MATH60026/MATH70026
Methods for Data Science
Lecture 5

based on materials by Kevin Webster

Barbara Bravi, Imperial College London

Department of Mathematics, Academic year 2024-2025

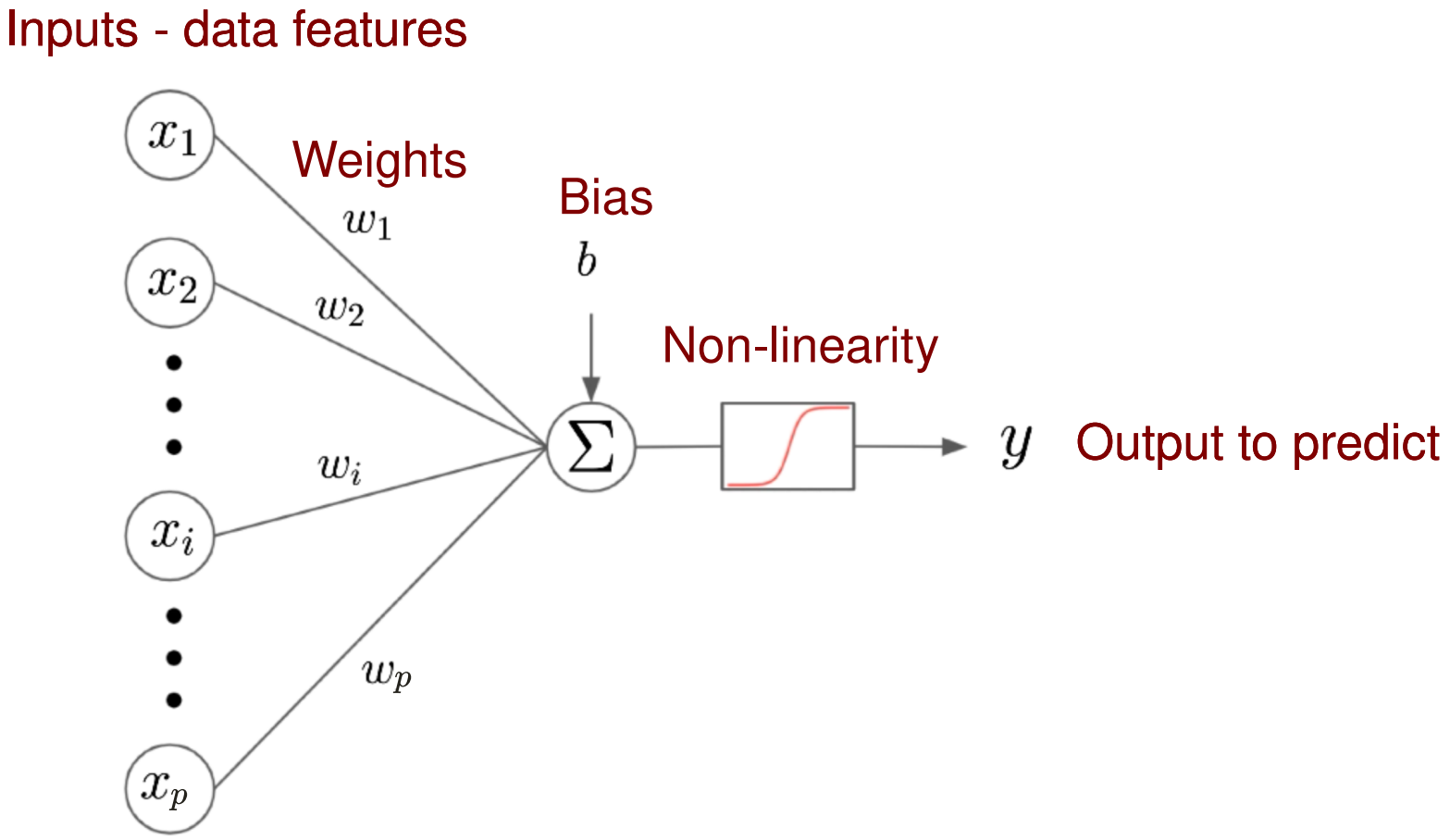
IMPERIAL

Artificial Neuron

Neural networks: ML model archiectures built by arranging together elementary computing units, *artificial neurons*

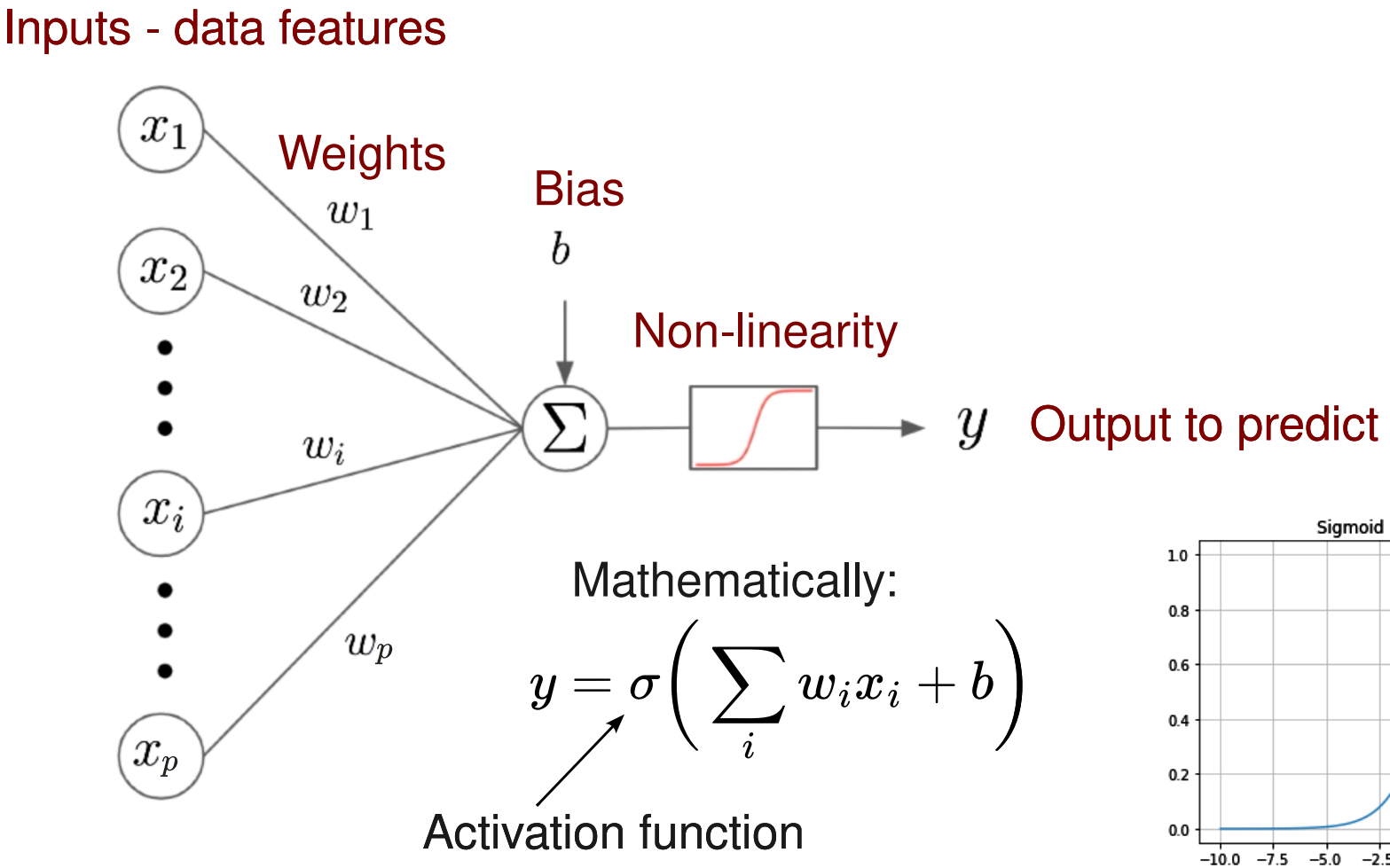
Artificial Neuron

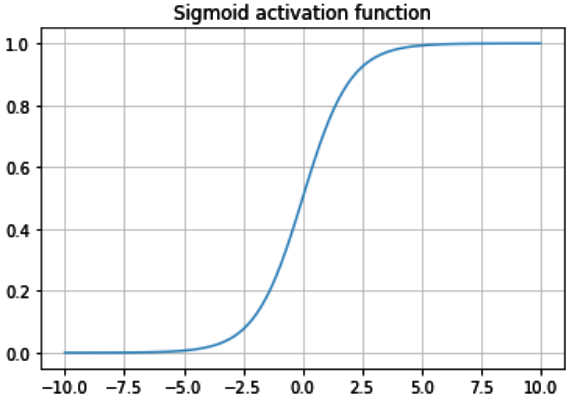
Neural networks: ML model archiectures built by arranging together elementary computing units, *artificial neurons*



Artificial Neuron

Neural networks: ML model archiectures built by arranging together elementary computing units, *artificial neurons*

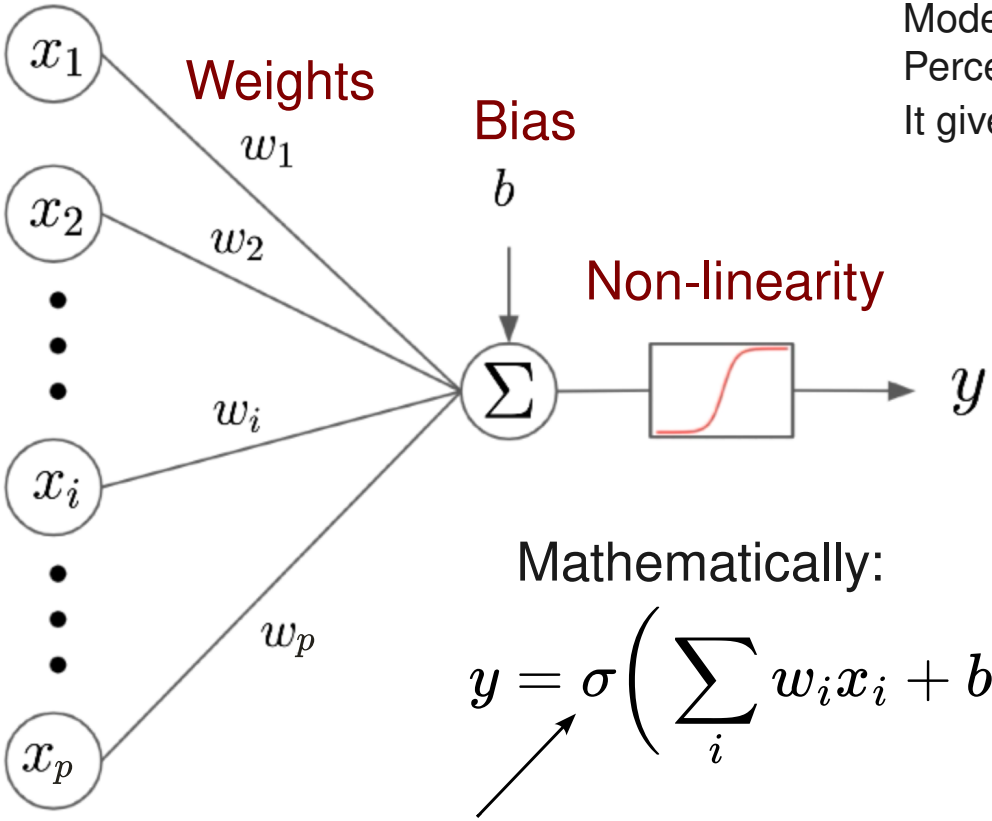




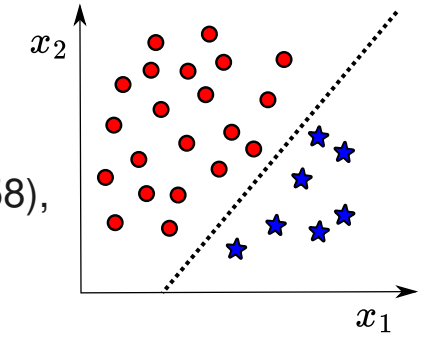
Artificial Neuron

Neural networks: ML model archiectures built by arranging together elementary computing units, *artificial neurons*

Inputs - data features



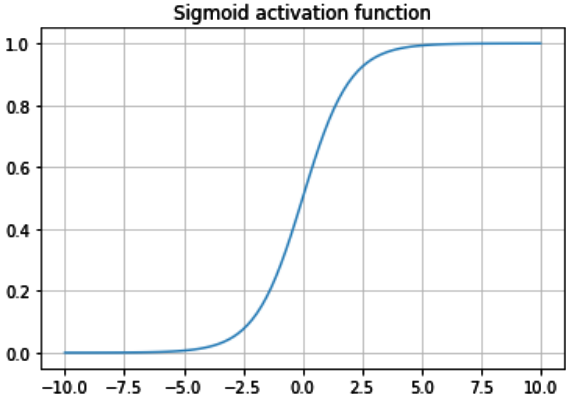
Model introduced as the:
Perceptron (Rosenblatt 1958),
It gives a *linear* classifier.



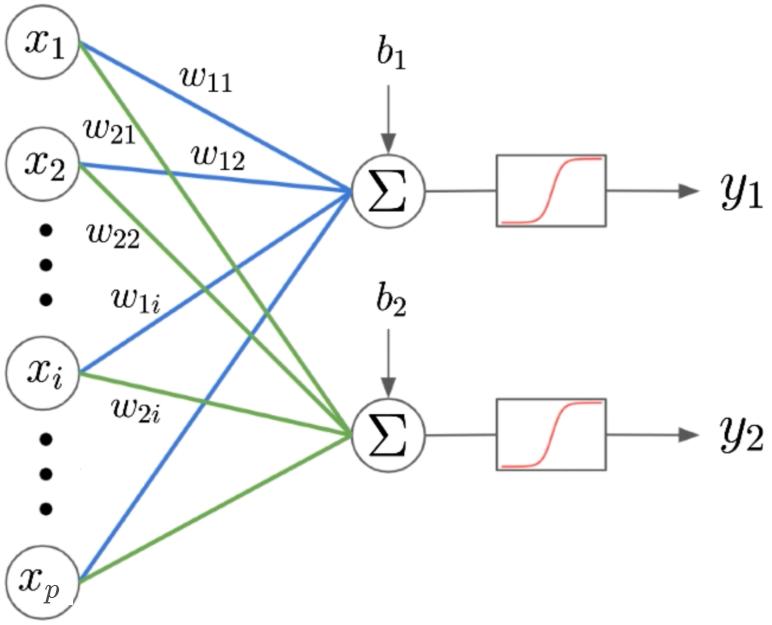
Mathematically:

$$y = \sigma\left(\sum_i w_i x_i + b\right)$$

Activation function



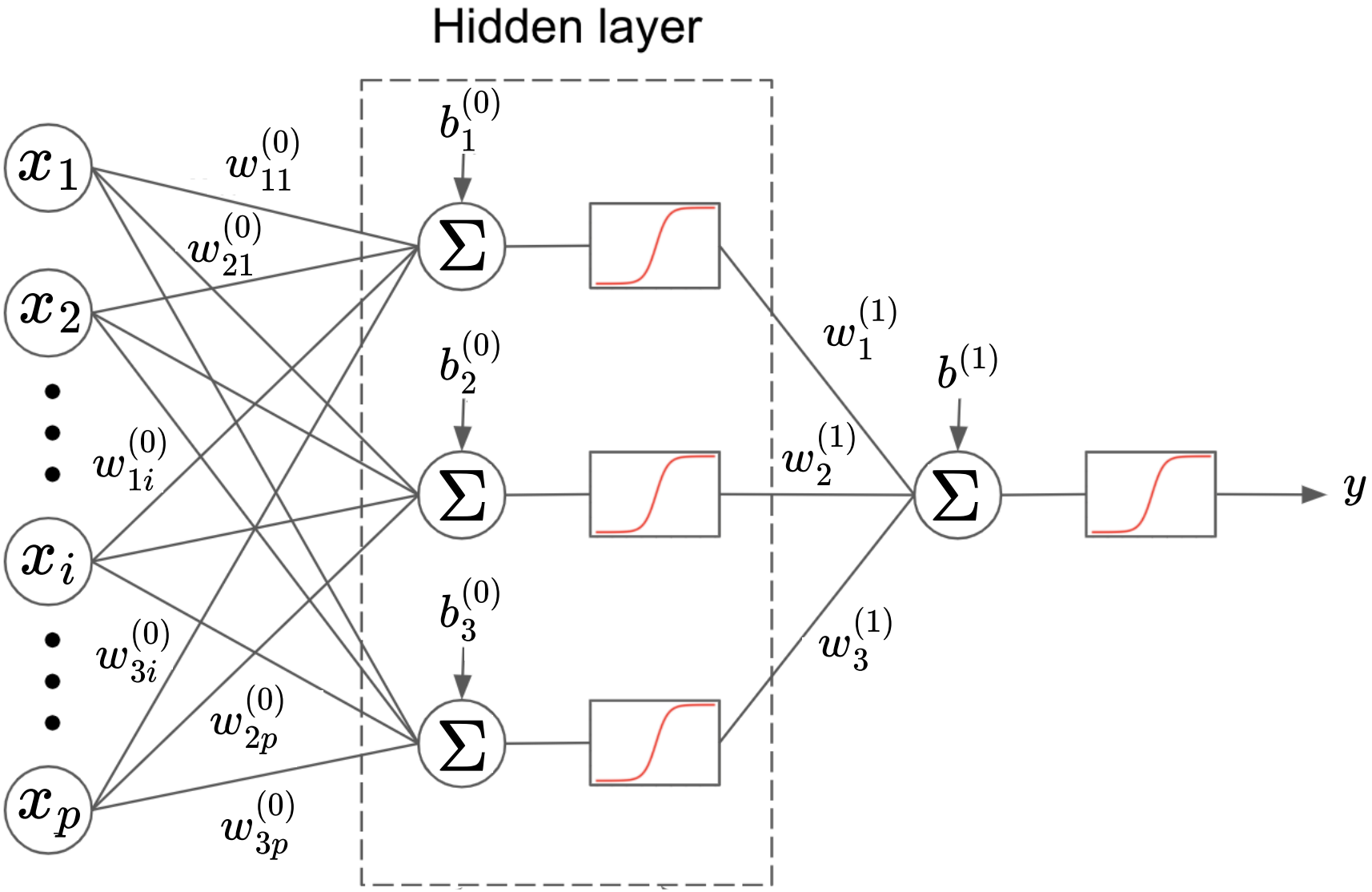
Artificial Neural Networks



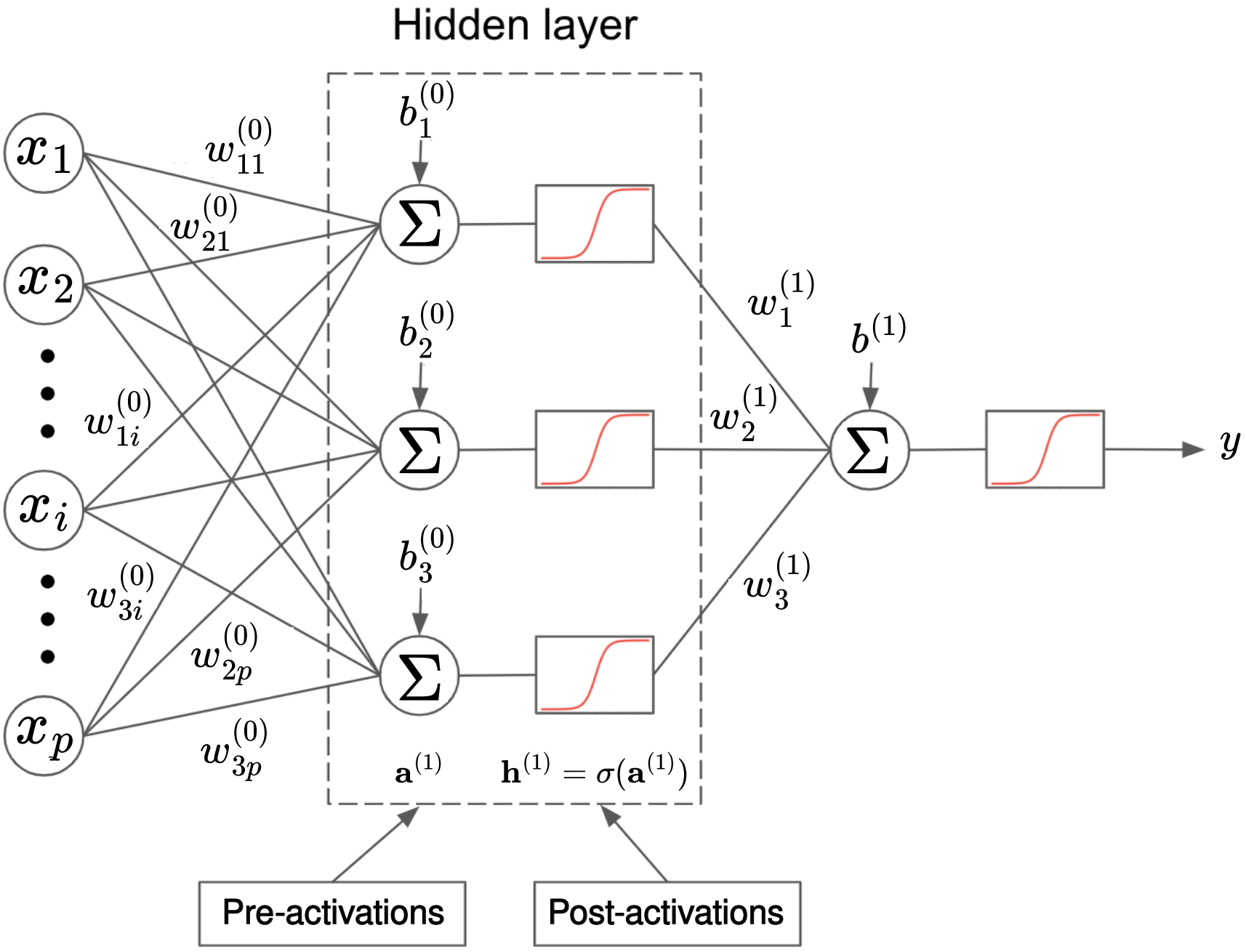
$$y_j = \sigma\left(\sum_i w_{ji} x_i + b_j\right)$$

One achieves higher fitting and predictive power by adding units, but especially by **composing** them into successive layers.

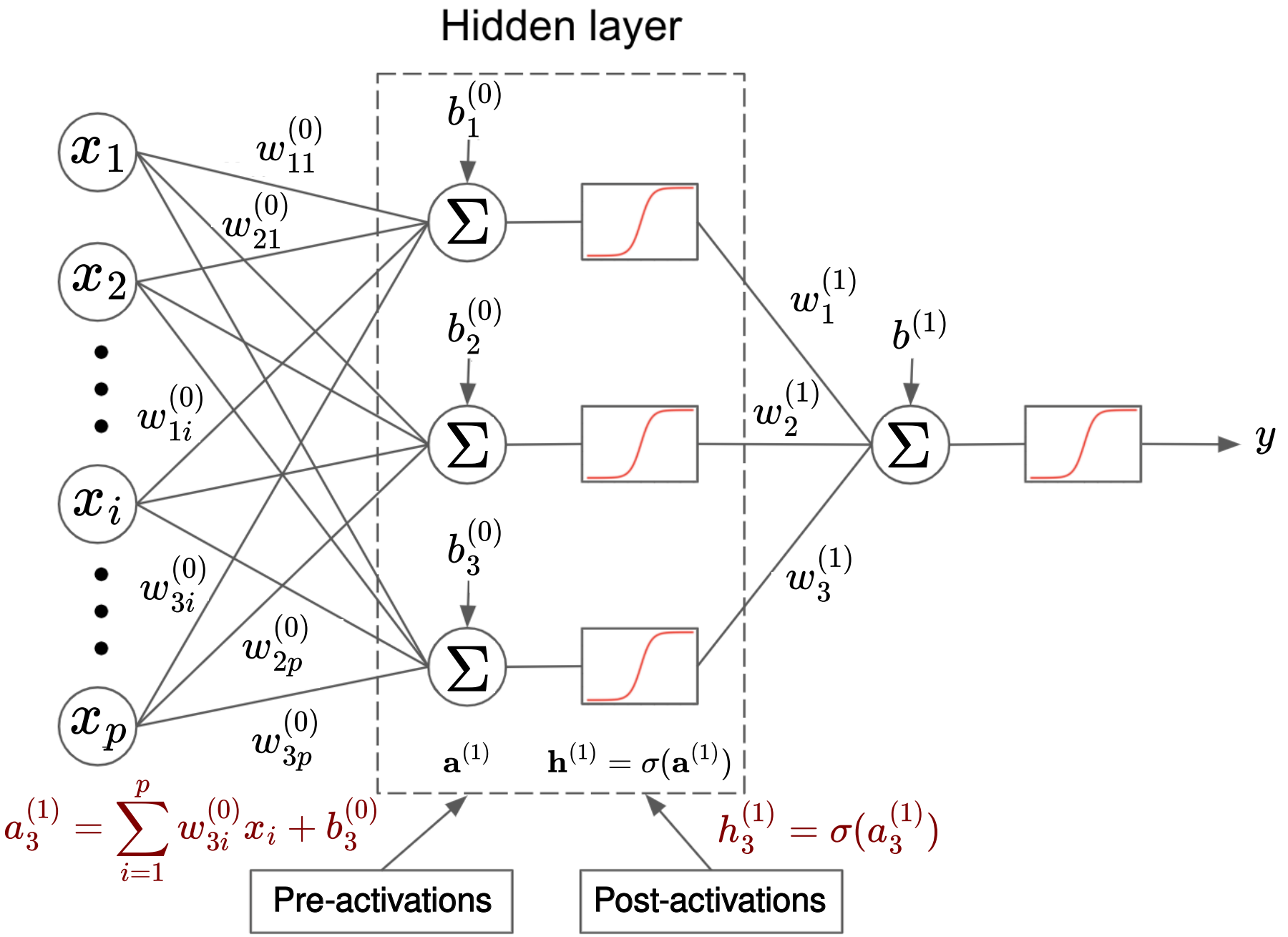
Artificial Neural Networks



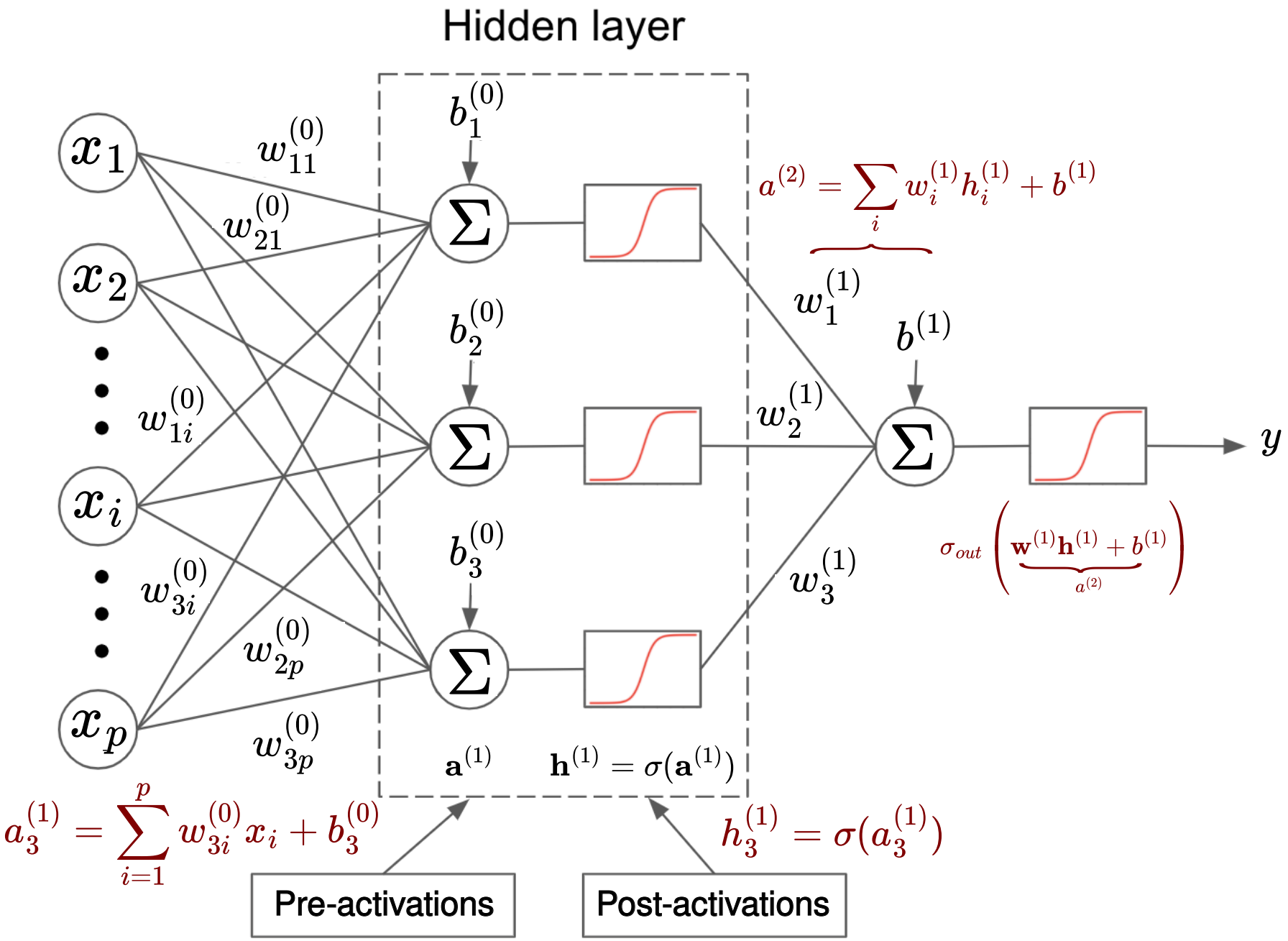
Artificial Neural Networks



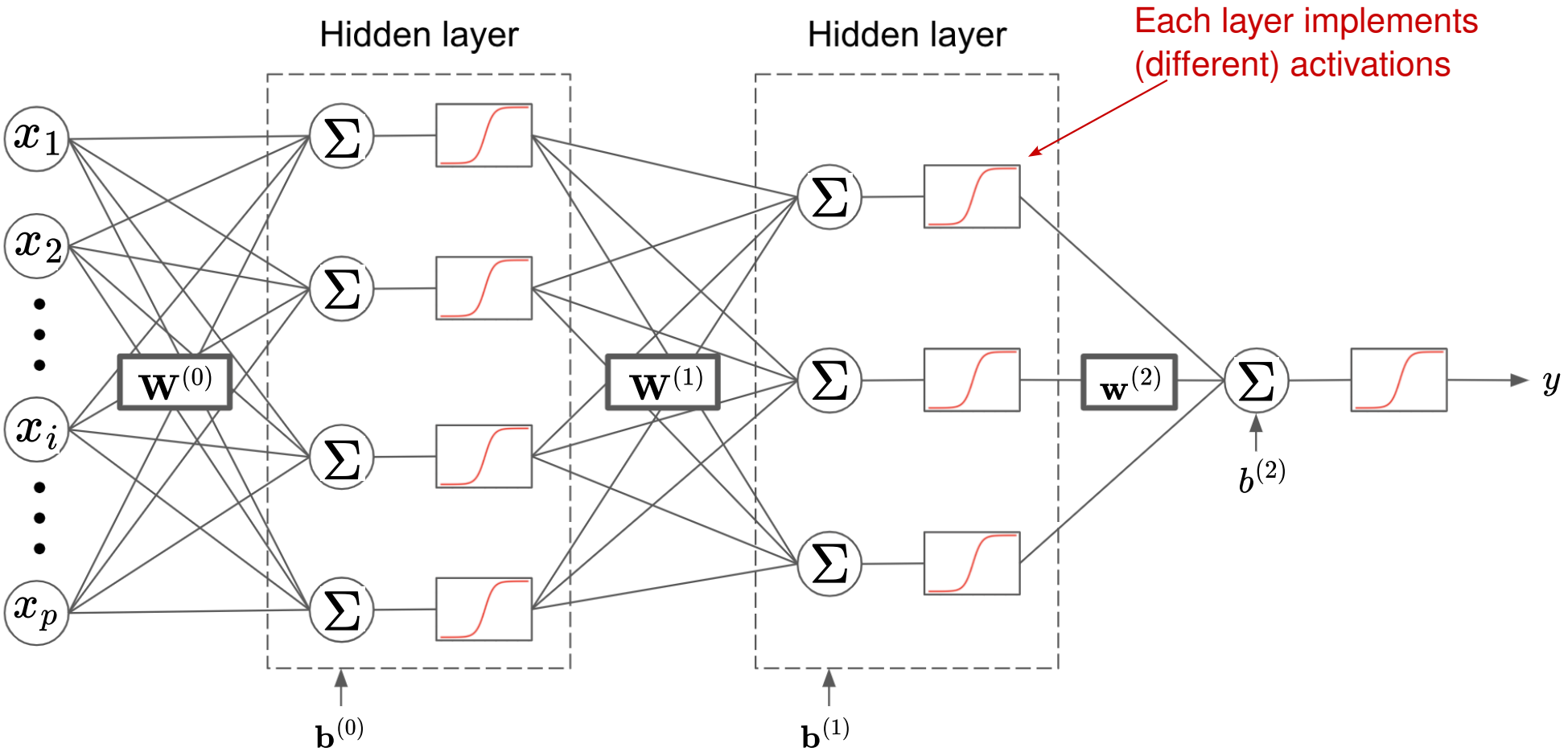
Artificial Neural Networks



Artificial Neural Networks

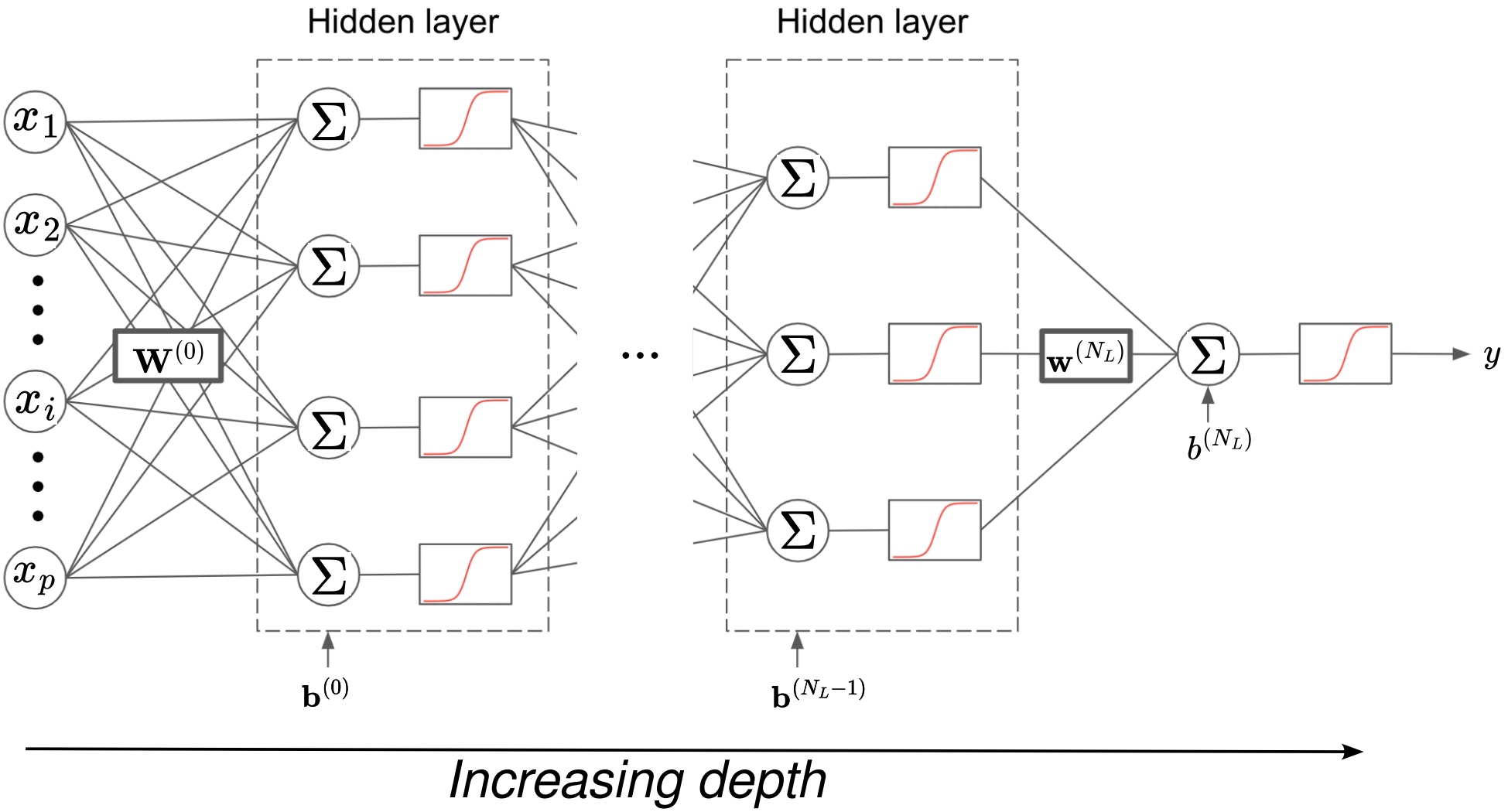


Artificial Neural Networks

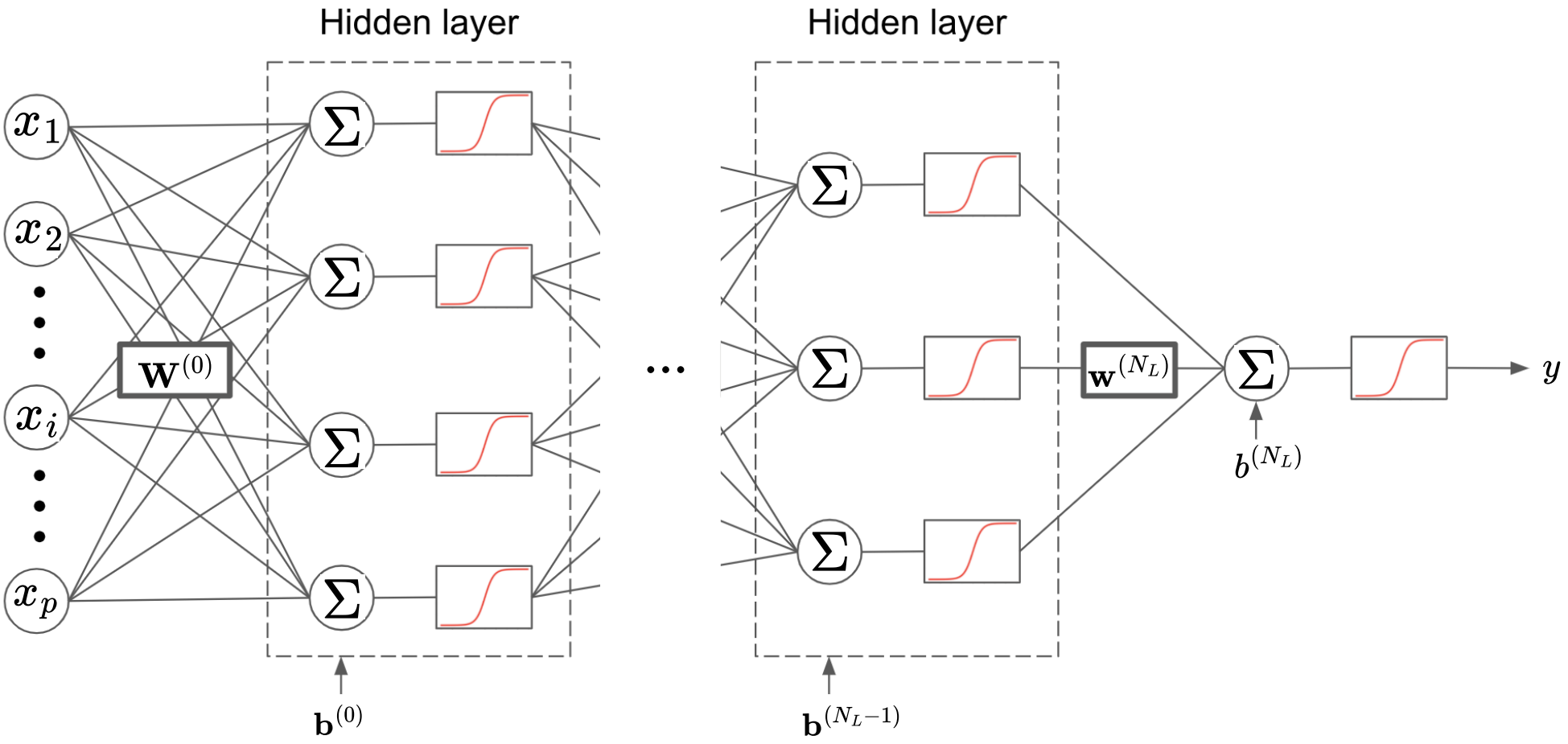


One can stack together an increasing number of layers
We get: **Multi-Layer Perceptron** (or **Feed-Forward NN**)

Artificial Neural Networks



Artificial Neural Networks



$$\mathbf{h}^{(0)} := \mathbf{x},$$

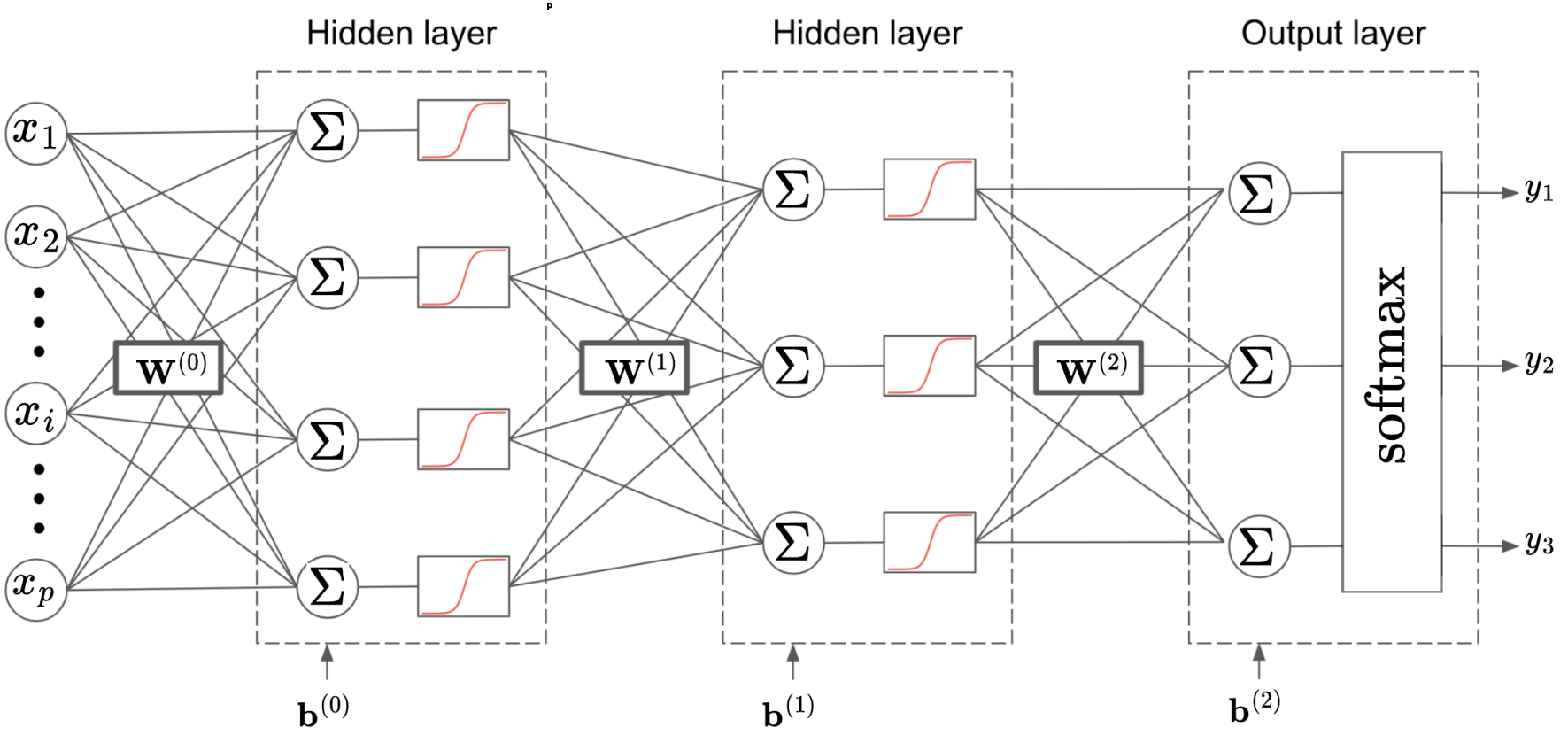
Mathematically: $$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)}\right), \qquad l = 1, \ldots, N_L,$$

$$\hat{y} = \sigma_{out}\left(\mathbf{w}^{(N_L)}\mathbf{h}^{(N_L)} + b^{(N_L)}\right), \qquad \mathbf{W}^{(l)} \in \mathbb{R}^{n_{l+1} \times n_l}$$

$$\mathbf{b}^{(l)} \in \mathbb{R}^{n_{l+1}} \quad \mathbf{h}^{(l)} \in \mathbb{R}^{n_l}$$

Artificial Neural Networks

Example of multi-output: Softmax layer for multi-class classification



It can be interpreted as a probability to belong to each class

$$\hat{\mathbf{y}}_q := \text{softmax}(\mathbf{a}^{(N_{L+1})})_q = \frac{\exp(a_q)}{\sum_{q'=1}^{Q} \exp(a_{q'})}, \qquad q = 1, \cdots, Q$$

Training: minibatch SGD

We have set up a model, the MLP, i.e. $f_{\boldsymbol{\theta}} : \mathbb{R}^p \mapsto Y$

weights and biases to learn during training

Training: minibatch SGD

We have set up a model, the MLP, i.e. $f_{\boldsymbol{\theta}} : \mathbb{R}^p \mapsto Y$

Initialise $\boldsymbol{\theta}$

Iterate:

Sample a minibatch $\mathcal{S}_m$

Compute the mean loss function:

$$L(\boldsymbol{\theta}_t; \mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{x}^{(i)}, y^{(i)} \in \mathcal{S}_m} L(y^{(i)}, f_{\boldsymbol{\theta}_t}(\mathbf{x}^{(i)}))$$

Training: minibatch SGD

We have set up a model, the MLP, i.e. $f_{\boldsymbol{\theta}} : \mathbb{R}^p \mapsto Y$

Initialise $\boldsymbol{\theta}$

Iterate:

Sample a minibatch $\mathcal{S}_m$

Compute the mean loss function:

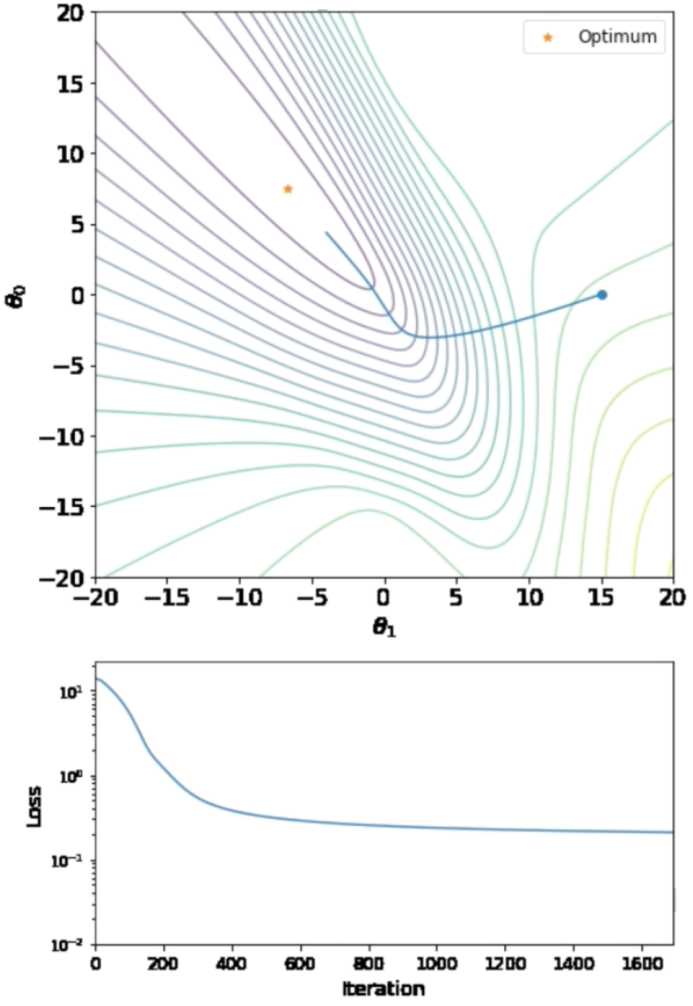
$$L(\boldsymbol{\theta}_t; \mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{x}^{(i)}, y^{(i)} \in \mathcal{S}_m} L(y^{(i)}, f_{\boldsymbol{\theta}_t}(\mathbf{x}^{(i)}))$$

Compute the gradient: $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$

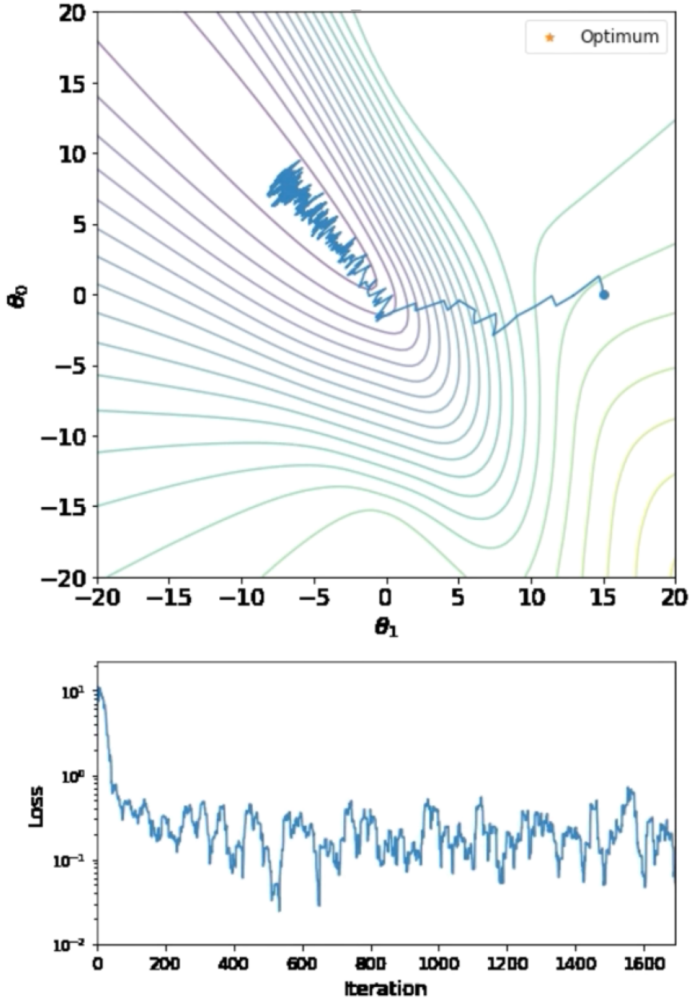
Update the parameters: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$

Training: minibatch SGD

Batch Gradient Descent (batch = full training set)

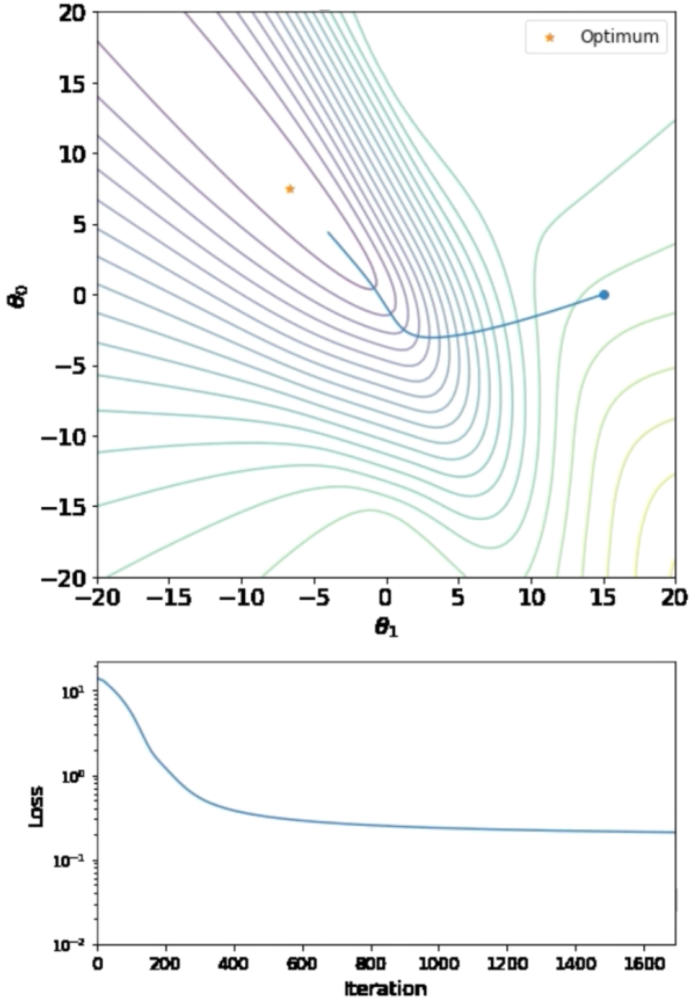


Minibatch Stochastic Gradient Descent

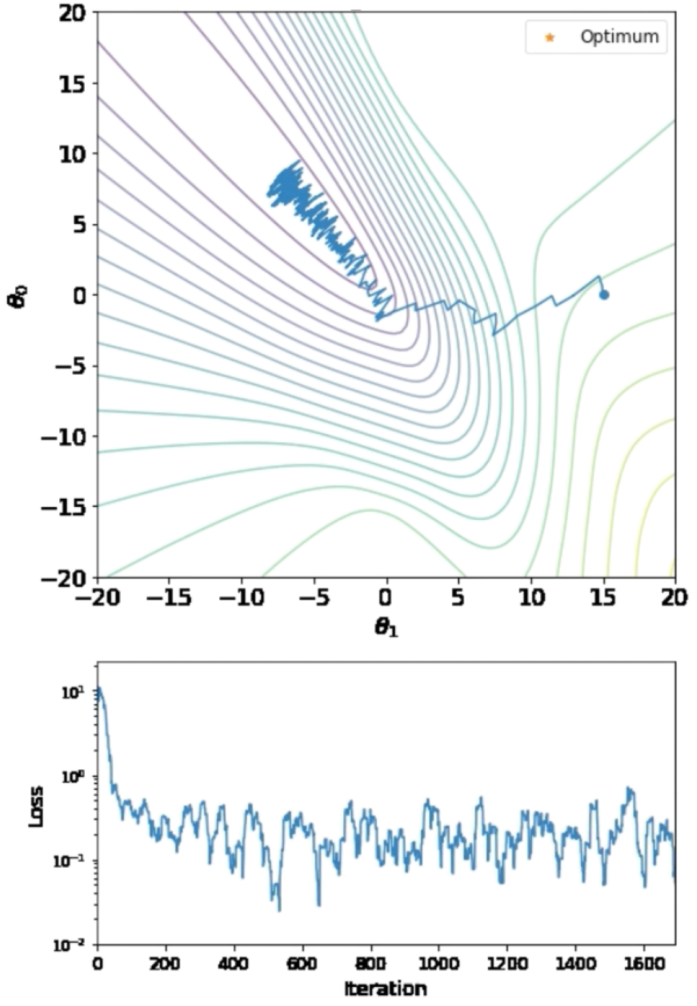


Training: minibatch SGD

Batch Gradient Descent (batch = full training set)



Minibatch Stochastic Gradient Descent



What Minibatch Size?

Larger batches:
More accurate estimation of gradient, but less than linear returns

Computationally & memory intensive

Smaller batches:
'Mini': 8, 16, 32, ... data points

Every update is faster, but need more

The noise can have a regularising effect (section 8.1.3, *Deep Learning*)

OK due to redundancies in big data

Training: minibatch SGD

We have set up a model, the MLP, i.e. $f_{\boldsymbol{\theta}} : \mathbb{R}^p \mapsto Y$

Initialise $\boldsymbol{\theta}$

Iterate:

Sample a minibatch $\mathcal{S}_m$

Compute the mean loss function:

$$L(\boldsymbol{\theta}_t; \mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{x}^{(i)}, y^{(i)} \in \mathcal{S}_m} L(y^{(i)}, f_{\boldsymbol{\theta}_t}(\mathbf{x}^{(i)}))$$

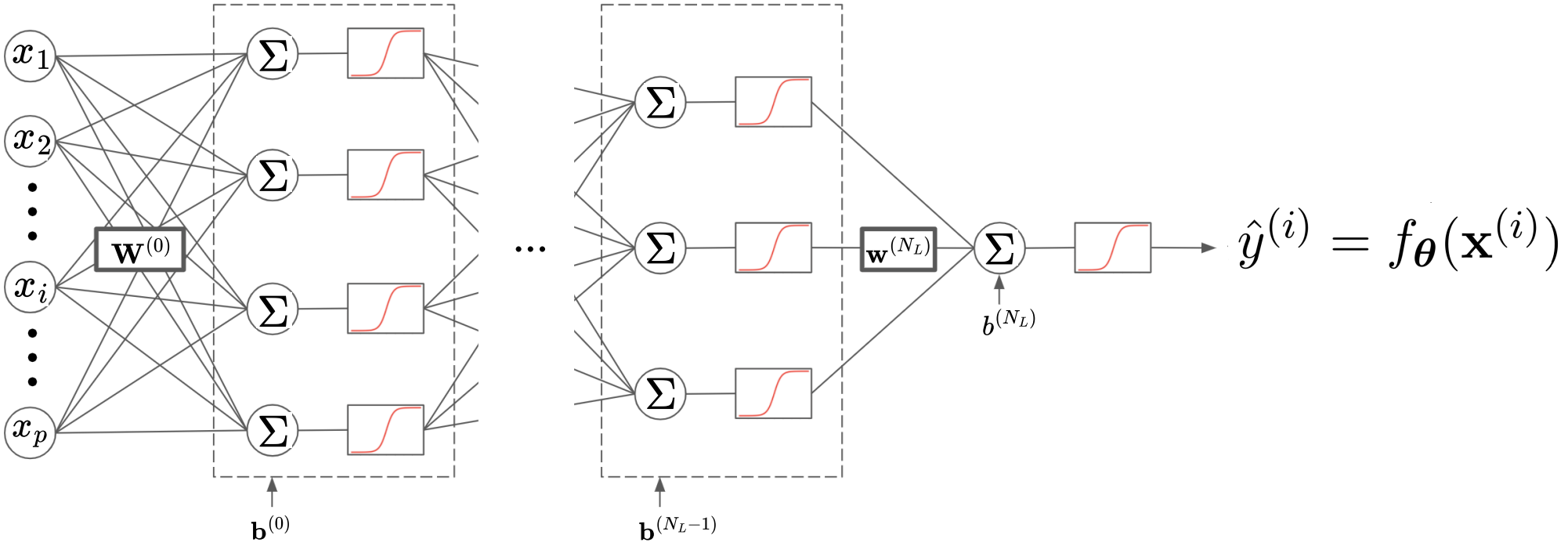
1. Compute the gradient: $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$
2. Update the parameters: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$

Error backpropagation

To compute efficiently $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$

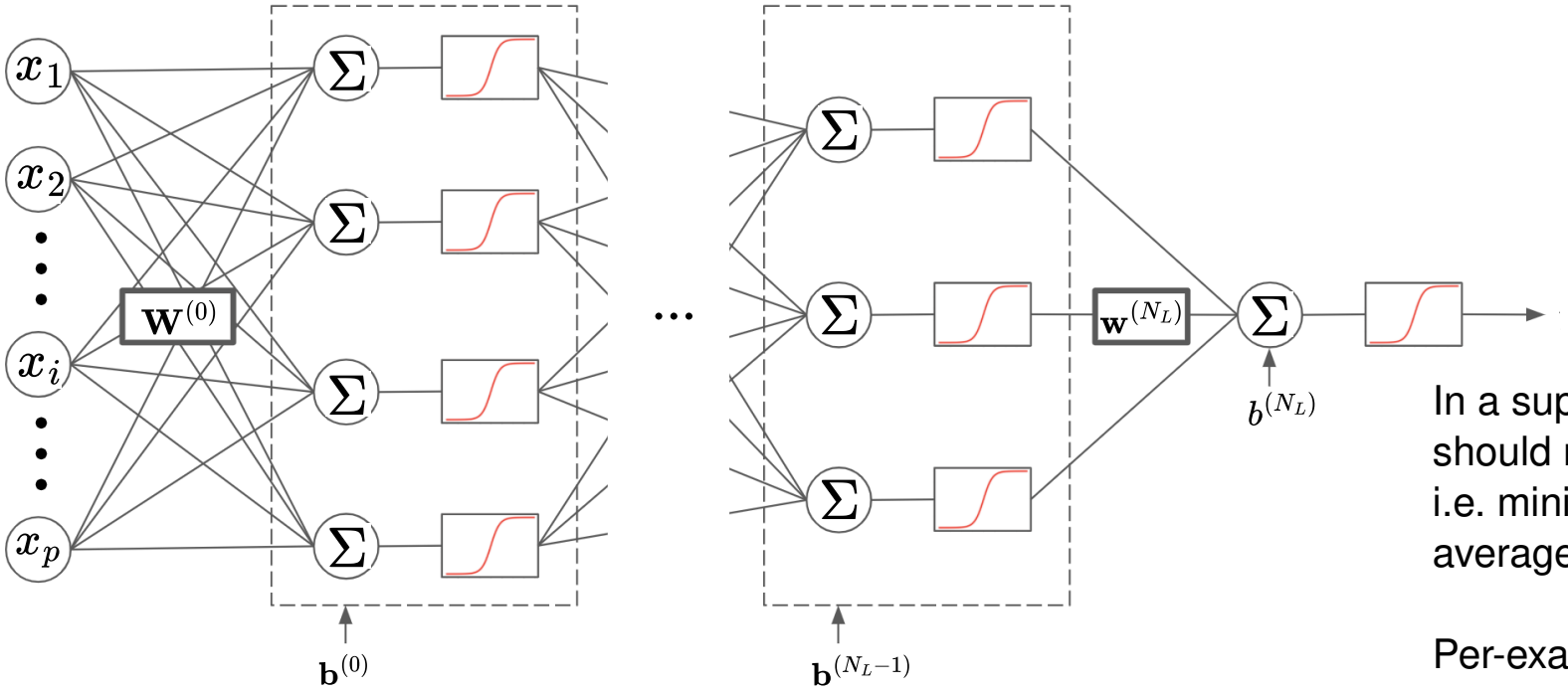
Error backpropagation

To compute efficiently $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$



Error backpropagation

To compute efficiently $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$



The prediction:

$$\hat{y}^{(i)} = f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})$$

In a supervised learning task, it should match the given outputs. i.e. minimising the minibatch-averaged per-example loss.

Per-example loss:

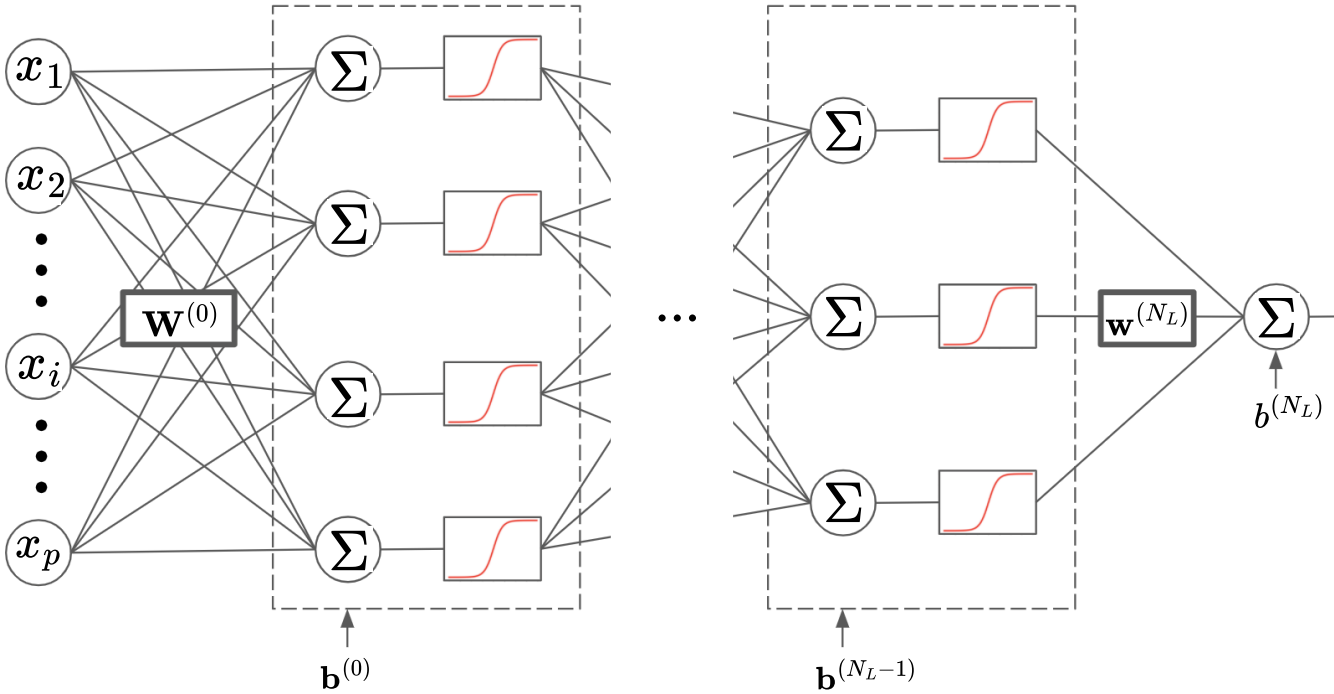
$$L_i := L(y^{(i)}, f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))$$

e.g. squared error (regressor):

$$L_i = (y^{(i)} - \hat{y}^{(i)})^2$$

Error backpropagation

To compute efficiently $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$



The prediction:

$$\hat{y}^{(i)} = f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})$$

In a supervised learning task, it should match the given outputs. i.e. minimising the minibatch-averaged per-example loss.

Per-example loss:

$$L_i := L(y^{(i)}, f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}))$$

e.g. squared error (regressor):

$$L_i = (y^{(i)} - \hat{y}^{(i)})^2$$

Chain rule of differentiation to backpropagate the gradient of the loss from the last layer to the previous layers

Error backpropagation

Target: compute a gradient wrt the weights (and biases)

$$\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} h_q^{(l)}$$

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

Error backpropagation

Target: compute a gradient wrt the weights (and biases)

$$\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} h_q^{(l)}$$

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

Define the **error**:

$$\delta_k^{(l)} := \frac{\partial L_i}{\partial a_k^{(l)}}$$

So the gradient is wriiten: $\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \delta_k^{(l+1)} h_q^{(l)}$

Error backpropagation

Target: compute a gradient wrt the weights (and biases)

$$\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} h_q^{(l)}$$

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

Define the **error**:

$$\delta_k^{(l)} := \frac{\partial L_i}{\partial a_k^{(l)}}$$

So the gradient is wriiten: $\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \delta_k^{(l+1)} h_q^{(l)}$

$$\delta_k^{(l)} \equiv \frac{\partial L_i}{\partial a_k^{(l)}} = \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = \sigma'(a_k^{(l)}) \sum_{j=1}^{n_{l+1}} w_{jk}^{(l)} \delta_j^{(l+1)}$$

$$a_j^{(l+1)} = \sum_{n=1}^{n_l} w_{jn}^{(l)} \sigma(a_n^{(l)}) + b_j^{(l)}$$

Error backpropagation

Target: compute a gradient wrt the weights (and biases)

$$\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} h_q^{(l)}$$

One needs post-activations and errors

Define the **error**:

$$\delta_k^{(l)} := \frac{\partial L_i}{\partial a_k^{(l)}}$$

So the gradient is wriiten: $\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \delta_k^{(l+1)} h_q^{(l)}$

$$\delta_k^{(l)} \equiv \frac{\partial L_i}{\partial a_k^{(l)}} = \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = \sigma'(a_k^{(l)}) \sum_{j=1}^{n_{l+1}} w_{jk}^{(l)} \delta_j^{(l+1)}$$

Error backpropagation

Target: compute a gradient wrt the weights (and biases)

$$\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial w_{kq}^{(l)}} = \frac{\partial L_i}{\partial a_k^{(l+1)}} h_q^{(l)}$$

One needs post-activations and errors

Define the **error**:

$$\delta_k^{(l)} := \frac{\partial L_i}{\partial a_k^{(l)}}$$

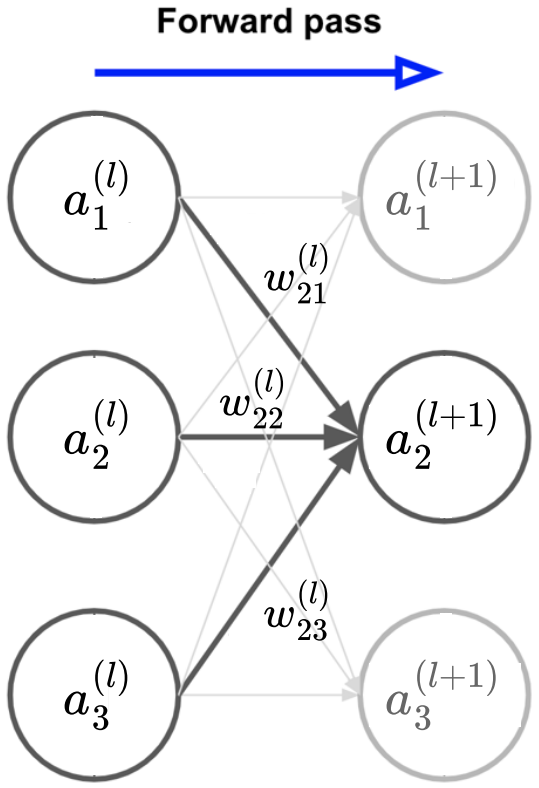
So the gradient is wriiten: $\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \delta_k^{(l+1)} h_q^{(l)}$

$$\boxed{\delta_k^{(l)}} \equiv \frac{\partial L_i}{\partial a_k^{(l)}} = \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} \frac{\partial a_j^{(l+1)}}{\partial a_k^{(l)}} = \sigma'(a_k^{(l)}) \sum_{j=1}^{n_{l+1}} w_{jk}^{(l)} \boxed{\delta_j^{(l+1)}}$$

Errors: found recursively from the last layer

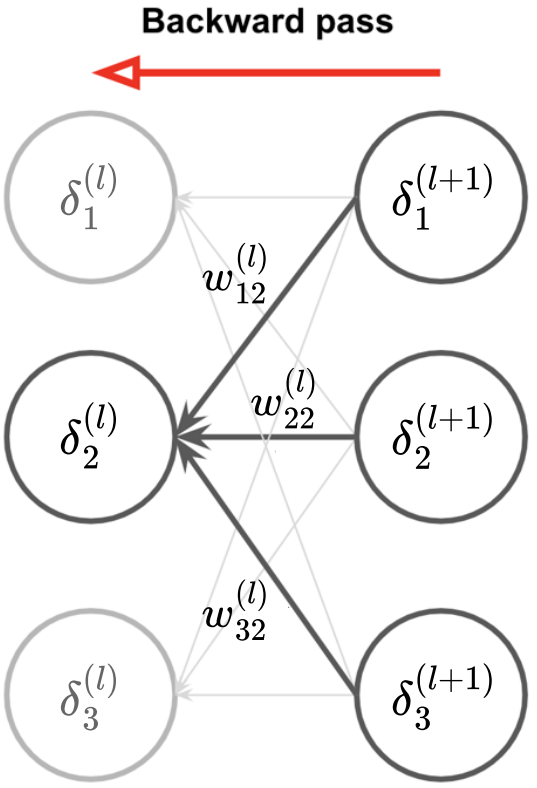
Error backpropagation

Building upon these observations, the error backpropagation algorithm combines:



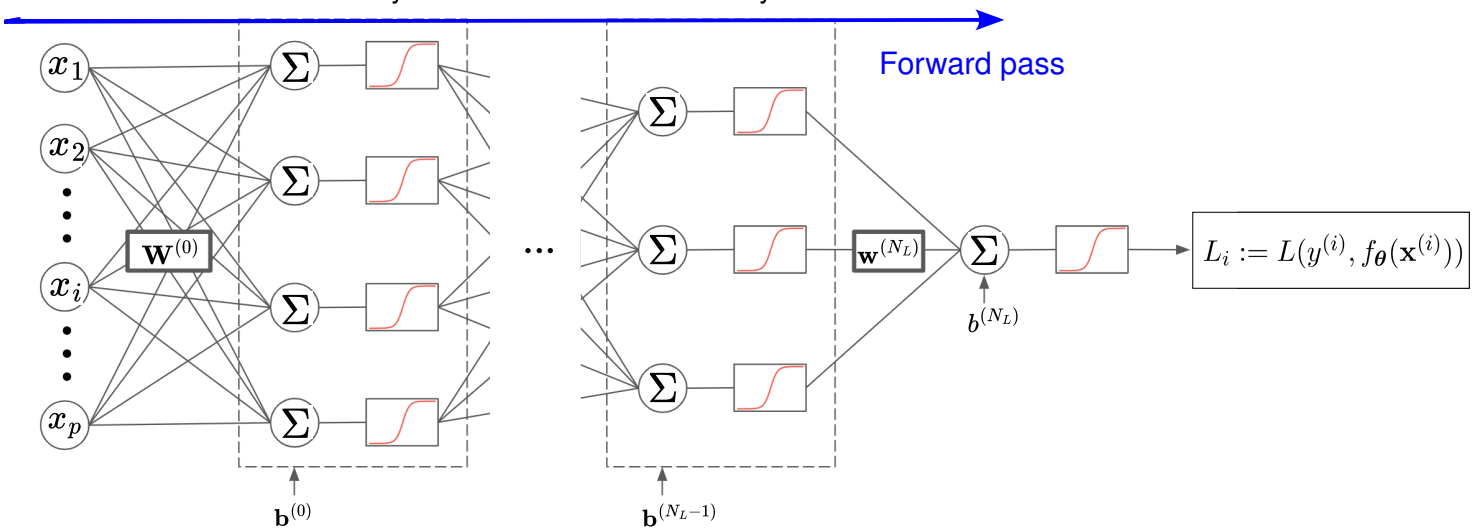
$$a_j^{(l+1)} = \sum_{n=1}^{n_l} w_{jn}^{(l)} \sigma(a_n^{(l)}) + b_j^{(l)}$$

calculates pre- and post-activations



$$\delta_k^{(l)} = \sigma'(a_k^{(l)}) \sum_{j=1}^{n_{l+1}} w_{jk}^{(l)} \delta_j^{(l+1)}$$

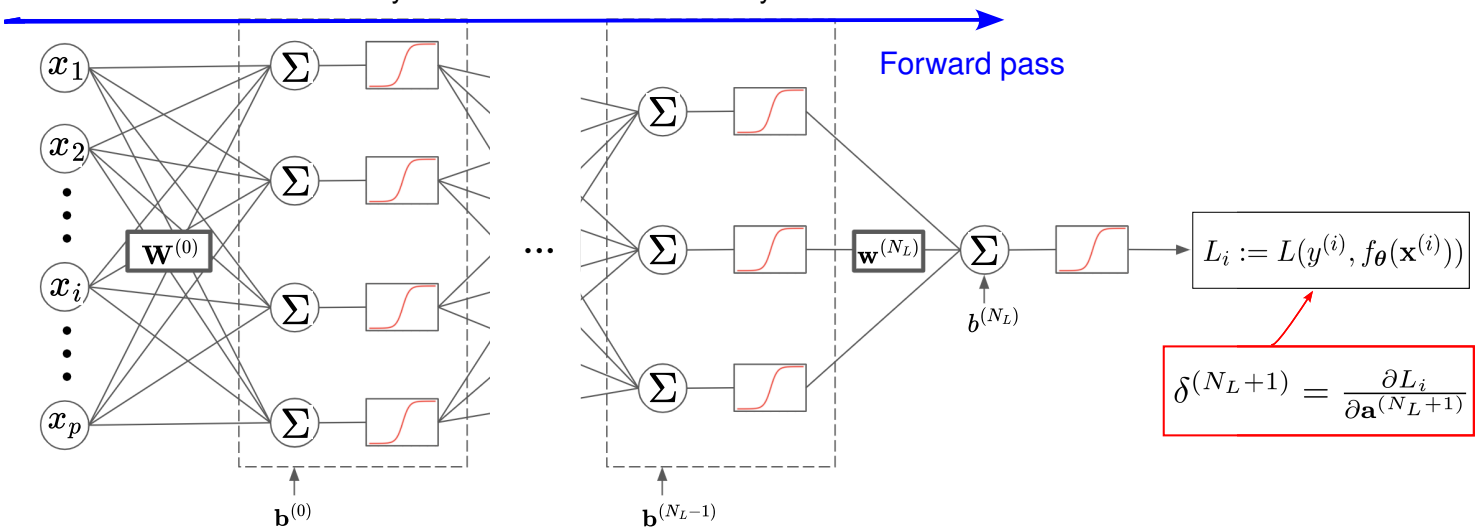
backpropagates the error



Key message: The backpropagation algorithm works as follows:

(1) Propagate the signal forwards by passing an input vector $\mathbf{x}^{(i)}$ through the network and computing all pre-activations and post-activations using:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)} \qquad l = 1, \cdots, N_L.$$

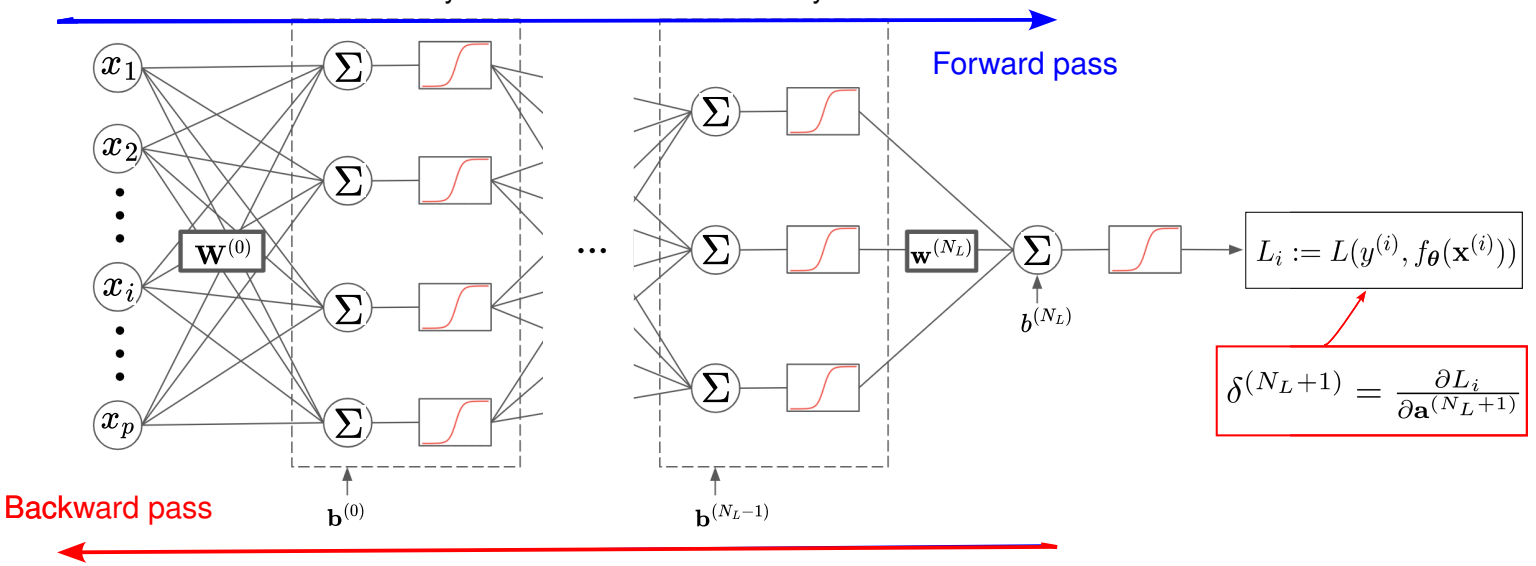


Key message: The backpropagation algorithm works as follows:

(1) Propagate the signal forwards by passing an input vector $\mathbf{x}^{(i)}$ through the network and computing all pre-activations and post-activations using:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)} \qquad l = 1, \cdots, N_L.$$

(2) Evaluate $\delta^{(N_L+1)} = \frac{\partial L_i}{\partial \mathbf{a}^{(N_L+1)}}$ for the output neurons (i.e., at layer N_L+1).



Key message: The backpropagation algorithm works as follows:

(1) Propagate the signal forwards by passing an input vector $\mathbf{x}^{(i)}$ through the network and computing all pre-activations and post-activations using:

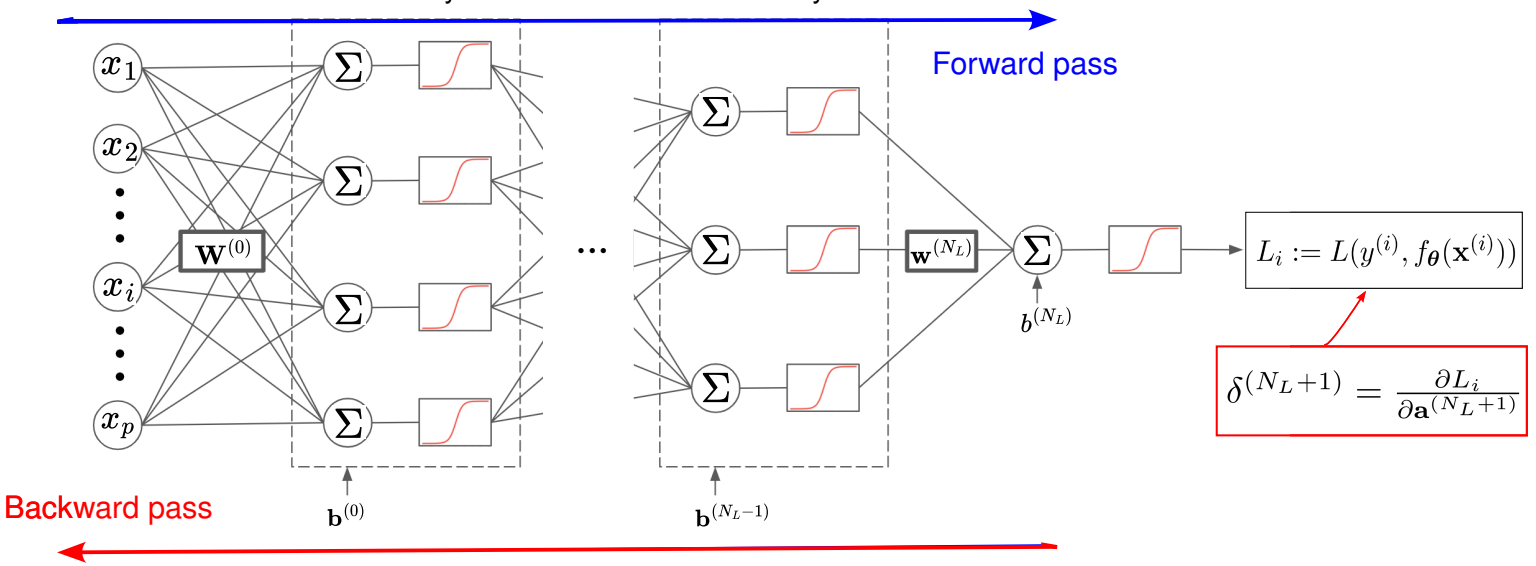
$$\mathbf{a}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)} \qquad l = 1, \cdots, N_L.$$

(2) Evaluate $\delta^{(N_L+1)} = \frac{\partial L_i}{\partial \mathbf{a}^{(N_L+1)}}$ for the output neurons (i.e., at layer N_L+1).

(3) Backpropagate the errors to compute $\delta^{(l)}$ for each hidden unit using Equation (7.22), whose compact form reads:

$$\delta^{(l)} = \sigma'(\mathbf{a}^{(l)})(\mathbf{W}^{(l)})^T \delta^{(l+1)},$$

where $\sigma'(\mathbf{a}^{(l)}) = \text{diag}([\sigma'(a_k^{(l)})]_{k=1}^{n_l})$ and the activation functions are applied element-wise.



Key message: The backpropagation algorithm works as follows:

(1) Propagate the signal forwards by passing an input vector $\mathbf{x}^{(i)}$ through the network and computing all pre-activations and post-activations using:

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l-1)} \qquad l = 1, \cdots, N_L.$$

(2) Evaluate $\delta^{(N_L+1)} = \frac{\partial L_i}{\partial \mathbf{a}^{(N_L+1)}}$ for the output neurons (i.e., at layer N_L+1).

(3) Backpropagate the errors to compute $\delta^{(l)}$ for each hidden unit using Equation (7.22), whose compact form reads:

$$\delta^{(l)} = \sigma'(\mathbf{a}^{(l)})(\mathbf{W}^{(l)})^T \delta^{(l+1)},$$

where $\sigma'(\mathbf{a}^{(l)}) = \text{diag}([\sigma'(a_k^{(l)})]_{k=1}^{n_l})$ and the activation functions are applied element-wise.

(4) Obtain the derivatives of L_i with respect to the weights and biases using: $\frac{\partial L_i}{\partial w_{kq}^{(l)}} = \delta_k^{(l+1)} h_q^{(l)}, \quad \frac{\partial L_i}{\partial b_k^{(l)}} = \delta_k^{(l+1)}.$

Training: minibatch SGD

We have set up a model, the MLP, i.e. $f_{\boldsymbol{\theta}} : \mathbb{R}^p \mapsto Y$

Initialise $\boldsymbol{\theta}$

Iterate:

Sample a minibatch $\mathcal{S}_m$

Compute the mean loss function:

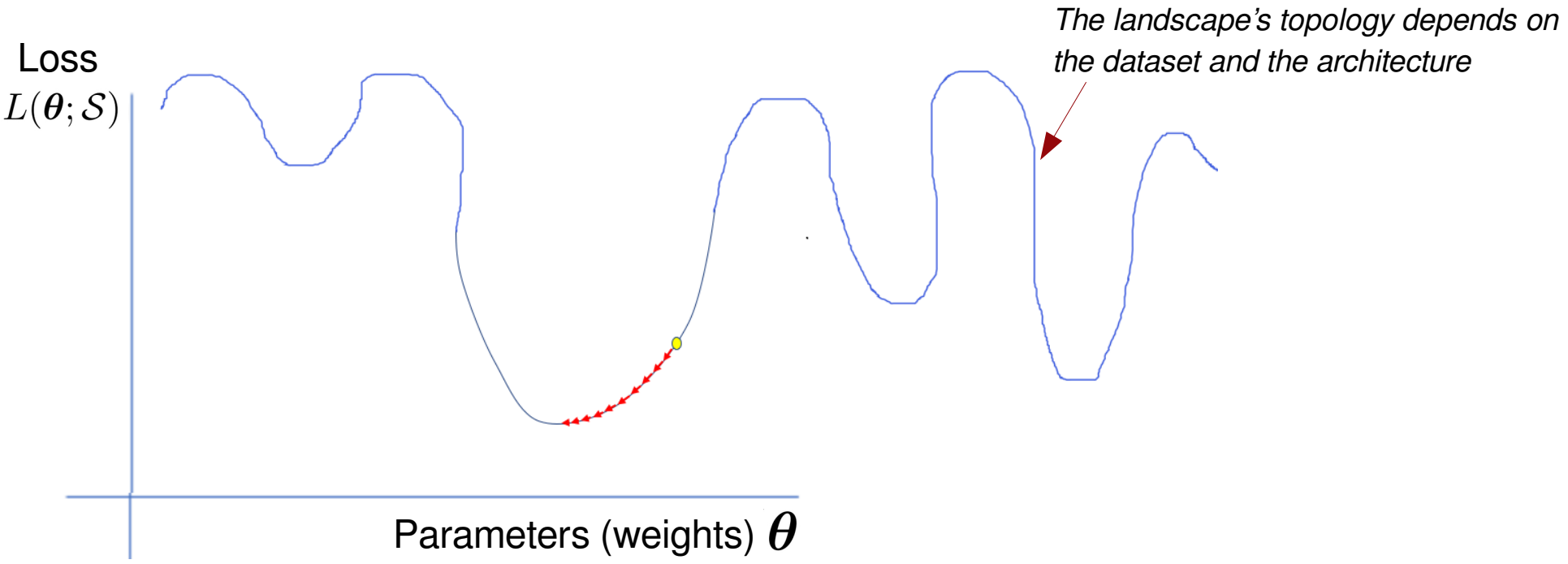
$$L(\boldsymbol{\theta}_t; \mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{x}^{(i)}, y^{(i)} \in \mathcal{S}_m} L(y^{(i)}, f_{\boldsymbol{\theta}_t}(\mathbf{x}^{(i)}))$$

Learning rate is crucial!

1. Compute the gradient: $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$
2. Update the parameters: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m)$

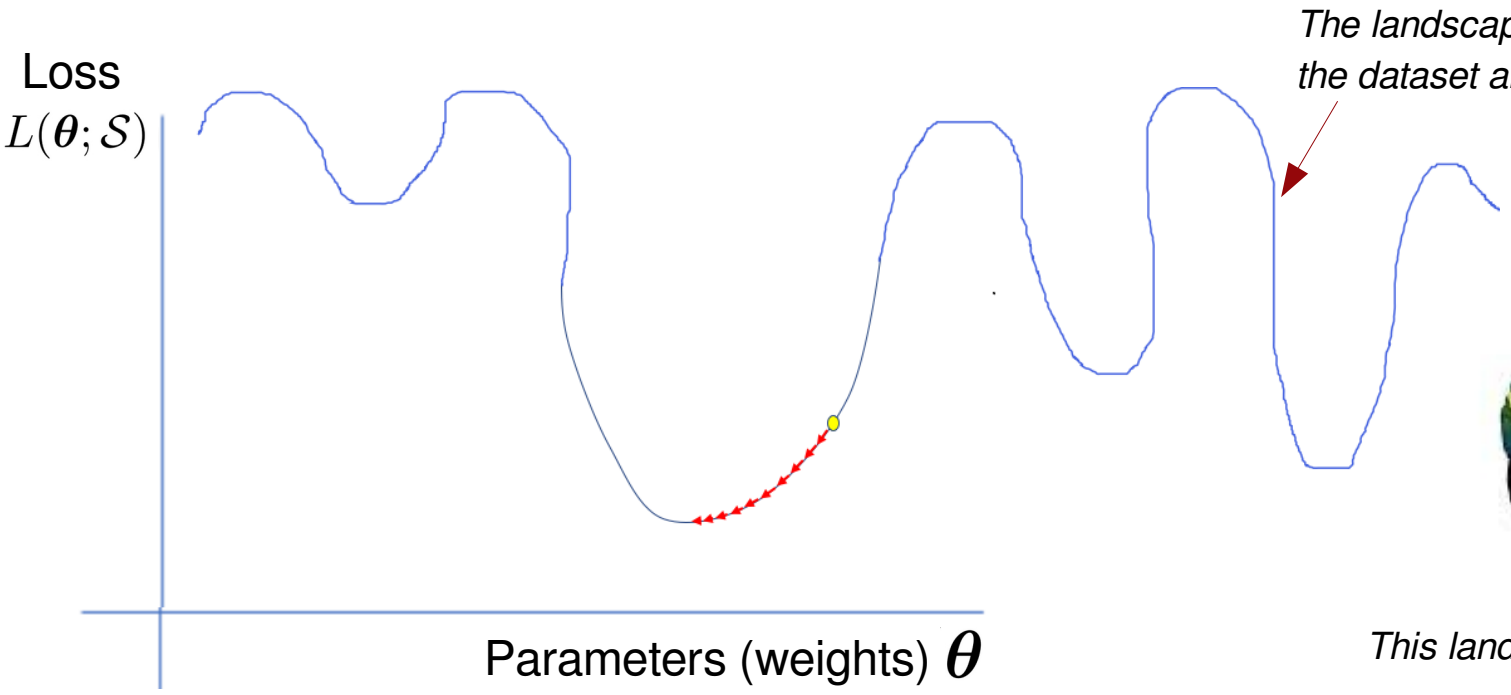
Learning rate

It controls the the magnitude of weights' change during training (step size)

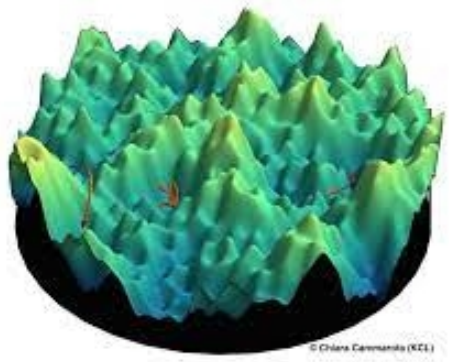


Learning rate

It controls the the magnitude of weights' change during training (step size)



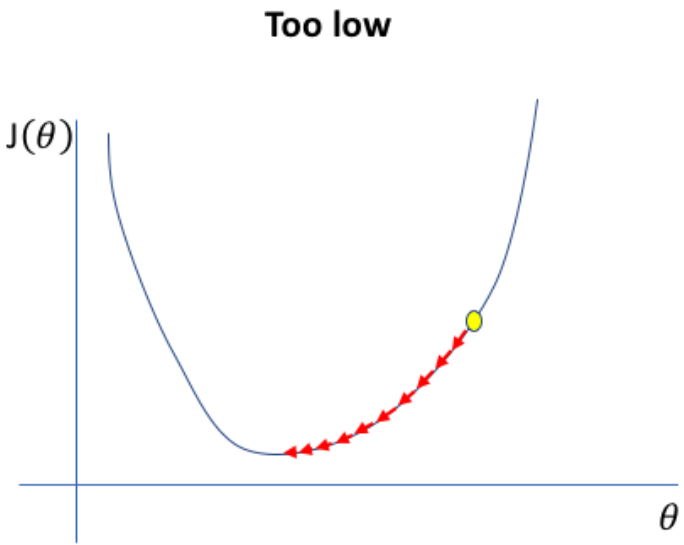
The landscape's topology depends on the dataset and the architecture

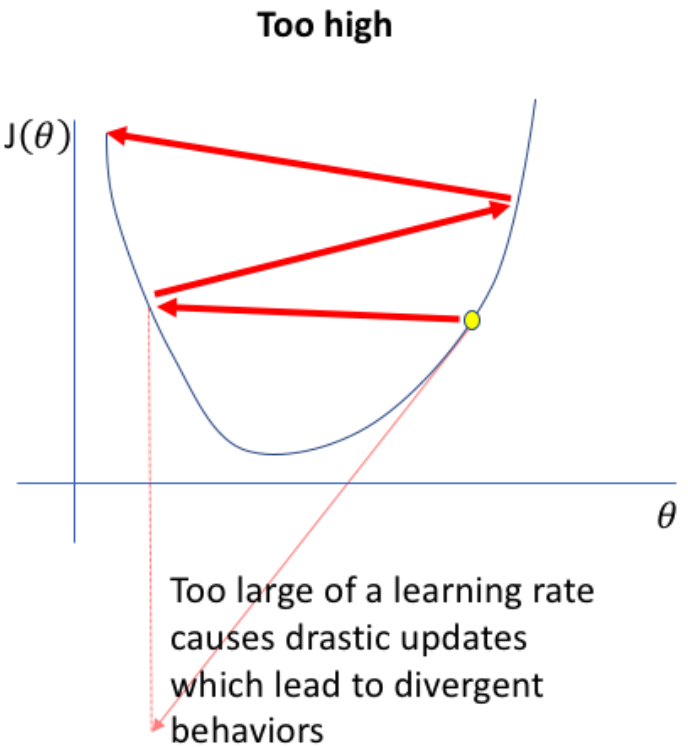


This landscape can be extremely rough

The loss landscape of a neural network is highly complex and **non-convex:** a large number of local minima (but many are good solutions!)

Learning rate



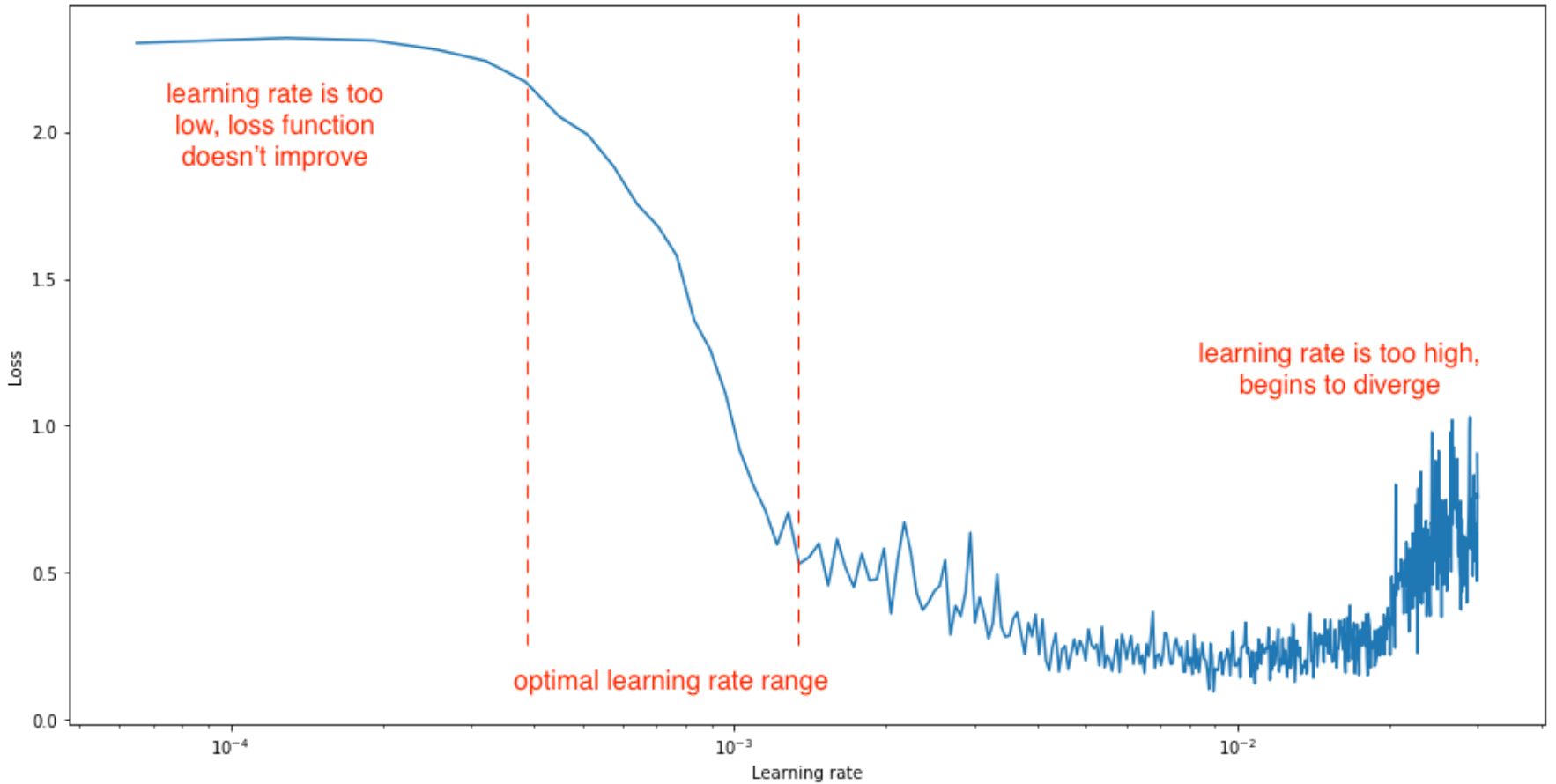


Long training, might become stuck with high loss

or suboptimal solutions

Image from https://www.jeremyjordan.me/nn-learning-rate/

Learning rate



Typical trend of the training loss with learning rate

Image from https://www.jeremyjordan.me/nn-learning-rate/

To avoid noise close to the solution, in practice one **progressively decreases the learning rate** along the iterations using **learning rate schedules**, for example:

Time-Based Decay

$$\frac{lr_0}{(1+kt)}$$

Iteration number

Step Decay

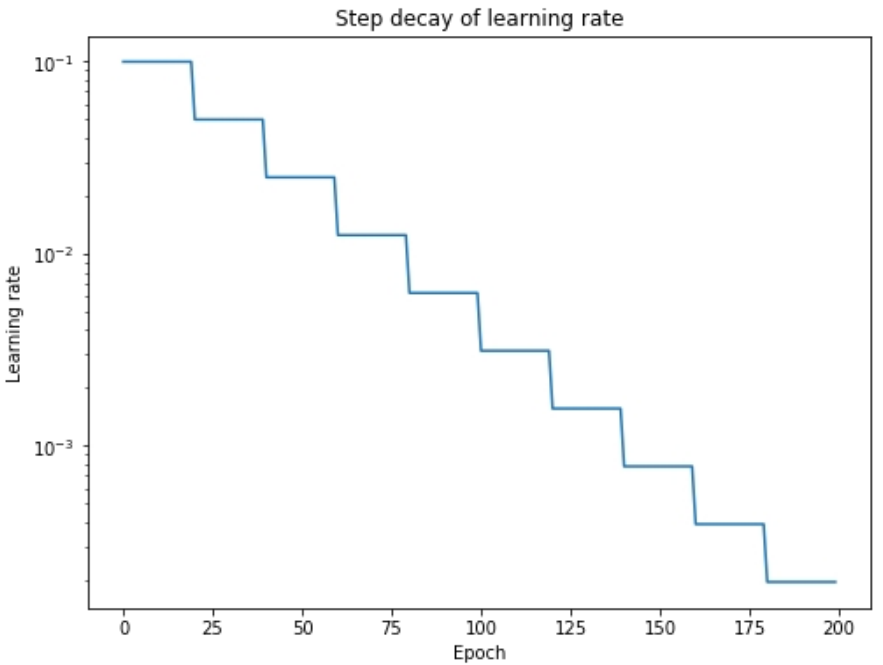


Image from https://www.jeremyjordan.me/nn-learning-rate/

To accelerate the learning with slow or noisy gradients, one can add **momentum**

Gradient Descent with momentum:

$$\mathbf{g}_t := \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m),$$

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \eta \mathbf{g}_t$$

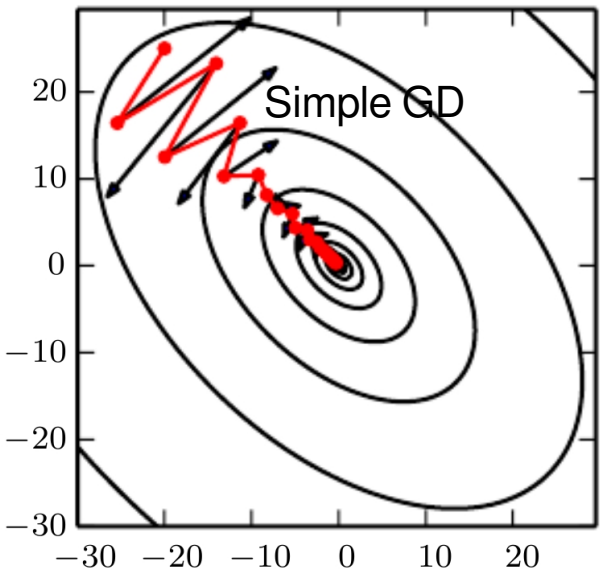
Velocity accumulates an exponentially decaying memory of past gradients

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_{t+1},$$

Parameter update rule contains a 'velocity'

To accelerate the learning with slow or noisy gradients, one can add **momentum**

With momentum



Gradient Descent with momentum:

$$\mathbf{g}_t := \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t; \mathcal{S}_m),$$

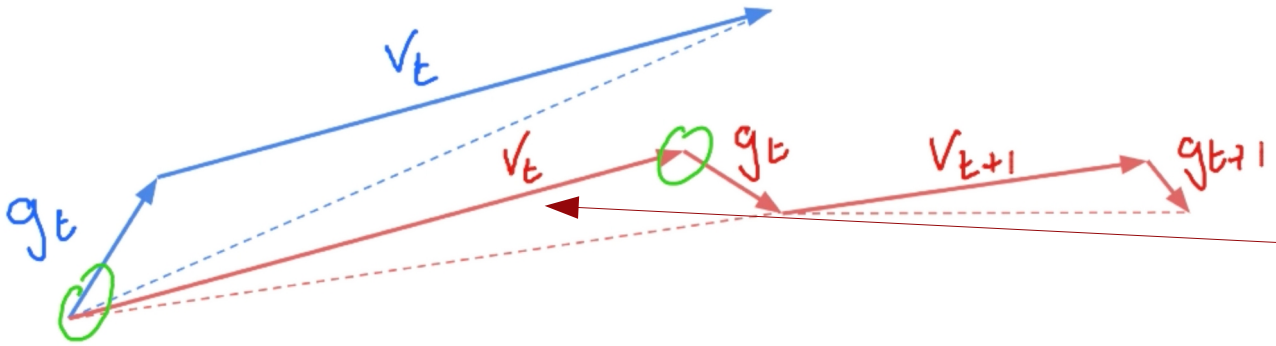
$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \eta \mathbf{g}_t$$ ← Velocity accumulates an exponentially decaying memory of past gradients

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_{t+1},$$ ← Parameter update rule contains a 'velocity'

The momentum allows the learning to efficiently traverse the valley lengthwise

Image from Goodfellow, Bengio, Courville, *Deep Learning*

A variant of momentum – **Nesterov Momentum**



Blue: SGD with momentum
Red: SGD with Nesterov momentum

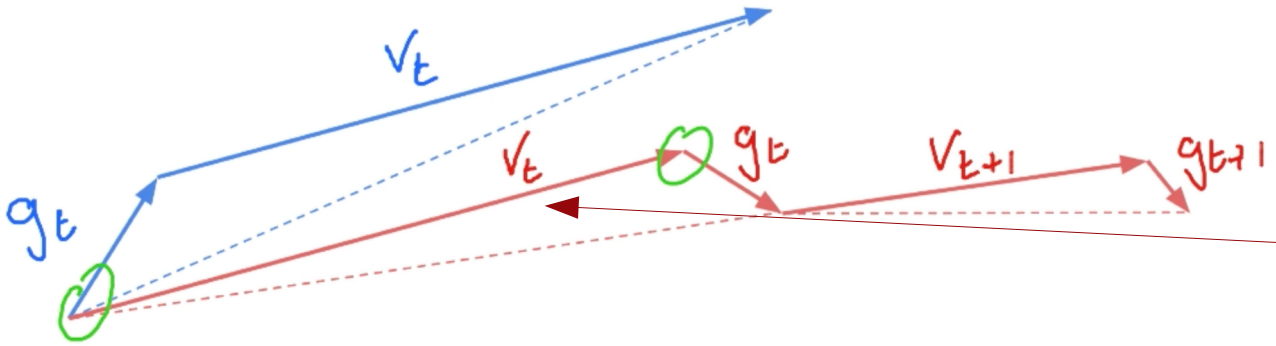
The accumulated gradient is calculated before the gradient step

$$\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t - \beta \mathbf{v}_t; \mathcal{S}_m),$$

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \eta \mathbf{g}_t$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_{t+1}.$$

A variant of momentum – **Nesterov Momentum**



Blue: SGD with momentum
Red: SGD with Nesterov momentum

The accumulated gradient is calculated before the gradient step

This gives an estimate of the next step in parameter space: i.e., an ability to 'look ahead' that **accelerates training convergence**

$$\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t - \beta \mathbf{v}_t; \mathcal{S}_m),$$

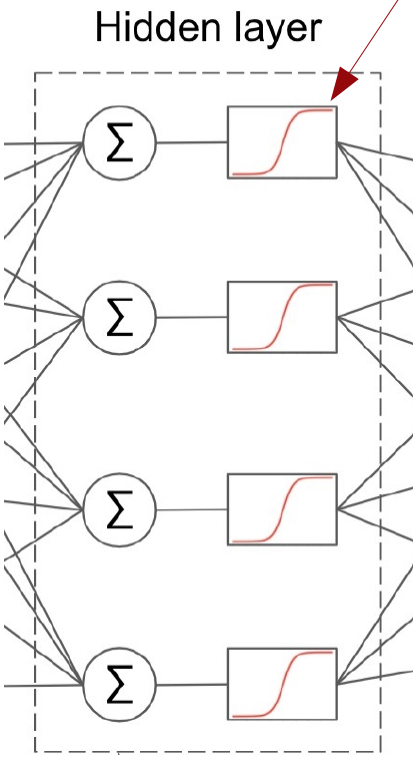
$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \eta \mathbf{g}_t$$

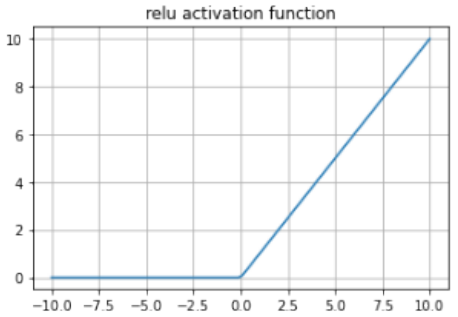
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_{t+1}.$$

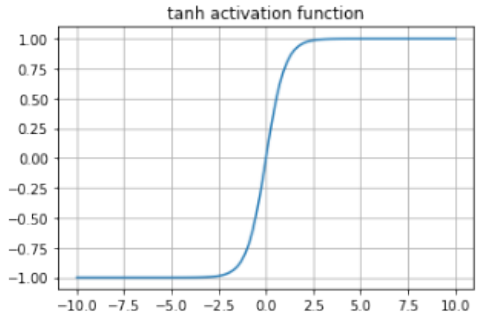
See Coursework 1, 2024

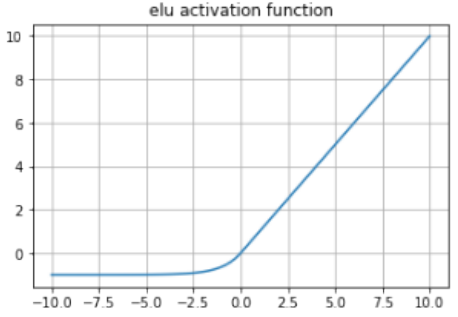
Why are neural networks so powerful?

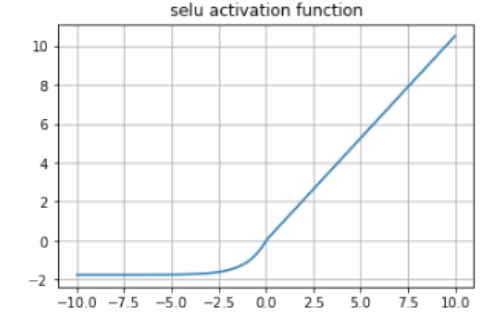
Nonlinearities: Activation functions like ReLU, tanh, softplus etc.

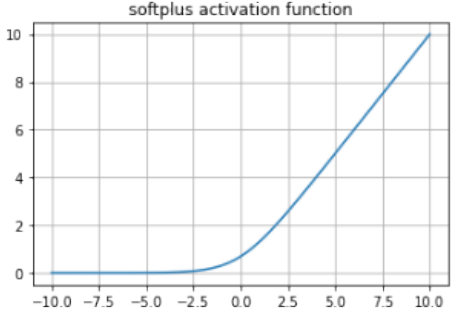


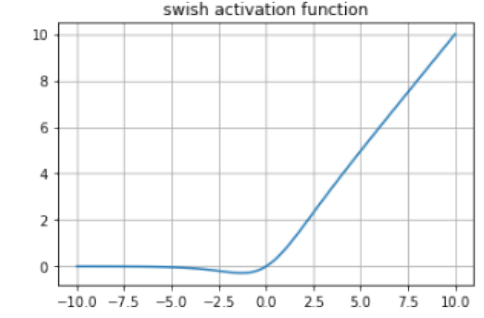






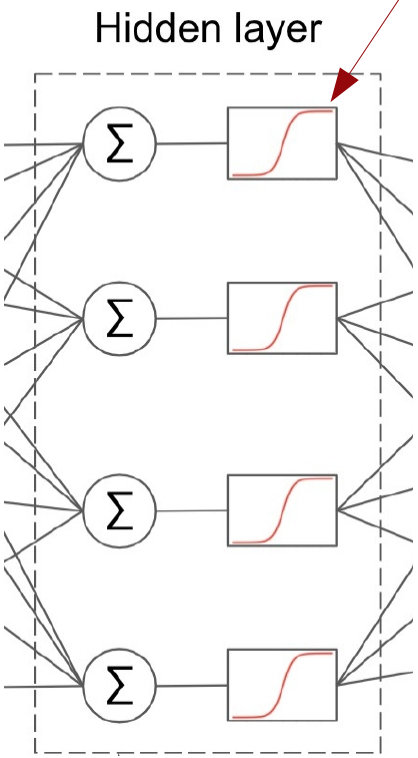




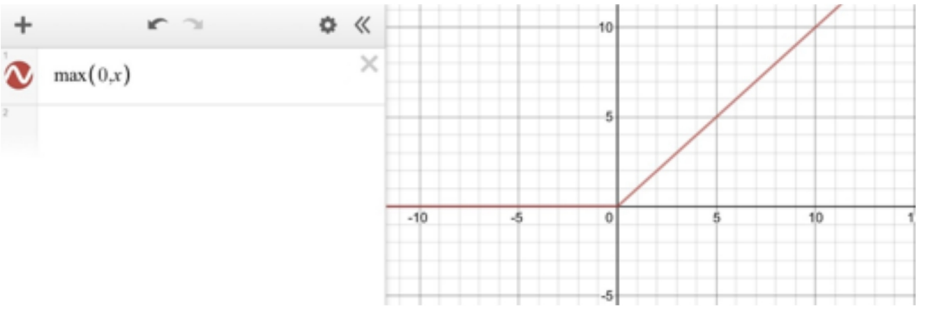


Why are neural networks so powerful?

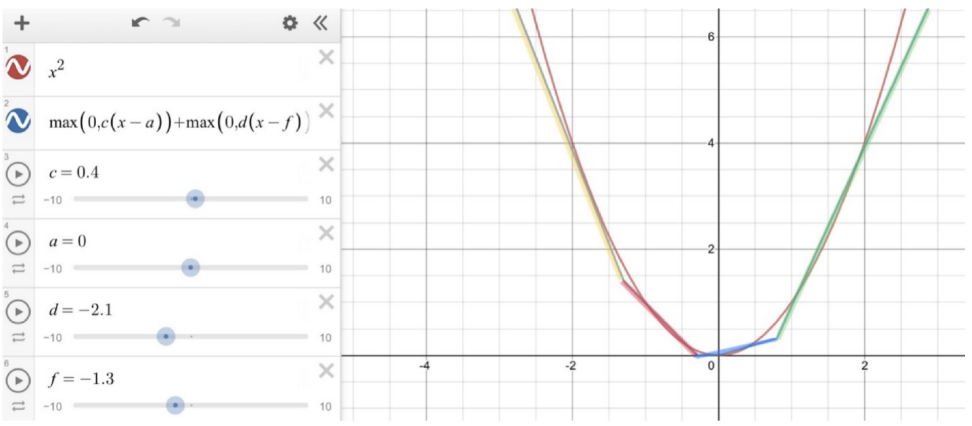
Nonlinearities: Activation functions like ReLU, tanh, softplus etc.



ReLU (Rectified Linear Unit)



Their composition allows one to approximate a generic function



Why are neural networks so powerful?

Universal approximation theorem:

a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (e.g. sigmoid) can approximate any Borel measurable function from one finite-dimensional space to another (hence any continuous function on a closed and bounded subset of the reals) with any desired non-zero amount of error, provided there are enough hidden units.

Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

Neural Networks, Vol. 2, pp. 359–366, 1989
Printed in the USA. All rights reserved.

ORIGINAL CONTRIBUTION

Multilayer Feedforward Networks are Universal Approximators

KURT HORNIK
Technische Universität Wien

Hornik1989.pdf (page 1 of 8)

MAXWELL STINCHCOMBE AND HALBERT WHITE
University of California, San Diego

(*Received* 16 *September* 1988; *revised and accepted* 9 *March* 1989)

Abstract—*This paper rigorously establishes that standard multilayer feedforward networks with as few as one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available. In this sense, multilayer feedforward networks are a class of universal approximators.*

Additional results for ReLU

On Approximation Capabilities of ReLU Activation and Softmax Output Layer in Neural Networks

Behnam Asadi
Department of Electrical Engineering and Computer Science
York University, 4700 Keele Street, Toronto, ON M3J1P3, Canada
benasadi@cse.yorku.ca

Hui Jiang
Department of Electrical Engineering and Computer Science
York University, 4700 Keele Street, Toronto, ON M3J1P3, Canada
hj@cse.yorku.ca

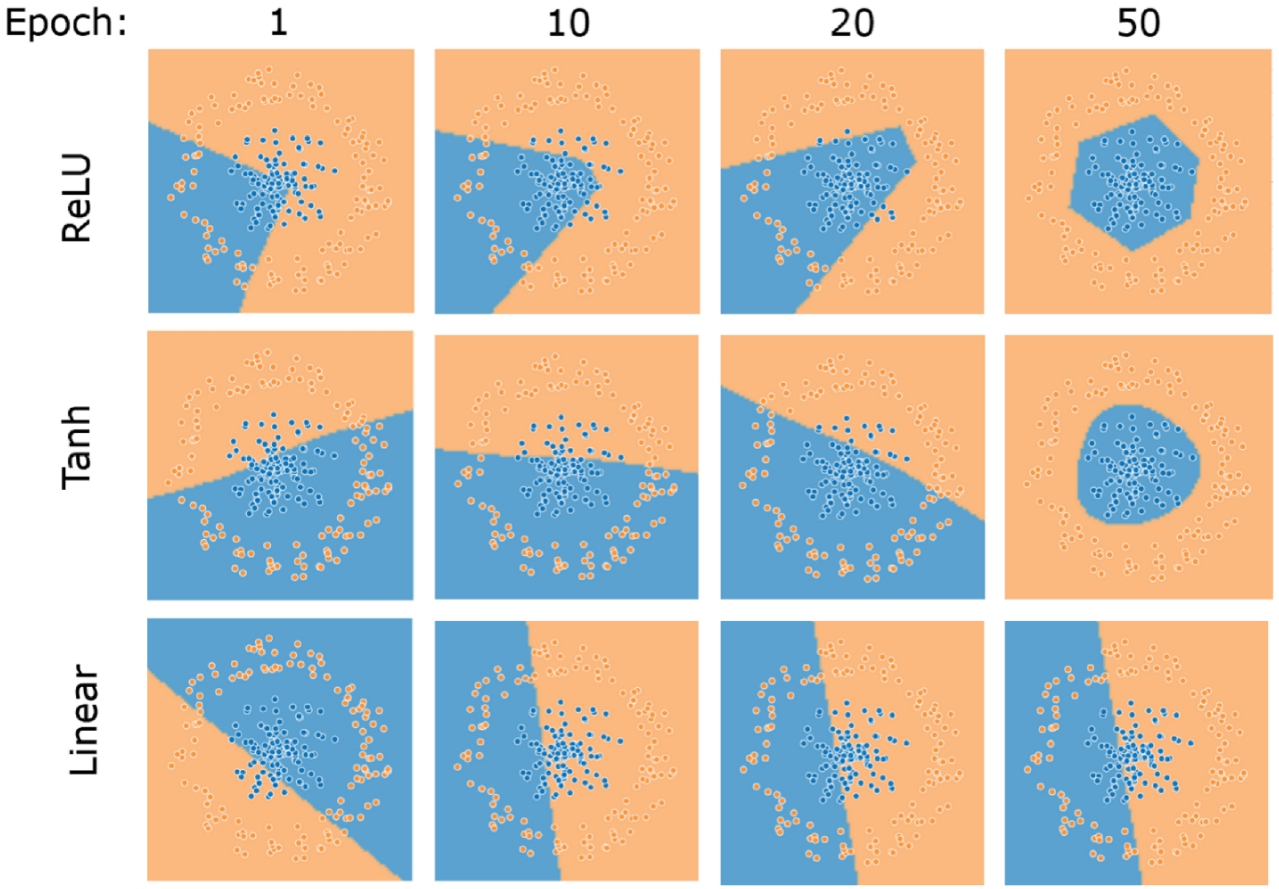
Abstract

In this paper, we have extended the well-established universal approximator theory to neural networks that use the unbounded ReLU activation function and a nonlinear softmax output layer. We have proved that a sufficiently large neural network using the ReLU activation function can approximate any function in L^1 up to any arbitrary precision. Moreover, our theoretical results have shown that a large enough neural network using a nonlinear softmax output layer can also approximate any indicator function in L^1, which is equivalent to mutually-exclusive class labels in any realistic multiple-class pattern classification problems. To the best of our knowledge, this work is the first theoretical justification for using the softmax output layers in neural networks for pattern classification.

See Goodfellow, Bengio, Courville, *Deep Learning,* Chap. 6

Why are neural networks so powerful?

Example of binary classification problem: non-linear activation functions allow one to fit **non-linear decision boundaries**



Each neural network had three hidden layers with three units in each one. The only difference was the activation function. Learning rate: 0.03, regularization

See Goodfellow, Bengio, Courville, *Deep Learning,* Chap. 6

Why Deep Learning?

- Depth (with non-linear activation functions) produce high representational capacity

Why Deep Learning?

- Depth (with non-linear activation functions) produce high representational capacity

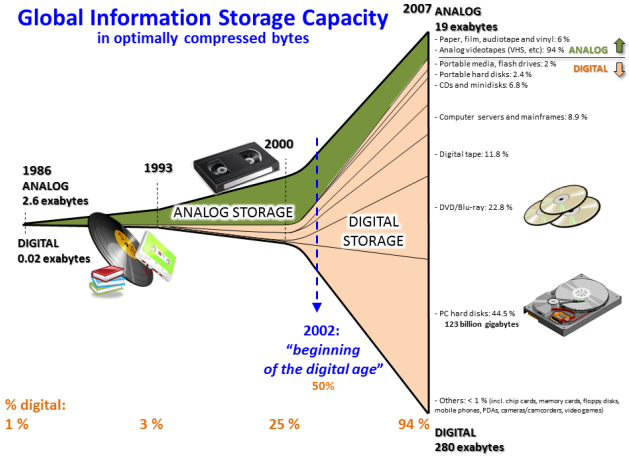
- They are overparametrised, leading to computationally demanding and data-hungry training. Why now?

Why Deep Learning?

- Depth (with non-linear activation functions) produce high representational capacity

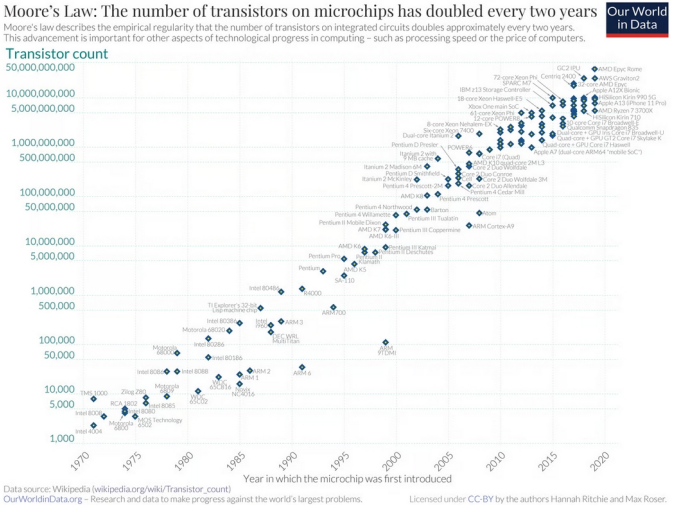
- They are overparametrised, leading to computationally demanding and data-hungry training. Why now?

Abundance of data



From Wikipedia

Computational power (GPU)



Why Deep Learning?

- Depth (with non-linear activation functions) produce high representational capacity

- However: the increase in number of parameters makes DL prone to overfitting. We need... **Regularisation!**

1. Penalties on weights

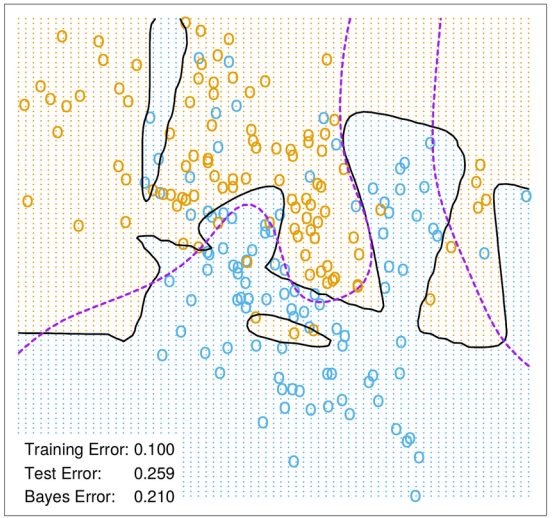
Penalties on weights: ℓ^2 regularization (weight decay)

see Hastie, Tibshirani, Friedman, *Elements of statistical learning,* chap. 11

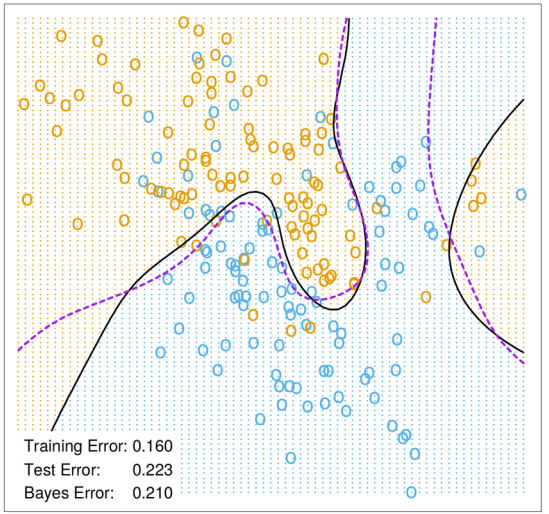
$$L(\mathbf{w}, \alpha) = L_0(\mathbf{w}) + \alpha_2 \sum_i w_i^2$$

Akin to Ridge regression!

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



Penalties on weights: ℓ^2 regularization (weight decay)

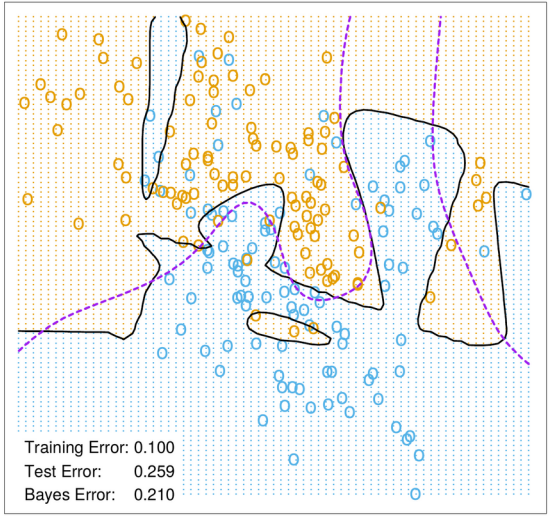
see Hastie, Tibshirani, Friedman, *Elements of statistical learning,* chap. 11

$$L(\mathbf{w}, \alpha) = L_0(\mathbf{w}) + \alpha_2 \sum_i w_i^2$$

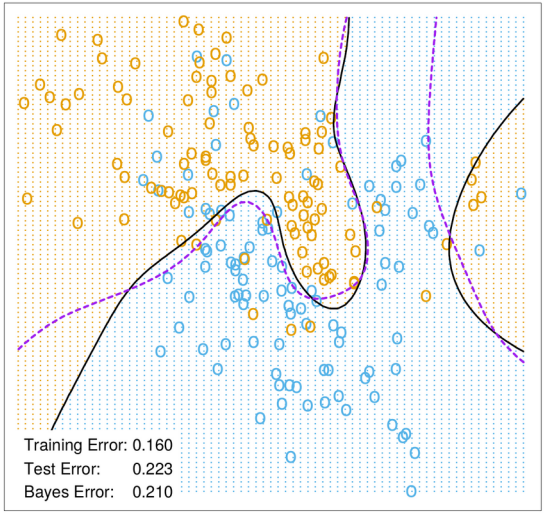
Akin to Ridge regression!

It prevents overfitting by shrinking weights' magnitude

Neural Network - 10 Units, No Weight Decay

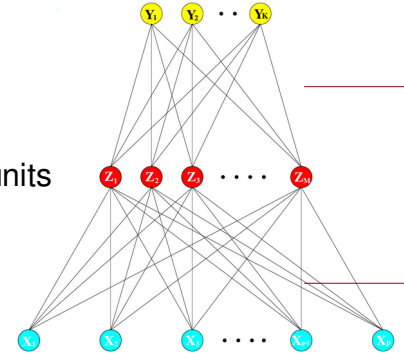


Neural Network - 10 Units, Weight Decay=0.02

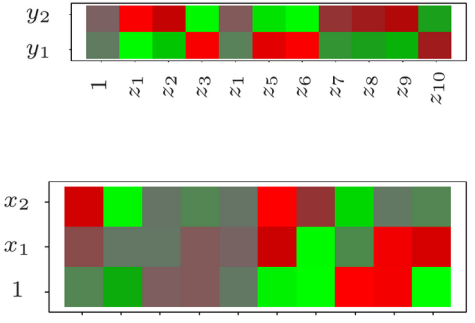


Classifier: 2 classes

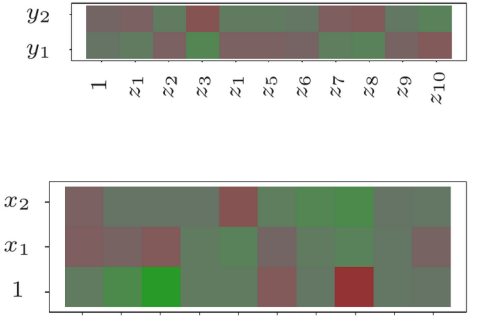
1 layer - 10 hidden units



No weight decay



Weight decay



Why Deep Learning?

- Depth (with non-linear activation functions) produce high representational capacity

- However: the increase in number of parameters makes DL prone to overfitting. We need... **Regularisation!**

1. Penalties on weights
2. Early stopping

Early stopping

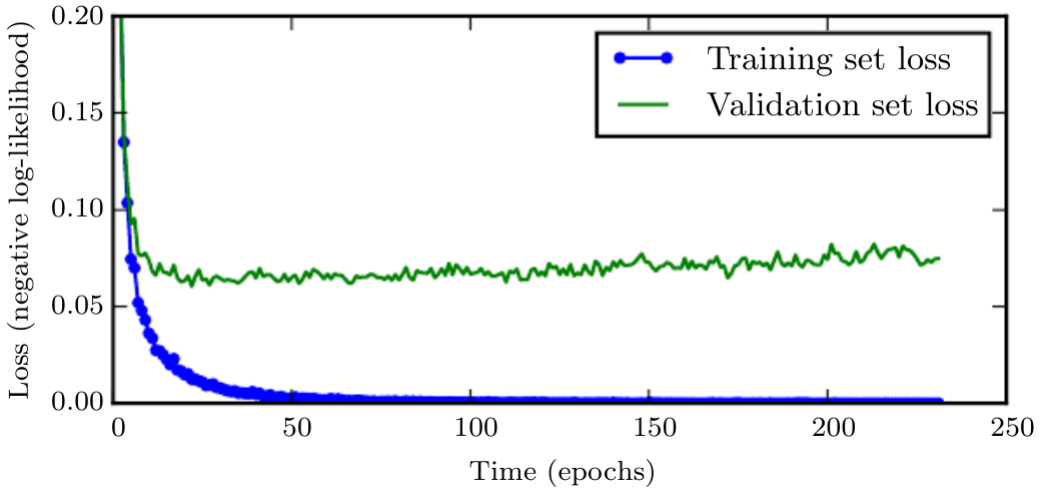


Image from Goodfellow, Bengio, Courville, *Deep Learning*

Early stopping: store the best performance (on validation) and terminate if no improvement

Early stopping

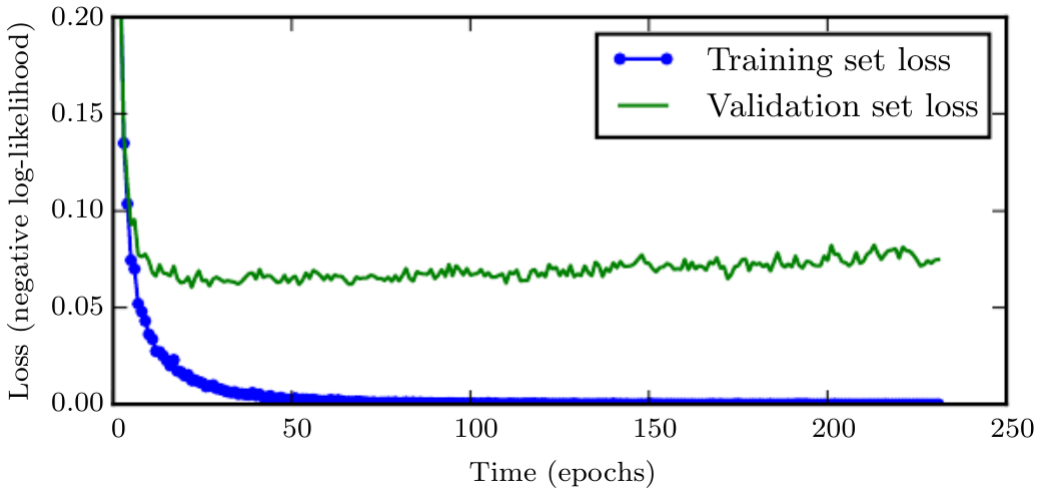


Image from Goodfellow, Bengio, Courville, *Deep Learning*

Early stopping: store the best performance (on validation) and terminate if no improvement

Early stopping is set by a hyper-parametric search over the number of epochs

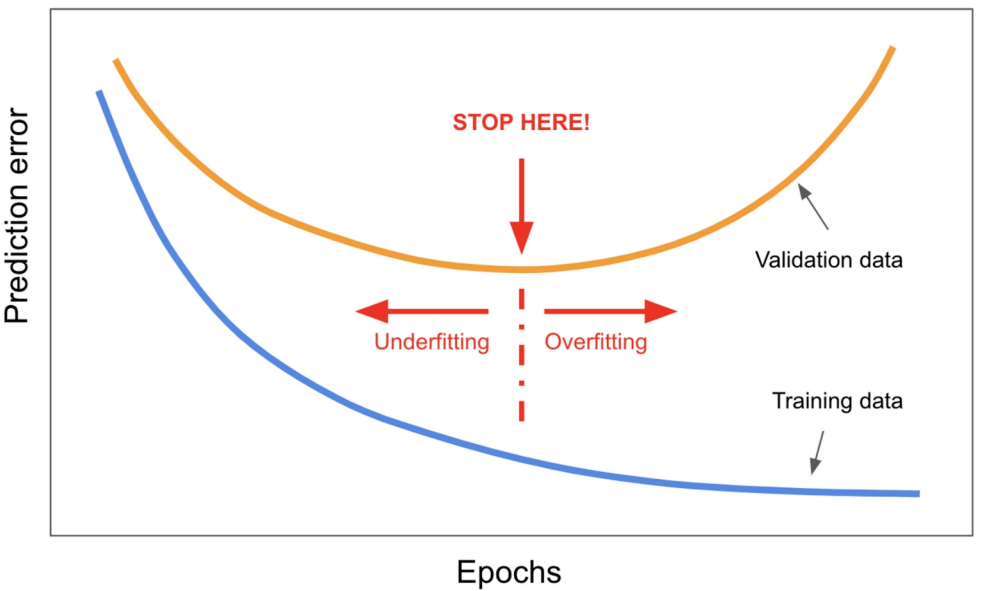


Figure 8.15. Prediction error vs number of training epochs

Early stopping

Given the validation loss can be noisy, in practice one: sets a **patience** (steps with no improvement before stopping) + saves model 'checkpoints'

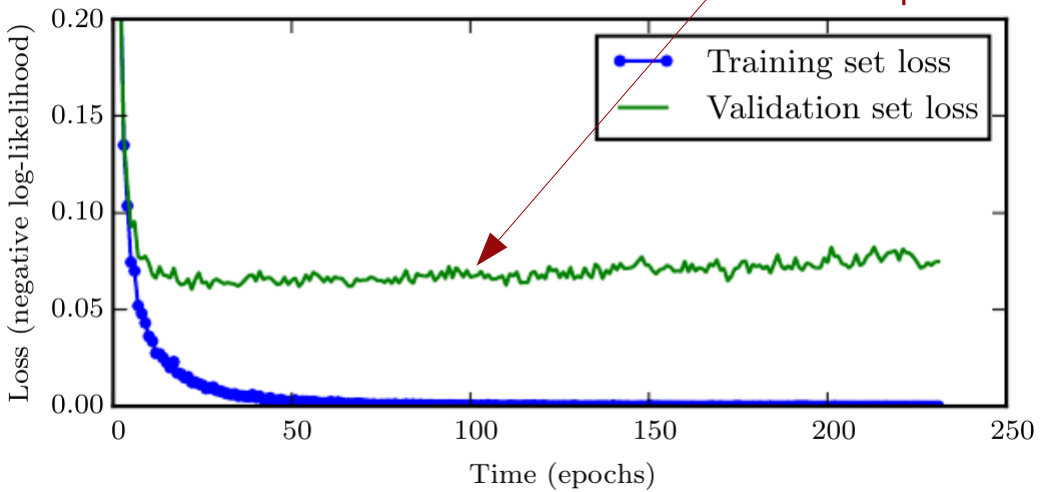


Image from Goodfellow, Bengio, Courville, *Deep Learning*

Early stopping: store the best performance (on validation) and terminate if no improvement

Early stopping is set by a hyper-parametric search over the number of epochs

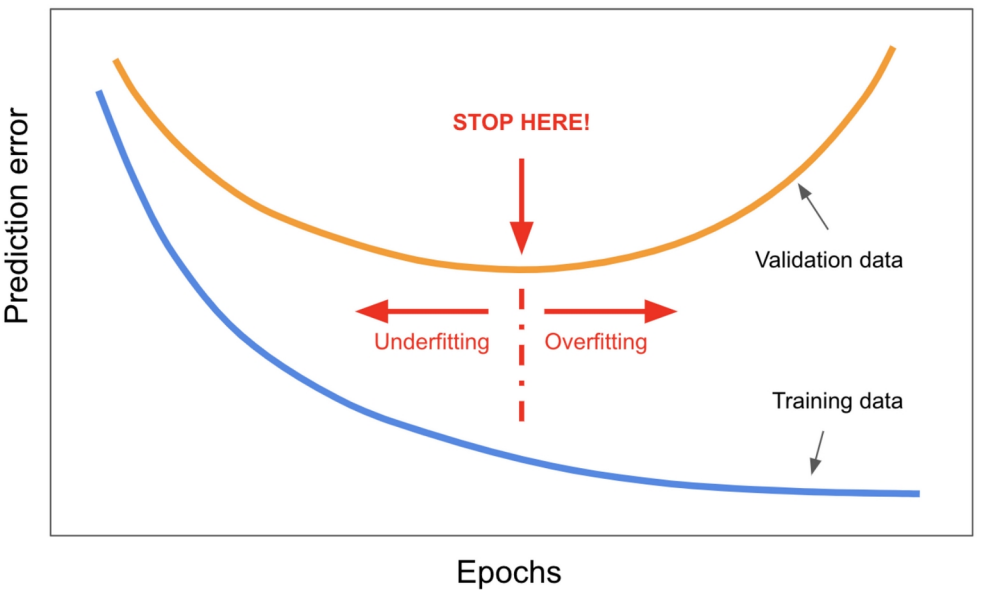


Figure 8.15. Prediction error vs number of training epochs

Early stopping

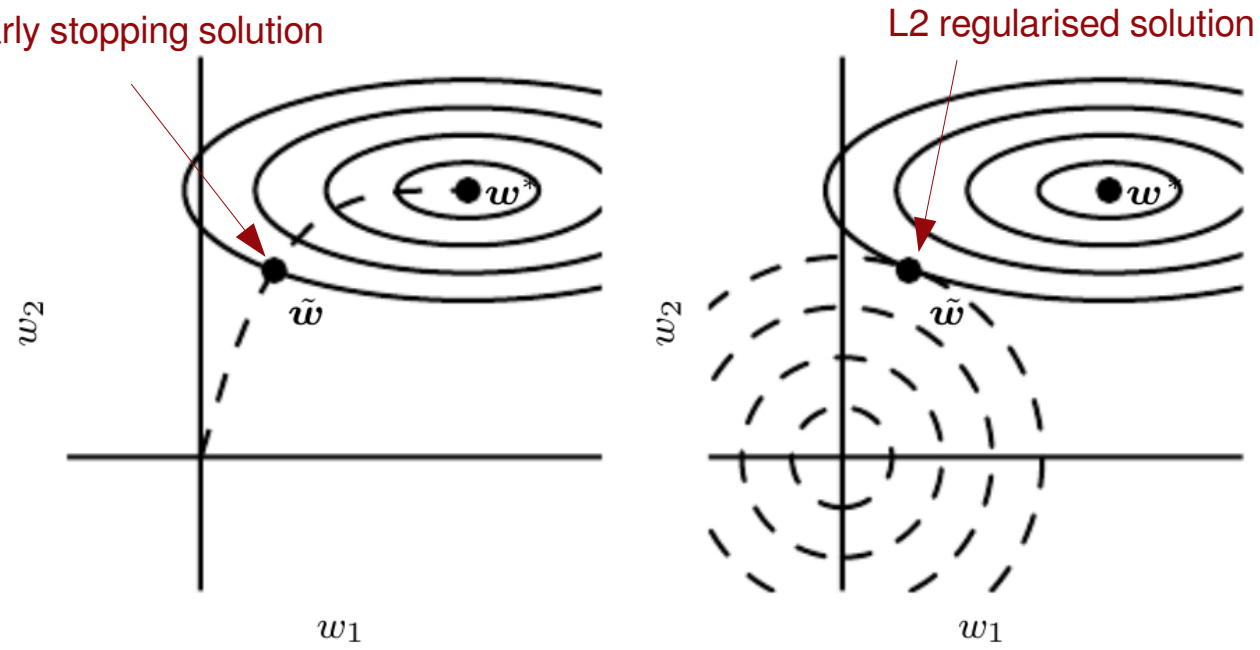


Image from Goodfellow, Bengio, Courville, *Deep Learning*

Early stopping is a form of regularisation of the weights of the neural network

In particular, it's equivalent to L2

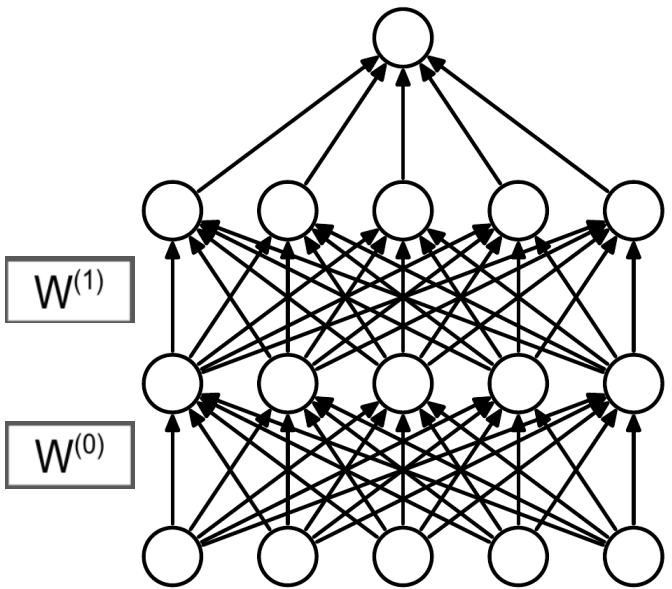
Why Deep Learning?

- Depth (with non-linear activation functions) produce high representational capacity

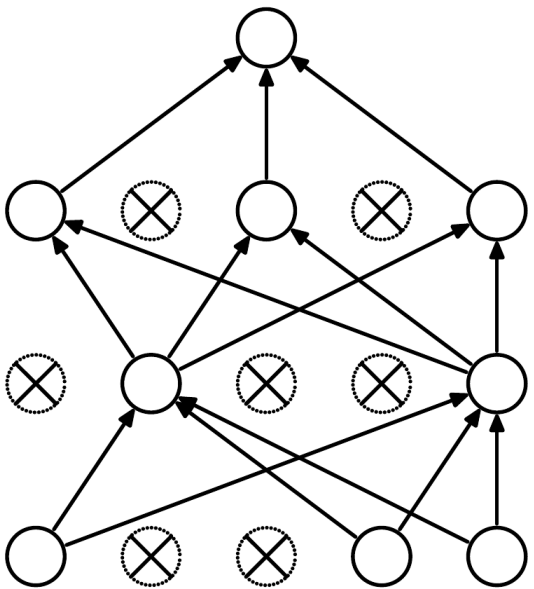
- However: the increase in number of parameters makes DL prone to overfitting. We need... **Regularisation!**

1. Penalties on weights
2. Early stopping
3. Dropout

A new type of regularization: **Dropout**



(a) Standard Neural Net

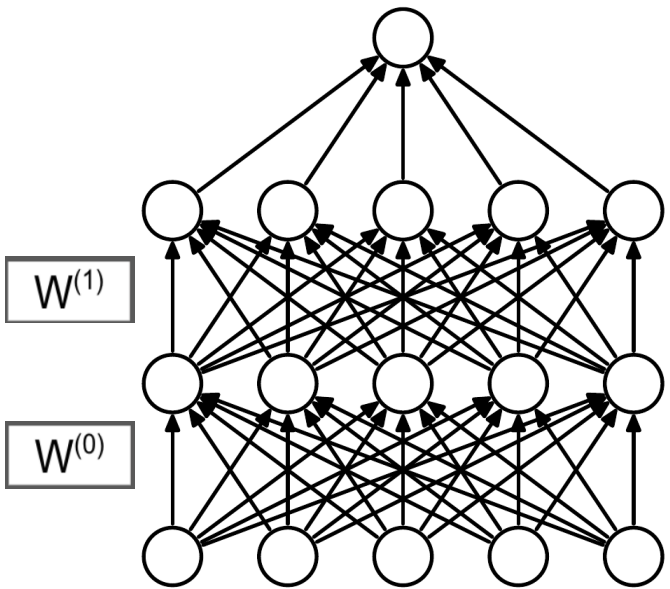


(b) After applying dropout.

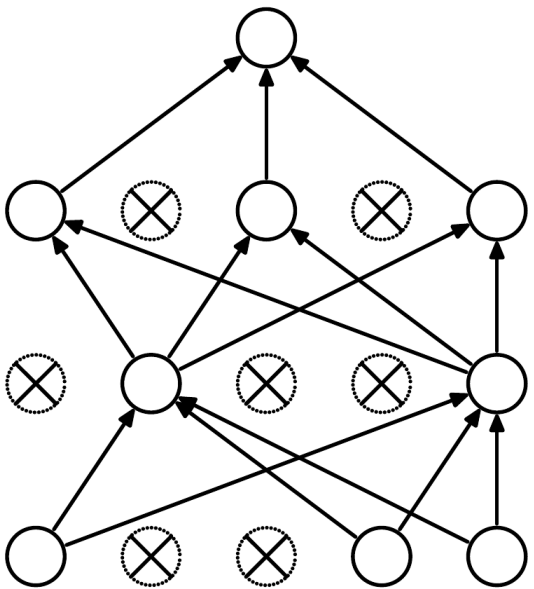
Randomly zero out neurons (+ weights) at each iteration with a certain dropout rate

Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

A new type of regularization: **Dropout**



(a) Standard Neural Net



(b) After applying dropout.

Randomly zero out neurons (+ weights) at each iteration with a certain dropout rate

Mathematically:

Mask

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} \cdot \mathrm{diag}(\mathbf{z}^l)$$

$$z_j^l \sim \mathrm{Bernoulli}(p_l)$$

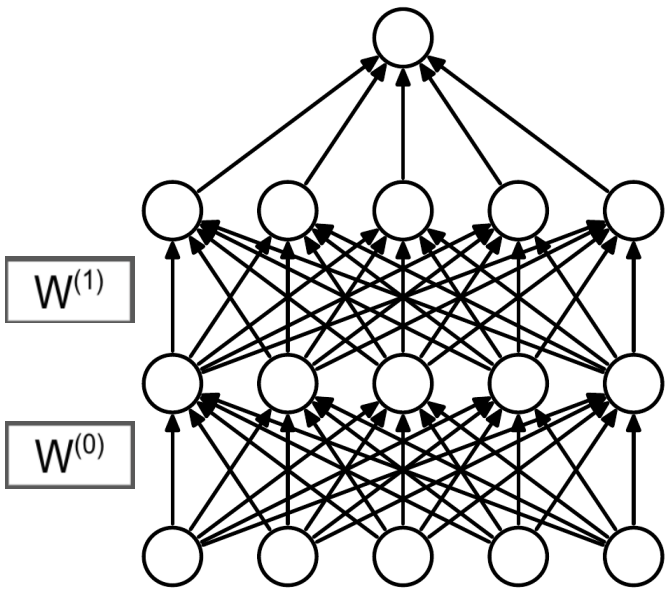
Keep layer *l* units with probability p_l

Dropout rate: $1 - p_l$

Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

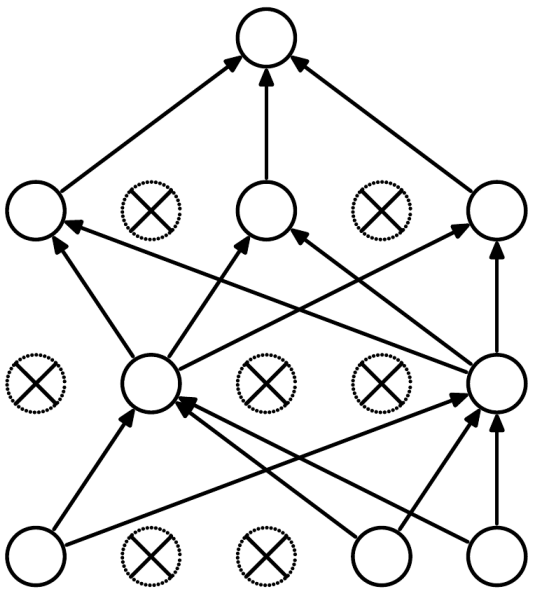
A new type of regularization: **Dropout**

Final structure when *testing* the network



(a) Standard Neural Net

At each *training* step



(b) After applying dropout.

Randomly zero out neurons (+ weights) at each iteration with a certain dropout rate

Mathematically: Mask

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} \cdot \mathrm{diag}(\mathbf{z}^l)$$

$$z_j^l \sim \mathrm{Bernoulli}(p_l)$$

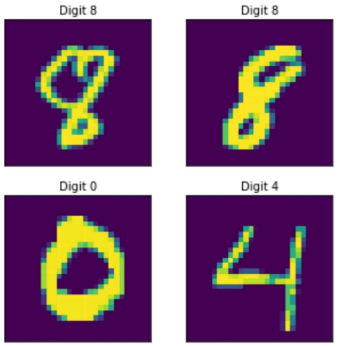
Keep layer *l* units with probability p_l

Dropout rate: $1 - p_l$

Thus: need to **rescale** weights by the probability by which they are kept

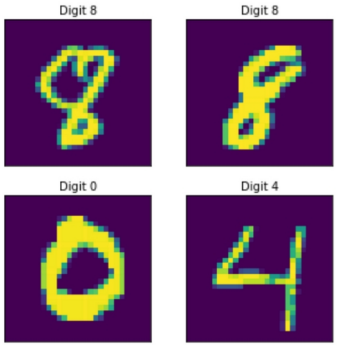
Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

MNIST dataset: collection of handwritten digits
Benchmark for digit recognition



Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

MNIST dataset: collection of handwritten digits
Benchmark for digit recognition



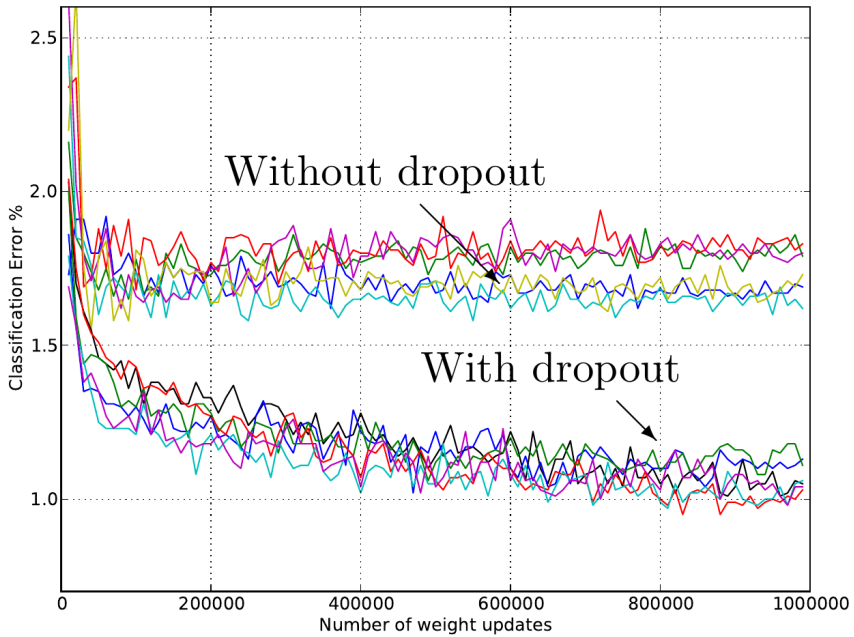


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Result: dropout improves accuracy across datasets and architectures

Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

Internal data representation: **pattern of weights**

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

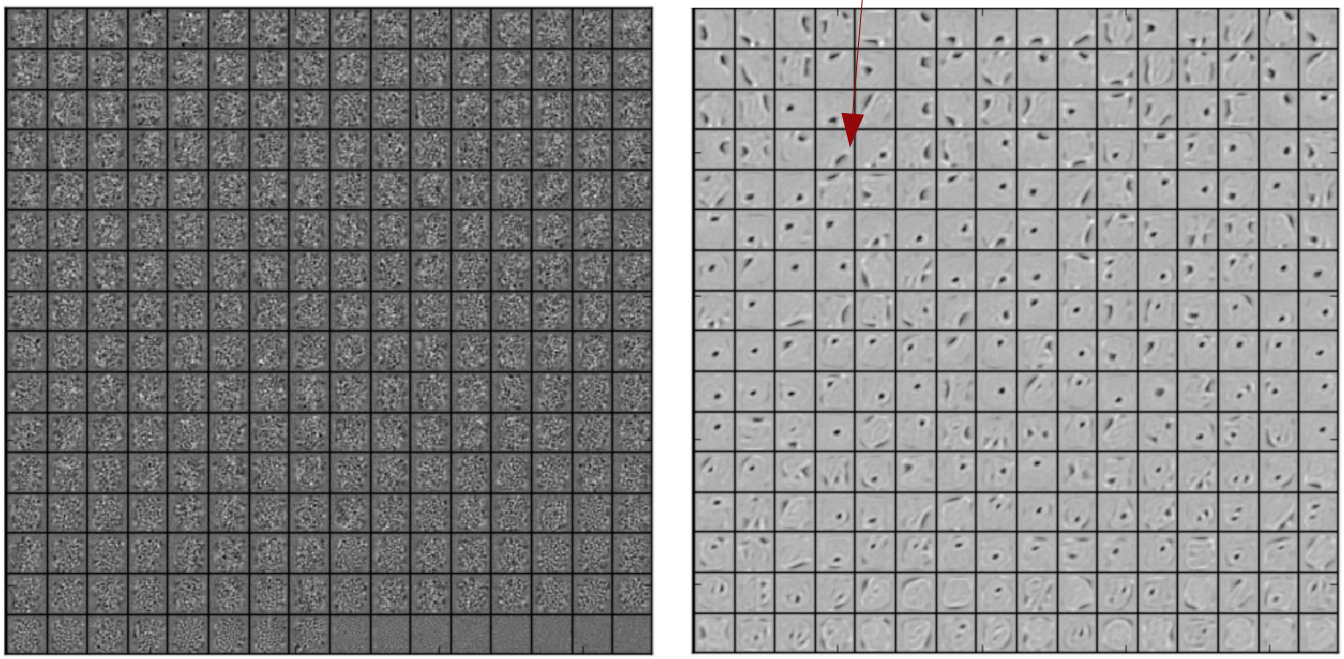
Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

Internal data representation: **pattern of weights**

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

Different sets of weights detect different features (spots, strokes, edges)

Dropout prevents hidden units (HU) from co-adaptation



(a) Without dropout (b) Dropout with $p = 0.5$.

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.

Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

Internal data representation: **pattern of hidden units activations**

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

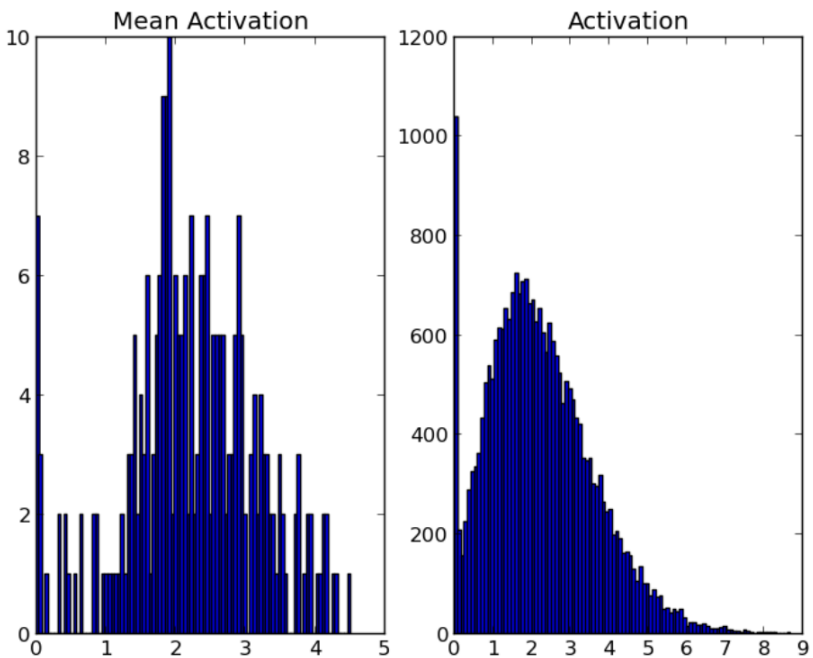
Internal data representation: **pattern of hidden units activations**

$$\mathbf{h}^{(1)} = \sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

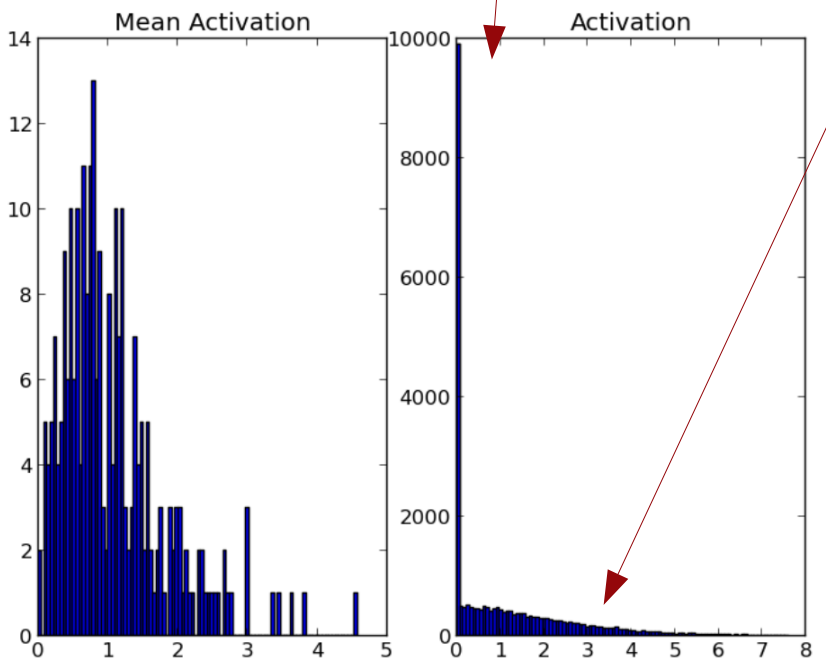
For a given image, **many HU have a few zero activation**

A few have non-zero activation
(the ones connected to the weights detecting the image's features)

Effect on Sparsity



(a) Without dropout



(b) Dropout with $p = 0.5$.

Srivastava et al., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (2014)

1. Depth increases the representational capacity of models, essential for very complex tasks (*but:* increases time and memory cost of training, need many data)

More on practical methodology (effect of hyper-parameters, grid search, debugging...): see Goodfellow, Bengio, Courville, *Deep Learning,* Chap. 11

1. Depth increases the representational capacity of models, essential for very complex tasks
(*but:* increases time and memory cost of training, need many data)

2. Several strategies to prevent overfitting (weight decay, early stopping, dropout)
- Dropout, peculiar of architectures with hidden units, avoids their co-adaptation, leading to more robust and interpretable data representations
(*but:* the gain in accuracy may depend on the size of the training set)

More on practical methodology (effect of hyper-parameters, grid search, debugging...): see Goodfellow, Bengio, Courville, *Deep Learning,* Chap. 11

1. Depth increases the representational capacity of models, essential for very complex tasks
(*but:* increases time and memory cost of training, need many data)

2. Several strategies to prevent overfitting (weight decay, early stopping, dropout)
- Dropout, peculiar of architectures with hidden units, avoids their co-adaptation, leading to more robust and interpretable data representations
(*but:* the gain in accuracy may depend on the size of the training set)

3. Learning rate is crucial and needs to be tuned/adapted – otherwise, optimisation failure

More on practical methodology (effect of hyper-parameters, grid search, debugging...):
see Goodfellow, Bengio, Courville, *Deep Learning,* Chap. 11

1. Depth increases the representational capacity of models, essential for very complex tasks
(*but:* increases time and memory cost of training, need many data)

2. Several strategies to prevent overfitting (weight decay, early stopping, dropout)
- Dropout, peculiar of architectures with hidden units, avoids their co-adaptation, leading to more robust and interpretable data representations
(*but:* the gain in accuracy may depend on the size of the training set)

Always need to assess benefits on your dataset

3. Learning rate is crucial and needs to be tuned/adapted – otherwise, optimisation failure

More on practical methodology (effect of hyper-parameters, grid search, debugging...): see Goodfellow, Bengio, Courville, *Deep Learning,* Chap. 11

1. Depth increases the representational capacity of models, essential for very complex tasks
(*but:* increases time and memory cost of training, need many data)

2. Several strategies to prevent overfitting (weight decay, early stopping, dropout)
- Dropout, peculiar of architectures with hidden units, avoids their co-adaptation, leading to more robust and interpretable data representations
(*but:* the gain in accuracy may depend on the size of the training set)

Always need to assess benefits on your dataset

3. Learning rate is crucial and needs to be tuned/adapted – otherwise, optimisation failure

4. Finding the 'best' architecture is a potentially **demanding task of hyper-parametric search,** involving: number and width of hidden layers, activations, regularisations *jointly*

More on practical methodology (effect of hyper-parameters, grid search, debugging...):
see Goodfellow, Bengio, Courville, *Deep Learning,* Chap. 11