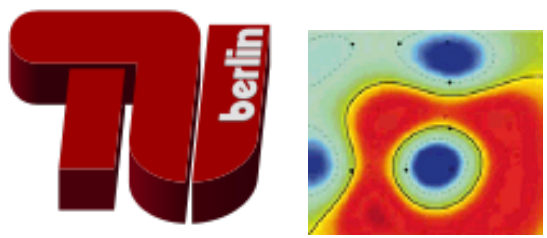
Lecture 12: Neural Networks

Machine Learning 1



TU Berlin – WiSe 2016/2017

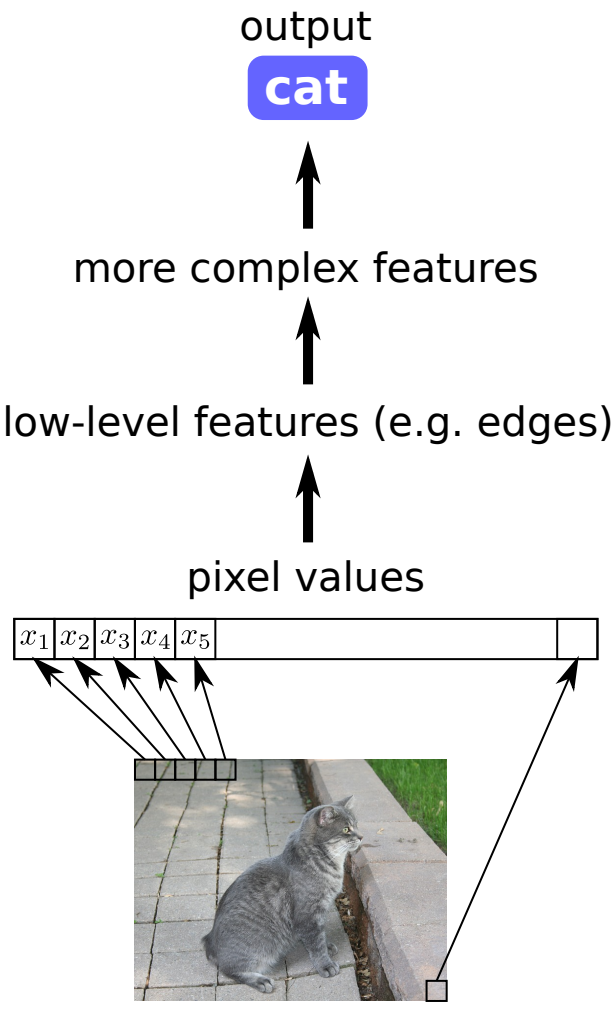
Overview

large-margin,
ridge, ...

feature map,
kernel

Learning

Engineering

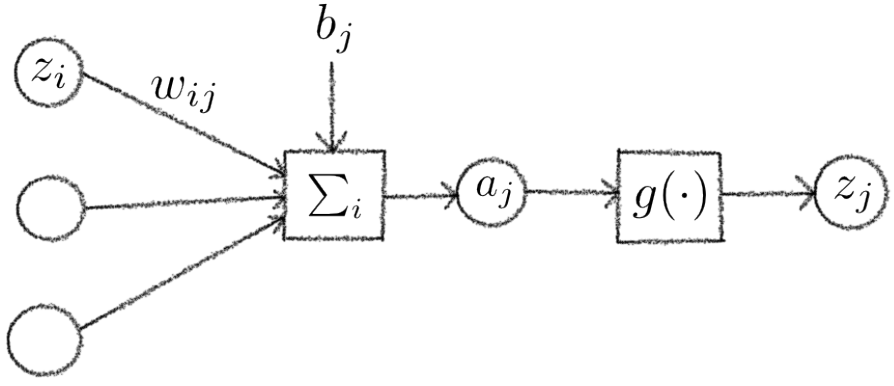


Learning

neurons,
update rule

The Neuron

- Set of inputs $\{z_i\}_{i=1}^{d}$
- Output z_j
- Nonlinear *activation* function $g(\cdot)$
- Set of weights $\{w_{ij}\}_{i=1}^{d}$ and bias b_j



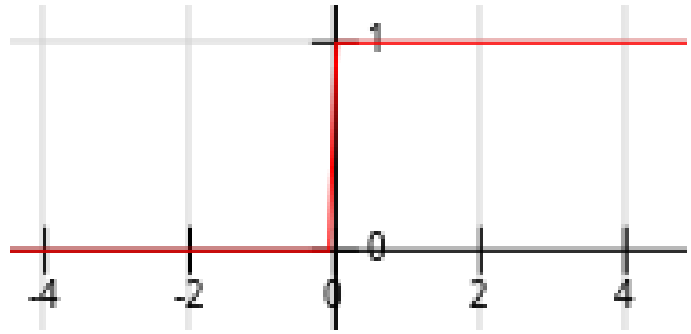
Forward equation:

$$a_j = \sum_{i=1}^{d} z_i \, w_{ij} + b_j \qquad \text{(linear transformation)}$$

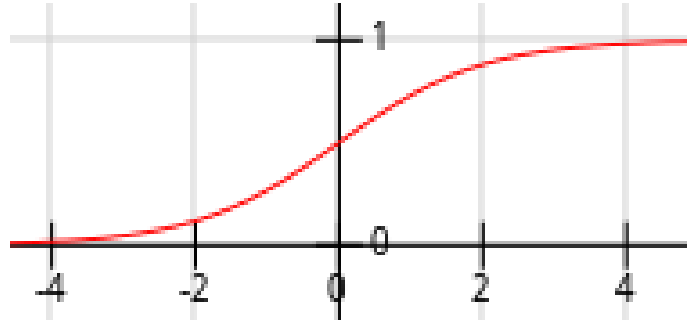
$$z_j = g(a_j) \qquad \text{(activation)}$$

Example of Activation Functions

- *Threshold* activation: $g(a_j) = 1_{\{a_j > 0\}}$

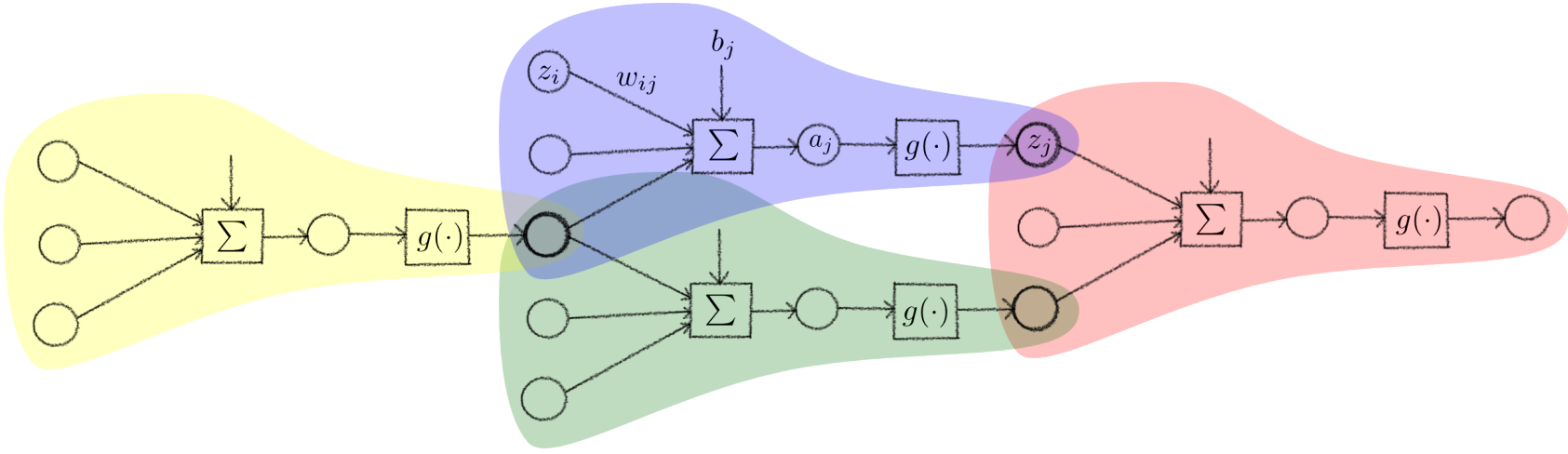


- *Sigmoid* activation: $g(a_j) = \frac{\exp(a_j)}{1+\exp(a_j)}$



Neural Network

Computational graph formed by interconnecting several neurons.

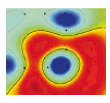


Observations:

- Output of a neuron (yellow) can be used as input by several upper-level neurons (blue, green).
- Output of several neurons (blue, green) can be used as input of an upper-level neuron (red).
- Each neuron has the same structure $(\Sigma, g(\cdot))$, but different parameters $(\{w_{ij}\}, b_j)$.

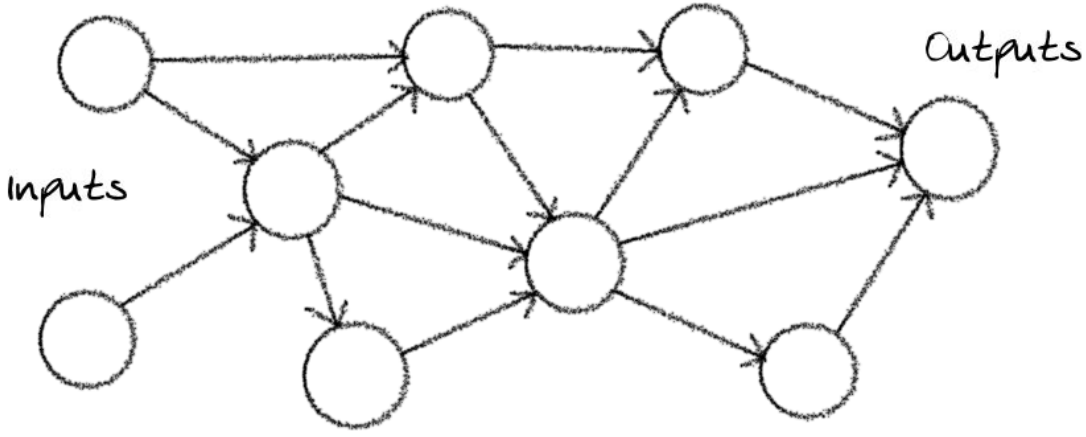
Forward Propagation

[Example on the blackboard]

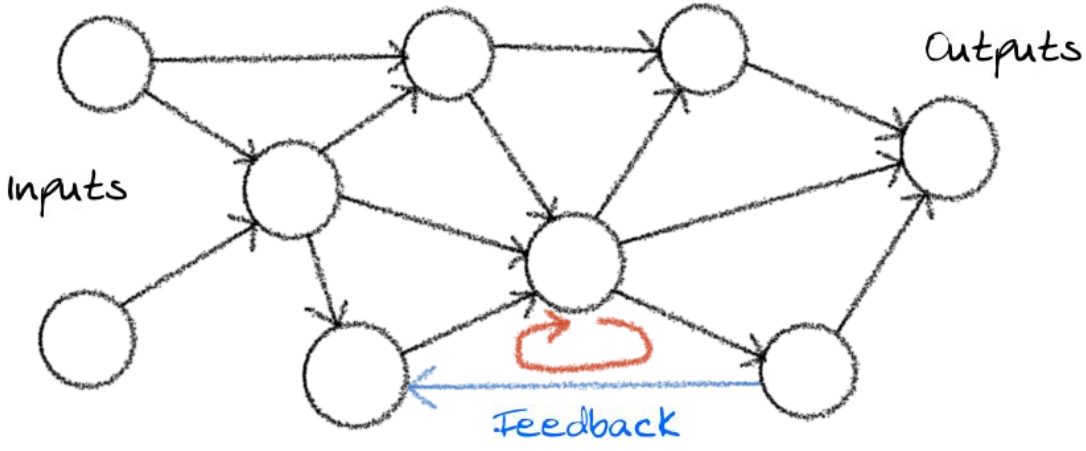


Example of Neural Networks

- **Feedforward** neural network



- **Recurrent** neural network (not covered in this lecture)

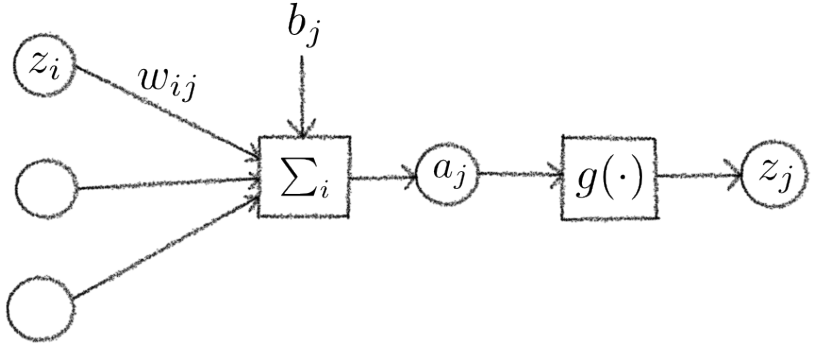


Implementing the Forward Pass

- **Reminder:** Forward equation

$$a_j = \sum_i z_i w_{ij} + b_j$$

$$z_j = g(a_j)$$



- **Idea:** Use a recursive procedure:

```
def forward(self):
    self.a = self.b
    for neuron,weight in self.incoming:
        neuron.forward()
        self.a += neuron.z * weight
    self.z = g(self.a)

for neuron in outputs: neuron.forward()
```

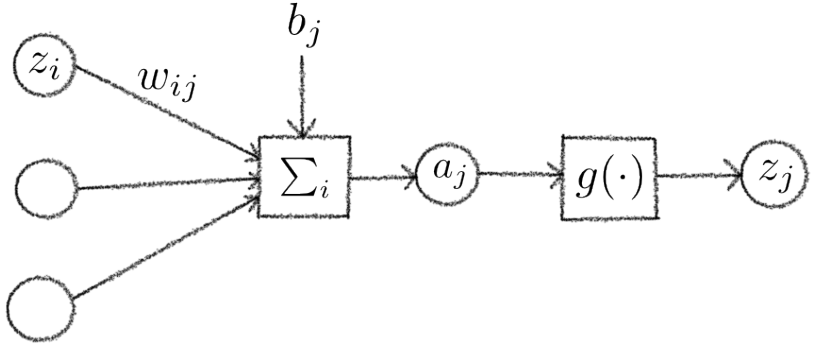
- **Question:** Why is this code slow?

Implementing the Forward Pass

- **Reminder:** Forward equation

$$a_j = \sum_i z_i w_{ij} + b_j$$

$$z_j = g(a_j)$$

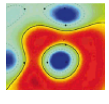


- **Solution:** Recursive procedure **+ memoization**

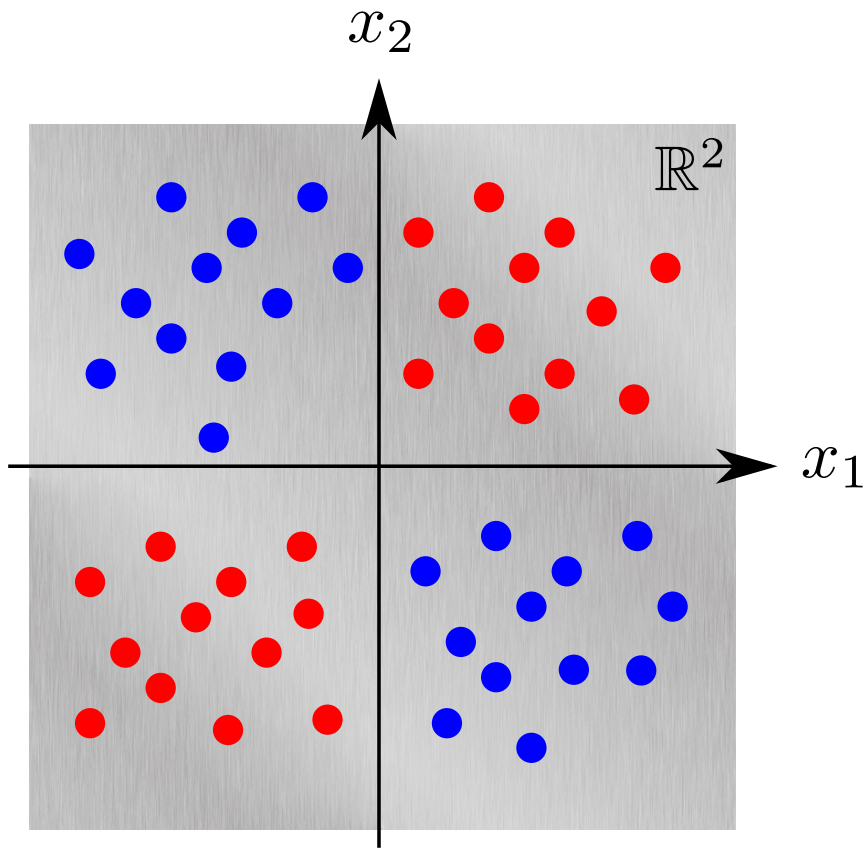
```
def forward(self):
    self.a = self.b
    for neuron,weight in self.incoming:
        if neuron.z == None: neuron.forward()
        self.a += neuron.z * weight
    self.z = g(self.a)

for neuron in outputs: neuron.forward()
```

- Accelerates the forward pass to $O(\#\text{connections})$.

Neural Network Representations



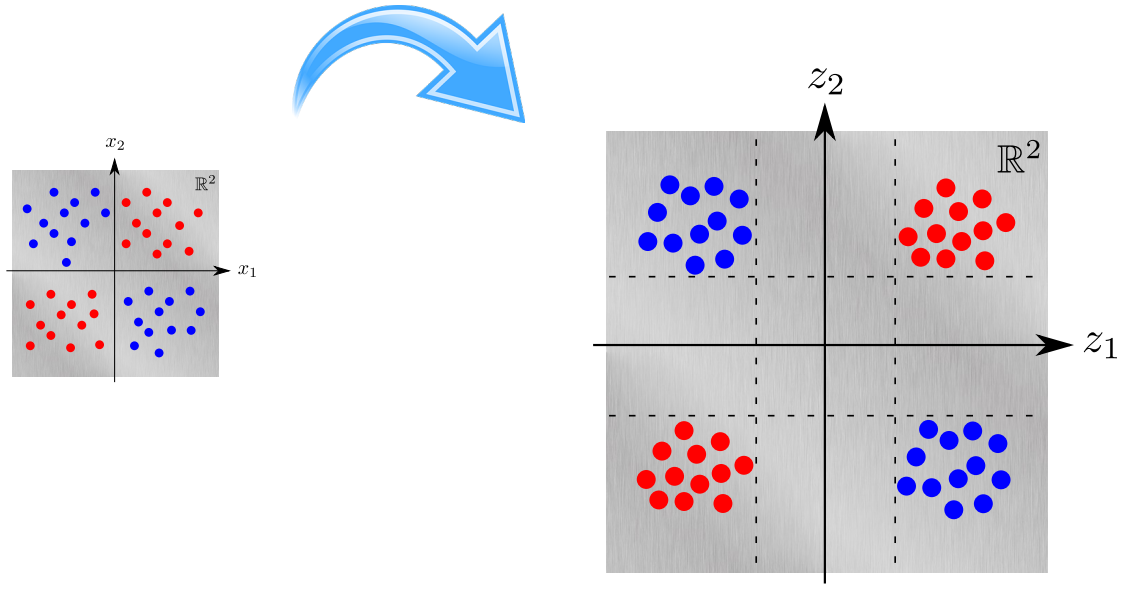
The XOR Problem



- **Question:** Can a neural network solve it?

Solving the XOR Problem (step 1)

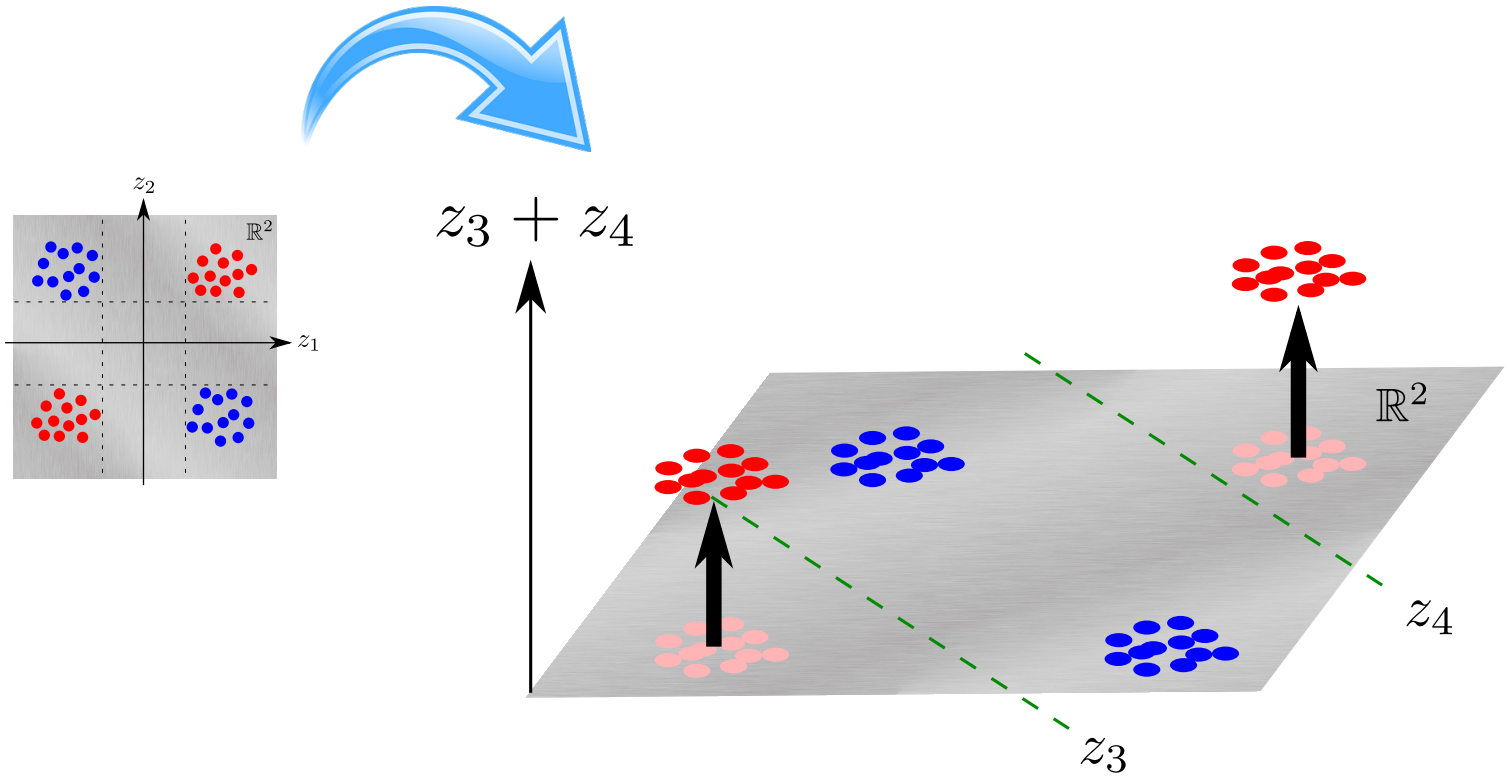
$$z_1 = \tanh(10 \cdot x_1)$$
$$z_2 = \tanh(10 \cdot x_2)$$



- Points are polarized near the coordinates $(-1, -1)$, $(-1, 1)$, $(1, -1)$, $(1, 1)$.
- Each cluster of points becomes linearly separable. (Not yet the problem as a whole)

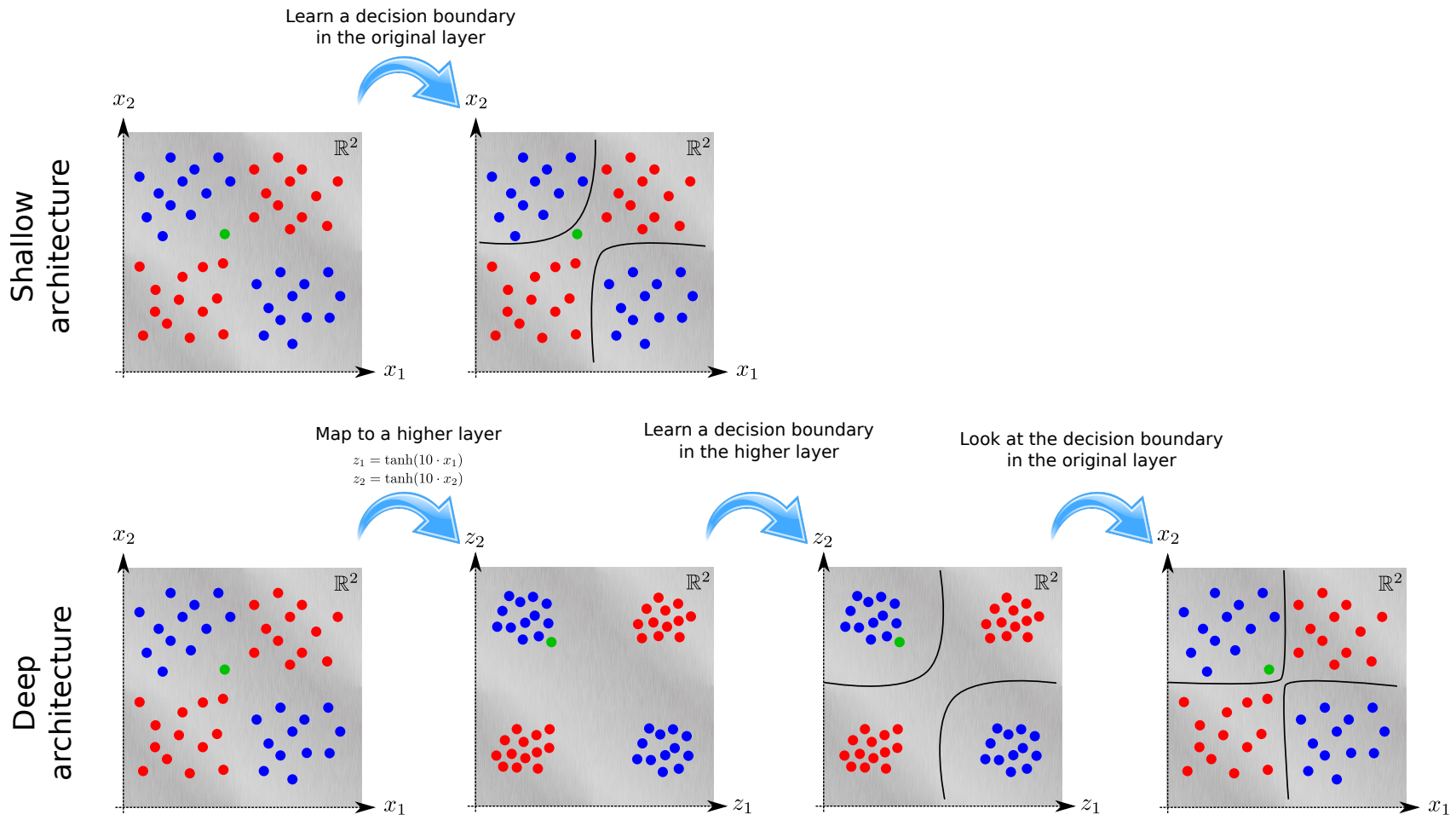
Solving the XOR Problem (step 2)

$$z_3 = \text{sigmoid}(-1 - z_1 - z_2)$$
$$z_4 = \text{sigmoid}(z_1 + z_2 - 1)$$



- Problem is now linearly separable with z_3 and z_4.

Deep Networks are Statistically Efficient

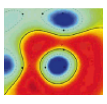


Remark: The above argument *assumes* that the map to the higher layer can be learned accurately without overfitting. It is the case when the same map produces features for *multiple* tasks in higher layers.

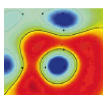
Question:

Can neural networks solve any binary classification problem?

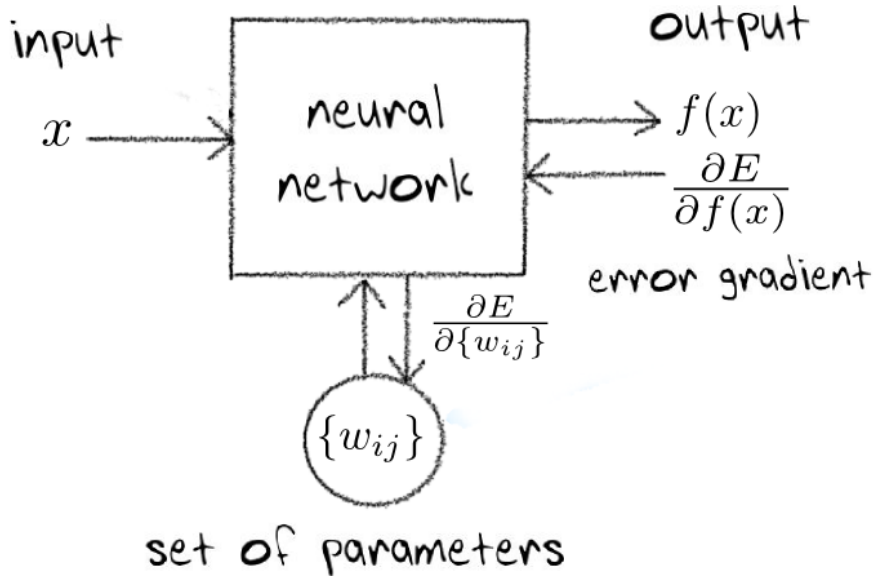
[Proof on the blackboard]

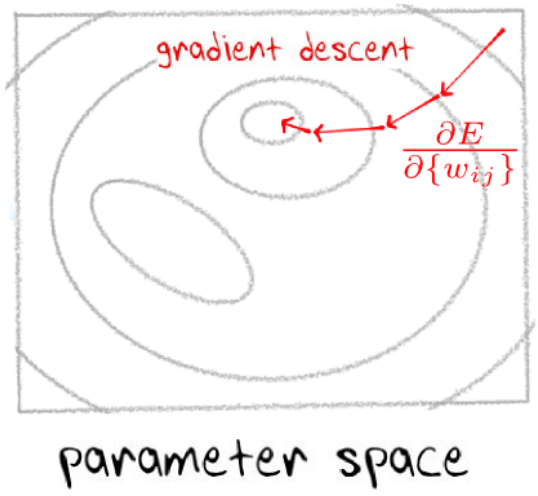


Training Neural Networks



Learning Neural Networks



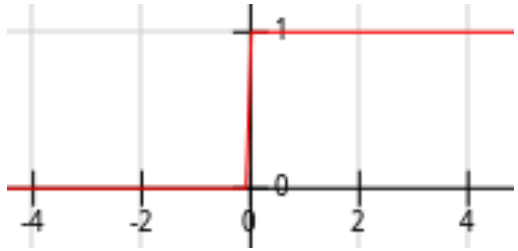


- **Idea:** Compute the gradient of the objective E (e.g. difference between network output and label), with respect to the neural network parameters w_{ij} and b_j and perform gradient descent.
- **Problem:** Gradient of a complex objective function (error of the neural network mapping) is hard to compute analytically.

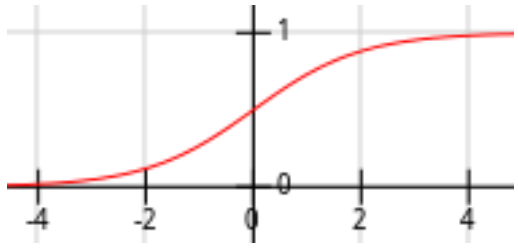
Question:

Which function $g(\cdot)$ is best for training a deep network?

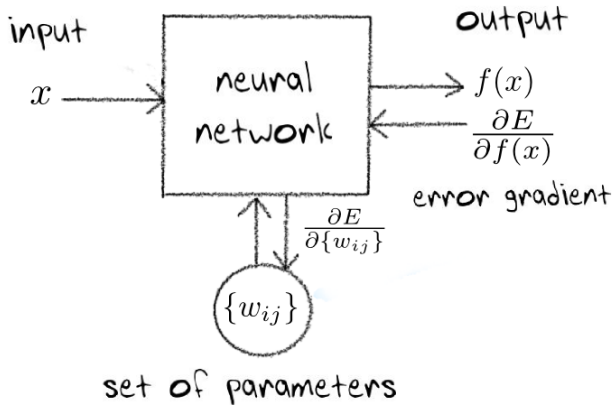
- 1. *Threshold* activation: $g(a_j) = 1_{\{a_j > 0\}}$

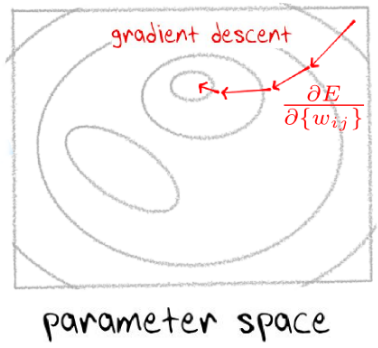


- 2. *Sigmoid* activation: $g(a_j) = \frac{\exp(a_j)}{1+\exp(a_j)}$



Approach 1: Finite Differences



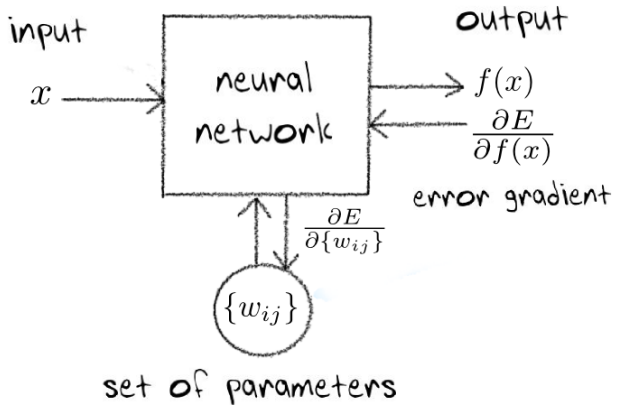


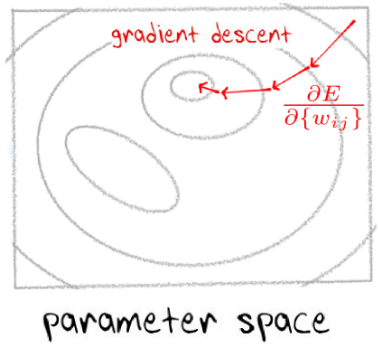
$$\frac{\partial E}{\partial w_{ij}} = \lim_{\varepsilon \to 0} \frac{E(\boldsymbol{w} + \varepsilon\, \delta_{ij}) - E(\boldsymbol{w})}{\varepsilon}$$

Advantage: No need to differentiate the error function with respect to the parameters.

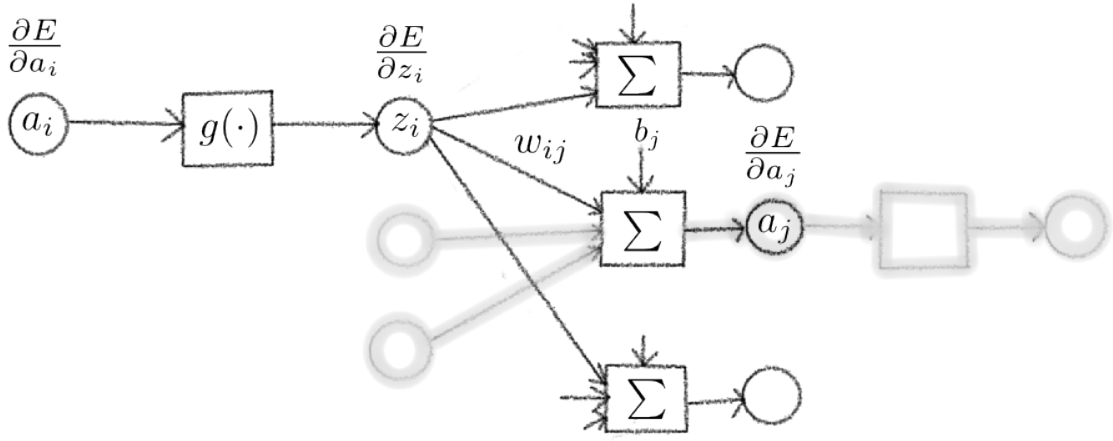
Disadvantage: Need to evaluate the error function $E(\cdot)$ as many times as we have parameters in the network (i.e. once for each component δ_{ij} of the gradient). This is computationally costly.

Approach 2: Error Backpropagation

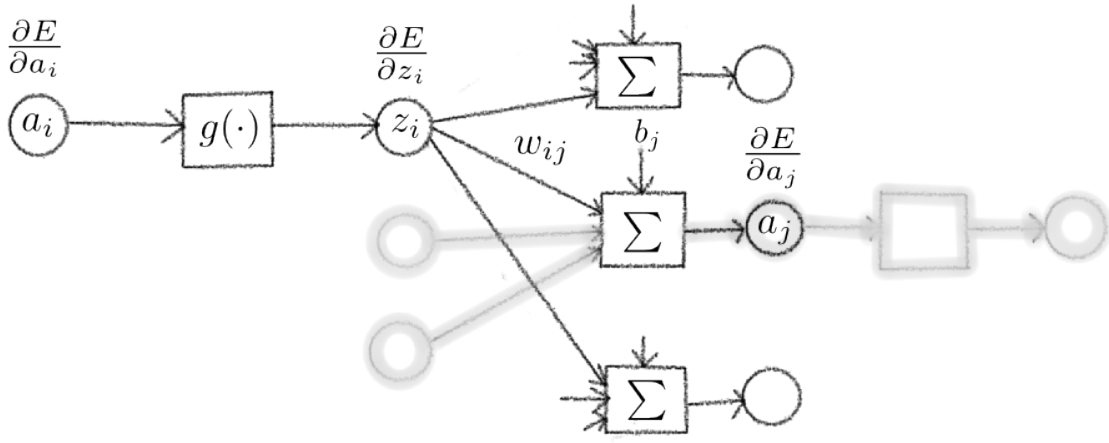




Idea: Error gradient with respect to quantities of a given neuron can be expressed as a function of its outgoing neurons.



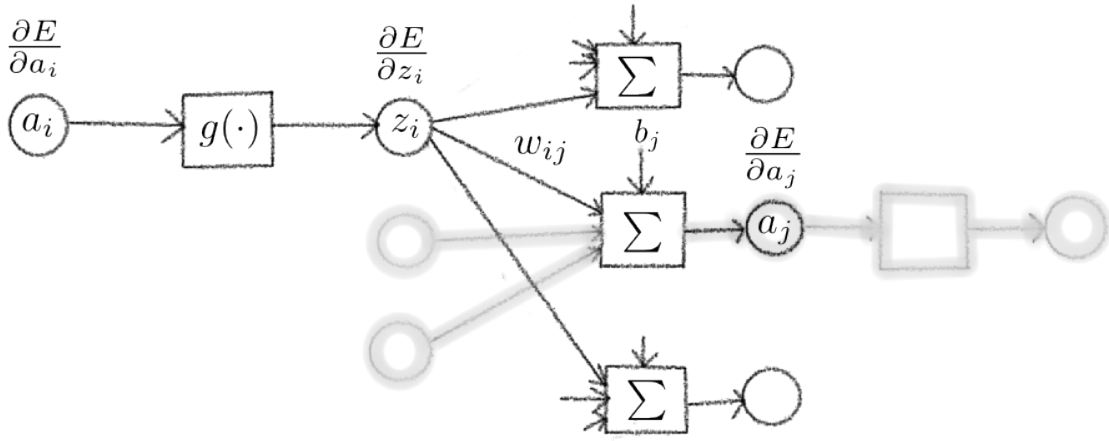
Error Backpropagation



Backward equation:

$$\frac{\partial E}{\partial a_i} = \underbrace{\frac{\partial z_i}{\partial a_i}}_{g'(a_i)} \sum_j \frac{\partial E}{\partial a_j} \underbrace{\frac{\partial a_j}{\partial z_i}}_{w_{ij}}$$

Computing the Parameter Gradient



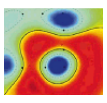
Parameter gradient:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j} \underbrace{\frac{\partial a_j}{\partial w_{ij}}}_{z_i} \quad \text{and} \quad \frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial a_j} \underbrace{\frac{\partial a_j}{\partial b_j}}_{1}$$

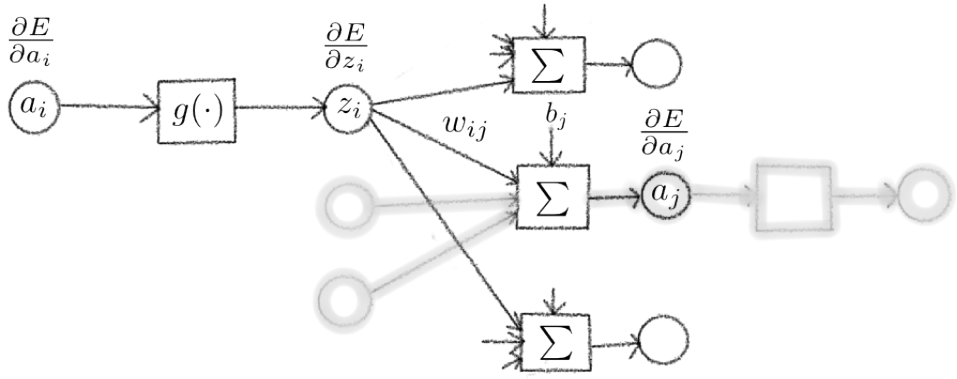
where z_i was computed in the *forward* pass and $\frac{\partial E}{\partial a_j}$ was computed in the *backward* pass.

Error Backpropagation

[Example on the blackboard]



Implementing the Backward Pass



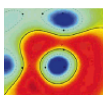
- **Backward equation:**

$$\frac{\partial E}{\partial a_i} = g'(a_i) \cdot \sum_j \frac{\partial E}{\partial a_j} \cdot w_{ij}$$

- **Recursive procedure:**

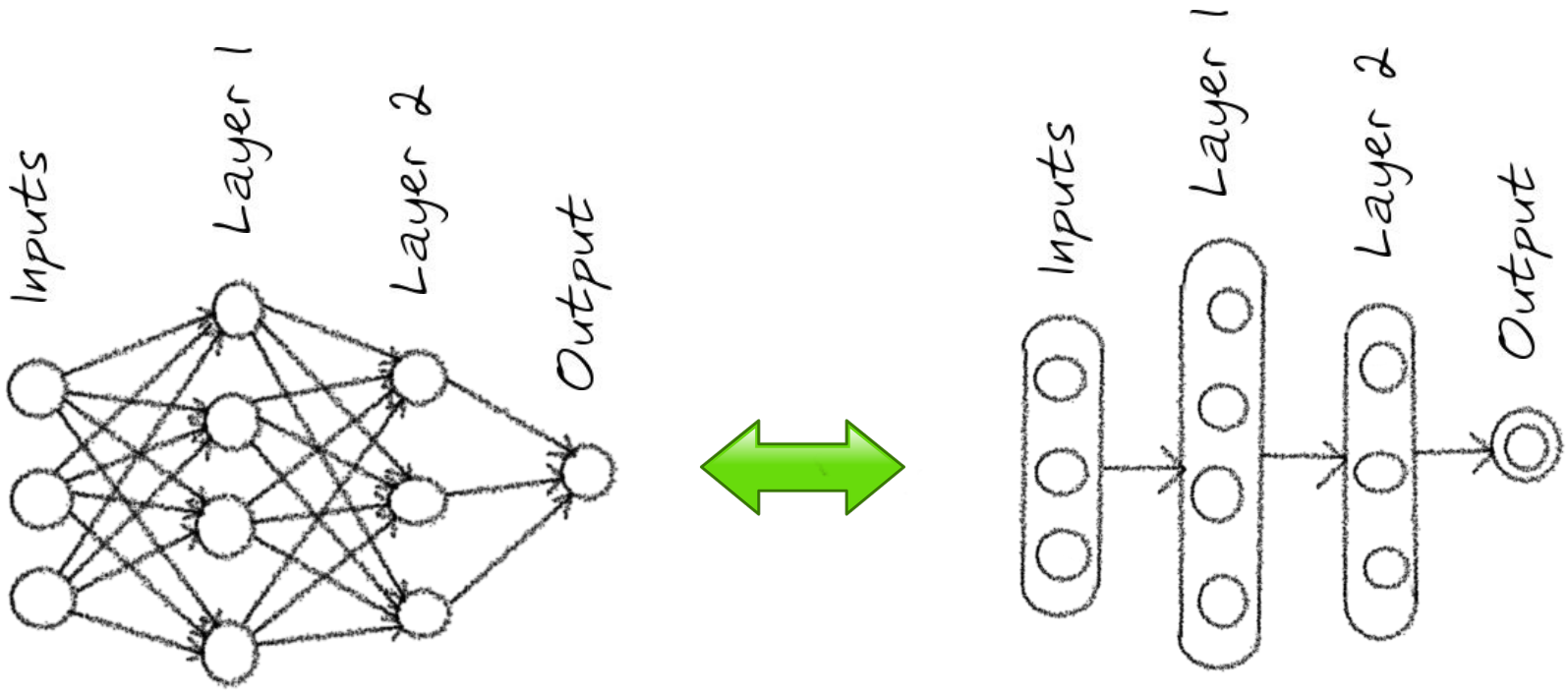
```
def backward(self):
    dEdz = 0
    for neuron,weight in self.outgoing:
        if neuron.dEda == None: neuron.backward()
        dEdz += weight * neuron.dEda
    self.dEda = dg(self.a) * dEdz
```

Layered Architectures

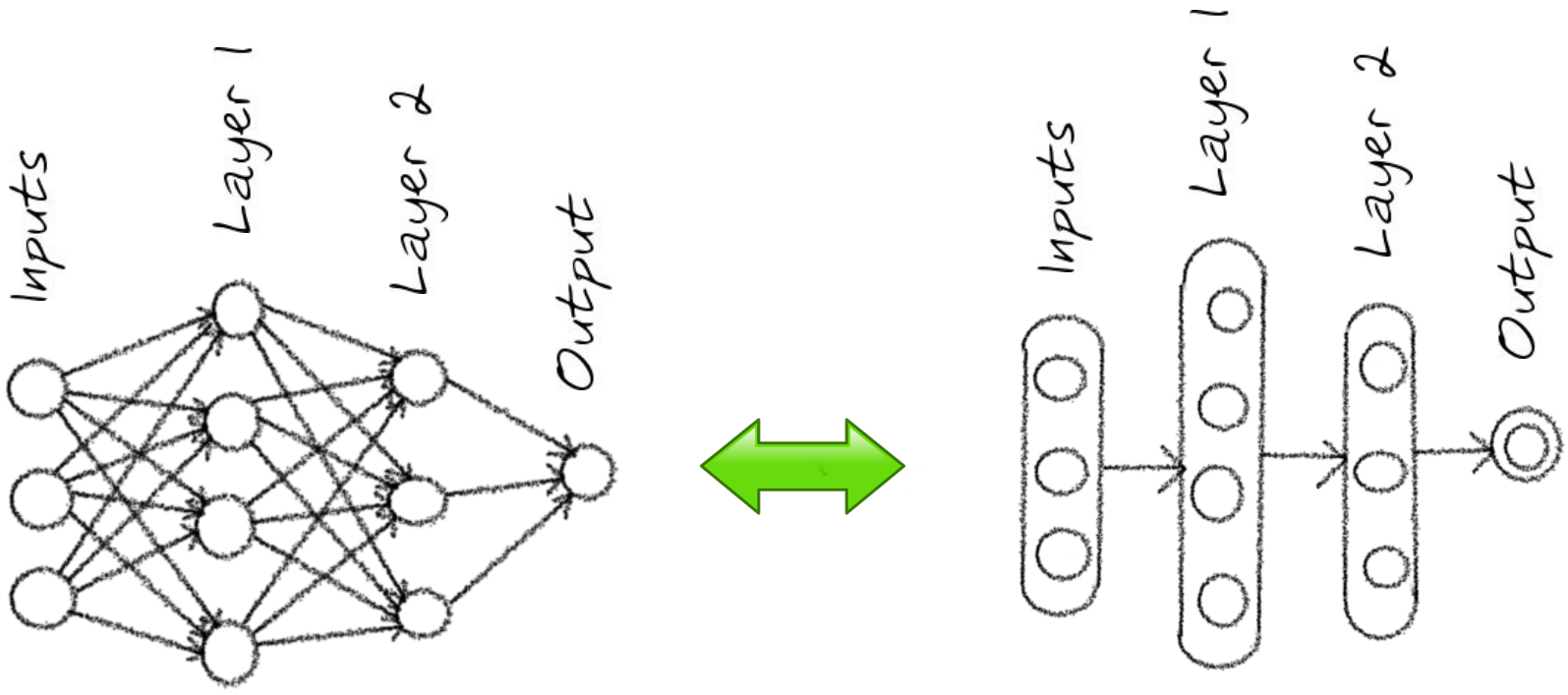


Layered Neural Network Architectures

- **Observation:** Graph-defined neural networks do not take advantage of fast matrix multiplication (e.g. `numpy.dot()`).
- **Idea:** Restrict neural network connectivity so that forward and backward equations can be expressed as matrix products.
- **Possible architecture:** Multi-layer neural network



Multi-Layer Neural Network



- $\boldsymbol{z}_l$: vector of neuron activations at layer l
- $W_{l,l+1}$: matrix of weights connecting layer l to layer $l+1$
- $\boldsymbol{b}_l$: vector of biases at layer l
- $\boldsymbol{g}(\cdot)$ function that applies elementwise the function $g(\cdot)$

Multi-Layer Neural Network

- $\boldsymbol{z}_l$: vector of neuron activations at layer l
- $W_{l,l+1}$: matrix of weights connecting layer l to layer $l+1$
- $\boldsymbol{b}_l$: vector of biases at layer l
- $\boldsymbol{g}(\cdot)$ function that applies elementwise the function $g(\cdot)$

Forward equations:

- $\boldsymbol{a}_{l+1} = W_{l,l+1}^{\top} \cdot \boldsymbol{z}_l + \boldsymbol{b}_{l+1}$
- $\boldsymbol{z}_{l+1} = \boldsymbol{g}(\boldsymbol{a}_{l+1})$

Backward equations:

- $\frac{\partial E}{\partial \boldsymbol{z}_l} = W_{l,l+1} \cdot \frac{\partial E}{\partial \boldsymbol{a}_{l+1}}$
- $\frac{\partial E}{\partial \boldsymbol{a}_l} = \boldsymbol{g}'(\boldsymbol{a}_l) \odot \frac{\partial E}{\partial \boldsymbol{z}_l}$ ($\odot$ is the element-wise product)

Parameter gradient:

- $\frac{\partial E}{\partial W_{l,l+1}} = \boldsymbol{z}_l \cdot \frac{\partial E}{\partial \boldsymbol{a}_{l+1}}^{\top}$
- $\frac{\partial E}{\partial \boldsymbol{b}_l} = \frac{\partial E}{\partial \boldsymbol{a}_l}$