffe2

theano

- There is a great deal of flexibility when it comes to constructing deep learning models
- We will look at the functional form of neural networks, including choice of nonlinear activation functions
- The parameters of the network are tuned by optimising the data likelihood
- The representational power of neural networks comes at the price of a non-convex loss function, and a potentially large set of parameters
- Neural networks are trained in a similar manner to logistic regression, using gradient-based methods
- We will look at the process of tuning network parameters, especially the *backpropagation* algorithm

# Outline

Regression and classification review

Artificial neural network

Deep learning background

Neural network basics

Activation functions

Output layers

Stochastic gradient descent

Parameter estimation

Maximum likelihood estimation

Multiple target dimensions

Backpropagation

# Feedforward network / multilayer perceptron

Neural network with  $L$  layers  $z_1, \dots, z_L$ , where  $z_i \in \mathbb{R}^{n_i}$ . Input  $\mathbf{x} = z_1$  and output  $\mathbf{y} = z_L$ . For  $i = 1, \dots, L - 1$ , we define

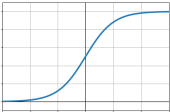
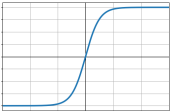
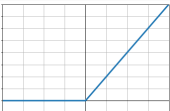
$$a_{i+1} = W^{(i)}z_i + b^{(i)}, \quad (\text{pre-activation})$$

where  $W^{(i)} \in \mathbb{R}^{n_{i+1} \times n_i}$  and  $b^{(i)} \in \mathbb{R}^{n_{i+1}}$ .

$$z_{i+1} = \sigma(a_{i+1}), \quad (\text{post-activation})$$

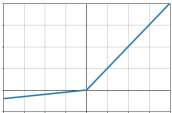
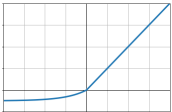
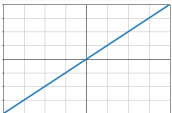
where  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  is an **activation function** that is applied element-wise.

# Activation functions

Activation function	Plot	Equation
sigmoid		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
tanh		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
ReLU		$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$



# Activation functions

Activation function	Plot	Equation
leaky ReLU		$f(x) = \begin{cases} \epsilon x, & x < 0 \\ x, & x \geq 0 \end{cases}$
ELU		$f(x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$
identity		$f(x) = x$

Frequently used output activations / layers:

- **Sigmoid output layer** is useful to predict probabilities as the range is in  $(0, 1)$ .
- **Softplus output layer** is useful to predict e.g. Gaussian variance parameters, as the range is  $(0, \infty)$ :

$$f(x) = \ln(1 + e^x)$$

- **Softmax output layer** is often used to predict parameters of a categorical distribution:

$$P(\text{Category } j) = \frac{e^{a_j}}{\sum_{k=1}^M e^{a_k}},$$

where  $a_k$  ( $k \in \{1, \dots, M\}$ ) are the pre-activations.

Recall that logistic regression can be viewed as a simple neural network.  
The model prediction is given by

$$y = \sigma(\mathbf{w}^T \mathbf{x} + b),$$

where the inputs  $\mathbf{x} \in \mathbb{R}^d$ , the weights  $\mathbf{w} \in \mathbb{R}^d$ , bias  $b \in \mathbb{R}$ ,  $\sigma$  is the sigmoid function and the output  $y \in (0, 1)$ .

# Loss functions

Loss functions used in deep learning are often attempting to maximise log-likelihood. Typical per-example loss functions are:

- **Cross-entropy loss.** Frequently used in classification, with  $M$  mutually exclusive classes:

$$H_{\hat{\mathbf{y}}}(\mathbf{y}) := -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M \hat{y}_i^{(j)} \log(y_i^{(j)})$$

- **Mean squared error.** Frequently used in regression tasks, e.g. in predicting a  $t$ -dimensional output:

$$S := \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^t (y_i^{(j)} - \hat{y}_i^{(j)})^2$$

Other loss functions are used for particular network frameworks such as the VAE or GAN (not covered in this course).

# Regularisation

As mentioned earlier, regularisation is important to avoid overfitting. There are several approaches to regularisation in deep learning:

- Model complexity
- Weight decay
- Patience/early stopping
- Dropout
- Weight sharing
- Ensemble predictions
- Dataset augmentation
- Noise robustness
- Multitask learning
- Adversarial training

# Stochastic gradient descent

Most Deep Learning models are trained with (a variant of) stochastic gradient descent.

The cost function usually decomposes as a sum over the training examples in the training set:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N L(\mathbf{x}_i, y_i, \theta),$$

where  $\theta$  are the parameters of the model,  $(\mathbf{x}_i, y_i)_{i=1}^N$  is the (labelled) training set, and  $L$  is the per-example loss.

# Stochastic gradient descent

**Gradient descent** involves computing

$$\nabla_{\theta} J(\theta) = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} L(\mathbf{x}_i, y_i, \theta),$$

which can be prohibitively expensive for large datasets.

**Stochastic gradient descent** approximates the expected gradient by sampling a **minibatch** of data examples  $(\mathbf{x}_{n_i}, y_{n_i})_{i=1}^{N'}$ ,  $N' \ll N$ .

The estimated gradient is then

$$\mathbf{g} = \frac{1}{N'} \sum_{i=1}^{N'} \nabla_{\theta} L(\mathbf{x}_{n_i}, y_{n_i}, \theta),$$

# Stochastic gradient descent

Stochastic gradient descent then uses the estimated gradient to adjust the parameters:

$$\theta \leftarrow \theta - \epsilon \mathbf{g},$$

where  $\epsilon$  is the learning rate.

- SGD makes efficient use of the dataset when training deep learning models
- The cost per SGD update does not depend on the training size  $N$
- Provides a scalable way of training nonlinear models on large datasets



# Outline

Regression and classification review

Artificial neural network

Deep learning background

Neural network basics

Activation functions

Output layers

Stochastic gradient descent

Parameter estimation

Maximum likelihood estimation

Multiple target dimensions

Backpropagation

# Maximum likelihood

Just as with the other parametric models we have considered, we need to estimate the parameters of the neural network. The parameters are the weight matrices  $W^{(i)} \in \mathbb{R}^{n_{i+1} \times n_i}$  and biases  $b^{(i)} \in \mathbb{R}^{n_{i+1}}$  for  $i = 1, \dots, L - 1$ .

Neural networks are often trained using maximum likelihood. Our aim is to maximise the likelihood function over the whole dataset:

$$\begin{aligned}\theta_{ML} &= \arg \max_{\theta} \mathcal{L}(\theta | \mathbf{x}, \mathbf{y}) \\ &= \arg \max_{\theta} \prod_{i=1}^N p(y_i | x_i, \theta)\end{aligned}$$

where we have assumed the data to be independent and identically distributed (i.i.d.).

# Maximum likelihood: regression

Consider first the regression setting. We are given a dataset consisting of  $N$  input points

$$\mathbf{x} = (x_1, x_2, \dots, x_N), \quad x_i \in \mathbb{R}^d \quad \forall i$$

and  $N$  corresponding output values

$$\mathbf{y} = (y_1, y_2, \dots, y_N), \quad y_i \in \mathbb{R} \quad \forall i$$

We wish to find a parametric function  $f(\mathbf{x}, \boldsymbol{\theta})$  that models the relationship between  $\mathbf{x}$  and  $\mathbf{y}$ .

We derive the loss function completely analogously to the linear regression case.

# Maximum likelihood: regression

- We again assume that the data is generated according to the following model

$$y = f(x, \theta) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2),$$

where  $\sigma^2$  is a hyperparameter

- Our assumption is that the data is contaminated by Gaussian noise, just as in the linear regression model
- However, now the function  $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto \mathbb{R}$  is our neural network, where  $p$  is the total number of model parameters (weights and biases)
- Note that in this case, the final output layer of the network consists of a single neuron with no nonlinear activation function
- The probability  $p(y_i | x_i, \theta)$  is given by the Gaussian pdf with mean  $f(x_i, \theta)$  and variance  $\sigma^2$  for each data example  $(x_i, y_i)$ .

# Maximum likelihood: regression

As usual, it is convenient to work with the negative of the log-likelihood function, as the arg min of the negative log-likelihood is the same as the arg max of the likelihood function.

$$\begin{aligned}\theta_{ML} &= \arg \min_{\theta} -\log \mathcal{L}(\theta|\mathbf{x}, \mathbf{y}) \\ &= \arg \min_{\theta} \sum_{i=1}^N -\log p(y_i|x_i, \theta) \\ &= \arg \min_{\theta} \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - f(x_i, \theta))^2 + \text{const.}\end{aligned}$$

where the constant is independent of the model parameters  $\theta$ . We have derived the **loss function** (MSE) to be optimised:

$$E(\theta) = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i, \theta))^2 = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (\hat{y}_i := f(x_i, \theta))$$

## Maximum likelihood: binary classification

We treat the classification in a similar way. Our dataset again consists of  $N$  input points

$$\mathbf{x} = (x_1, x_2, \dots, x_N), \quad x_i \in \mathbb{R}^d \quad \forall i$$

and  $N$  corresponding output values

$$\mathbf{y} = (y_1, y_2, \dots, y_N), \quad y_i \in \{0, 1\} \quad \forall i$$

where  $y_i = 1$  if  $x_i \in \mathcal{C}_1$  and vice versa.

In the above setting, our neural network output layer consists of a single neuron with a logistic sigmoid activation function. This ensures that the network output is interpretable as a probability:  $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto (0, 1)$ .

As in the case of logistic regression, we use the network function  $f(\mathbf{x}, \boldsymbol{\theta})$  to model the probability  $p(\mathcal{C}_1|\mathbf{x})$ , and  $p(\mathcal{C}_2|\mathbf{x}) = 1 - p(\mathcal{C}_1|\mathbf{x})$ .

# Maximum likelihood: binary classification

Again, we seek the parameters that minimise the negative log-likelihood:

$$\begin{aligned}\theta_{ML} &= \arg \min_{\theta} -\log \mathcal{L}(\theta|\mathbf{x}, \mathbf{y}) \\ &= \arg \min_{\theta} \sum_{i=1}^N -\log p(y_i|x_i, \theta) \\ &= \arg \min_{\theta} \sum_{i=1}^N -y_i \log f(x_i, \theta) - (1 - y_i) \log(1 - f(x_i, \theta))\end{aligned}$$

We have derived the **loss function** (binary cross entropy) to be optimised:

$$E(\theta) = \frac{1}{N} \sum_{i=1}^N -y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i)$$

where we denote the model prediction  $\hat{y}_i := f(x_i, \theta) = p(\mathcal{C}_1|x_i)$ .

## Multiple target dimensions: regression

The previous derivations can be extended in a straightforward way to a multi-dimensional target variable.

In the case of regression, we consider inputs  $\mathbf{x} = (x_1, x_2, \dots, x_N)$  with  $x_i \in \mathbb{R}^d$  and outputs  $\mathbf{y} = (y_1, y_2, \dots, y_N)$  with  $y_i \in \mathbb{R}^t$ .

We denote the  $j$ -th dimension of  $y_i$  as  $y_i^{(j)}$ . Our network is now a mapping  $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto \mathbb{R}^t$  with  $t$  (linear) output neurons.



## Multiple target dimensions: regression

We make the similar modelling assumption that the data is generated according to the following model

$$y = f(x, \theta) + \epsilon, \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I}_t)$$

Similar arguments follow to show that the maximum likelihood solution is given by minimising the following MSE loss function:

$$E(\theta) = \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^t (y_i^{(j)} - \hat{y}_i^{(j)})^2$$

where we denote the model prediction  $\hat{y}_i = f(x_i, \theta) \in \mathbb{R}^t$ .

## Multiple target dimensions: classification

For classifications, we may consider two different settings for multiple target dimensions.

In the simpler case, we have inputs  $\mathbf{x} = (x_1, x_2, \dots, x_N)$  with  $x_i \in \mathbb{R}^d$  and  $M$  binary output labels  $\mathbf{y} = (y_1, y_2, \dots, y_N)$  with  $y_i \in \{0, 1\}$ . In this case we have  $M$  separate binary classifications to perform.

In this setting, our neural network output layer consists of  $M$  neurons with a logistic sigmoid activation function, and  $f : \mathbb{R}^d \times \mathbb{R}^p \mapsto (0, 1)^M$ .

With similar arguments to before, we derive the loss function

$$E(\theta) = \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M -y_i^{(j)} \log \hat{y}_i^{(j)} - (1 - y_i^{(j)}) \log(1 - \hat{y}_i^{(j)})$$

## Multiple target dimensions: classification

For multiclass classification, we have inputs  $\mathbf{x} = (x_1, x_2, \dots, x_N)$  with  $x_i \in \mathbb{R}^d$  and outputs  $\mathbf{y} = (y_1, y_2, \dots, y_N)$  with each  $y_i \in \{\mathcal{C}_1, \dots, \mathcal{C}_M\}$ .

In this case, we typically encode the data so that each class  $\mathcal{C}_j$  is identified with a **one-hot vector** in  $\mathbb{R}^M$ .

For an input  $x_i$  belonging to class  $\mathcal{C}_j$ , the one-hot vector  $y_i \in \mathbb{R}^M$  is a binary vector with a 1 in the  $j$ th position and zeros everywhere else:

$$y_i = [0, \ 0, \ \dots, \ 0, \ 1, \ 0, \ \dots, \ 0]$$

$\uparrow$   
 $j$

## Multiple target dimensions: classification

In this case, the network model prediction needs to predict a categorical distribution over  $M$  classes. This can be achieved using a **softmax** output layer:

$$a_L = W^{(L-1)} z_{L-1} + b^{(L-1)}, \quad (\text{pre-activations / logits})$$

$$z_L^{(j)} = \frac{e^{a_L^{(j)}}}{\sum_{k=1}^M e^{a_L^{(k)}}} \quad (\text{softmax output})$$

In the above,  $W^{(L-1)} \in \mathbb{R}^{M \times n_{L-1}}$ ,  $b^{(L-1)} \in \mathbb{R}^M$  and  $a_L, z_L \in \mathbb{R}^M$ .

Note that  $z_L^{(j)} \geq 0$  and  $\sum_{j=1}^M z_L^{(j)} = 1$ .

The  $j$ -th coordinate of the output layer  $z_L = f(x, \theta)$  gives the model probability  $p(\mathcal{C}_j | x)$ .

## Multiple target dimensions: classification

As usual, we denote the model prediction  $\hat{y}_i = f(x_i, \theta)$ . The likelihood function in this case is given by

$$\begin{aligned}\mathcal{L}(\theta|\mathbf{x}, \mathbf{y}) &= \prod_{i=1}^N p(y_i|x_i, \theta) \\ &= \prod_{i=1}^N \prod_{j=1}^M (\hat{y}_i^{(j)})^{y_i^{(j)}}\end{aligned}$$

The parameters that minimise the negative log-likelihood are given by:

$$\theta_{ML} = \arg \min_{\theta} \sum_{i=1}^N \sum_{j=1}^M -y_i^{(j)} \log \hat{y}_i^{(j)}$$

and we have the following cross entropy loss function:

$$E(\theta) = \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M -y_i^{(j)} \log \hat{y}_i^{(j)}$$

# Outline

Regression and classification review

Artificial neural network

Deep learning background

Neural network basics

Activation functions

Output layers

Stochastic gradient descent

Parameter estimation

Maximum likelihood estimation

Multiple target dimensions

Backpropagation

# Backpropagation

Neural network training can be viewed iterating the following two separate stages:

1. Computation of the derivatives of the loss function with respect to the model parameters
2. Use the computed gradient to update the parameters

The first of these stages is done through applying the chain rule of differentiation. The process of computing these derivatives in an efficient manner is known as **backpropagation**.

In the remainder of this lecture, we will derive the backpropagation algorithm for finding the loss function derivatives for a general feedforward neural network architecture.

## Backpropagation: preactivations $a_j$ and activations $z_j$

- We view the network as a collection of neurons units
- Each unit has an *preactivation* value  $a_j$ , computed as

$$a_j = \sum_{k \in \uparrow j} w_{jk} z_k + b_j \quad (1)$$

where  $\uparrow j$  denotes the set of indices of neurons upstream from unit  $j$ , i.e. that feed directly into unit  $j$

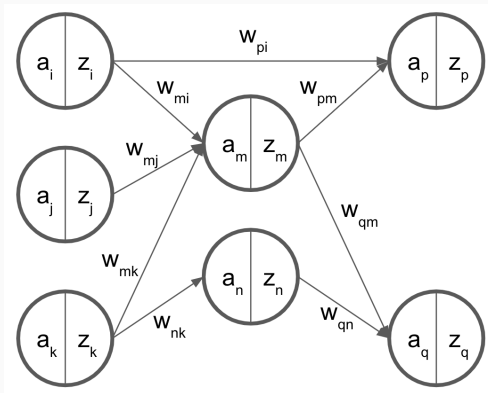
- Each neuron also has an *activation* value  $z_j$  given by

$$z_j = \sigma(a_j) \quad (2)$$

where  $\sigma(\cdot)$  is an activation function



# Backpropagation: directed acyclic graph



Note the variables  $z_j$  could be used for input or output nodes to the network. The loss function can be seen as a function of one or more node activations and data labels.

# Backpropagation: per-example loss function

- We note that due to the i.i.d. assumption, the loss is a sum of terms for each data example, e.g.

$$E(\theta) = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i, \theta))^2$$

- therefore we will consider the loss derivatives computed on a single data example, in this example:

$$E_i(\theta) = (y_i - f(x_i, \theta))^2$$

- Recall that  $\theta$  denotes the network parameters
- We want to compute derivatives of  $E_i(\theta)$  with respect to the weights  $w_{jk}$  and biases  $b_j$  of the network
- The derivatives of  $E(\theta)$  are computed by averaging  $E_i(\theta)$  over the minibatch

# Error backpropagation

We first compute the *forward pass*, in which the input data is fed to the network and the preactivations  $a_j$  and activations  $z_j$  are computed in sequence corresponding to (1) and (2), for the current parameters  $\theta$ .

Consider the derivative of  $E_i$  with respect to  $w_{jk}$  and  $b_j$ . We have:

$$\frac{\partial E_i}{\partial w_{jk}} = \frac{\partial E_i}{\partial a_j} \frac{\partial a_j}{\partial w_{jk}} = \frac{\partial E_i}{\partial a_j} z_k$$

and

$$\frac{\partial E_i}{\partial b_j} = \frac{\partial E_i}{\partial a_j} \frac{\partial a_j}{\partial b_j} = \frac{\partial E_i}{\partial a_j}$$

# Error backpropagation

Introducing the notation  $\delta_j := \frac{\partial E_i}{\partial a_j}$ , called the **error**. We then write

$$\frac{\partial E_i}{\partial w_{jk}} = \delta_j z_k, \quad \frac{\partial E_i}{\partial b_j} = \delta_j \quad (3)$$

We therefore need to compute the quantity  $\delta_j$  for each hidden and output unit in the network. Again using the chain rule, we have

$$\begin{aligned} \delta_j \equiv \frac{\partial E_i}{\partial a_j} &= \sum_{k \in \downarrow j} \frac{\partial E_i}{\partial a_k} \frac{\partial a_k}{\partial a_j} \\ &= \sum_{k \in \downarrow j} \delta_k \frac{\partial a_k}{\partial a_j} \end{aligned}$$

where  $\downarrow j$  denotes the set of indices of neurons downstream from unit  $j$ ; that is, the neurons that unit  $j$  feeds into.

# Error backpropagation

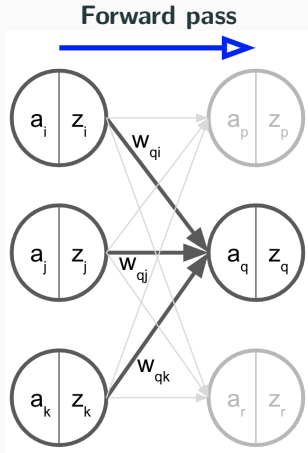
Combining equations (1) and (2) we see that

$$\begin{aligned}a_k &= \sum_{j \in \uparrow k} w_{kj} \sigma(a_j) + b_k \\ \frac{\partial a_k}{\partial a_j} &= w_{kj} \sigma'(a_j)\end{aligned}$$

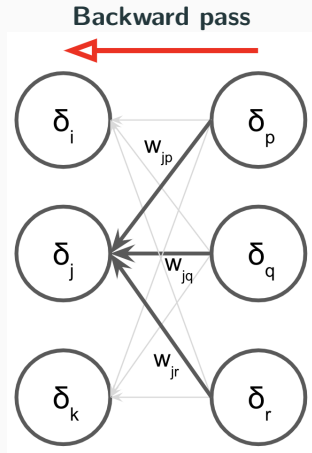
where  $\sigma'$  is the derivative of the activation function. So we have

$$\begin{aligned}\delta_j \equiv \frac{\partial E_i}{\partial a_j} &= \sum_{k \in \downarrow j} \delta_k \frac{\partial a_k}{\partial a_j} \\ &= \sigma'(a_j) \sum_{k \in \downarrow j} w_{kj} \delta_k\end{aligned}$$

# Forward and backward passes



$$a_q = \sum_{j \in \uparrow k} w_{qj} \sigma(a_j) + b_q$$



$$\delta_j = \sigma'(a_j) \sum_{k \in \downarrow j} w_{kj} \delta_k$$

# Backpropagation algorithm

$$\text{Forward pass: } a_k = \sum_{j \in \uparrow k} w_{kj} \sigma(a_j) + b_k$$

$$\text{Backward pass: } \delta_j = \sigma'(a_j) \sum_{k \in \downarrow j} w_{kj} \delta_k$$

$$\text{Derivatives: } \frac{\partial E_i}{\partial w_{jk}} = \delta_j z_k, \quad \frac{\partial E_i}{\partial b_j} = \delta_j$$

1. First, propagate the signal forwards by passing an input vector  $x_i$  to the network and computing all pre-activations and activations
2. Evaluate  $\delta_k = \frac{\partial E_i}{\partial a_k}$  for the output neuron pre-activations
3. Backpropagate the errors  $\delta$  to compute  $\delta_j$  for each hidden unit
4. Compute the derivatives using the errors  $\delta_j$