UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Data Analysis Lab

ICT/Electronic Lab

Lecture 03: Neural networks

SPEAKER
Prof. Antonio Ferramosca

PLACE
University of Bergamo

Outline

1. Introduction to neural networks
2. Feed-forward neural networks
3. Training neural networks
    a. Cost function
    b. Backpropagation
    c. Stochastic Gradient Descent
    d. Regularization in neural networks
4. Comparison of supervised learning methods

Outline

1. **Introduction to neural networks**
2. Feed-forward neural networks
3. Training neural networks
    a. Cost function
    b. Backpropagation
    c. Stochastic Gradient Descent
    d. Regularization in neural networks
4. Comparison of supervised learning methods

From linear/logistic regression to neural networks

Neural networks are a powerful alternative to linear regression, logistic regression and decision trees when model interpretability is **not** of primary concern

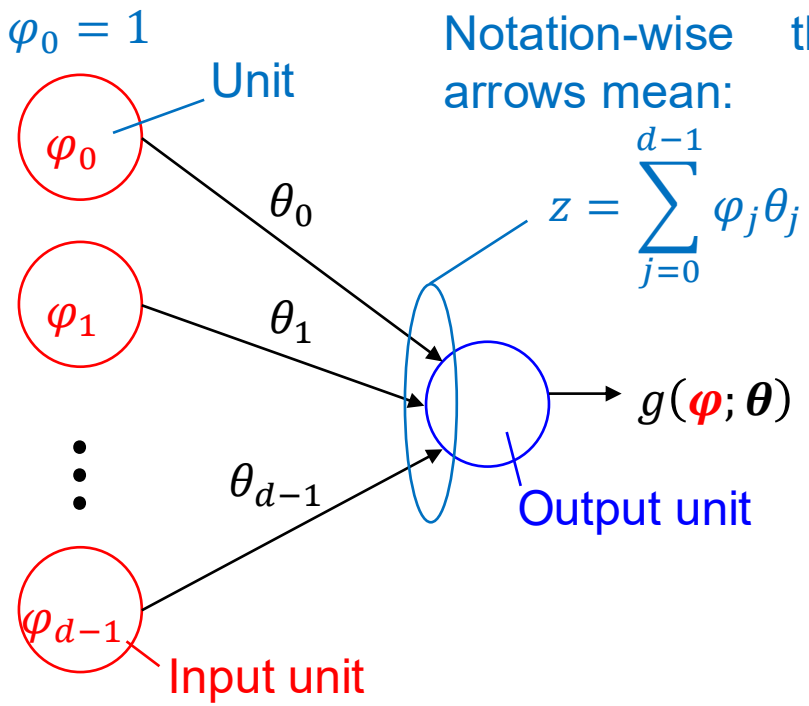
Nowadays, neural networks are the state-of-the-art in many data science tasks that involve **unstructured data** (e.g. in computer vision, natural language processing, …)

Audio files, text files, video files, image files, …

In practice, linear regression and logistic regression models can be viewed as simple neural networks

From linear/logistic regression to neural networks

We can represent linear regression and logistic regression models using a **network diagram**



In practice, linear regression and logistic regression estimate either the **output** or the **probability of belonging to the positive class** by applying a suitable function $g(\boldsymbol{\varphi}; \boldsymbol{\theta})$ to the linear combination $z = \boldsymbol{\varphi}^\top \boldsymbol{\theta}$

Linear regression $y \approx g(\boldsymbol{\varphi}; \boldsymbol{\theta}) = \boldsymbol{\varphi}^\top \boldsymbol{\theta}$ (Identity function)

Logistic regression $P(y = 1|\boldsymbol{\varphi}) \approx g(\boldsymbol{\varphi}; \boldsymbol{\theta}) = \frac{1}{1 + e^{-\boldsymbol{\varphi}^\top \boldsymbol{\theta}}}$ (Sigmoid)

$\boldsymbol{\theta}$ is found by minimizing a suitable cost function, obtaining the estimate $\widehat{\boldsymbol{\theta}}$

From linear/logistic regression to neural networks

What if the data-generating function $f(\boldsymbol{\varphi})$ is **nonlinear** in $\boldsymbol{\varphi}$?

- In practice, we can still use linear regression and logistic regression to learn a nonlinear $f(\boldsymbol{\varphi})$ by applying suitable **nonlinear transformations or basis functions to the inputs**, for example:

$$\underbrace{\begin{bmatrix}\varphi_1\\ \varphi_2\end{bmatrix}}_{\boldsymbol{\varphi}} \Rightarrow \underbrace{\begin{bmatrix}\varphi_1\\ \varphi_2\\ \varphi_1^2\\ \varphi_1\varphi_2\\ \varphi_2^2\\ \varphi_1^3\\ \vdots\end{bmatrix}}_{\boldsymbol{\psi}(\boldsymbol{\varphi})}$$

Polynomial feature transformation

How do we select the nonlinear transformations/basis function $\boldsymbol{\psi}(\boldsymbol{\varphi})$?

- One option is to use a very complex $\boldsymbol{\psi}(\boldsymbol{\varphi})$. However, we risk **overfitting** the data
- Alternatively, we could **manually engineer** $\boldsymbol{\psi}(\boldsymbol{\varphi})$ based on the task at hand (very time consuming)

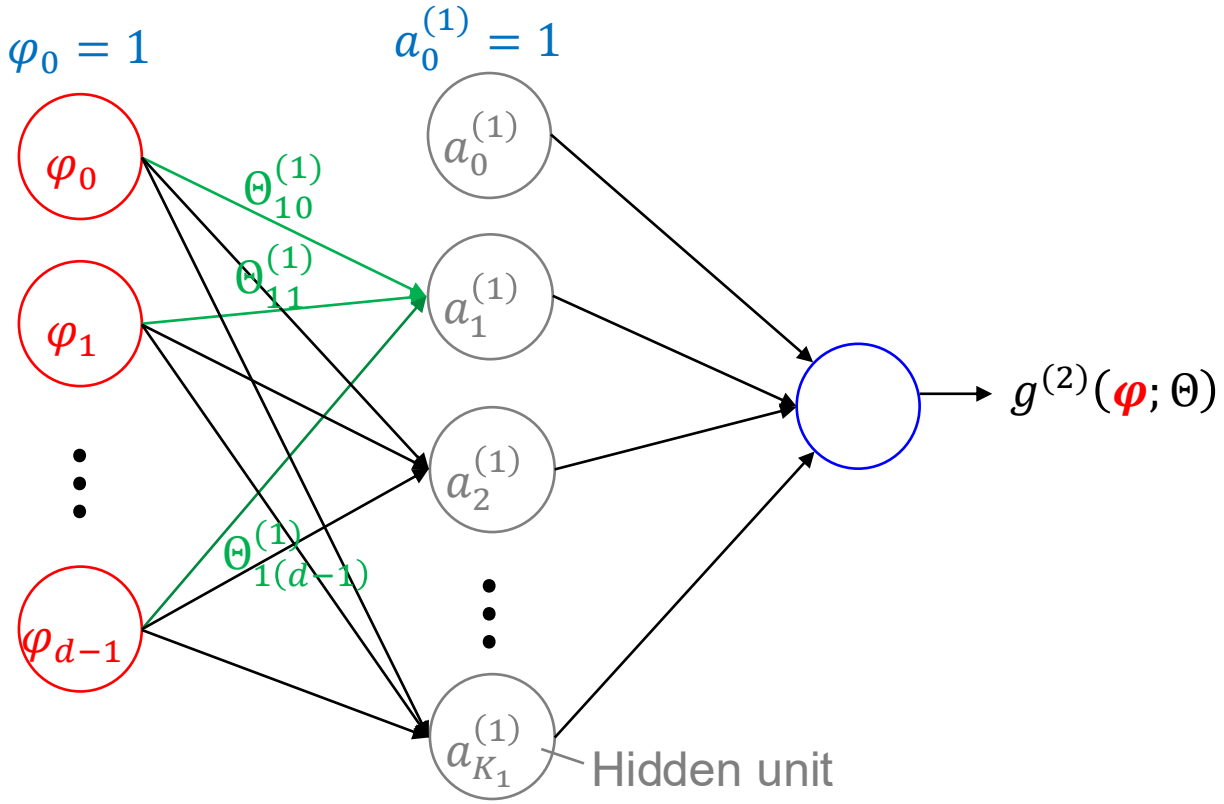
⇒ **Neural networks** provide a way of learning nonlinear functions without the need to define $\boldsymbol{\psi}(\boldsymbol{\varphi})$

Outline

1. Introduction to neural networks
2. **Feed-forward neural networks**
3. Training neural networks
    a. Cost function
    b. Backpropagation
    c. Stochastic Gradient Descent
    d. Regularization in neural networks
4. Comparison of supervised learning methods

Feed-forward neural networks – single output

A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows



$$a_1^{(1)} = g^{(1)}\left(\Theta_{10}^{(1)}\varphi_0 + \Theta_{11}^{(1)}\varphi_1 + \cdots + \Theta_{1(d-1)}^{(1)}\varphi_{d-1}\right)$$

$$= g^{(1)}\left(\sum_{j=0}^{d-1}\Theta_{1j}^{(1)}\varphi_j\right) = g^{(1)}\left(z_1^{(1)}\right)$$

$\Theta_{kj}^{(l)}$

- l: Index of the next layer (where the connections are going to)
- j: Index of the unit from the previous layer
- k: Index of the unit in the next layer

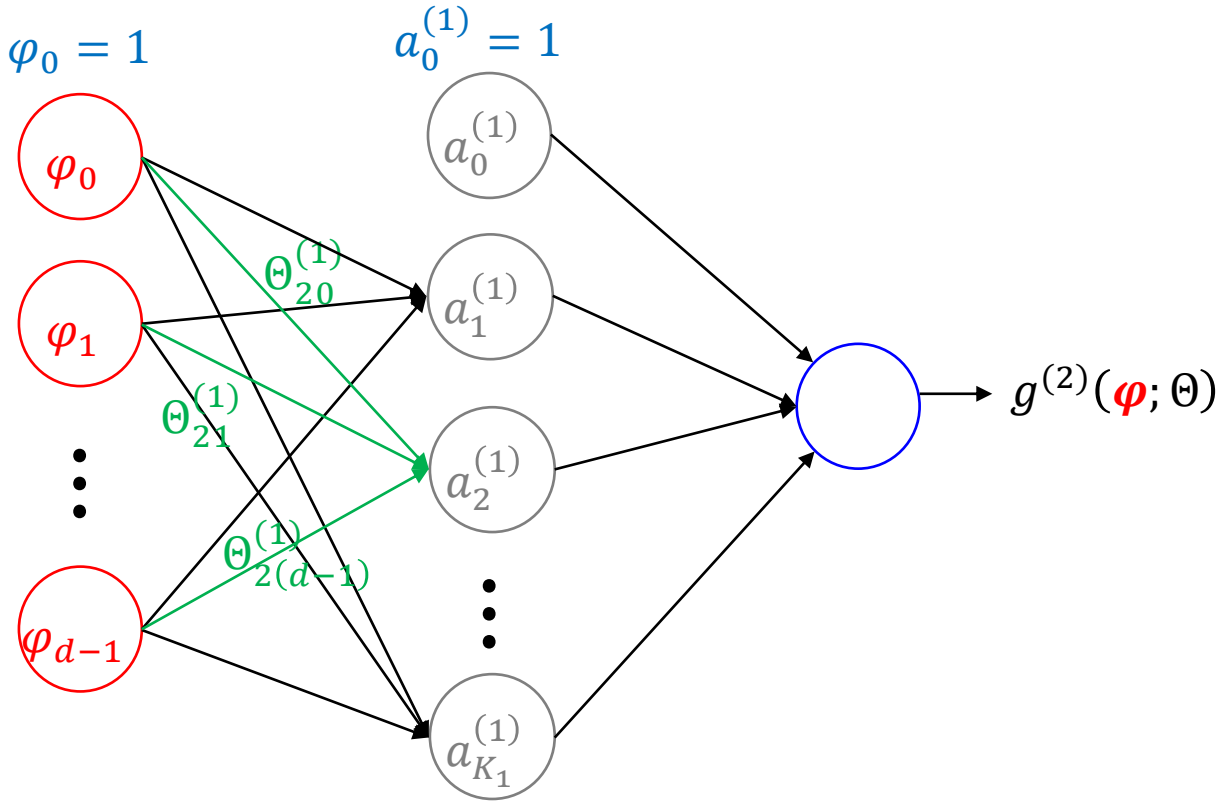
$g^{(l)}$: **activation function** of the l-th layer

$a_k^{(l)}$: k-th **derived feature** in the l-th layer

$z_k^{(l)}$: linear combination from which the k-th **derived feature** in the l-th layer is computed

Feed-forward neural networks – single output

A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows



Input layer | **Hidden layer 1** | **Output layer**

$$a_1^{(1)} = g^{(1)}\left(\Theta_{10}^{(1)}\varphi_0 + \Theta_{11}^{(1)}\varphi_1 + \cdots + \Theta_{1(d-1)}^{(1)}\varphi_{d-1}\right)$$

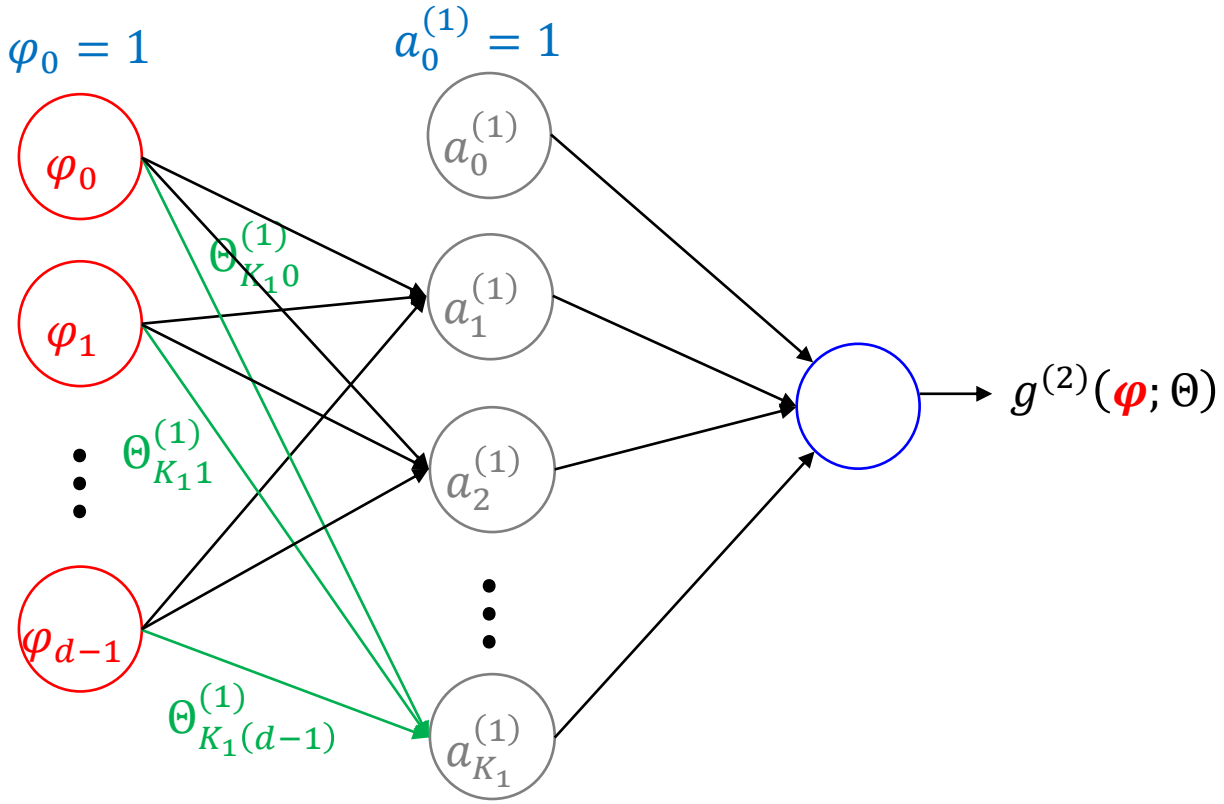
$$= g^{(1)}\left(\sum_{j=0}^{d-1}\Theta_{1j}^{(1)}\varphi_j\right) = g^{(1)}\left(z_1^{(1)}\right)$$

$$a_2^{(1)} = g^{(1)}\left(\Theta_{20}^{(1)}\varphi_0 + \Theta_{21}^{(1)}\varphi_1 + \cdots + \Theta_{2(d-1)}^{(1)}\varphi_{d-1}\right)$$

$$= g^{(1)}\left(\sum_{j=0}^{d-1}\Theta_{2j}^{(1)}\varphi_j\right) = g^{(1)}\left(z_2^{(1)}\right)$$

Feed-forward neural networks – single output

A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows



$$a_1^{(1)} = g^{(1)}\left(\Theta_{10}^{(1)}\varphi_0 + \Theta_{11}^{(1)}\varphi_1 + \cdots + \Theta_{1(d-1)}^{(1)}\varphi_{d-1}\right)$$

$$= g^{(1)}\left(\sum_{j=0}^{d-1}\Theta_{1j}^{(1)}\varphi_j\right) = g^{(1)}\left(z_1^{(1)}\right)$$

$$a_2^{(1)} = g^{(1)}\left(\Theta_{20}^{(1)}\varphi_0 + \Theta_{21}^{(1)}\varphi_1 + \cdots + \Theta_{2(d-1)}^{(1)}\varphi_{d-1}\right)$$

$$= g^{(1)}\left(\sum_{j=0}^{d-1}\Theta_{2j}^{(1)}\varphi_j\right) = g^{(1)}\left(z_2^{(1)}\right)$$

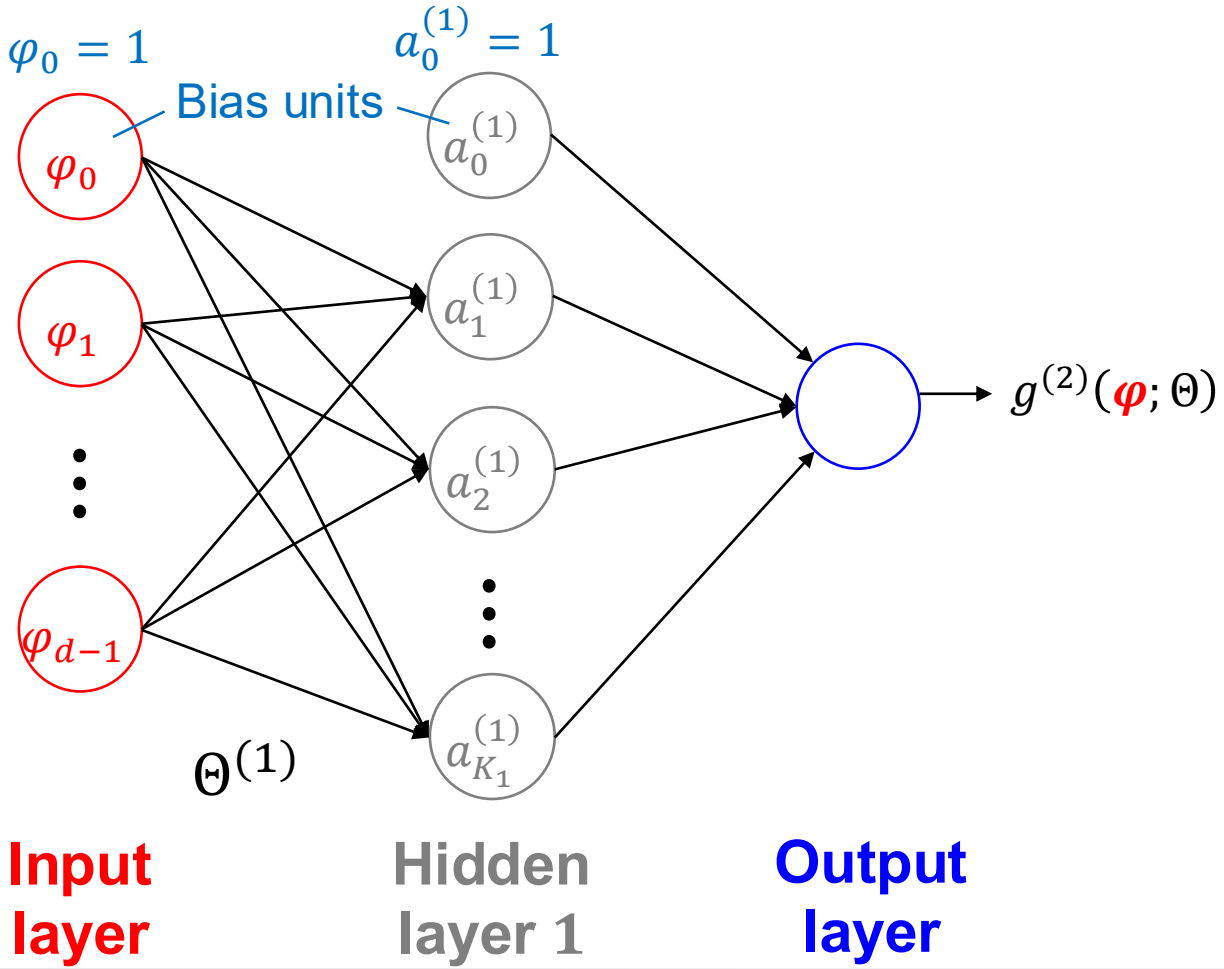
$$\vdots$$

$$a_{K_1}^{(1)} = g^{(1)}\left(\Theta_{K_1 0}^{(1)}\varphi_0 + \Theta_{K_1 1}^{(1)}\varphi_1 + \cdots + \Theta_{K_1 (d-1)}^{(1)}\varphi_{d-1}\right)$$

$$= g^{(1)}\left(\sum_{j=0}^{d-1}\Theta_{K_1 j}^{(1)}\varphi_j\right) = g^{(1)}\left(z_{K_1}^{(1)}\right)$$

Feed-forward neural networks – single output

A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows

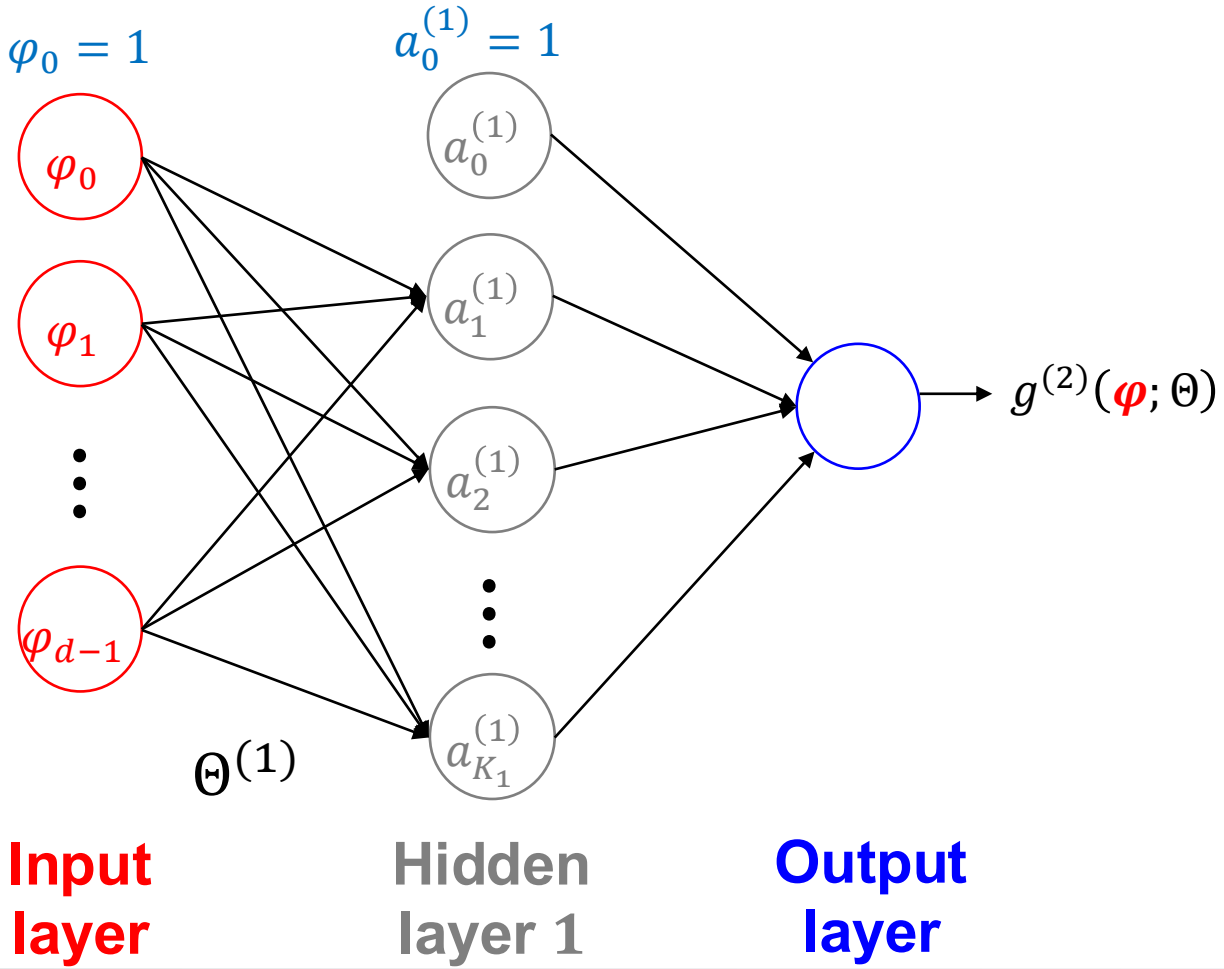


Overall, we have a parameter $\Theta_{kj}^{(1)} \in \mathbb{R}$ associated to **each** connection from the units in the **input layer** to the units in **hidden layer 1**

- $\Theta^{(1)} \in \mathbb{R}^{K_1 \times d}$ is the **parameters matrix** of **hidden layer 1** (we need to estimate this)
  - $\Theta_{k0}^{(1)}, 1 \leq k \leq K_1$, are called **biases**
  - $\Theta_{kj}^{(1)}, 1 \leq k \leq K_1, 1 \leq j \leq d-1$, are the **weights**

Feed-forward neural networks – single output

A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows



- We can view each $a_k^{(1)}, 1 \leq k \leq K_1$, as **"new" features** generated by the neural network by applying a suitable **nonlinear** activation function $g^{(1)}(\cdot)$ to several linear combinations $z_k^{(1)}$ of the inputs $\boldsymbol{\varphi}$

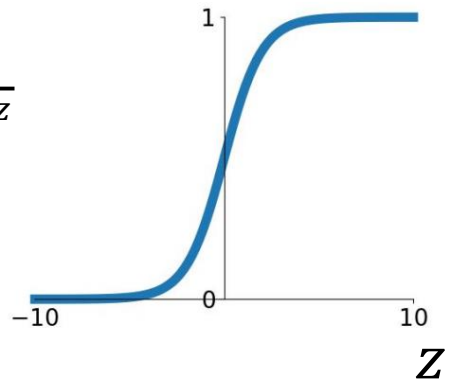
- Since **each** unit in **hidden layer 1** is connected to **each** unit in the **input layer**, **hidden layer 1** is referred to as a **fully connected layer**

Activation functions for the hidden units

Some commonly used activation functions $g^{(1)}(z)$ for the **hidden layers** are:

Sigmoid

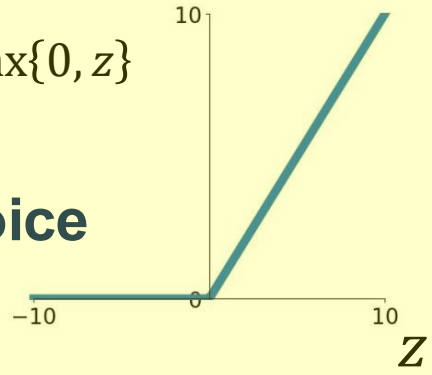
$$g^{(1)}(z) = \frac{1}{1 + e^{-z}}$$



ReLU

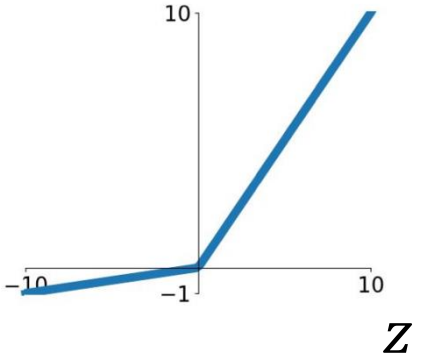
$$g^{(1)}(z) = \max\{0, z\}$$

Default choice



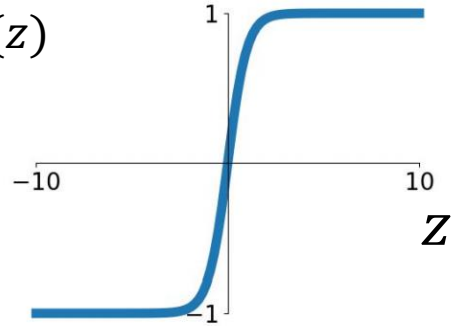
Leaky ReLU

$$g^{(1)}(z) = \max\{0.1z, z\}$$



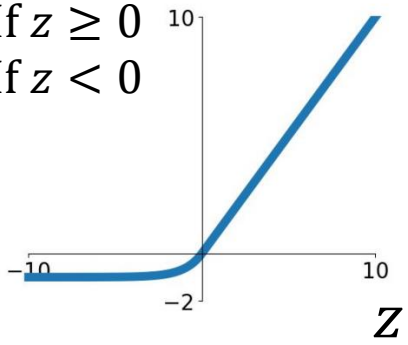
Hyperbolic tangent

$$g^{(1)}(z) = \tanh(z)$$



ELU

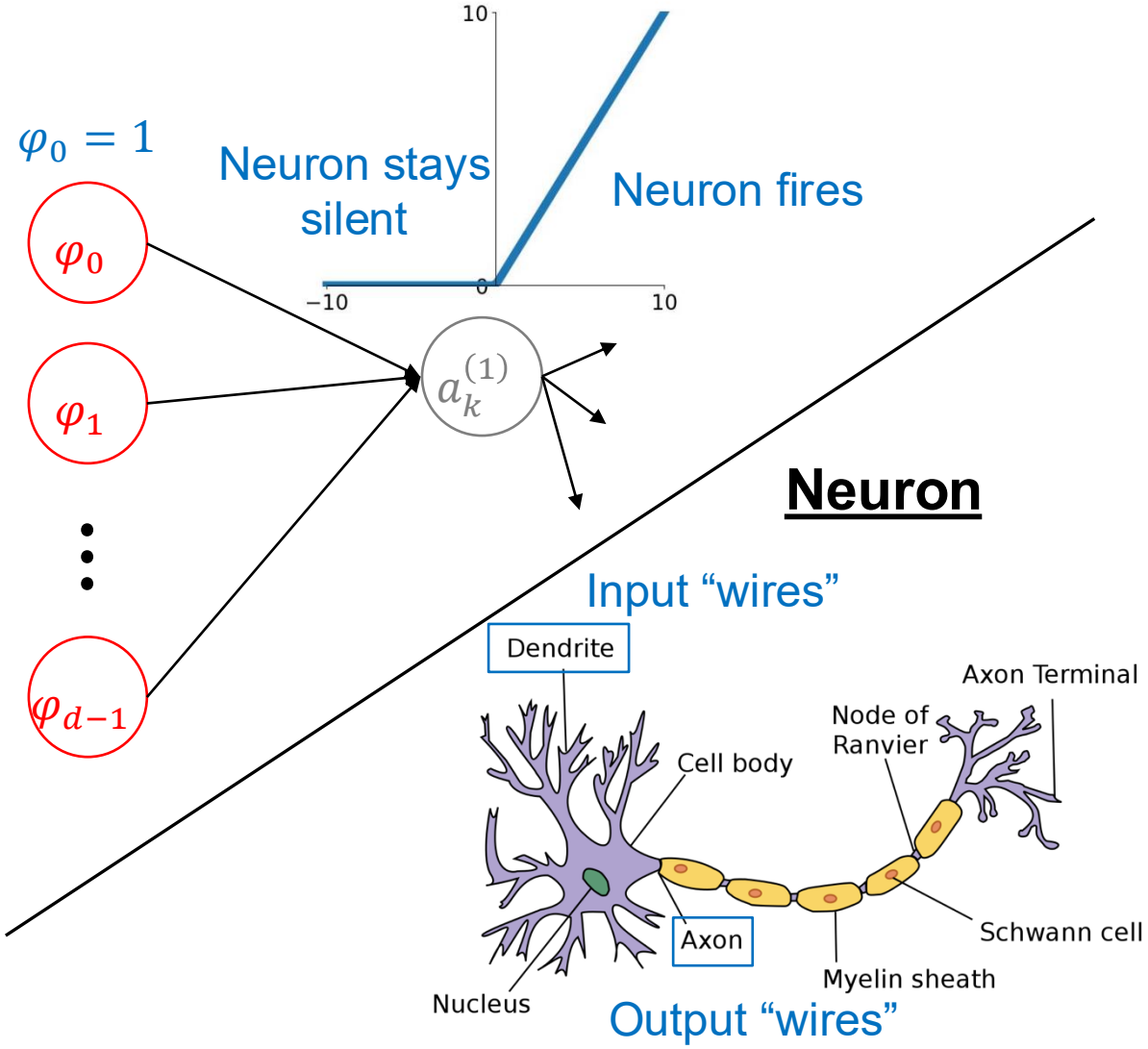
$$g^{(1)}(z) = \begin{cases} z & \text{If } z \geq 0 \\ \alpha(e^z - 1) & \text{If } z < 0 \end{cases}$$



Notice that all these functions are **nonlinear** in z!

Analogy to the human brain

These networks are called **neural** because they are loosely inspired by neuroscience [5]

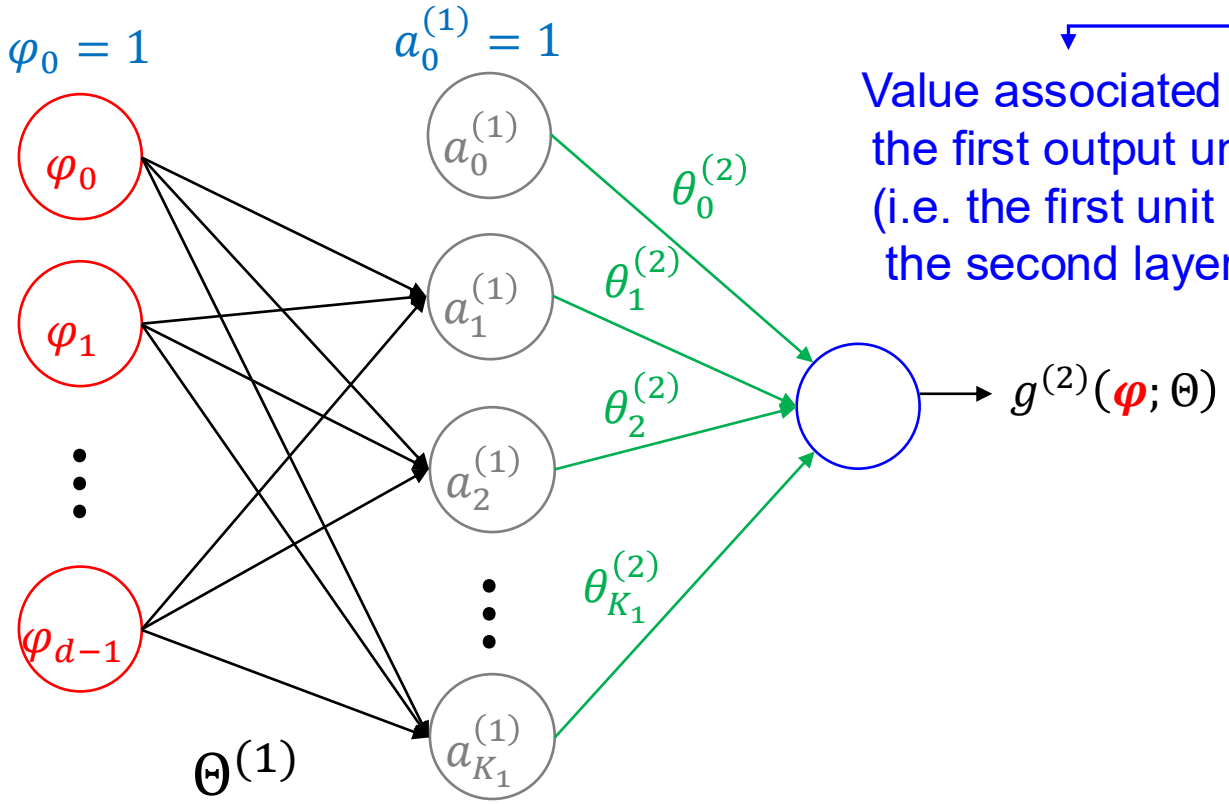


Picture taken from [5]

- Each unit of the network represents a **neuron**
- Neurons receive signals from neighboring neurons
- The action taken by a neuron depends on the strength of the signals received by other neurons (cf. the parameters $\Theta^{(1)}$ and the linear combinations of the inputs $\boldsymbol{\varphi}$)
- If the signals are "strong enough" the neuron **fires**, otherwise it **stays silent**

Feed-forward neural networks – single output

A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows



Value associated to the first output unit (i.e. the first unit of the second layer)

$$a_1^{(2)} = g^{(2)}\left(\theta_0^{(2)} a_0^{(1)} + \theta_1^{(2)} a_1^{(1)} + \cdots + \theta_{K_1}^{(2)} a_{K_1}^{(1)}\right)$$

$$= g^{(2)}\left(\sum_{k=0}^{K_1} \theta_k^{(2)} a_k^{(1)}\right) = g^{(2)}\left(z_1^{(2)}\right)$$

Parameters of the second layer (output layer), $\boldsymbol{\theta}^{(2)} \in \mathbb{R}^{1\times(K_1+1)}$

"New" features generated by the network

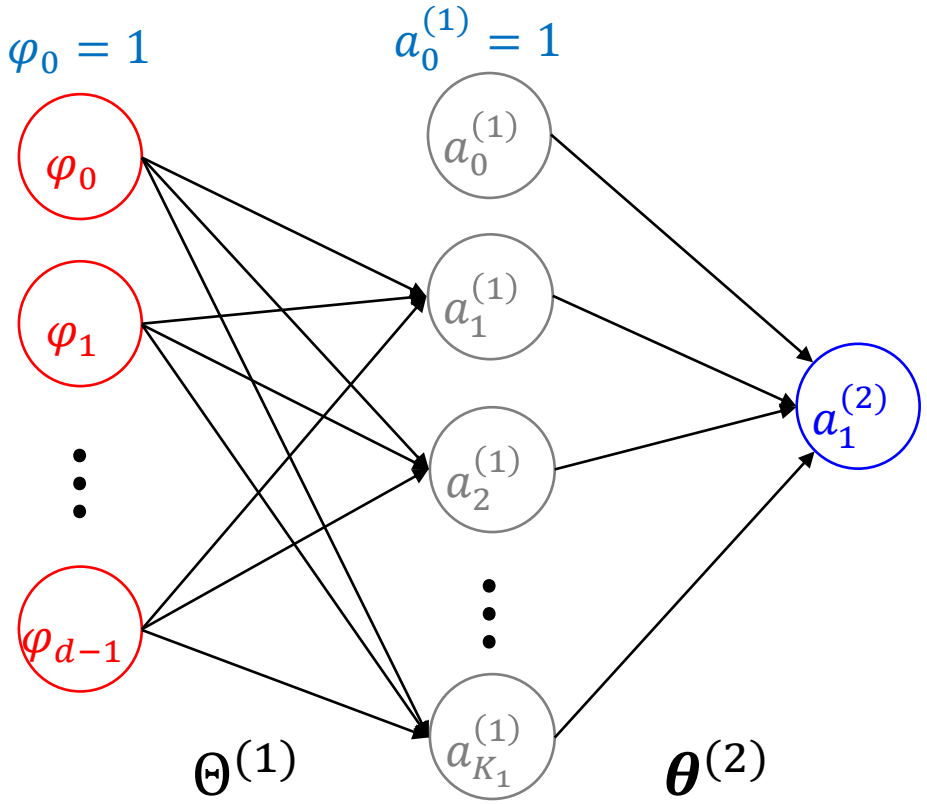
$$= g^{(2)}\left(\theta_0^{(2)} + \sum_{k=1}^{K_1} \theta_k^{(2)} g^{(1)}\left(\sum_{j=0}^{d-1} \Theta_{kj}^{(1)} \varphi_j\right)\right)$$

$$= g^{(2)}(\boldsymbol{\varphi}; \Theta)$$

Set of parameters associated to the connections of the network (in this case, $\Theta^{(1)}$ and $\boldsymbol{\theta}^{(2)}$) → needs to be estimated

Activation functions for the output unit

A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows



$$a_1^{(2)} = g^{(2)}\left(\sum_{k=0}^{K_1} \theta_k^{(2)} a_k^{(1)}\right) = g^{(2)}\left(z_1^{(2)}\right)$$

The activation function $g^{(2)}(\cdot)$ of the second layer of the network (i.e. the **output layer**) needs to be chosen **according to the data science task** that we are trying to solve:

- **Regression** → **identity function**:

$$g^{(2)}\left(z_1^{(2)}\right) = z_1^{(2)}$$

  - $a_1^{(2)}$ is the predicted output y associated to the input $\boldsymbol{\varphi}$
- **Binary classification** → **sigmoid**:

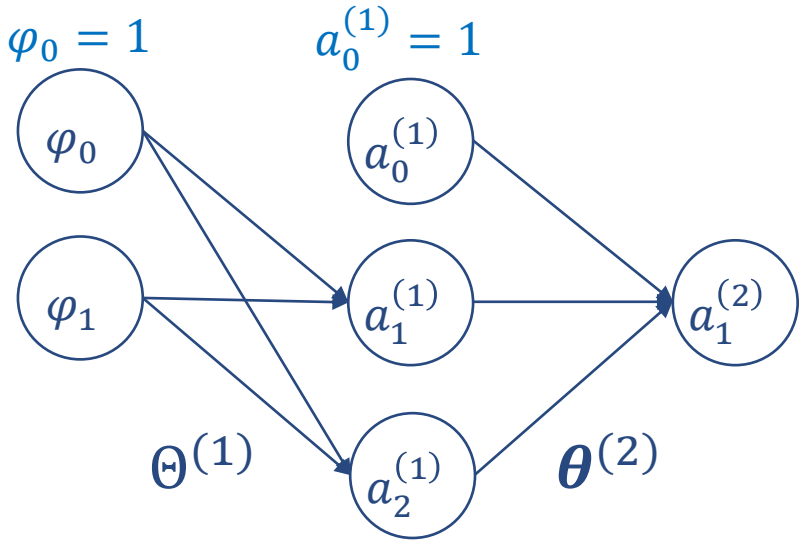
$$g^{(2)}\left(z_1^{(2)}\right) = \frac{1}{1+e^{-z_1^{(2)}}}$$

  - $a_1^{(2)}$ is the estimated probability of belonging to the positive class $P(y = 1|\boldsymbol{\varphi})$

Check out again the network diagram for linear and logistic regression at the beginning of this lecture!

Example: why are the activation functions nonlinear?

Consider the following neural network with a **linear activation function** $g^{(1)}\left(z_k^{(1)}\right) = z_k^{(1)}$ in the first hidden layer and for a **regression** task → $g^{(2)}\left(z_1^{(2)}\right) = z_1^{(2)}$ (identity function)



$$a_1^{(1)} = g^{(1)}\left(\Theta_{10}^{(1)}\varphi_0 + \Theta_{11}^{(1)}\varphi_1\right) = \Theta_{10}^{(1)} + \Theta_{11}^{(1)}\varphi_1$$

$$a_2^{(1)} = g^{(1)}\left(\Theta_{20}^{(1)}\varphi_0 + \Theta_{21}^{(1)}\varphi_1\right) = \Theta_{20}^{(1)} + \Theta_{21}^{(1)}\varphi_1$$

$$a_1^{(2)} = g^{(2)}\left(\theta_0^{(2)}a_0^{(1)} + \theta_1^{(2)}a_1^{(1)} + \theta_2^{(2)}a_2^{(1)}\right) = \theta_0^{(2)} + \theta_1^{(2)}a_1^{(1)} + \theta_2^{(2)}a_2^{(1)}$$

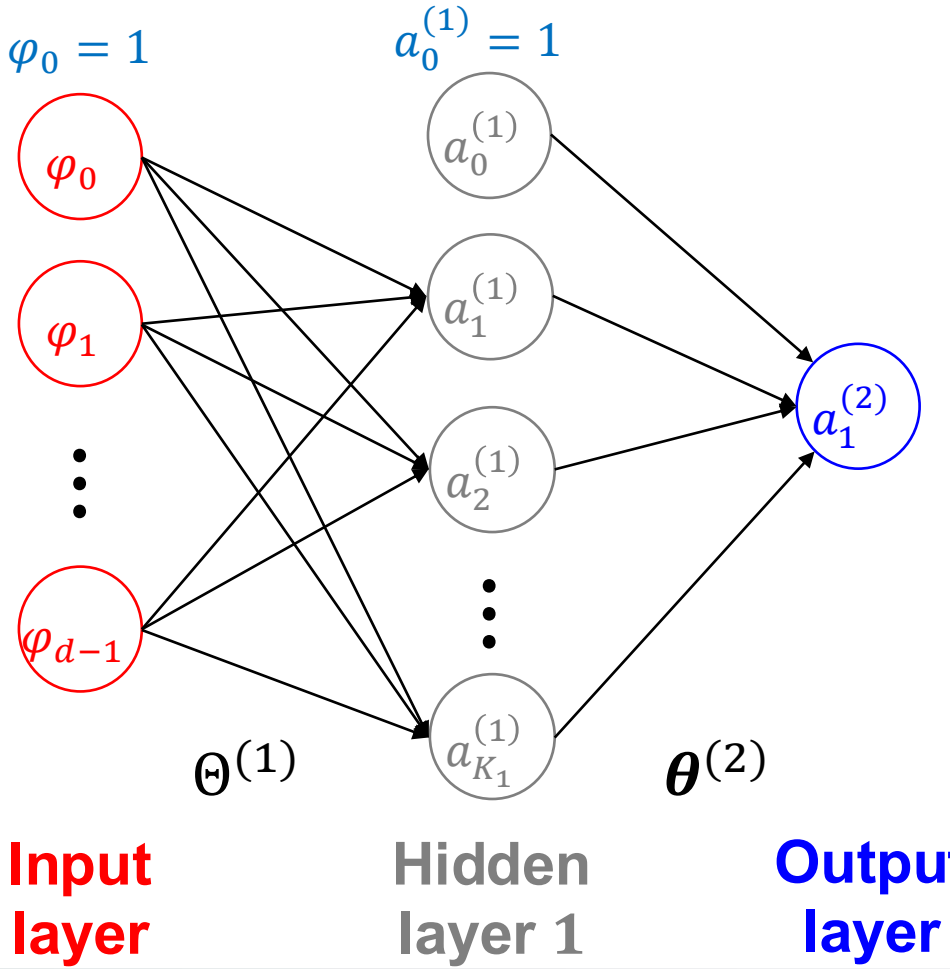
$$= \theta_0^{(2)} + \theta_1^{(2)}\left[\Theta_{10}^{(1)} + \Theta_{11}^{(1)}\varphi_1\right] + \theta_2^{(2)}\left[\Theta_{20}^{(1)} + \Theta_{21}^{(1)}\varphi_1\right]$$

$$= \underbrace{\left[\theta_0^{(2)} + \theta_1^{(2)}\Theta_{10}^{(1)} + \theta_2^{(2)}\Theta_{20}^{(1)}\right]}_{\tilde{\theta}_0} + \underbrace{\left[\theta_1^{(2)}\Theta_{11}^{(1)} + \theta_2^{(2)}\Theta_{21}^{(1)}\right]}_{\tilde{\theta}_1}\varphi_1$$

The network collapses into a linear model!

Neural networks with multiple hidden layers

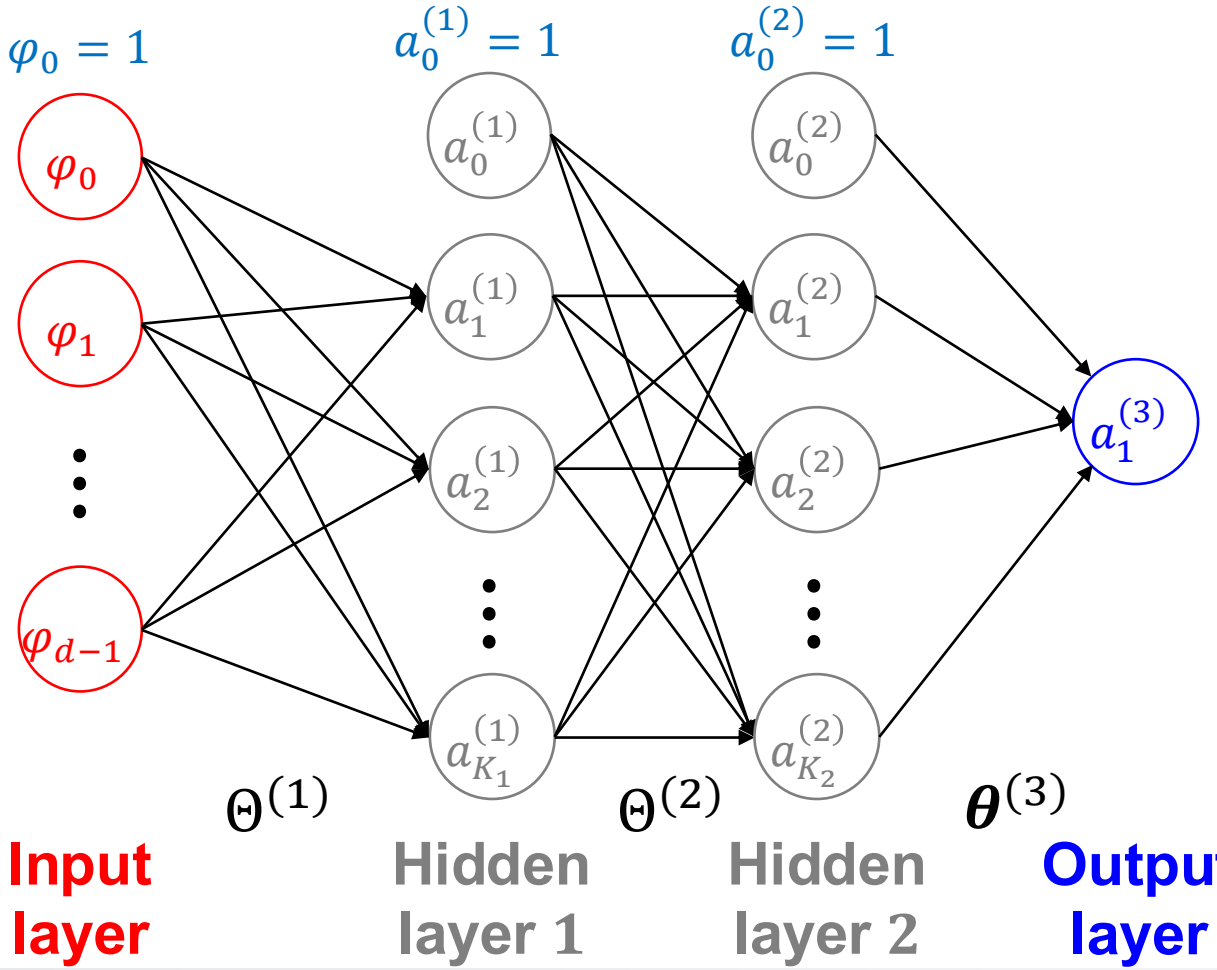
A **feed-forward Neural Network (NN)** with 1 hidden layer is defined as follows



- The just covered neural network is called a **two-layer network** (count the number of matrices/vectors of parameters)
- In practice, neural networks can have **multiple hidden layers**

Neural networks with multiple hidden layers

Consider now a **Neural Network (NN)** with multiple hidden layers



Three-layer network:

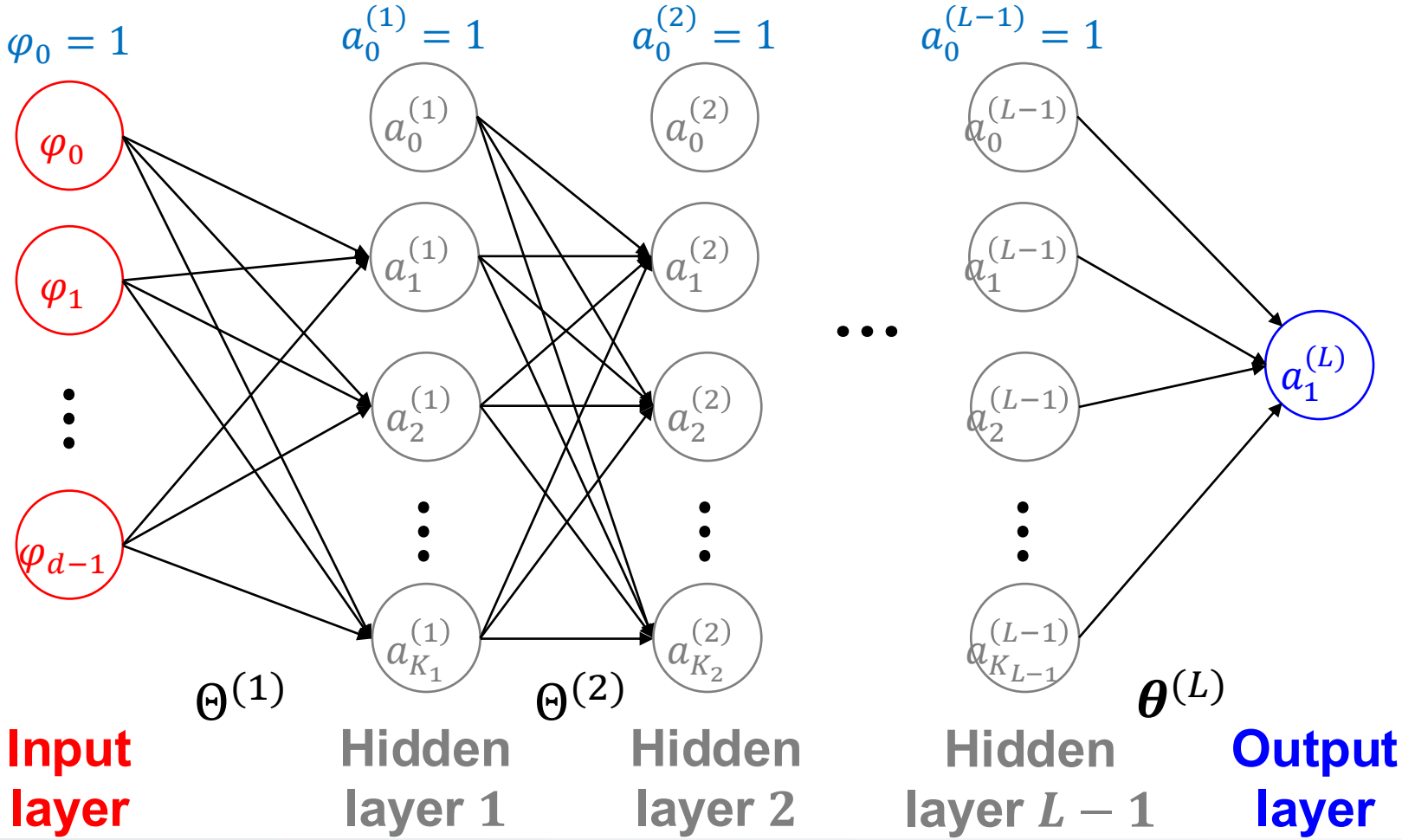
- $\Theta^{(1)} \in \mathbb{R}^{K_1 \times d}$
- $\Theta^{(2)} \in \mathbb{R}^{K_2 \times (K_1+1)}$

Due to the biases

- $\boldsymbol{\theta}^{(3)} \in \mathbb{R}^{1 \times (K_2+1)}$

Neural networks with multiple hidden layers

Consider now a **Neural Network (NN)** with multiple hidden layers

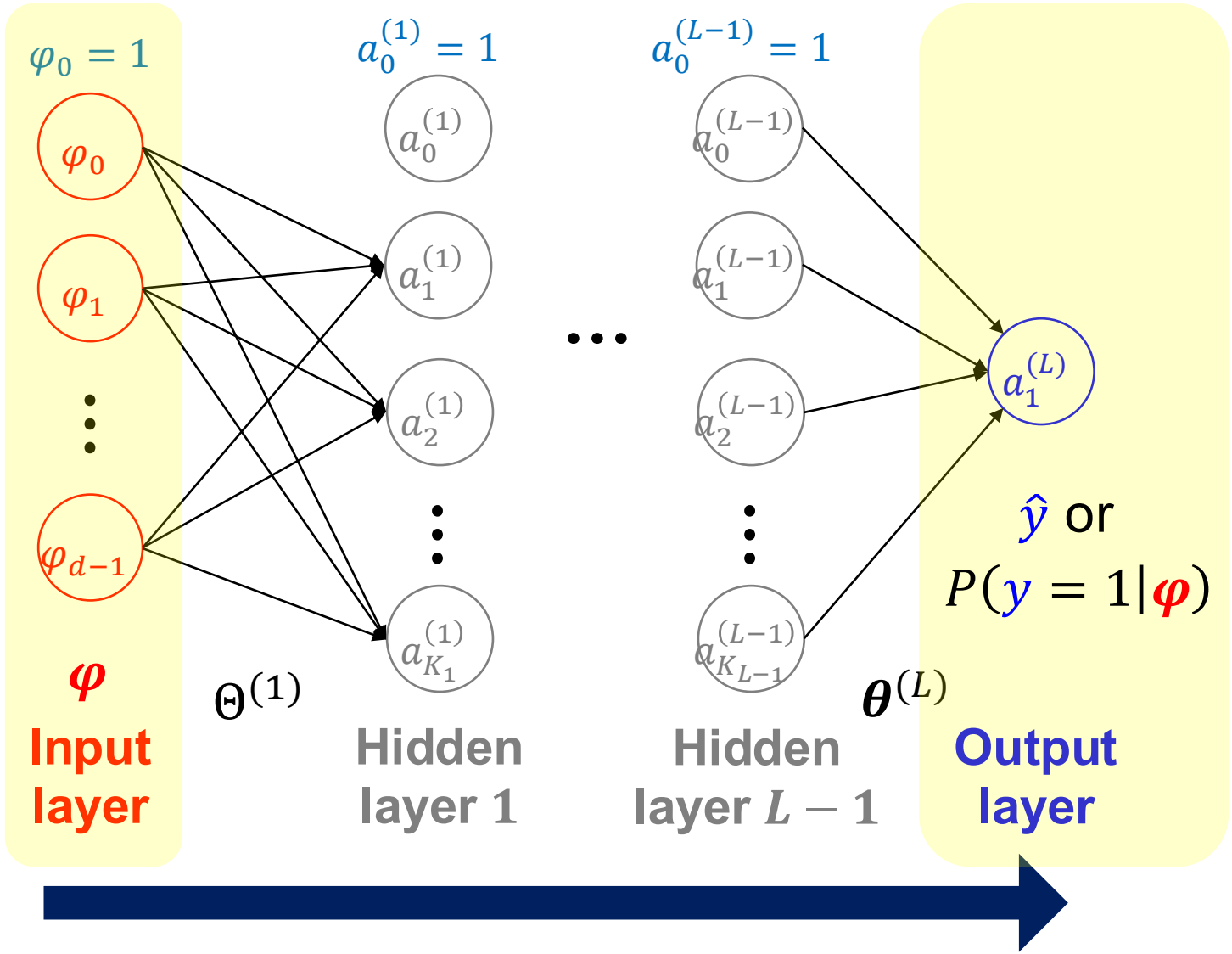


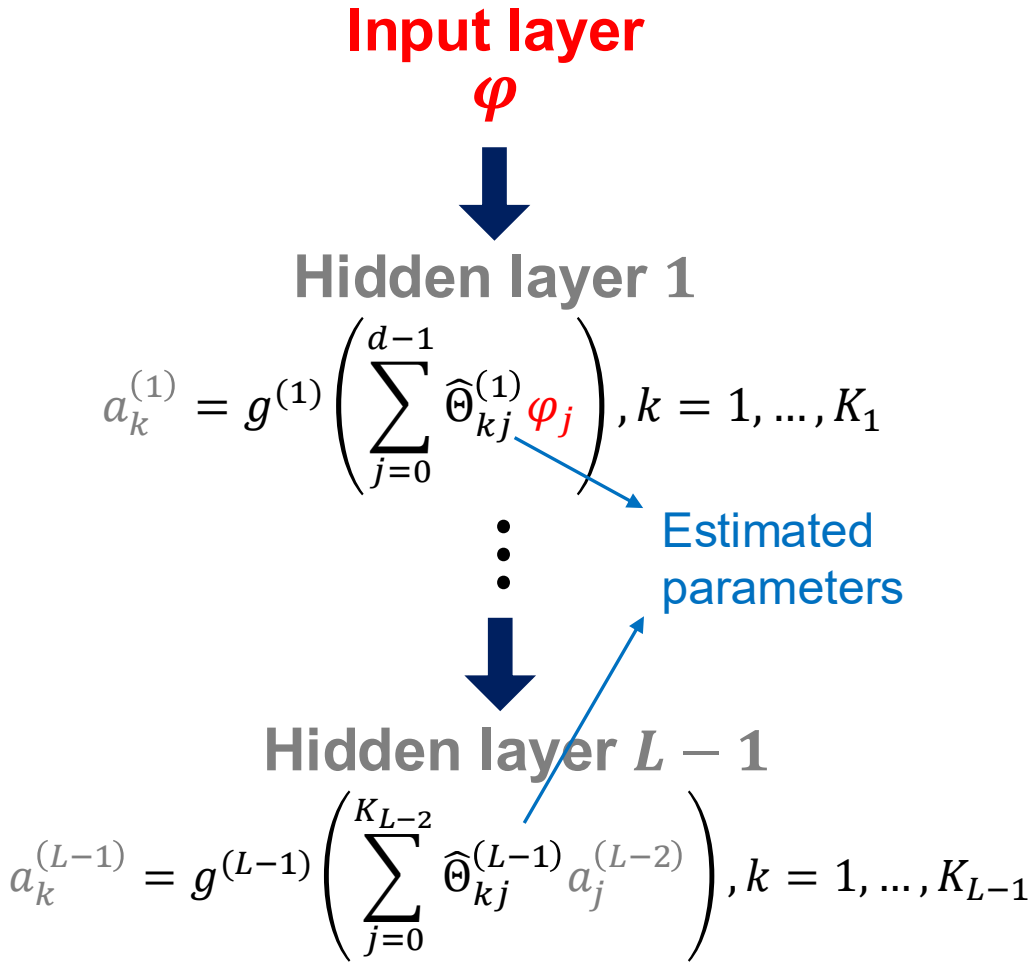
L-layer network:

- $\Theta^{(1)} \in \mathbb{R}^{K_1 \times d}$
- $\Theta^{(2)} \in \mathbb{R}^{K_2 \times (K_1+1)}$
- $\Theta^{(3)} \in \mathbb{R}^{K_3 \times (K_2+1)}$
- $\vdots$
- $\Theta^{(L-1)} \in \mathbb{R}^{K_{L-1} \times (K_{L-2}+1)}$
- $\boldsymbol{\theta}^{(L)} \in \mathbb{R}^{1 \times (K_{L-1}+1)}$

Feed-forward neural networks – making predictions

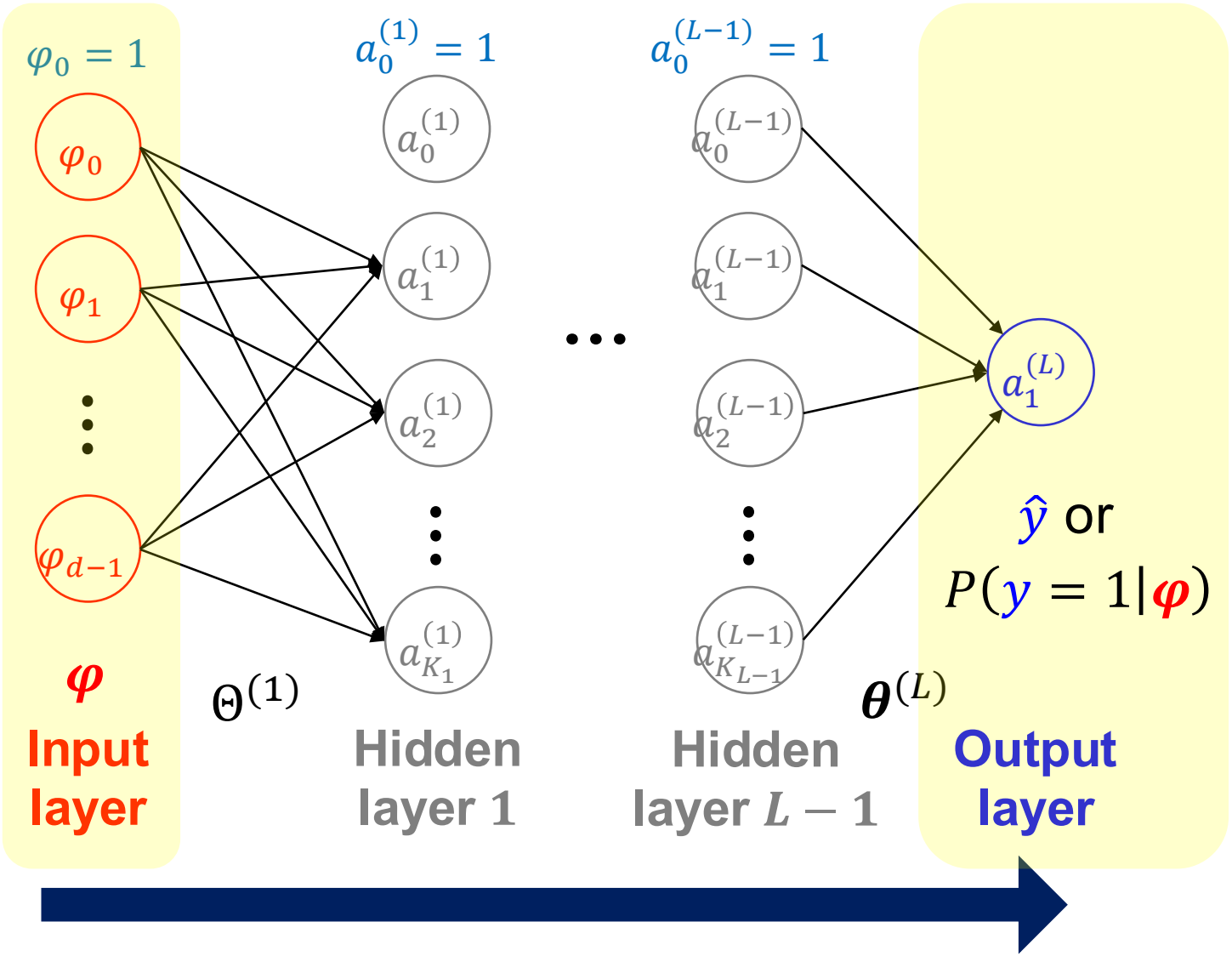
In neural networks, predictions are made through **forward propagation**





Feed-forward neural networks – making predictions

In neural networks, predictions are made through **forward propagation**



Hidden layer $L-1$

$$a_k^{(L-1)} = g^{(L-1)}\left(\sum_{j=0}^{K_{L-2}} \widehat{\Theta}_{kj}^{(L-1)} a_j^{(L-2)}\right), k = 1, \dots, K_{L-1}$$

Output layer

- **Regression**

$$\hat{y} = \hat{f}(\boldsymbol{\varphi}) = \sum_{j=0}^{K_{L-1}} \hat{\theta}_j^{(L)} a_j^{(L-1)}$$

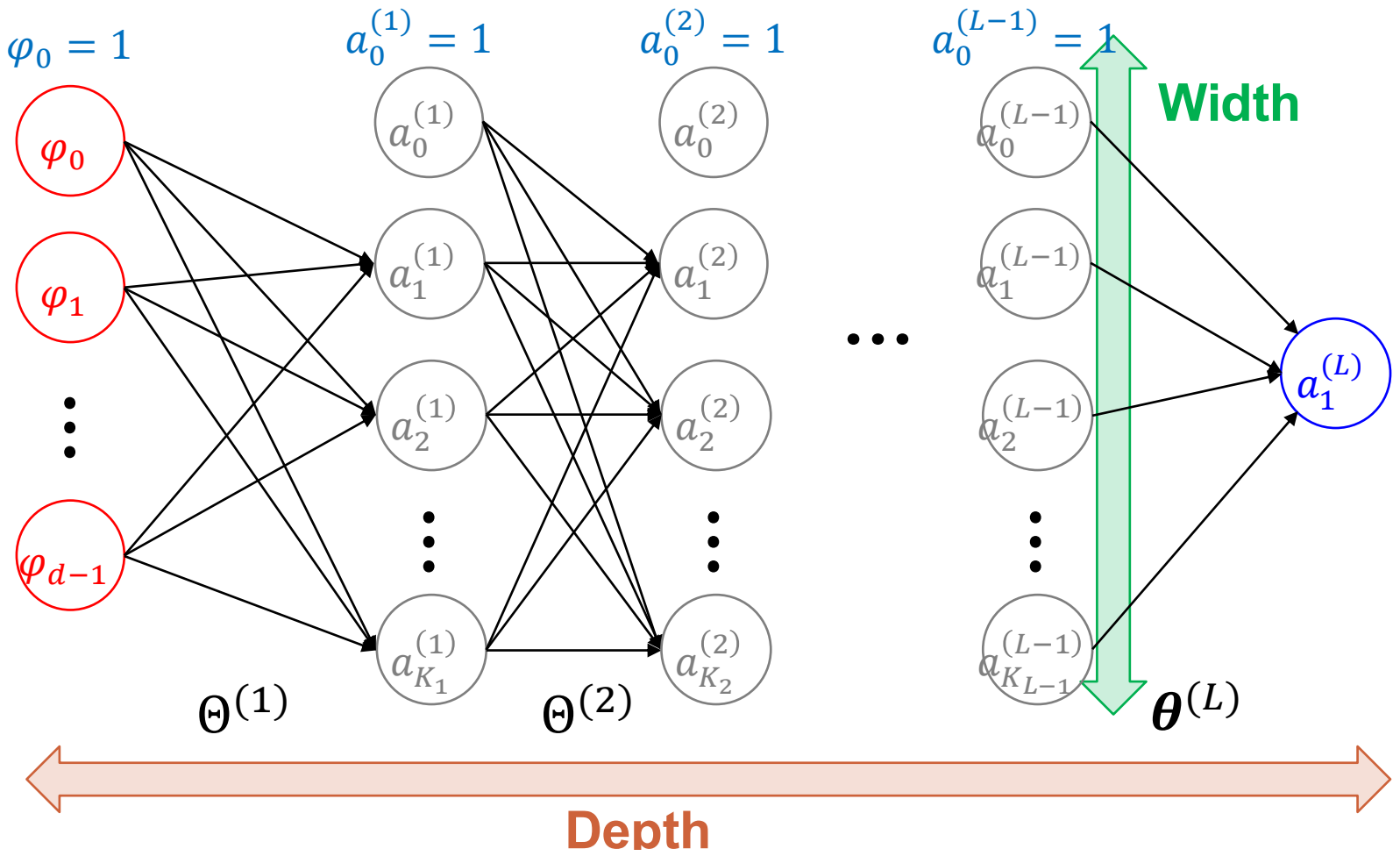
- **Binary classification**

$$P(y = 1|\boldsymbol{\varphi}) = \frac{1}{1 + e^{-\sum_{j=0}^{K_{L-1}} \hat{\theta}_j^{(L)} a_j^{(L-1)}}}$$

$\hat{y} \in \{0,1\}$ is then found by imposing a threshold γ on $P(y = 1|\boldsymbol{\varphi})$

Depth and width of a neural network

Consider now a **Neural Network (NN)** with multiple hidden layers



- The **depth** of a neural network is its number of layers (**excluding** the input layer) → L
- The **width** of
  - a **layer** is its number of units $(K_1, K_2, ...)$
  - the **network** is the width of the widest layer of the neural network, i.e.

$$\max\{K_1, K_2, ...\}$$

(do not count the biases)

The term **deep learning** refers to those supervised and unsupervised learning methods that rely on neural networks with **many** hidden layers (high depth)

Hyper-parameters of neural networks

Neural networks have **several hyper-parameters** that need to be carefully tuned:

- The number of hidden layers (i.e. the **depth** of the network)
- The number of units contained in each hidden layer (i.e. the **width** of the network)
- How the units are **connected** (so far, we have only seen the **fully connected layer** but there are several other types)
- The **activation functions** for each hidden layer

Network architecture

Increasing the **width** and the **depth** of the network amounts to adding additional parameters which, in turn, can increase the chances of **overfitting** the training data

Neural networks as universal approximators

Universal approximation theorem: a feed-forward neural network with **at least** one hidden layer and adequate activation functions (ReLU, sigmoid, …) can **approximate** any continuous function $f: \mathcal{F} \to \mathcal{Y}$ on a compact feature space $\mathcal{F}$ to arbitrary accuracy, provided that the network has a **sufficiently large number of hidden units**

Remarks:

→ If the network is large enough then, in some sense, $f(\boldsymbol{\varphi}) \in \mathcal{H}$ ($\mathcal{H}$ being the hypothesis space associated with the network)

- A sufficiently **wide** neural network is powerful enough to represent any (continuous) **nonlinear** function. However, we do **not** know a-priori how large this network should be
- With just one hidden layer, in the **worst case**, we need a number of hidden units that is **exponential** in the number of inputs $\boldsymbol{\varphi}$ to approximate $f(\boldsymbol{\varphi})$ to arbitrary accuracy

Neural networks as universal approximators

There exists a set of parameters $\hat{\Theta}$ such that the corresponding model $\hat{f}(\boldsymbol{\varphi}) \approx f(\boldsymbol{\varphi})$ to arbitrary accuracy

Remarks:

- Even though the network can represent $f(\boldsymbol{\varphi})$, we have **no** guarantee that the estimated model is such that $\hat{f}(\boldsymbol{\varphi}) \approx f(\boldsymbol{\varphi})$:
  - The optimization algorithm used to estimate Θ **might not find** the values of the parameters that correspond to $f(\boldsymbol{\varphi})$
  - $\hat{f}(\boldsymbol{\varphi})$ might **overfit** the data, leading to bad generalization

- In many circumstances, **using deeper models can reduce the number of units required to represent $f(\boldsymbol{\varphi})$ and limit the generalization error**
  - Do **not** use only one very **wide** hidden layer. Instead, use a neural network architecture with many "narrow" hidden layers
  - A lot of **trial and error** is needed to select the best architecture for the task at hand (monitor the performances on a validation set!)

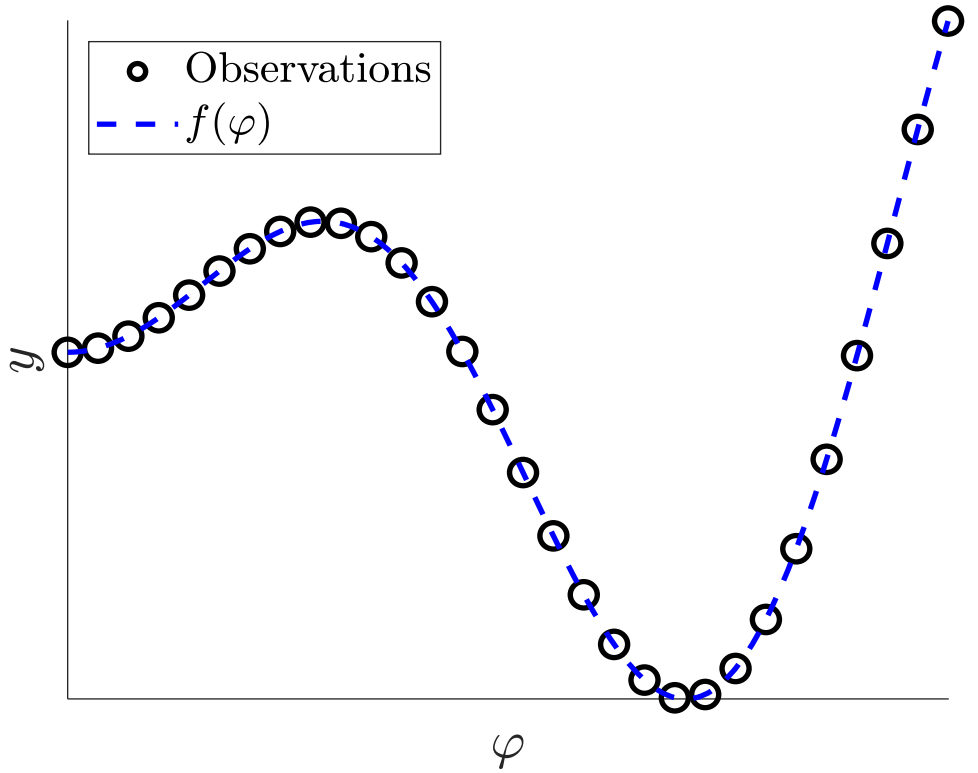
Example: performances of NN of varying width/depth

In this example, we want to assess the performances of a feed-forward neural network as its width and its depth vary

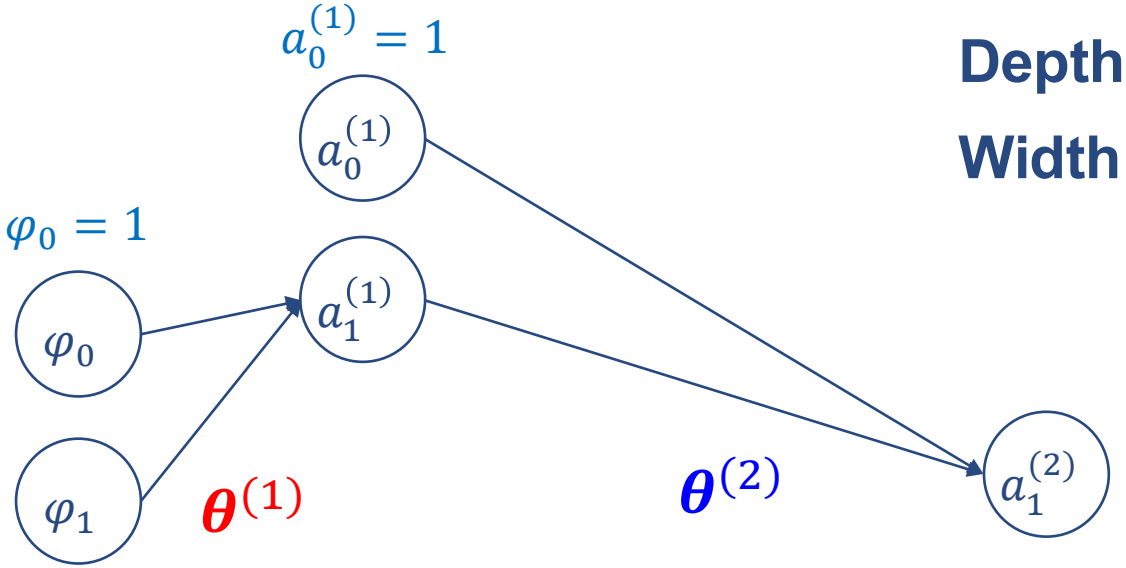
- We consider a **regression** task. The data-generating function is nonlinear:

$$f(\varphi) = \varphi \cdot \sin(\varphi)$$

- We acquire $N = 30$ data
- The activation functions used in each hidden layer are **ReLU**s
- For simplicity, we only consider the performances on training data



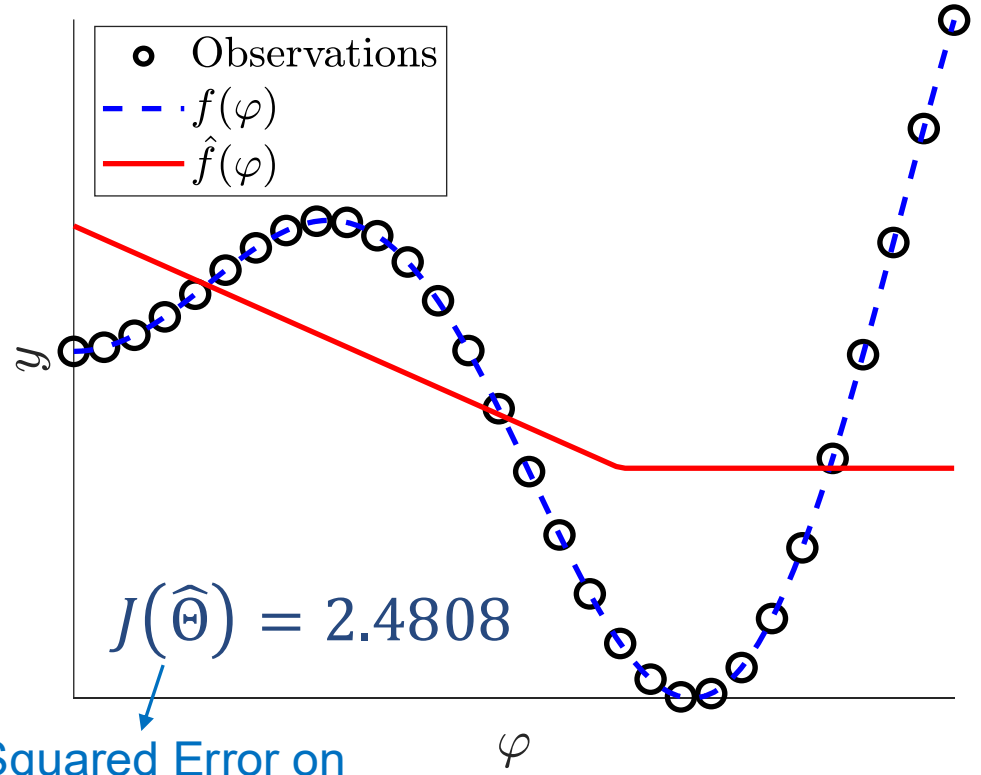
Example: performances of NN of varying width/depth



Depth: 2

Width: 1

1 hidden layer with 1 unit
(do not count the biases)



Mean Squared Error on training data

Total **number of parameters** is:

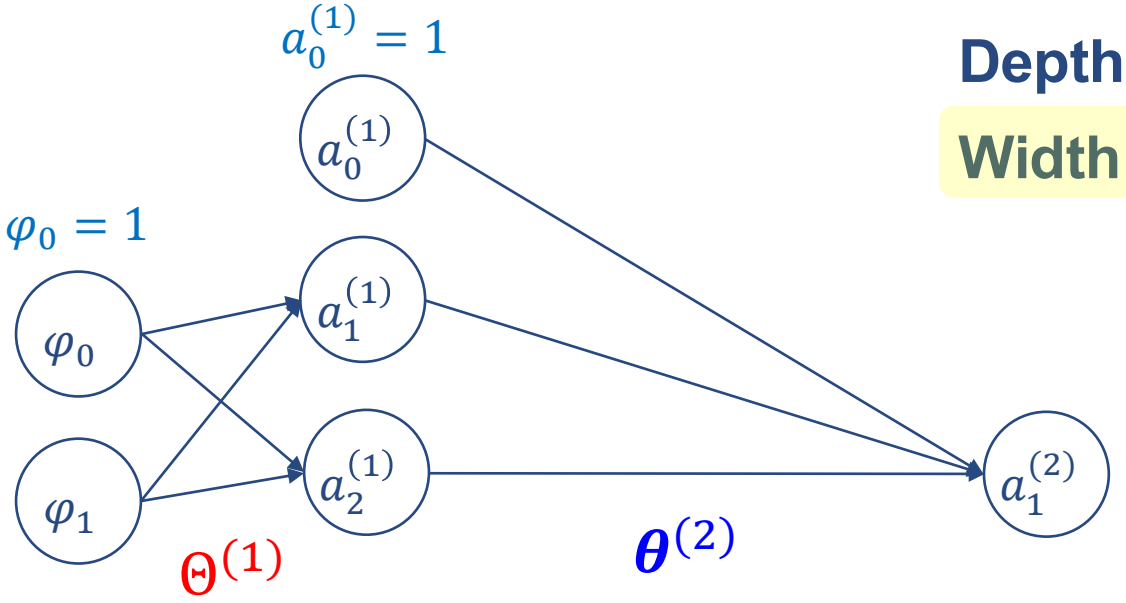
$K_1 \times d + 1 \times (K_1 + 1) = \mathbf{4}$

$\boldsymbol{\theta}^{(1)}$ $\boldsymbol{\theta}^{(2)}$

$d = 2$

$K_1 = 1$

Example: performances of NN of varying width/depth

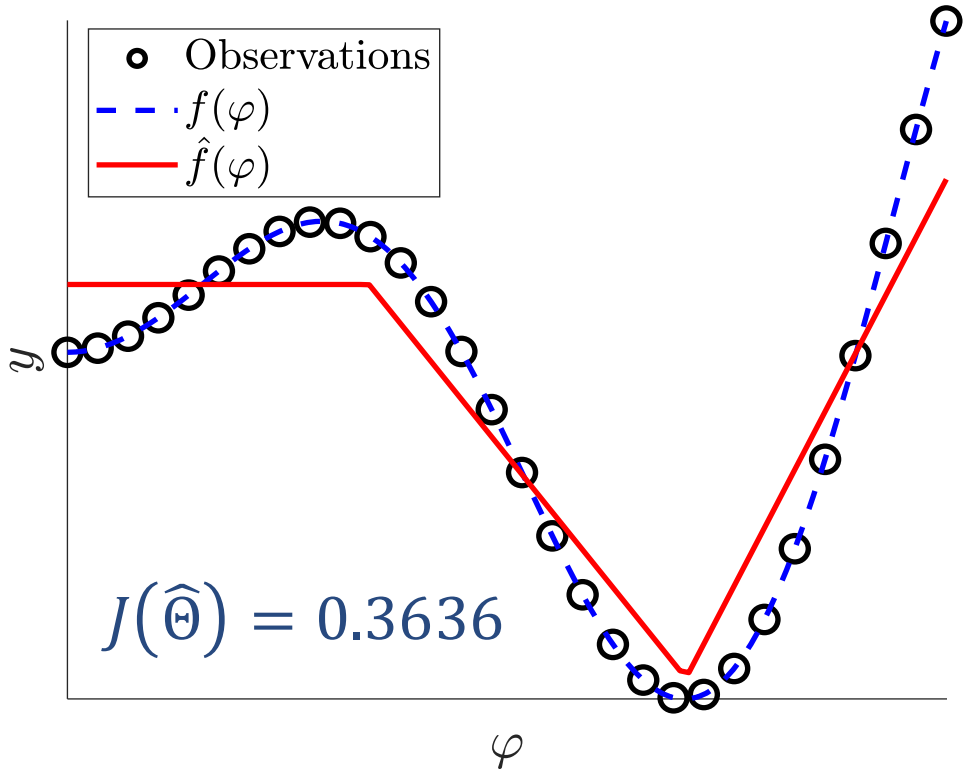


Depth: 2

Width: 2

1 hidden layer with 2 units

(do not count the biases)



Total **number of parameters** is:

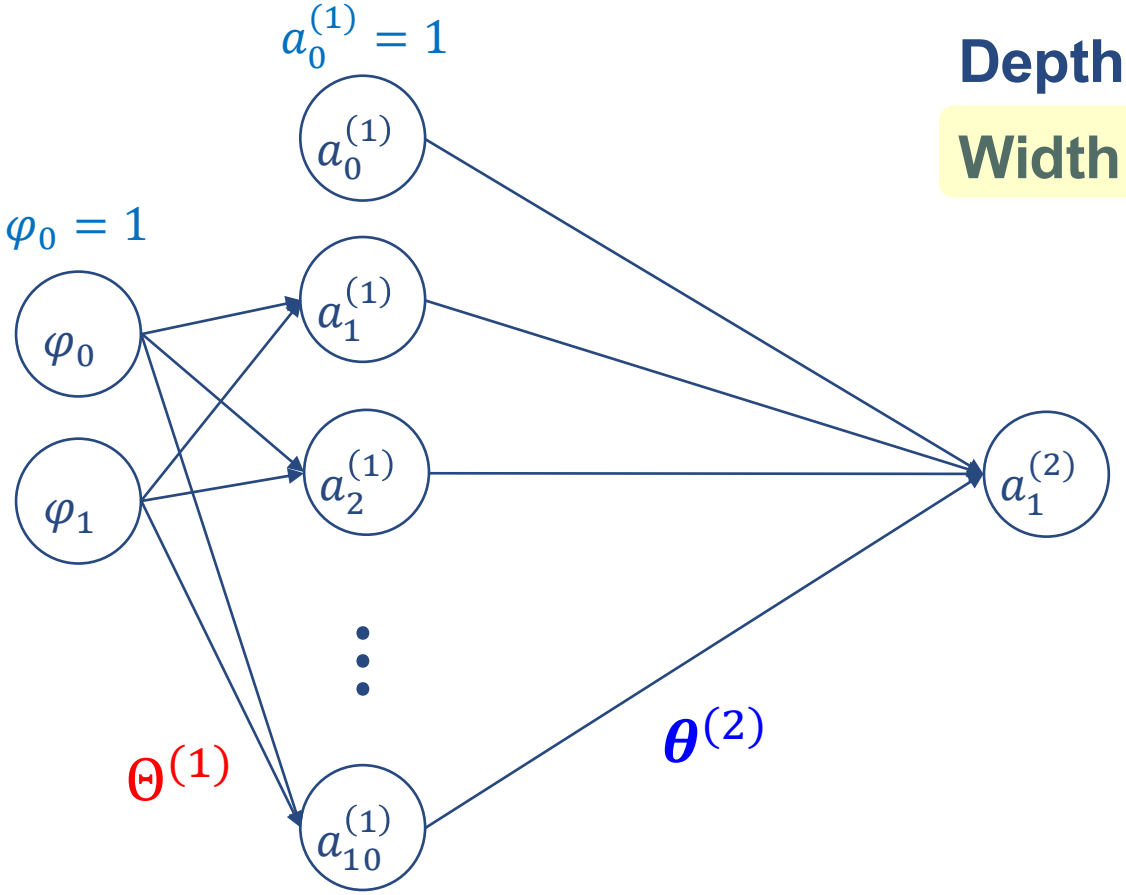
$K_1 \times d + 1 \times (K_1 + 1) = 7$

$\Theta^{(1)}$ $\boldsymbol{\theta}^{(2)}$

$d = 2$

$K_1 = 2$

Example: performances of NN of varying width/depth

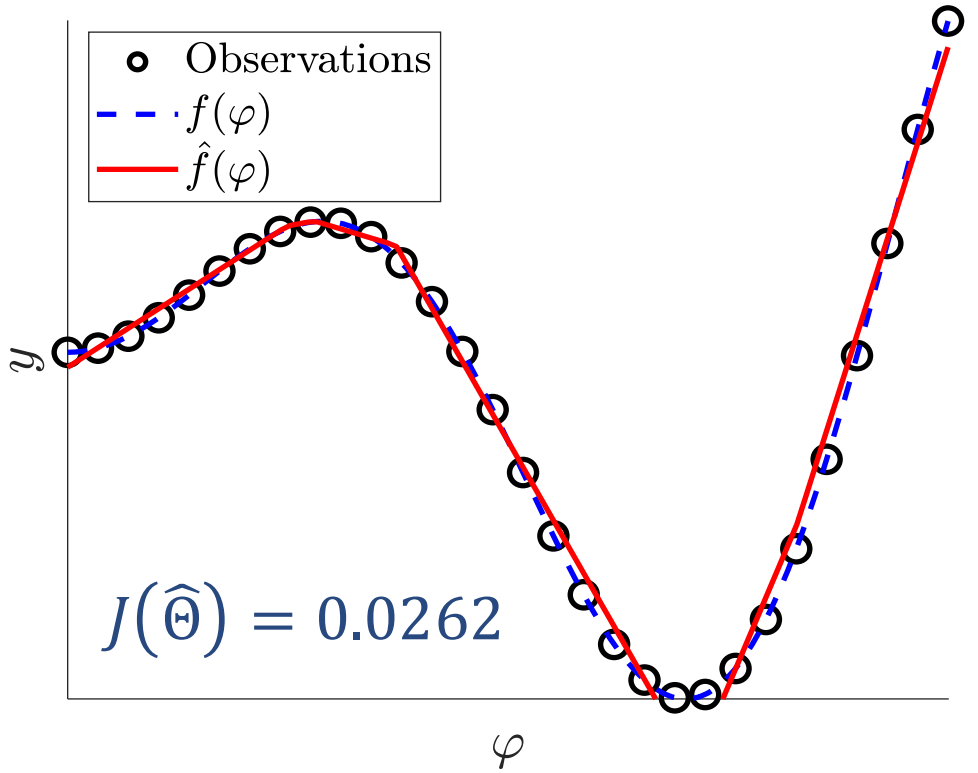


Depth: 2

Width: 10

1 hidden layer with 10 units

(do not count the biases)



Total **number of parameters** is:

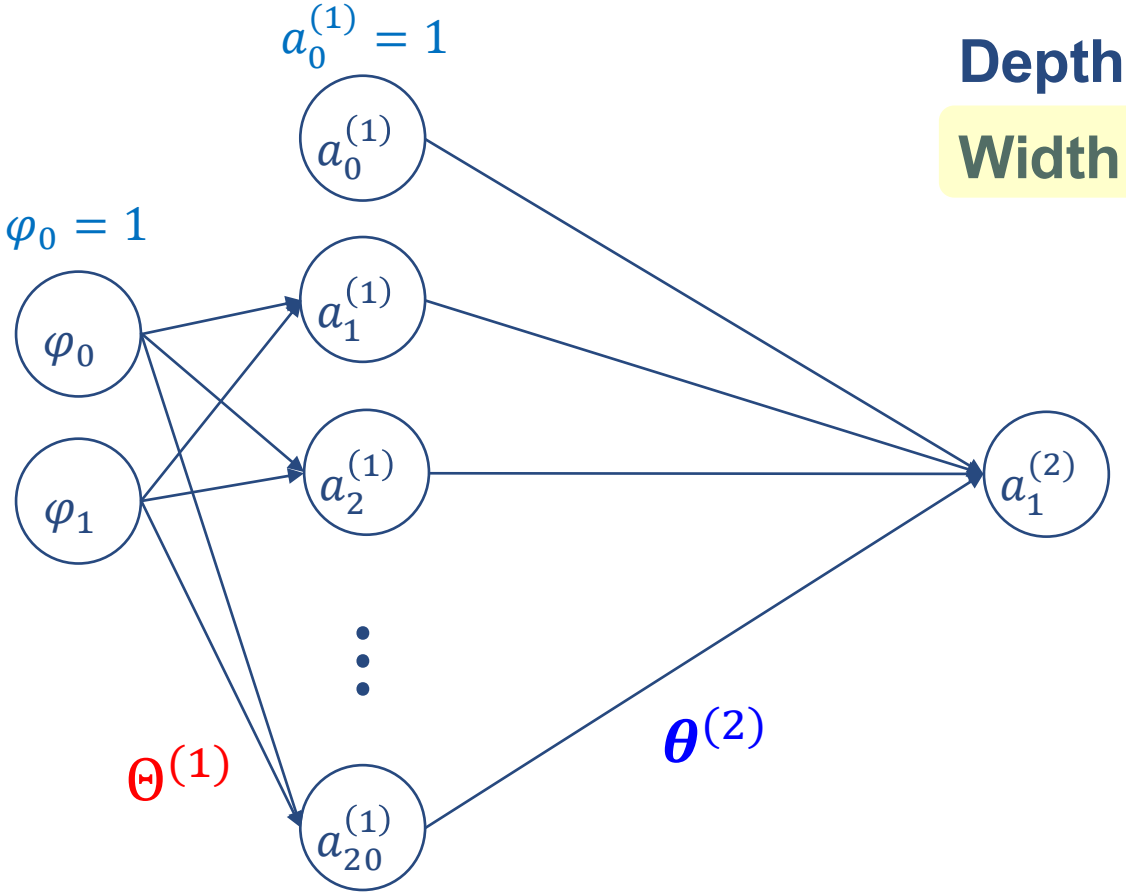
$K_1 \times d + 1 \times (K_1 + 1) = \mathbf{31}$

$\Theta^{(1)}$ $\boldsymbol{\theta}^{(2)}$

$d = 2$

$K_1 = 10$

Example: performances of NN of varying width/depth

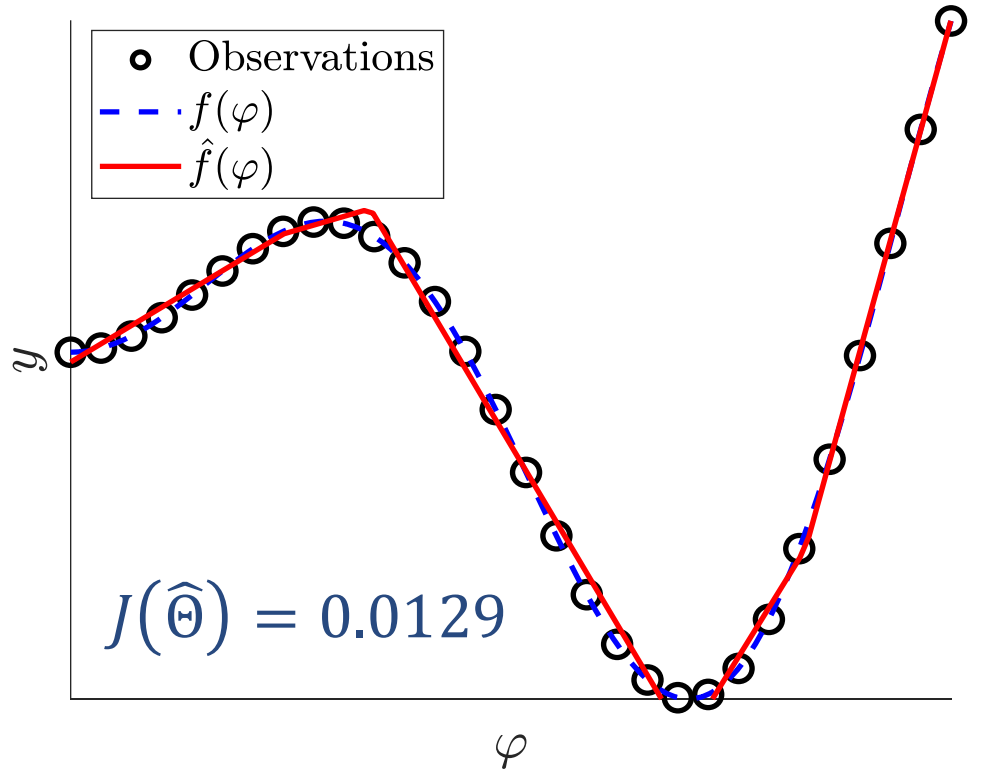


Depth: 2

Width: 20

1 hidden layer with 20 units

(do not count the biases)



Total **number of parameters** is:

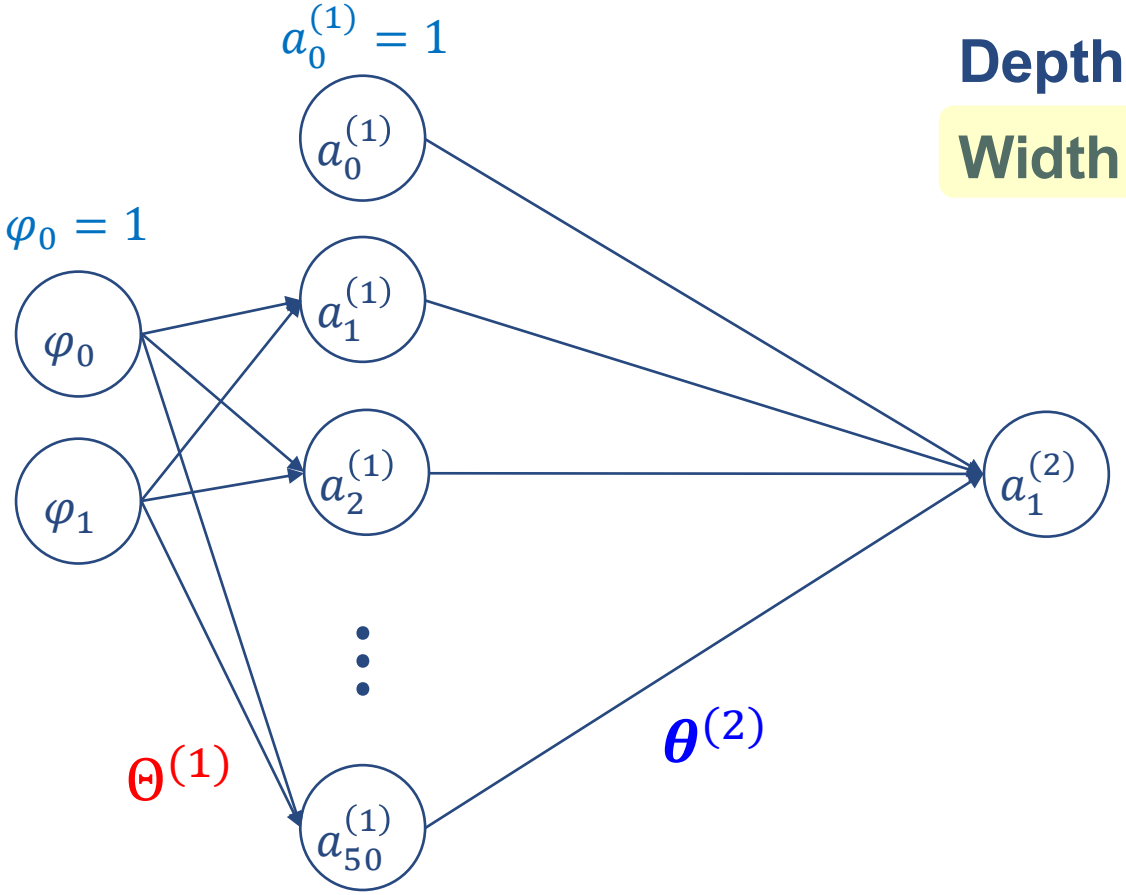
$K_1 \times d + 1 \times (K_1 + 1) = \mathbf{61}$

$\Theta^{(1)}$ $\boldsymbol{\theta}^{(2)}$

$d = 2$

$K_1 = 20$

Example: performances of NN of varying width/depth

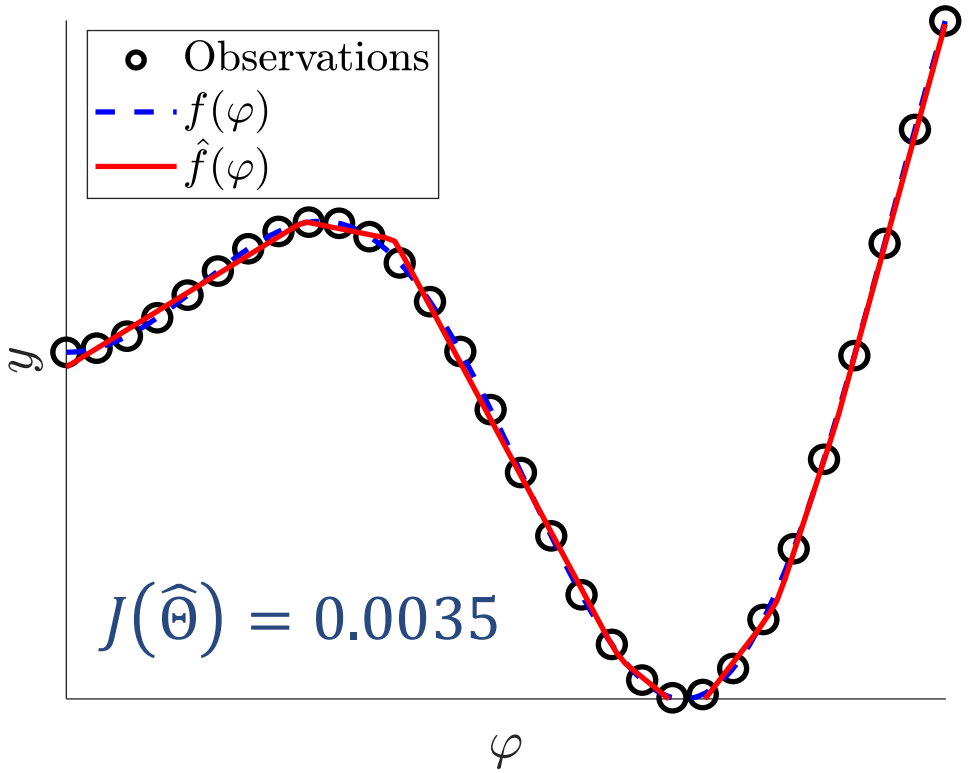


Depth: 2

Width: 50

1 hidden layer with 50 units

(do not count the biases)



Total **number of parameters** is:

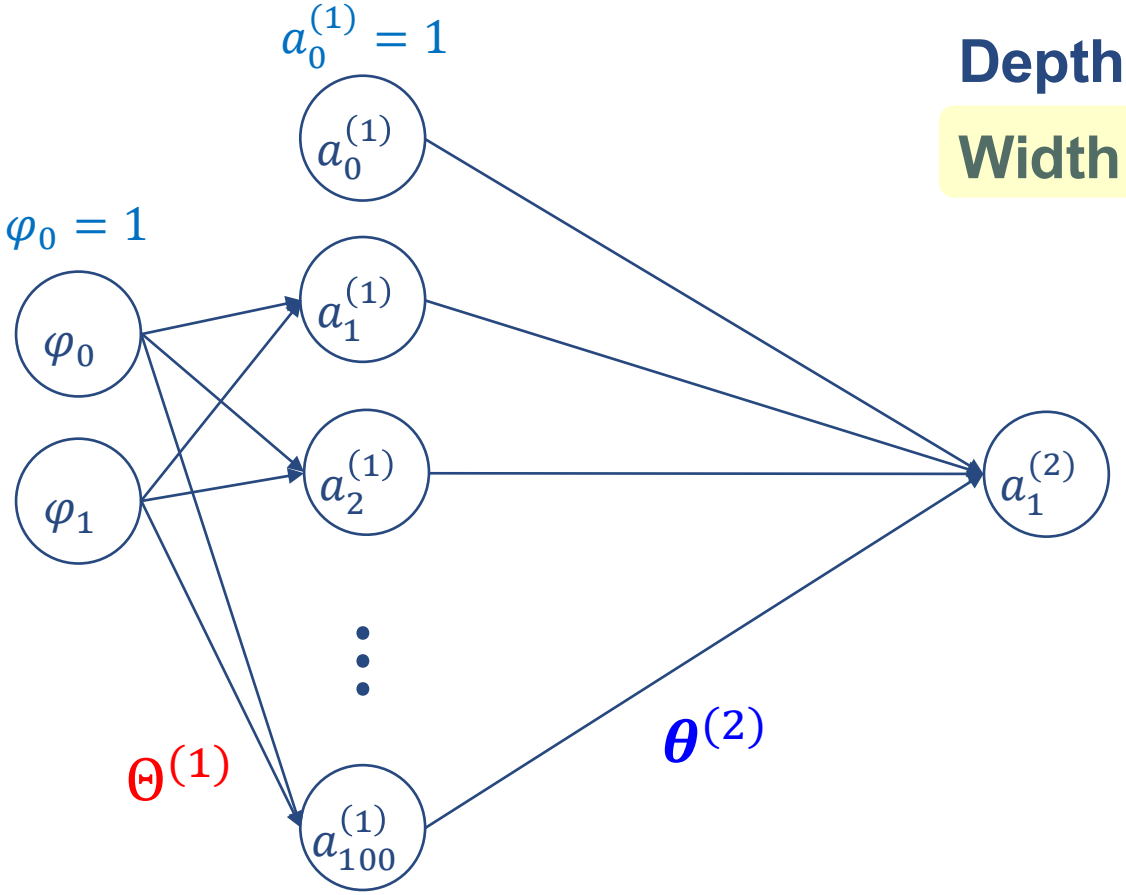
$K_1 \times d + 1 \times (K_1 + 1) = \mathbf{151}$

$\Theta^{(1)}$ $\boldsymbol{\theta}^{(2)}$

$d = 2$

$K_1 = 50$

Example: performances of NN of varying width/depth

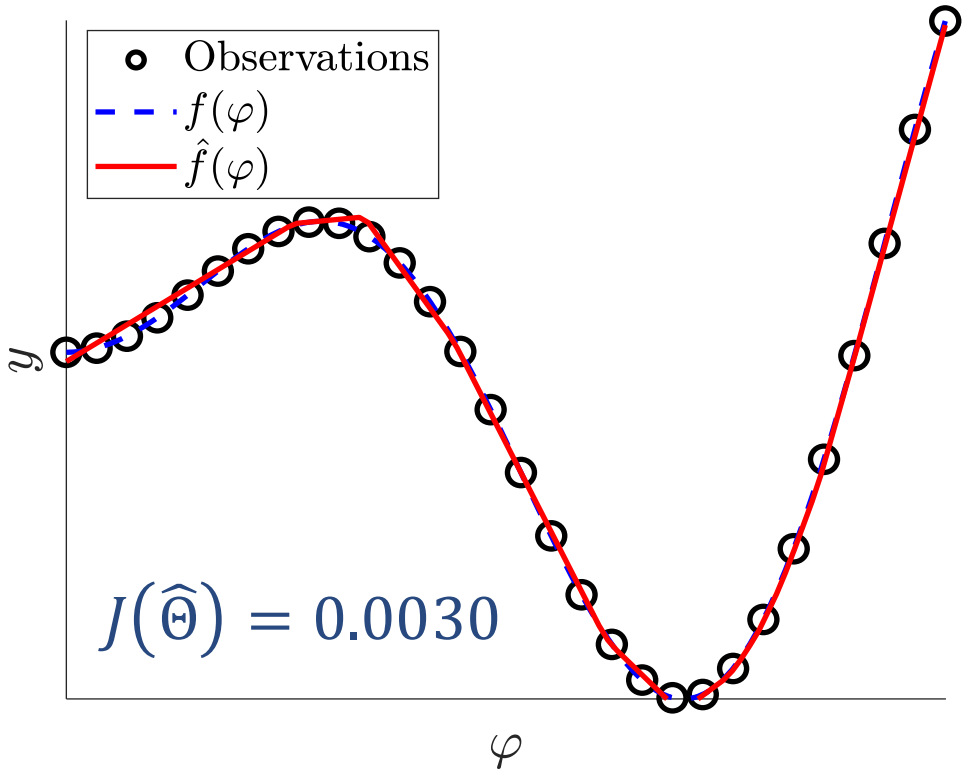


Depth: 2

Width: 100

1 hidden layer with 100 units

(do not count the biases)



Total **number of parameters** is:

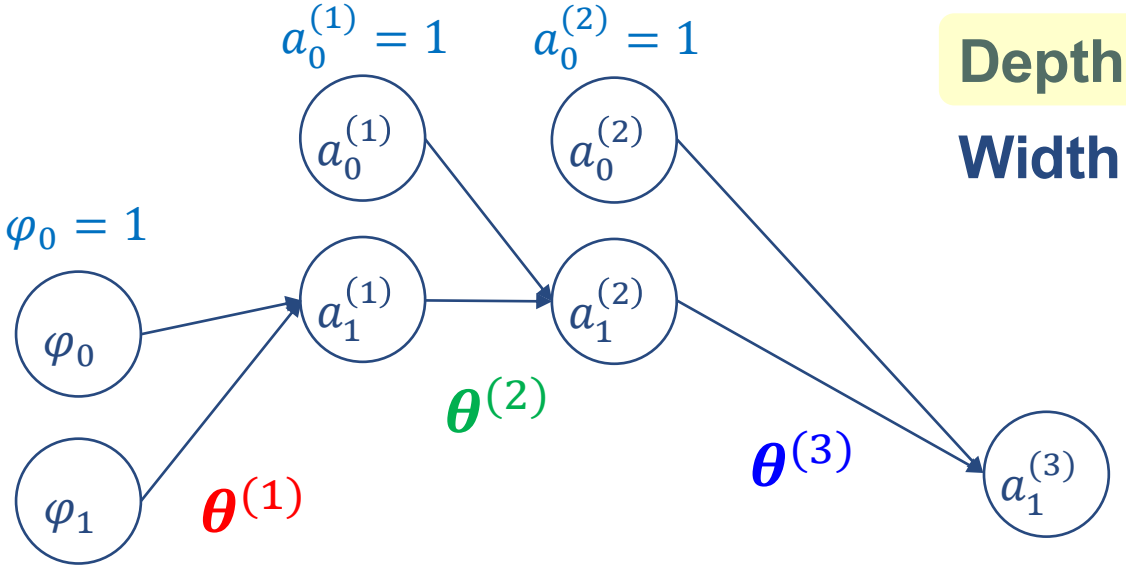
$K_1 \times d + 1 \times (K_1 + 1) = \mathbf{301}$

$\Theta^{(1)}$ $\boldsymbol{\theta}^{(2)}$

$d = 2$

$K_1 = 100$

Example: performances of NN of varying width/depth

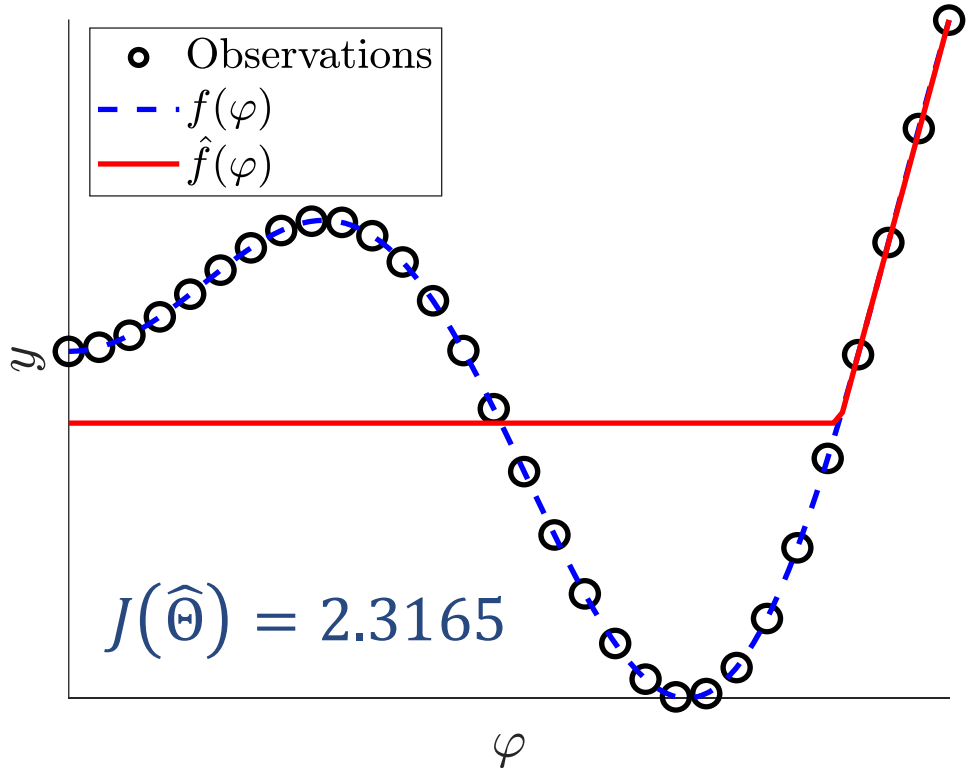


Depth: 3

Width: 1

2 hidden layers with 1 unit each

(do not count the biases)



Total **number of parameters** is:

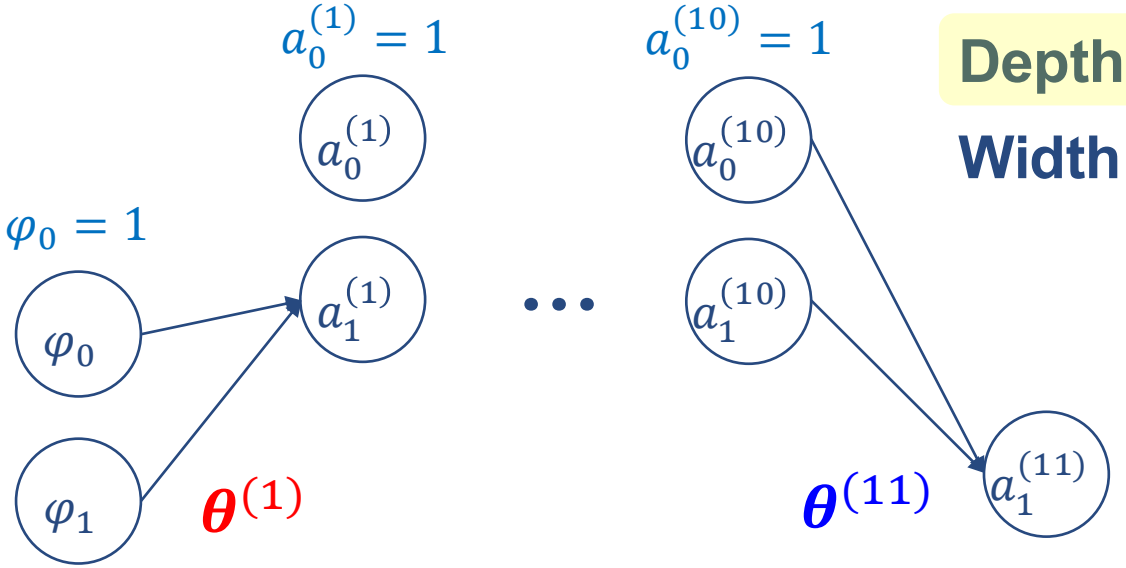
$K_1 \times d + K_2 \times (K_1 + 1) + 1 \times (K_2 + 1) = 6$

$\boldsymbol{\theta}^{(1)}$ $\boldsymbol{\theta}^{(2)}$ $\boldsymbol{\theta}^{(3)}$

$d = 2$

$K_1 = K_2 = 1$

Example: performances of NN of varying width/depth

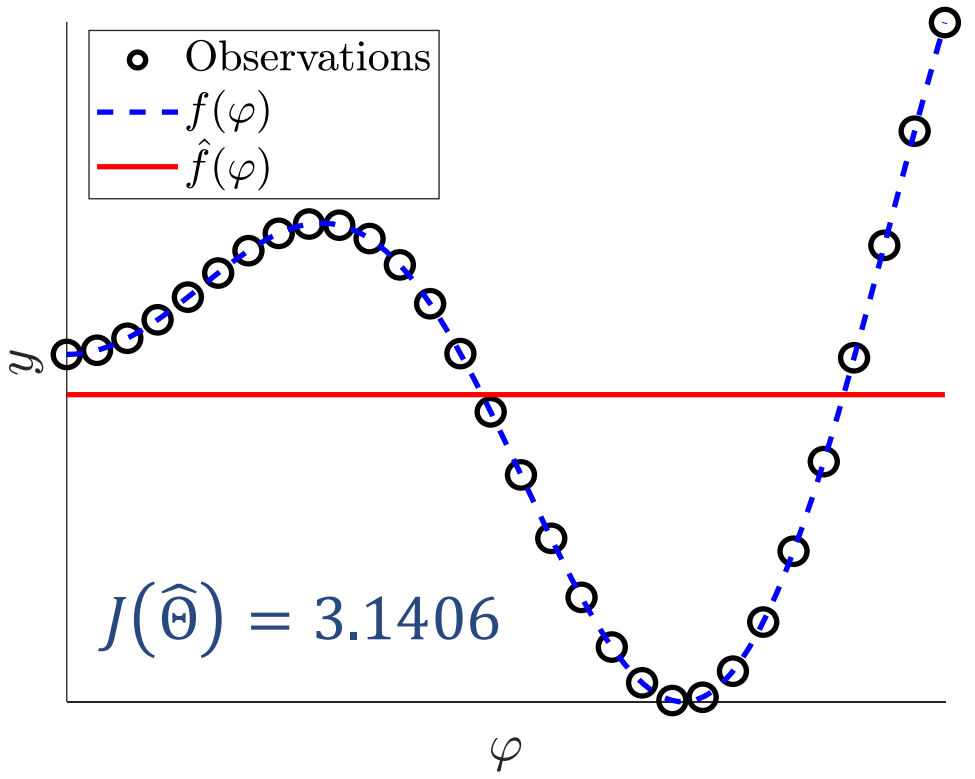


Depth: 11

Width: 1

10 hidden layers with 1 unit each

(do not count the biases)



Simply increasing the depth without increasing the width can lead to bad performances!

Total **number of parameters** is:

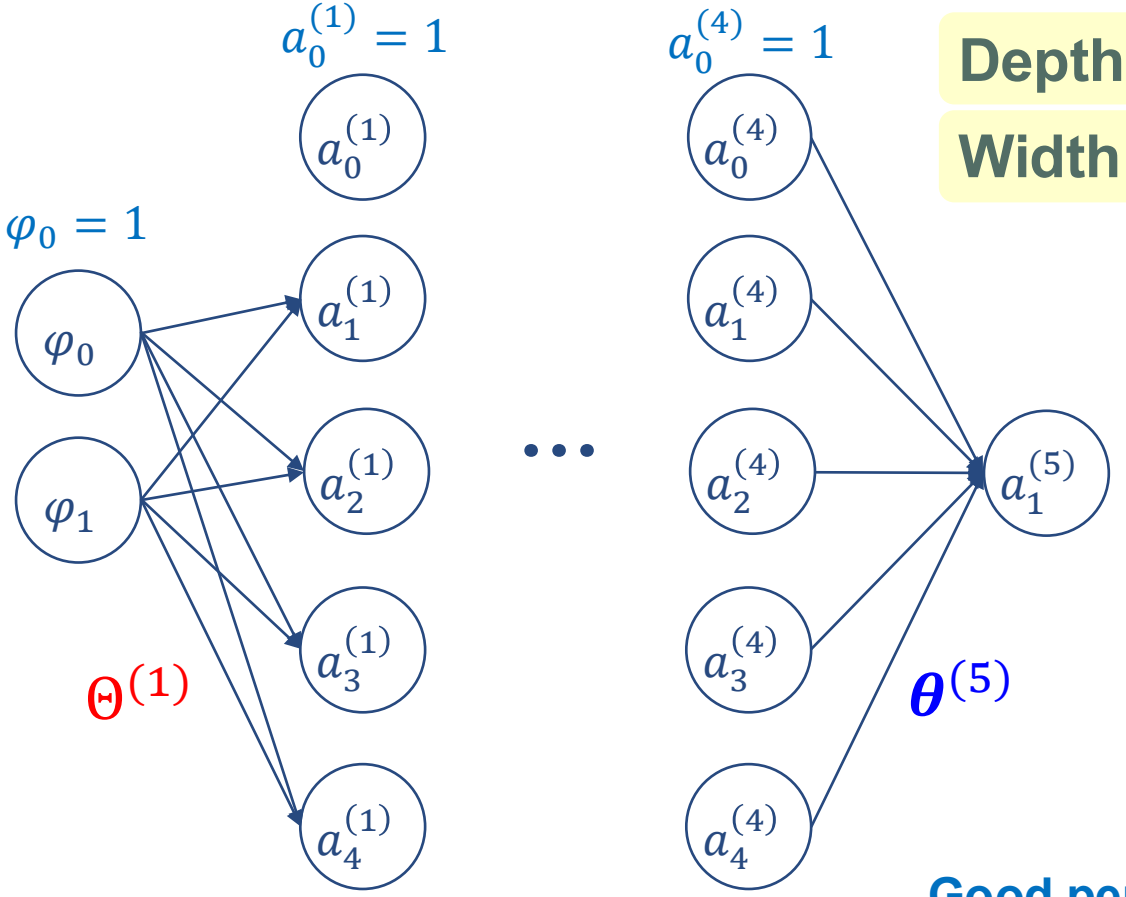
$$K_1 \times d + K_2 \times (K_1 + 1) + \cdots + 1 \times (K_{10} + 1) = \mathbf{22}$$

$\Theta^{(1)}$ $\Theta^{(2)}$ $\cdots$ $\boldsymbol{\theta}^{(11)}$

$$d = 2$$

$$K_1 = K_2 = \cdots = K_{10} = 1$$

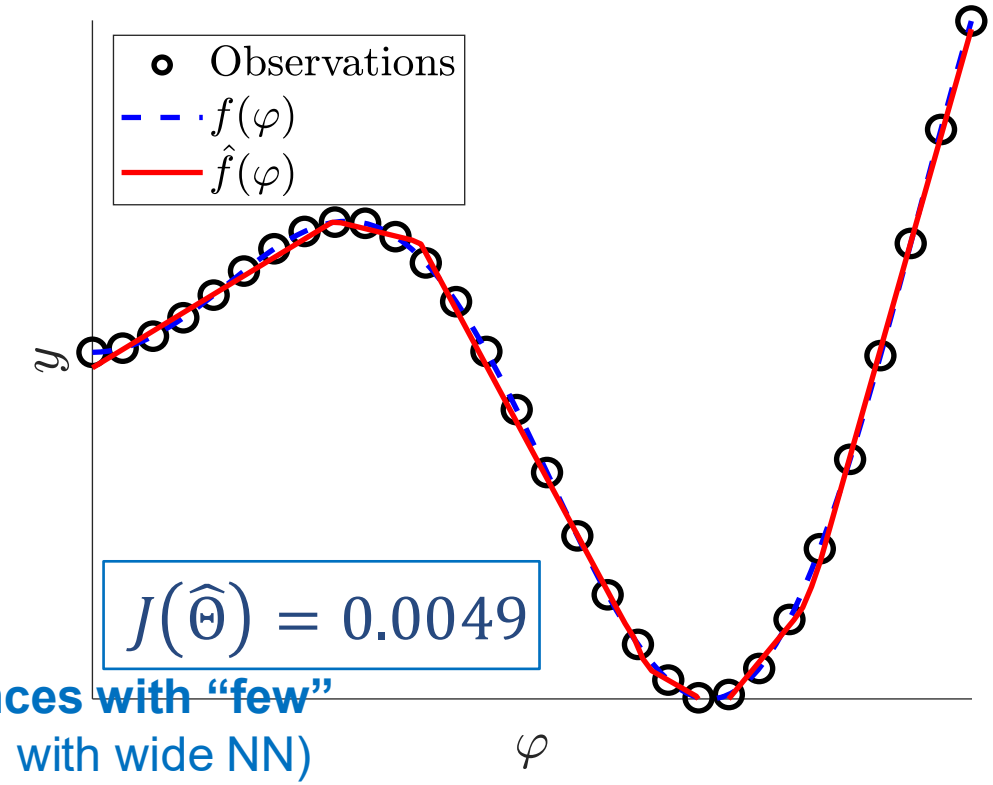
Example: performances of NN of varying width/depth



Depth: 5

Width: 4

4 hidden layers, each with 4 units
(do not count the biases)



Total **number of parameters** is:

$$K_1 \times d + K_2 \times (K_1 + 1) + K_3 \times (K_2 + 1) + K_4 \times (K_3 + 1) + 1 \times (K_4 + 1) = \boxed{73}$$

$\Theta^{(1)}$ $\Theta^{(2)}$ $\Theta^{(3)}$ $\Theta^{(4)}$ $\boldsymbol{\theta}^{(5)}$

Good performances with "few" parameters! (cf. with wide NN)

$d = 2$

$K_1 = K_2 = K_3 = K_4 = 4$

Feed-forward neural networks – multiple outputs and multi-class classification

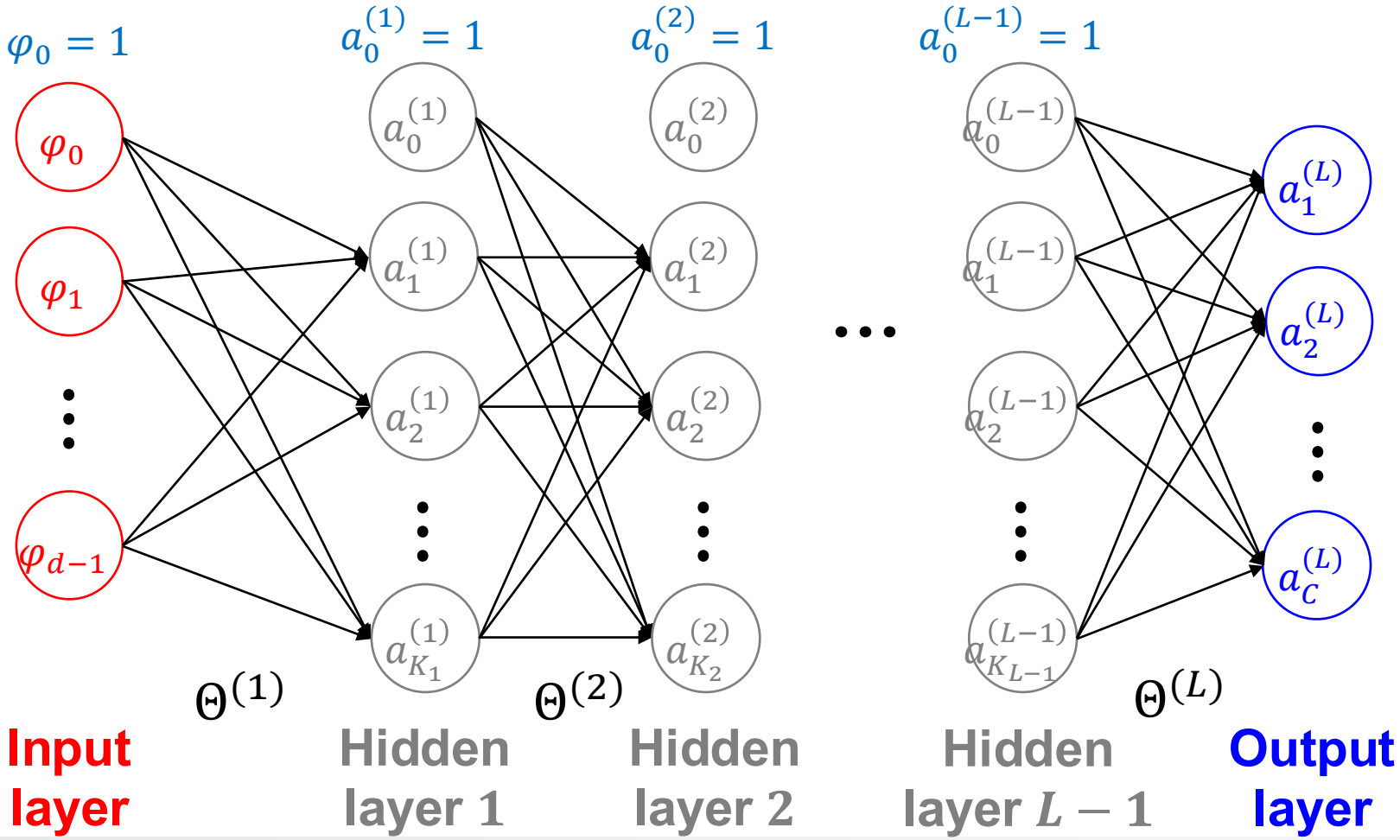
- Neural networks can easily handle **multiple outputs** by adding a unit to the **output layer** for each output and choosing a suitable activation function
- Similarly, it is straightforward to deal with **multi-class classification** problems, i.e. when $y(i) \in \{1,2,\dots,C\}$. Recall the **one-hot encoding**

$$y(i) = c, y(i) \in \{1,\dots,C\} \quad (1 \times 1) \Rightarrow \boldsymbol{y}^{(OH)}(i) = \underset{C \times 1}{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}} \begin{matrix} \text{Class } 1 \\ \text{Class } 2 \\ \\ \text{Class } c \\ \text{Class } c+1 \\ \\ \text{Class } C \end{matrix}$$

It's like having multiple outputs!

- One-hot encoding converts a single output, which can assume C values, into C outputs, each of which can either be 0 or 1
- Each element of $\boldsymbol{y}^{(OH)}(i)$ is to be interpreted as follows:
  - $y_c^{(OH)}(i) = 1 \Rightarrow$ class c
  - $y_c^{(OH)}(i) = 0 \Rightarrow$ **not** class c (it can be any of the remaining $C - 1$ classes)

Feed-forward neural networks for multi-class classification



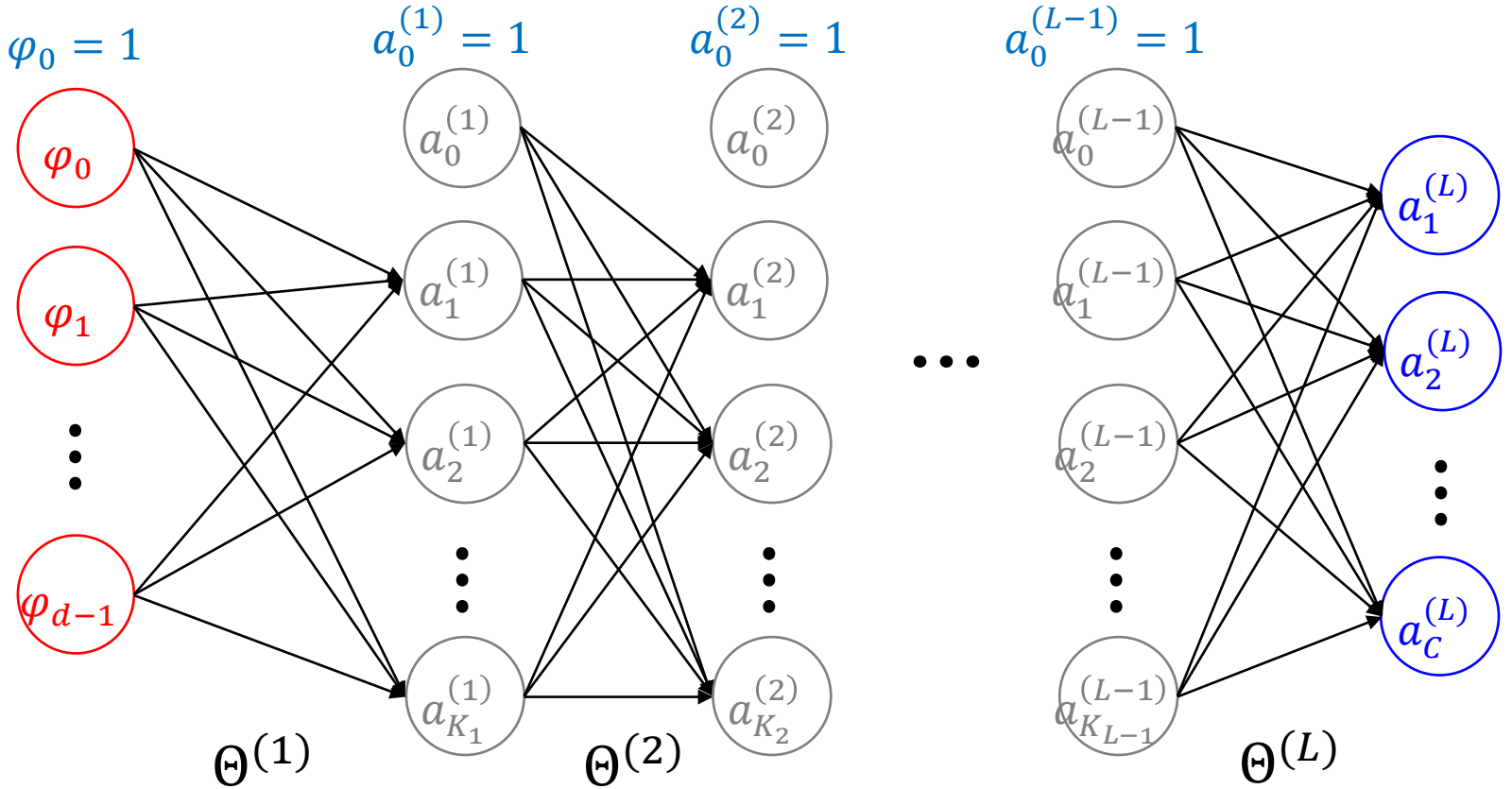
$\Rightarrow a_1^{(L)} = P(y = 1|\boldsymbol{\varphi})$

$\Rightarrow a_2^{(L)} = P(y = 2|\boldsymbol{\varphi})$

$\Rightarrow a_C^{(L)} = P(y = C|\boldsymbol{\varphi})$

Given an input $\boldsymbol{\varphi}$, the network estimates the probabilities of belonging to each class

Feed-forward neural networks for multi-class classification



This time, we use the **softmax** activation function instead of the sigmoid:

$$a_c^{(L)} = P(y = c|\boldsymbol{\varphi}) = g^{(L)}\left(z_c^{(L)}\right) = \frac{\exp\left[\sum_{j=0}^{K_{L-1}} \Theta_{cj}^{(L)} a_j^{(L-1)}\right]}{\sum_{k=1}^{C} \exp\left[\sum_{j=0}^{K_{L-1}} \Theta_{kj}^{(L)} a_j^{(L-1)}\right]} = \frac{\exp\left[z_c^{(L)}\right]}{\sum_{k=1}^{C} \exp\left[z_k^{(L)}\right]}$$

$$c = 1, \dots, C$$

$$\Theta^{(L)} \in \mathbb{R}^{C \times (K_{L-1}+1)}$$

Outline

1. Introduction to neural networks
2. Feed-forward neural networks
3. **Training neural networks**
   a. **Cost function**
   b. Backpropagation
   c. Stochastic Gradient Descent
   d. Regularization in neural networks
4. Comparison of supervised learning methods

Training NN: cost function

Consistently with most supervised learning methods, **the set of parameters Θ of a neural network is found by minimizing a suitable cost function**

$$\widehat{\Theta} = \arg\min_{\Theta} J(\Theta)$$

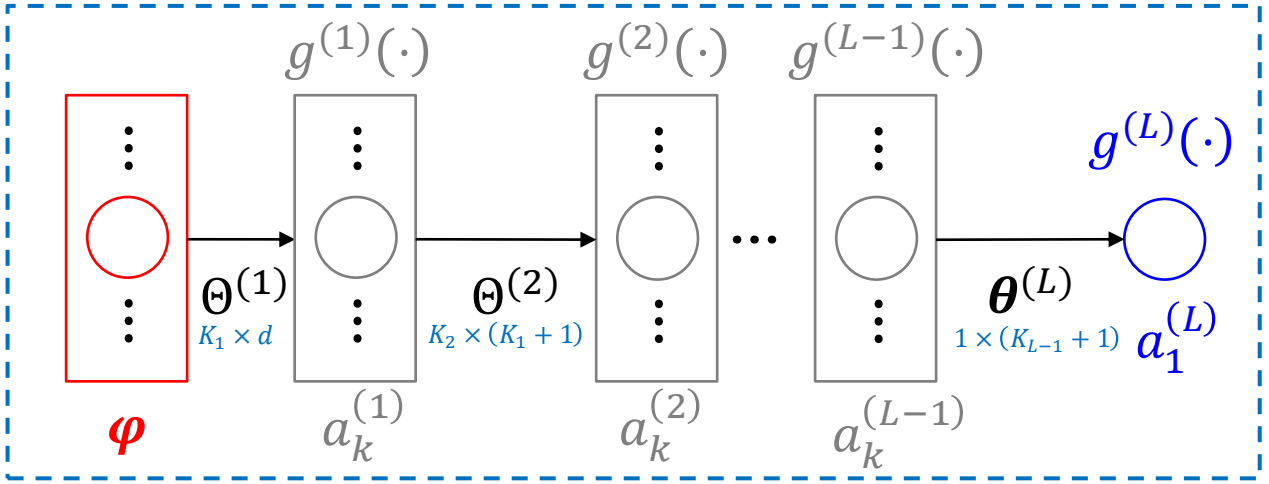
Assume to have at our disposal a set of N data $\mathcal{D} = \{(\boldsymbol{\varphi}(1), y(1)), \dots, (\boldsymbol{\varphi}(N), y(N))\}$ where $\boldsymbol{\varphi}(i) \in \mathbb{R}^{(d-1)\times 1}$ and either:

- $y(i) \in \mathbb{R}$ (**regression**)
- $y(i) \in \{0,1\}$ (**binary classification**)
- $y(i) \in \{1,2,\dots,C\}$ (**multi-class classification**)

Training NN: cost function - regression

Assume to have at our disposal a set of N data $\mathcal{D} = \{(\boldsymbol{\varphi}(1), y(1)), \dots, (\boldsymbol{\varphi}(N), y(N))\}$ where $\boldsymbol{\varphi}(i) \in \mathbb{R}^{(d-1)\times 1}$ and:

- $y(i) \in \mathbb{R}$ (**regression**)



We minimize the **mean squared error**:

$$J(\Theta) = \frac{1}{N}\sum_{i=1}^{N}\left(y(i) - a_1^{(L)}(\boldsymbol{\varphi}(i);\Theta)\right)^2$$

- **Prediction of the output** (i.e. value of the output unit) obtained by forward-propagating the input $\boldsymbol{\varphi}(i)$ through the network
- We explicit the dependency on both $\boldsymbol{\varphi}(i)$ and Θ

$$a_1^{(L)}(\boldsymbol{\varphi}(i);\Theta) = g^{(L)}\left(\sum_{j=0}^{K_{L-1}} \theta_j^{(L)} a_j^{(L-1)}(\boldsymbol{\varphi}(i);\Theta)\right) \equiv z_1^{(L)}(\boldsymbol{\varphi}(i);\Theta)$$

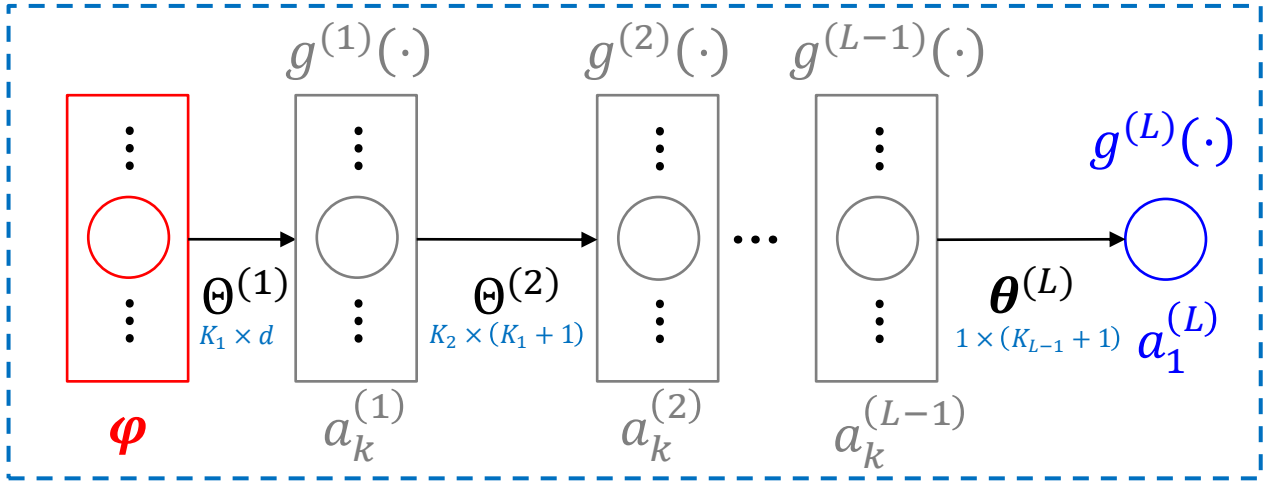
$$= z_1^{(L)}(\boldsymbol{\varphi}(i);\Theta)$$

Activation function: identity function

Training NN: cost function – binary classification

Assume to have at our disposal a set of N data $\mathcal{D} = \{(\boldsymbol{\varphi}(1), y(1)), \dots, (\boldsymbol{\varphi}(N), y(N))\}$ where $\boldsymbol{\varphi}(i) \in \mathbb{R}^{(d-1)\times 1}$ and:

- $y(i) \in \{0,1\}$ (**binary classification**)



We minimize the **cross-entropy**: (basically, the minus log-likelihood)

$$J(\Theta) = -\sum_{i=1}^{N} \left(y(i) \ln a_1^{(L)}(\boldsymbol{\varphi}(i); \Theta) + (1 - y(i)) \ln \left[1 - a_1^{(L)}(\boldsymbol{\varphi}(i); \Theta) \right] \right)$$

- **Prediction of the probability of belonging to the positive class** (i.e. value of the output unit) obtained by forward-propagating the input $\boldsymbol{\varphi}(i)$ through the network
- We explicit the dependency on both $\boldsymbol{\varphi}(i)$ and Θ

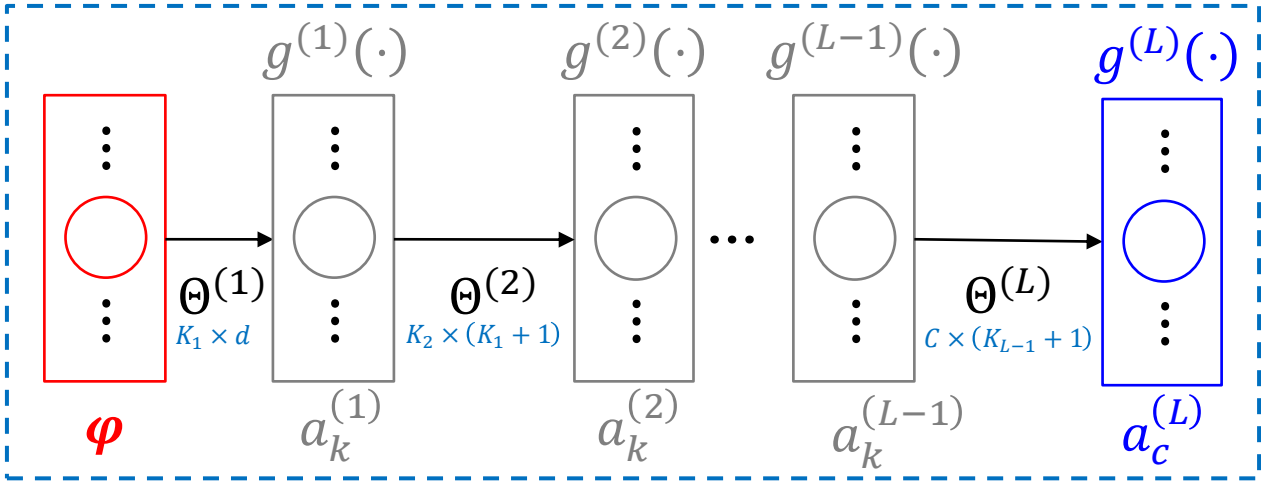
$$a_1^{(L)}(\boldsymbol{\varphi}(i); \Theta) = g^{(L)} \left(\underbrace{\sum_{j=0}^{K_{L-1}} \theta_j^{(L)} a_j^{(L-1)}(\boldsymbol{\varphi}(i); \Theta)}_{\equiv z_1^{(L)}(\boldsymbol{\varphi}(i); \Theta)} \right) = \frac{1}{1 + \exp\left[-z_1^{(L)}(\boldsymbol{\varphi}(i); \Theta) \right]}$$

Activation function: sigmoid

Training NN: cost function – multi-class classification

Assume to have at our disposal a set of N data $\mathcal{D} = \{(\boldsymbol{\varphi}(1), y(1)), \dots, (\boldsymbol{\varphi}(N), y(N))\}$ where $\boldsymbol{\varphi}(i) \in \mathbb{R}^{(d-1)\times 1}$ and:

- $y(i) \in \{1,2,\dots,C\}$ (**multi-class classification**)



We minimize the **multi-class cross-entropy**: (basically, the minus log-likelihood)

$$J(\Theta) = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^{(OH)}(i) \ln a_c^{(L)}(\boldsymbol{\varphi}(i);\Theta)$$

- **Prediction of the probability of belonging to the c-th class** (i.e. value of the c-th output unit) obtained by forward-propagating the input $\boldsymbol{\varphi}(i)$ through the network
- We explicit the dependency on both $\boldsymbol{\varphi}(i)$ and Θ

$$a_c^{(L)}(\boldsymbol{\varphi}(i);\Theta) = g^{(L)}\left(\sum_{j=0}^{K_{L-1}} \Theta_{cj}^{(L)} a_j^{(L-1)}(\boldsymbol{\varphi}(i);\Theta)\right)$$

$$\equiv z_c^{(L)}(\boldsymbol{\varphi}(i);\Theta)$$

$$= \frac{\exp\left[z_c^{(L)}(\boldsymbol{\varphi}(i);\Theta)\right]}{\sum_{k=1}^{C}\exp\left[z_k^{(L)}(\boldsymbol{\varphi}(i);\Theta)\right]}$$

Activation function: softmax

Training NN: cost function - challenges

- Differently from the supervised learning methods that we have seen so far, the **nonlinearity** introduced by the activation functions $g^{(l)}(\cdot), 1 \leq l \leq L$, make the cost functions $J(\Theta)$ **nonconvex**
  - $J(\Theta)$ can have **multiple local and global minima and maxima**

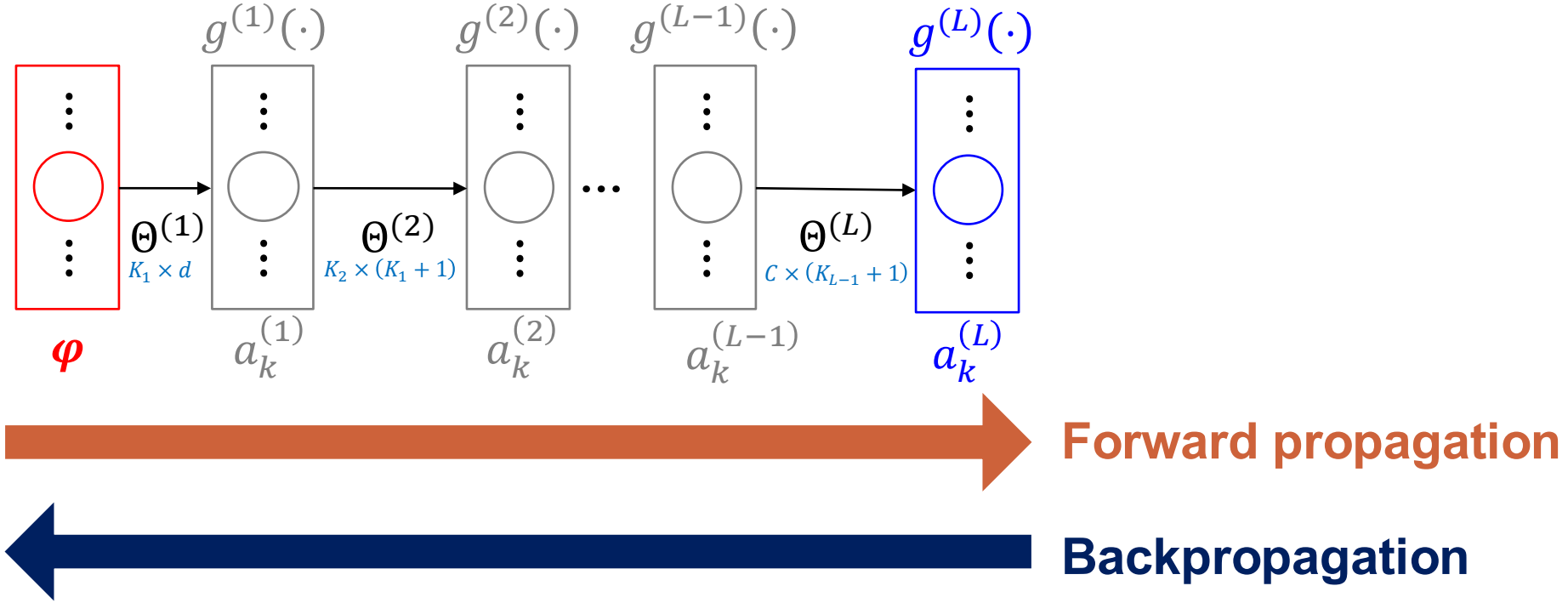
    We are not necessarily interested in finding a global minimizer of $J(\Theta)$, a local minimizer might suffice (or a Θ with sufficiently small cost)
- $J(\Theta)$ is **differentiable** almost everywhere and thus we can employ **gradient-based optimization procedures** (such as gradient descent) to minimize it
  - We need a way to compute the gradient $\nabla J(\Theta)$ that is **scalable** (since the architecture of the network can change) and **computationally efficient**. That is the role of **backpropagation**

Outline

1. Introduction to neural networks
2. Feed-forward neural networks
3. **Training neural networks**
   a. Cost function
   b. **Backpropagation**
   c. Stochastic Gradient Descent
   d. Regularization in neural networks
4. Comparison of supervised learning methods

Training NN: backpropagation

- **Backpropagation** is the standard way of **computing the gradient** of the cost function $J(\Theta)$ in neural networks
- It is **not** a way to minimize the cost function $J(\Theta)$. That is the role of a suitable gradient-based optimization procedure
- Backpropagation relies heavily on the **chain rule of differentiation**



Advantages of backpropagation

- Backpropagation is extremely **scalable**:
  - ➢ It can be applied to **any** neural network architecture. We only need to specify:
    - ❑ How to compute **the errors at the output layer** (these depend on the choice of the cost function $J(\Theta)$ and the activation function of the output layer)
    - ❑ **The derivatives of the activation functions** in each hidden layer
  - ➢ Practically, this means that we do **not** have to find the gradient by hand for any possible neural network architecture

- Backpropagation is **computationally efficient**: for a sufficiently large number of parameters d_Θ, the number of operations needed to evaluate the gradient of $J(\Theta)$ is **linear** in d_Θ (mathematically speaking, using big O notation, it is $O(d_\Theta)$)

Outline

1. Introduction to neural networks
2. Feed-forward neural networks
3. **Training neural networks**
   a. Cost function
   b. Backpropagation
   c. **Stochastic Gradient Descent**
   d. Regularization in neural networks
4. Comparison of supervised learning methods

Minibatch gradient-based optimization methods

Idea: rather than computing the gradient of the point-wise loss $\nabla J_i(\Theta)$ associated to **each** observation in $\mathcal{D}$, we divide (**randomly**) the dataset into several **batches**

$$\{1, \dots, N\} = \mathcal{B}_1 \cup \mathcal{B}_2 \cup \cdots \cup \mathcal{B}_B \qquad \mathcal{B}_i \cap \mathcal{B}_j = \emptyset, \forall i, j = 1, \dots, B, i \neq j$$

Each $\mathcal{B}_b$ is basically a set of indexes of observations

$$\mathcal{D}_b = \{(\boldsymbol{\varphi}(i), y(i)) \in \mathcal{D} : i \in \mathcal{B}_b\} \qquad b = 1, \dots, B$$

The dataset is partitioned into B **disjoint** batches

and **estimate** the gradient using only a batch $\mathcal{D}_b$ rather than the whole $\mathcal{D}$:

$$\nabla \hat{J}(\Theta) = \frac{1}{|\mathcal{B}_b|} \sum_{i \in \mathcal{B}_b} \nabla J_i(\Theta)$$

Number of elements in $\mathcal{B}_b$

Average gradient of the loss for the batch $\mathcal{B}_b$ instead of $\mathcal{D}$ ($|\mathcal{B}_b| < |\mathcal{D}| = N$)

Stochastic gradient descent

Stochastic in a sense that we minimize $J(\Theta)$ using an estimate of its gradient. It's like adding some form of **noise**

Common **batch sizes** are powers of 2, from 32 to 256, especially when computations are parallelized over different GPUs

The batches are fed, **sequentially**, to the optimization procedure. We say that we have trained a model for an **epoch** when $\left[\widehat{\Theta}\right]^{(k)}$ has been updated on every batch (i.e., the network has "seen" the whole dataset)

Needs to be **gradually decreased** due to the noise introduced by the gradient's estimate. $\nabla\hat{J}\left(\left[\widehat{\Theta}\right]^{(k)}\right)$ does **not** vanish at a minimizer of $J(\Theta)$

Stochastic gradient descent

- **Given** an initial value $\left[\widehat{\Theta}\right]^{(0)}$
- **Given** the initial learning rate $\alpha^{(0)} \in \mathbb{R}_{>0}$

1. Randomly divide the dataset into batches $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_B$

Repeat for $k = 0, 1, \ldots$ until stopping criterion not met

Typically, we either stop it ourselves (**early stopping**) or we use heuristics

2. Select the batch $\mathcal{B}_b$
3. Estimate the gradient

$$\nabla\hat{J}\left(\left[\widehat{\Theta}\right]^{(k)}\right) = \frac{1}{|\mathcal{B}_b|}\sum_{i\in\mathcal{B}_b}\nabla J_i\left(\left[\widehat{\Theta}\right]^{(k)}\right)$$

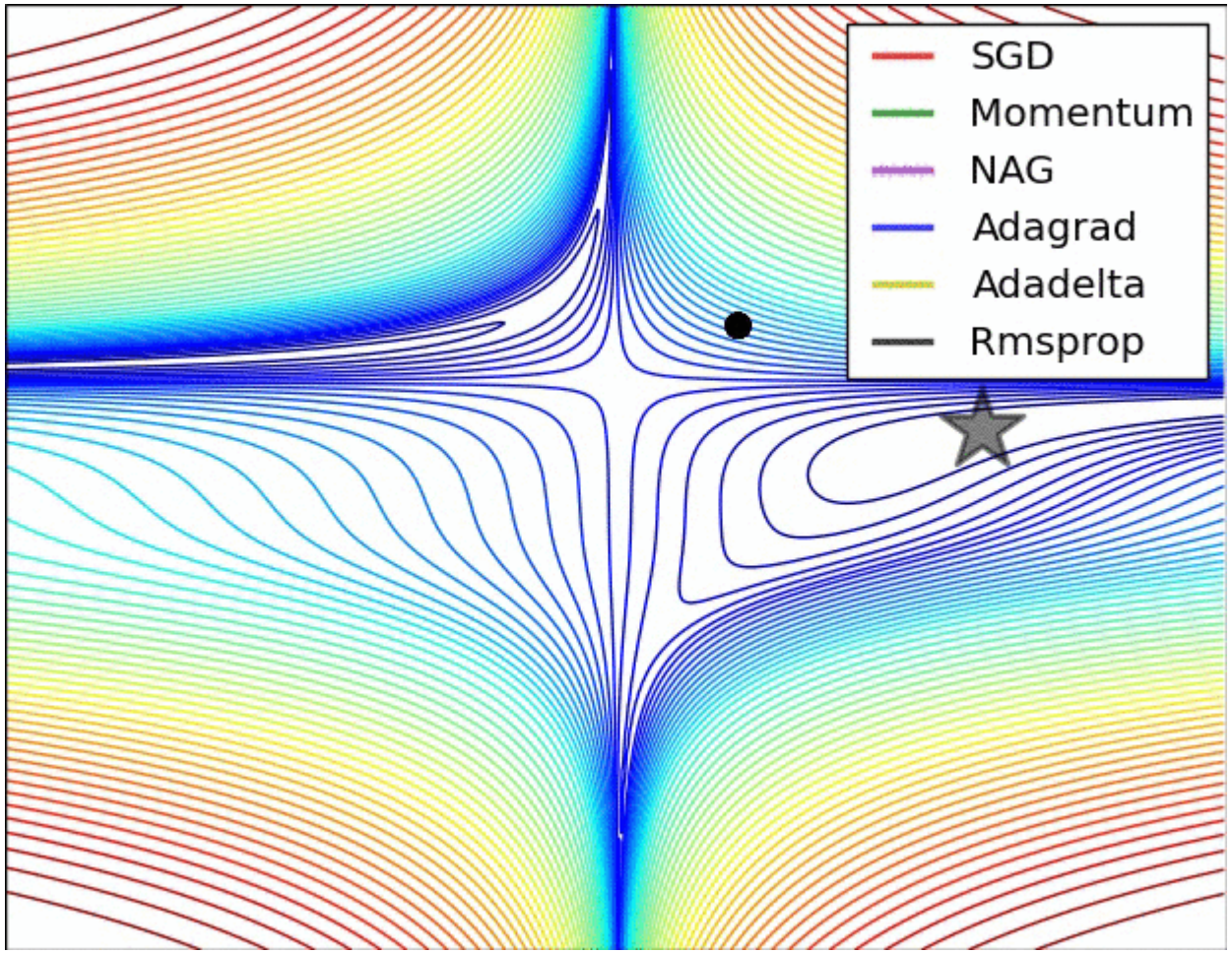
4. Update parameters' estimate

$$\left[\widehat{\Theta}\right]^{(k+1)} = \left[\widehat{\Theta}\right]^{(k)} - \alpha^{(k)}\cdot\nabla\hat{J}\left(\left[\widehat{\Theta}\right]^{(k)}\right)$$

5. Update learning rate $\rightarrow \alpha^{(k+1)}$

End repeat

Other optimization methods for neural networks



GIF taken from [6]

- There exist many other **gradient-based** optimization methods for neural networks such as **AdaGrad**, **RMSProp** and **Adam**
- In general, **no algorithm works best in every case**
- Typically, the choice of which algorithm to use depends largely on the user's familiarity with the method (for ease of hyper-parameters tuning)

Challenges in neural networks' optimization

There are several challenges connected to the minimization of $J(\Theta)$:

1. Due to the nonlinearity of the activation functions, $J(\Theta)$ is nonconvex. Deep neural networks exhibit **extremely large numbers of local minima, maxima and saddle points**
   - ➢ Typically, there are more saddle points than local minima
2. Differently from standard supervised learning methods, we are **not** necessarily interested in finding a **global minimizer** of $J(\Theta)$. That is because the NN will most likely **overfit** the data when using such parametrization. Instead, **we typically select a $\hat{\Theta}$ that works well on a validation set** (during training, we monitor the performances of the network both on the training set and on a validation set)

Challenges in neural networks' optimization

There are several challenges connected to the minimization of $J(\Theta)$:

3. **Initialization of the parameters**: due to the nonconvexity of $J(\Theta)$, the solutions returned by gradient-based optimization procedure will **depend** greatly on $[\widehat{\Theta}]^{(0)}$. A common strategy is:
   - Initialize the **weights** randomly, following a Gaussian distribution **(remember to normalize the features!)**
   - Initialize the **biases** to zero

4. **Exploding and vanishing gradients**: during training, $\nabla J\left([\widehat{\Theta}]^{(k)}\right)$ can become too big, making learning unstable, or too small, making it difficult to know in which direction we should move. These shortcomings are compensated by suitable "tricks"

Outline

1. Introduction to neural networks
2. Feed-forward neural networks
3. **Training neural networks**
   a. Cost function
   b. Backpropagation
   c. Stochastic Gradient Descent
   d. **Regularization in neural networks**
4. Comparison of supervised learning methods

Regularization in neural networks

Previously, we have seen that **the models associated with neural networks can be very complex**, especially if these include several hidden units

The **depth** and the **width** of the neural network determine the number of hidden units and, consequently, the **complexity** of the model

Controlling the complexity of the model is not a simple matter of finding the neural network of the right size, with the right number of parameters. In practical deep learning scenarios, **the best model** (i.e. the one that minimizes the test error) **is a large model that has been regularized properly**

(many hidden units)

Regularization in neural networks

In general, we can view **regularization** as **any** modification we make to a learning algorithm that is intended to reduce its out-of-sample error but not its in-sample error

- In linear or logistic regression, we can penalize the complexity of a model by adding a regularization term to the cost functions

There are several **regularization strategies for neural networks**:

1. Parameter norm penalties
2. Data augmentation
3. Early stopping
4. Parameter tying and parameter sharing
5. Dropout
6. ...

Can be used in conjunction with each other!

Parameter norm penalties

This regularization strategy can be done by either **the L_1 or the L_2 regularization approaches**

We add a regularization term to the cost function that **penalizes** the model's complexity

L_2 regularization
(also known as **weight decay**)

$$J_{\text{aug}}(\Theta) = J(\Theta) + \lambda_{\text{reg}} \cdot \sum_{l=1}^{L} \sum_{k=1}^{K_l} \sum_{j=1}^{K_{l-1}} \left[\Theta_{kj}^{(l)}\right]^2$$

L_1 regularization

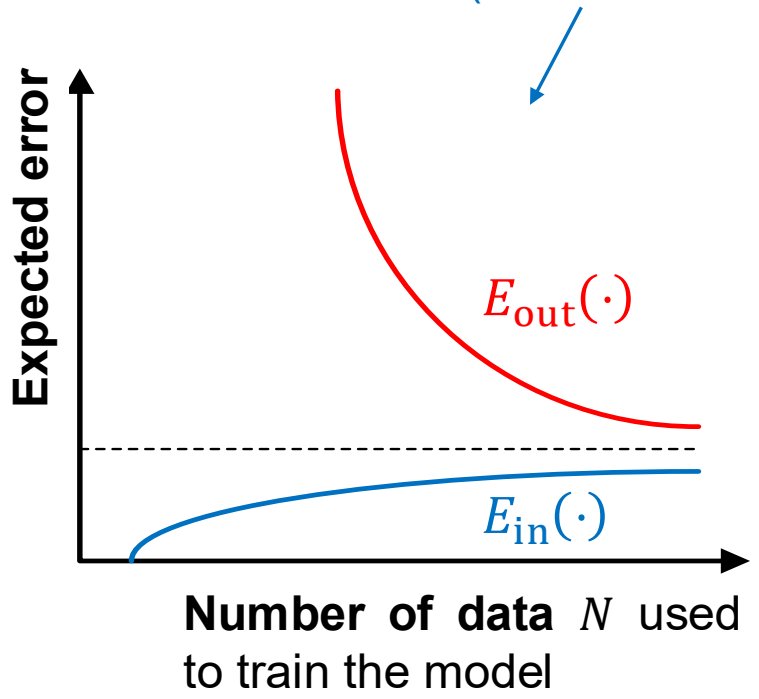
$$J_{\text{aug}}(\Theta) = J(\Theta) + \lambda_{\text{reg}} \cdot \sum_{l=1}^{L} \sum_{k=1}^{K_l} \sum_{j=1}^{K_{l-1}} \left|\Theta_{kj}^{(l)}\right|$$

- $K_0 = d - 1$ (input layer)
- $\lambda_{\text{reg}} \in \mathbb{R}_{\geq 0}$
- **The biases are not penalized**

Remark: remember to **normalize** the features!

Dataset augmentation

The best way to make a complex model generalize better is to **train it on more data** (remember the learning curves)



In practice, the amount of data that we have is **limited**. However, in some data science tasks, it is reasonable to generate new "fake" data

$\Longrightarrow$ **dataset augmentation**

Dataset augmentation is particularly effective in **classification problems where the input is an image**

Original Picture taken from [1] **Generated**

These "distorted" images should all be classified as tiger!

In this context, new data are generated by **zooming, shifting, cropping, flipping and rotating the images while preserving the same class label**

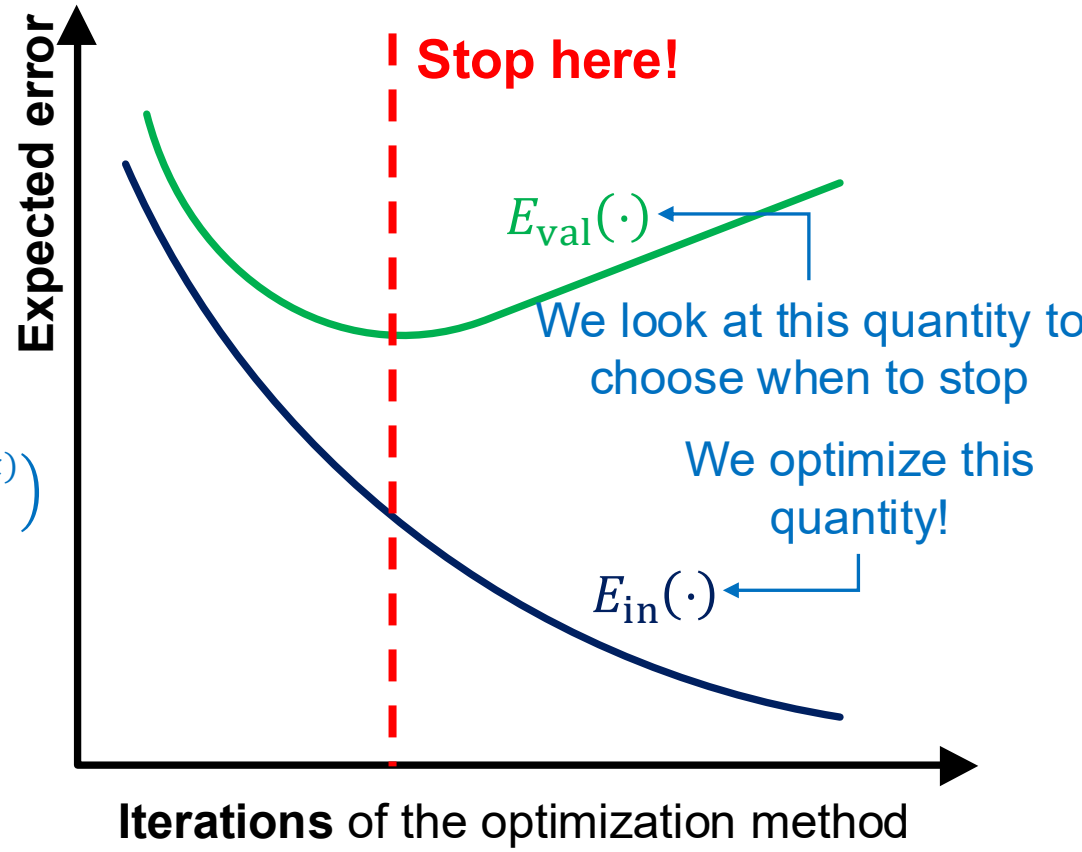
Early stopping

Previously, we have seen that training a neural network amounts to estimating the parameters Θ by minimizing $J(\Theta)$ using a suitable gradient-based optimization procedure

When training very complex models, we are not necessarily interested in finding a set of parameters $\widehat{\Theta}$ that is a **global minimizer** for $J(\Theta) \Longrightarrow$ likely to **overfit** the data

i.e. parameters' updates $\left[\widehat{\Theta}\right]^{(k+1)} = \left[\widehat{\Theta}\right]^{(k)} - \alpha^{(k)} \cdot \nabla \hat{J}\left(\left[\widehat{\Theta}\right]^{(k)}\right)$

Instead, we **monitor** the performances of the model **as the iterations go on** and stop the optimization procedure once we have found a set of parameters that performs well on a **validation set** (**early stopping**)

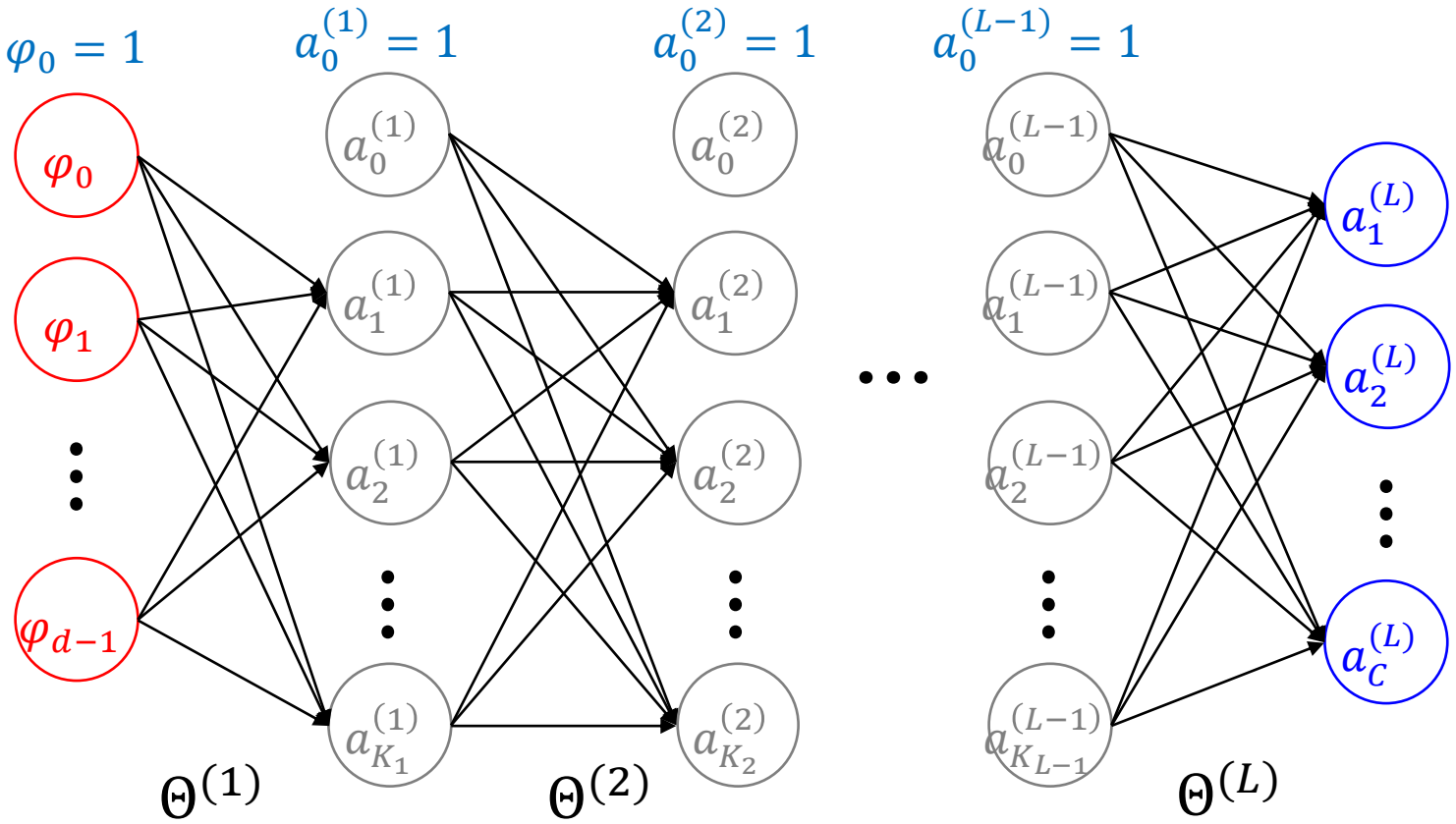


In practice, we are stopping the optimization procedure once we have found a set of parameters that achieves a cost that is sufficiently small, **not** when a minimizer of $J(\Theta)$ is found

Parameter tying and parameter sharing

So far, we have considered each parameter $\Theta_{kj}^{(l)}$ associated to each connection on its own, i.e.

we have assumed that there are <u>no</u> dependencies between the parameters



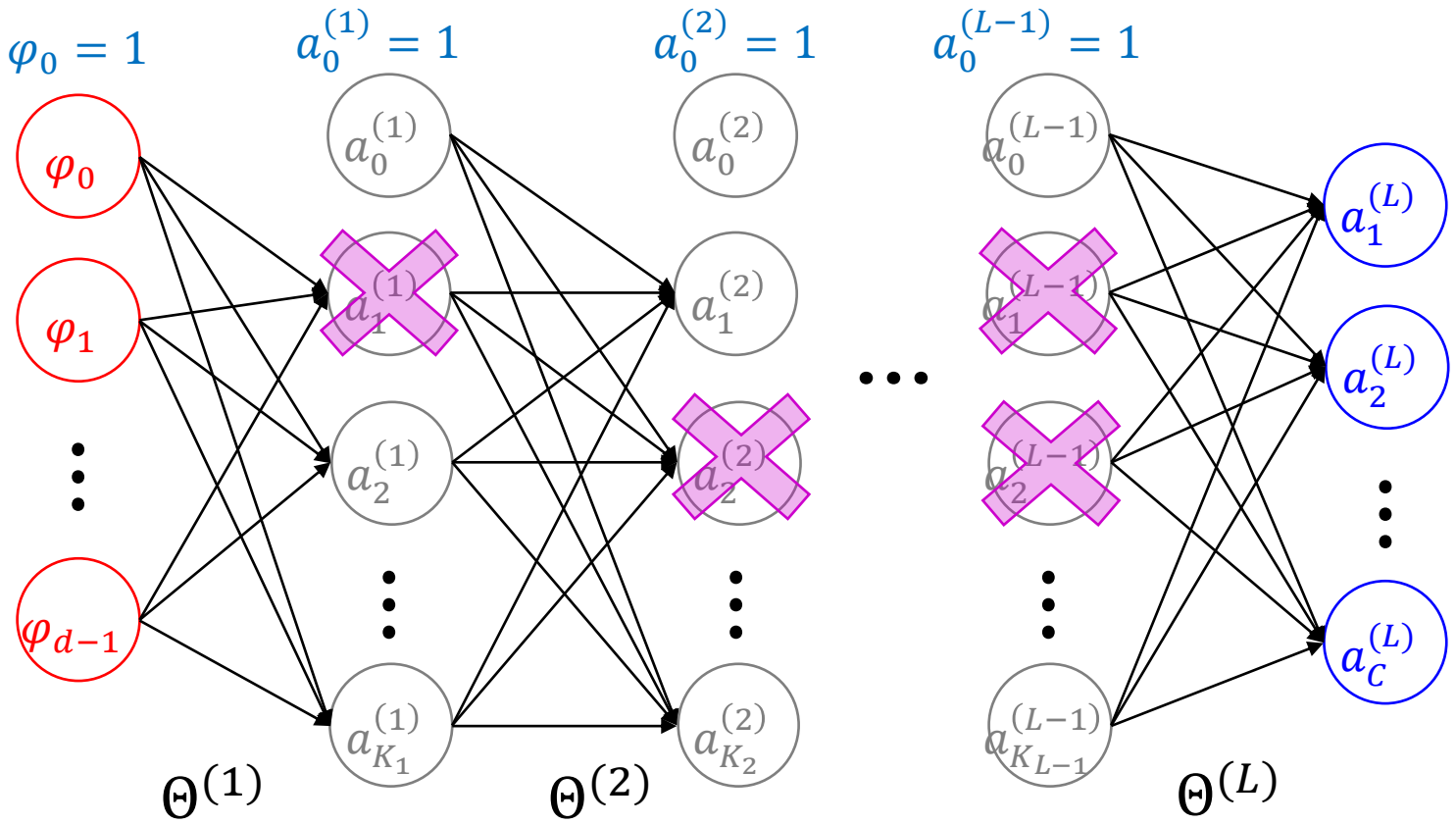
We can perform **regularization** by:

- Forcing some parameters to be **close to one another**
- Forcing some parameters to be **equal**

Particularly relevant for **convolutional neural networks**

Dropout

Dropout is a regularization technique **specifically designed for neural networks**. It consists of:



- (Probabilistically) **turn off (drop) random hidden units** of the network during the training phase
  - These units are **<u>not</u>** considered during forward propagation and backpropagation
- The surviving units stand in for those missing, and **their weights are scaled** appropriately

<u>Remark</u>: need to specify the **probability of dropping a unit** in each hidden layer

Outline

1. Introduction to neural networks
2. Feed-forward neural networks
3. Training neural networks
    a. Cost function
    b. Backpropagation
    c. Stochastic Gradient Descent
    d. Regularization in neural networks
4. **Comparison of supervised learning methods**

Comparison of supervised learning methods

We choose the most suited supervised learning method for the data science task at hand

based on the desired characteristics of the resulting model

Method	Interpretability	Predictive power
Linear regression and logistic regression	▲	▼
Linear regression and logistic regression with **nonlinear** transformations/basis functions	◆ Depends quite a lot on the chosen transformation	◆
Decision trees	▲	▼
Random forests	◆	◆
Neural networks	▼	▲

▲ : good ◆ : fair ▼ : poor

Comparison of supervised learning methods

There are two key principles when selecting which supervised learning method to use:

- **No free lunch theorem**: no single supervised learning method is universally the best-performing algorithm for all problems

- **Occam's razor**: among methods that show similar performances on a given problem, choose the **simplest** one (i.e. the most interpretable)

UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione