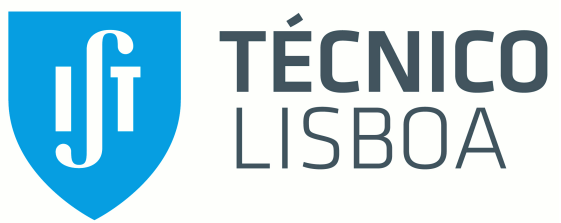
Lecture 5: Neural Networks I

André Martins, Francisco Melo, Mário Figueiredo



Deep Learning Course, Fall 2021

Today's Roadmap

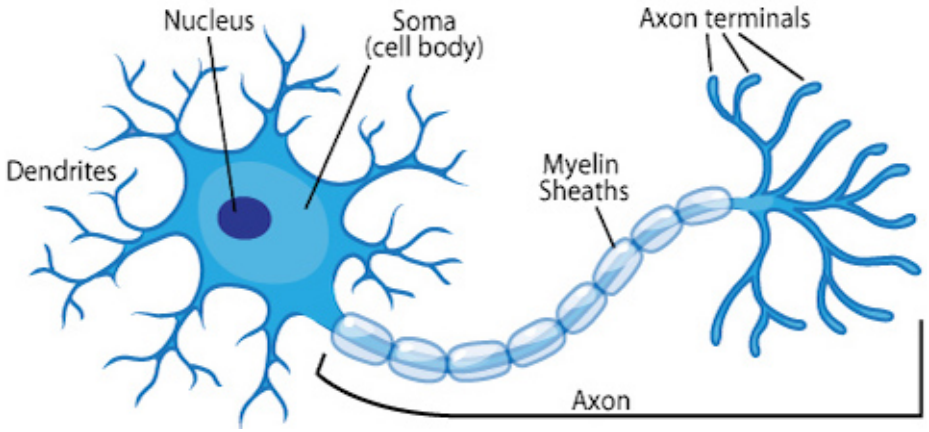
Today's lecture is about neural networks:

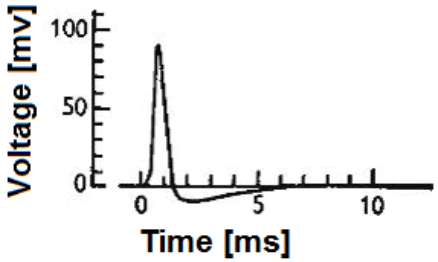
- From perceptron to multi-layer perceptron
- Feed-forward neural networks
- Activation functions: sigmoid, tanh, relu, ...
- Activation maps: softmax, sparsemax, ...
- Non-convex optimization and local minima
- Universal approximation theorem
- Gradient backpropagation

Outline

Outline

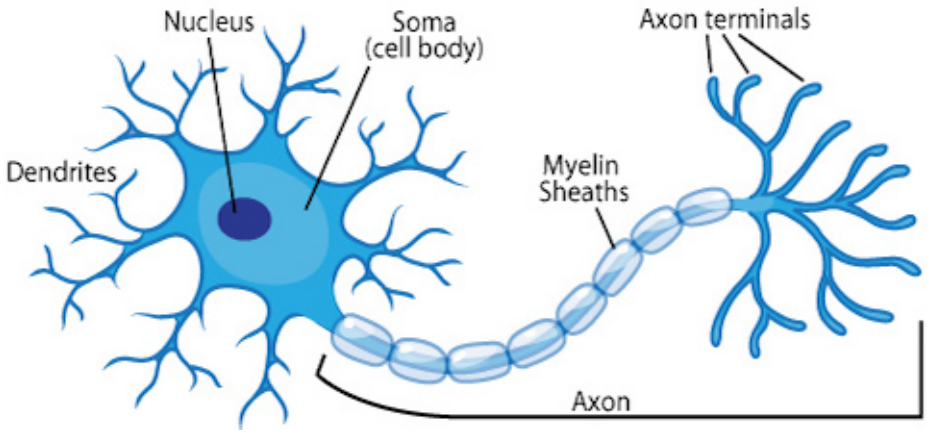
Biological Neuron

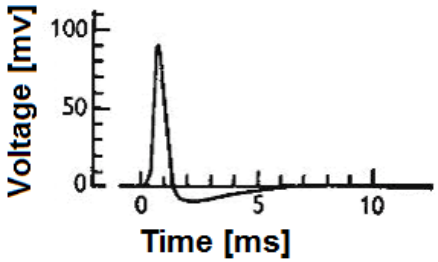




- Three main parts: the main body (soma), dendrites and an axon

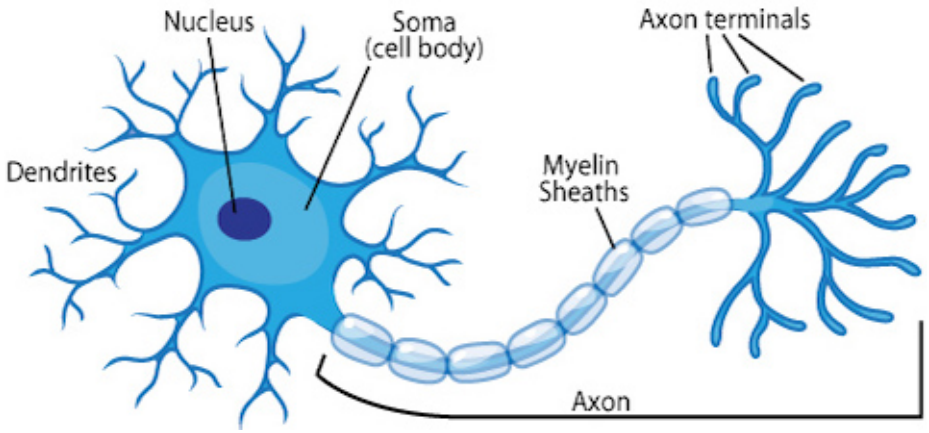
Biological Neuron

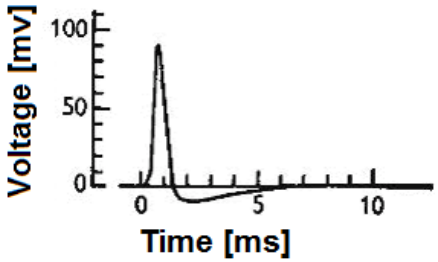




- Three main parts: the main body (soma), dendrites and an axon
- The neuron receives input signals from dendrites, and outputs its own signals through the axon

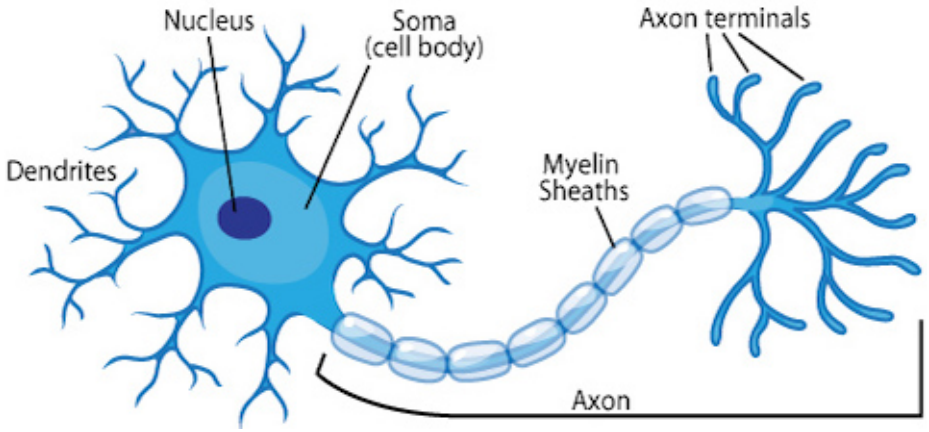
Biological Neuron

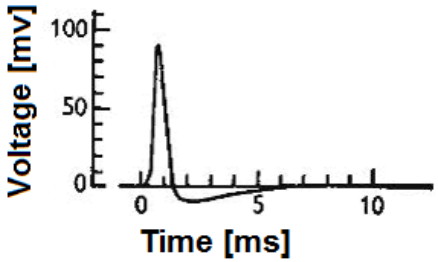




- Three main parts: the main body (soma), dendrites and an axon
- The neuron receives input signals from dendrites, and outputs its own signals through the axon
- Axons in turn connect to the dendrites of other neurons; the connections called synapses

Biological Neuron

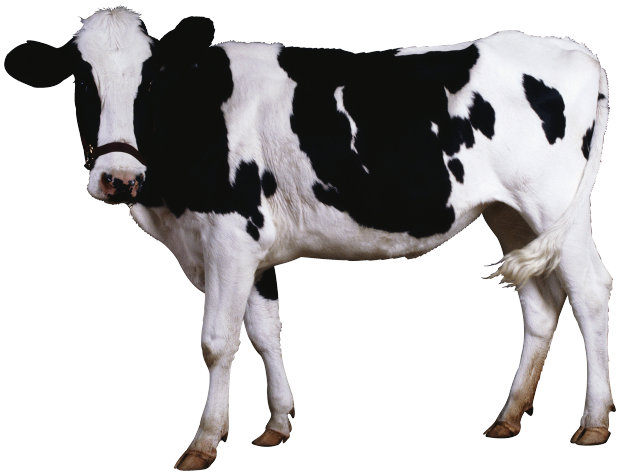




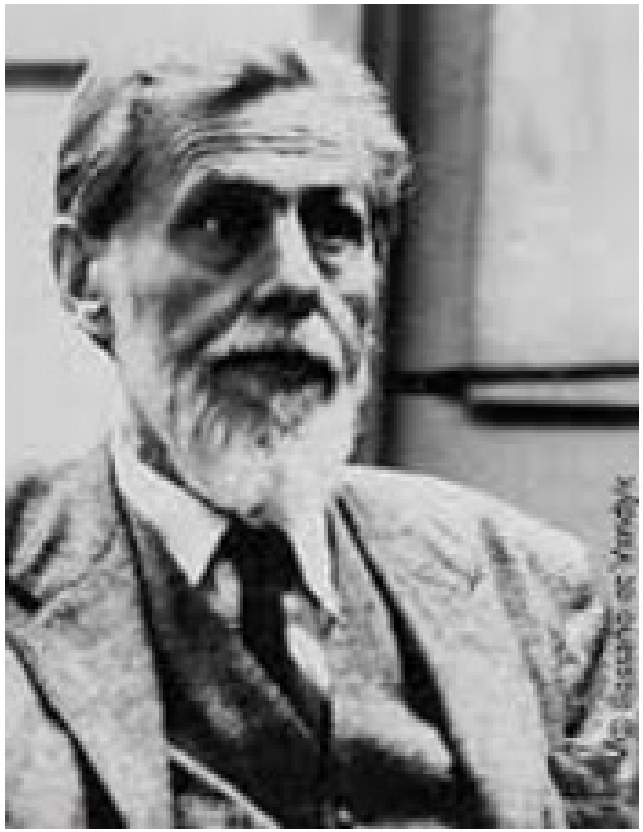
- Three main parts: the main body (soma), dendrites and an axon
- The neuron receives input signals from dendrites, and outputs its own signals through the axon
- Axons in turn connect to the dendrites of other neurons; the connections called synapses
- Generate sharp electrical potentials across their cell membrane (spikes), the main signalling unit of the nervous system

Word of Caution

- Artificial neurons are inspired by biological neurons, but...



Warren McCulloch and Walter Pitts



- The earliest computational model of a neuron, via threshold logic (McCulloch and Pitts, 1943).

Artificial Neuron (McCulloch and Pitts, 1943)

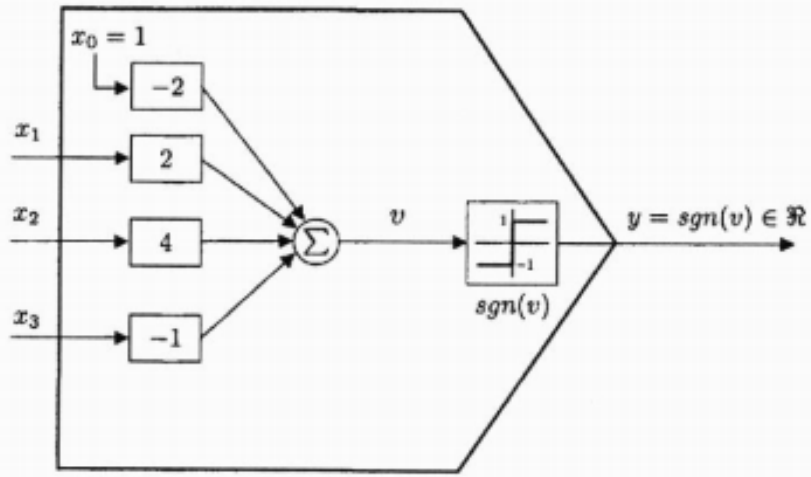


Figure 3.6 Example 3.2: a threshold neural logic for $y = x_2(x_1 + \overline{x}_3)$.

Table 3.6 **Truth table for Example 3.2**

Neural Inputs			$v = \boldsymbol{w}_a^T \boldsymbol{x}_a$	$y = sgn(v)$
x_1	x_2	x_3	$= -2 + 2x_1 + 4x_2 - x_3$	$= sgn(\boldsymbol{w}_a^T \boldsymbol{x}_a)$
-1	-1	-1	-7	-1
-1	-1	1	-9	-1
-1	1	-1	1	1
-1	1	1	-1	-1
1	-1	-1	-3	-1
1	-1	1	-5	-1
1	1	-1	5	1
1	1	1	3	1

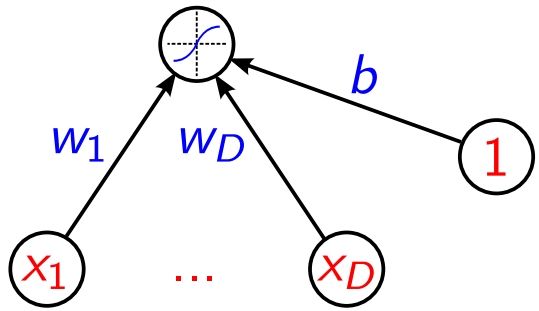
- Later models replaced the hard threshold by more general activations

Artificial Neuron

- **Pre-activation** (input activation):

$$z(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b = \sum_{i=1}^{D} w_i x_i + b,$$

where $\boldsymbol{w}$ are the connection weights and b is a bias term.

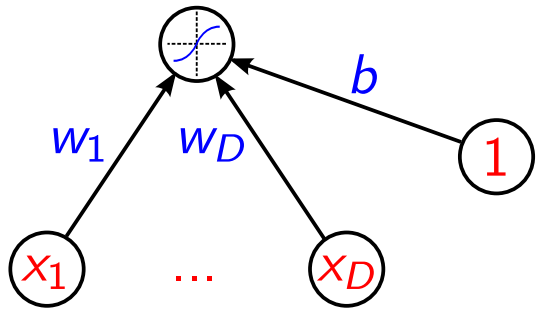


Artificial Neuron

- **Pre-activation** (input activation):

$$z(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b = \sum_{i=1}^{D} w_i x_i + b,$$

where $\boldsymbol{w}$ are the connection weights and b is a bias term.



- **Activation:**

$$h(\boldsymbol{x}) = g(z(\boldsymbol{x})) = g(\boldsymbol{w} \cdot \boldsymbol{x} + b),$$

where $g : \mathbb{R} \to \mathbb{R}$ is the activation function.

Outline

Activation Function

Typical choices:

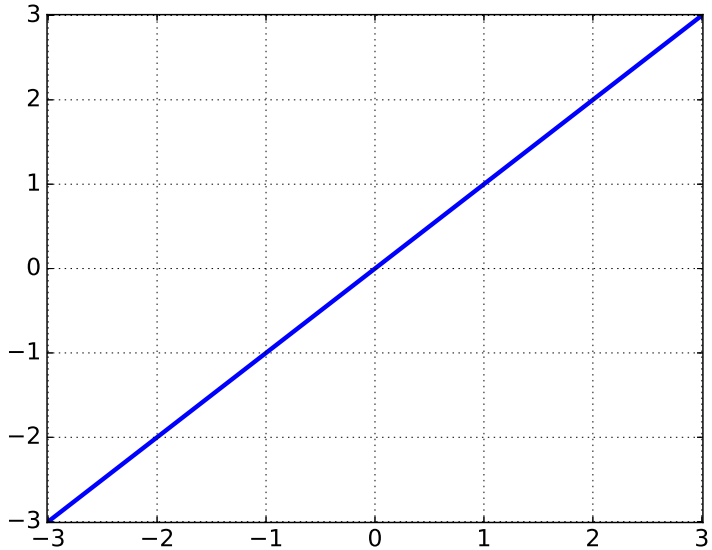
- Linear
- Sigmoid (logistic function)
- Hyperbolic Tangent
- Rectified Linear

Later:

- Softmax
- Sparsemax
- Max-pooling

Linear Activation

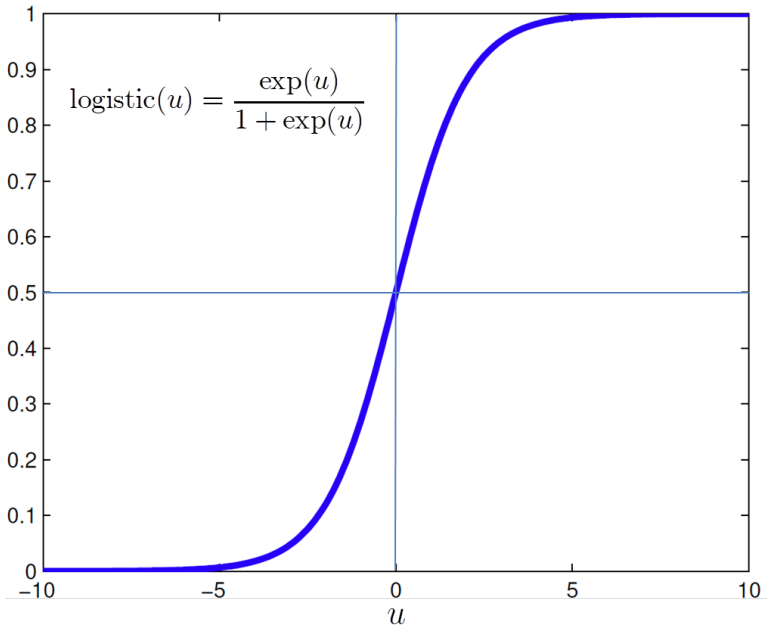
$$\boxed{g(z) = z}$$



- No "squashing" of the input
- Composing layers of linear units is equivalent to a single linear layer: no expressive power increase by using multiple layers (more later)
- Still useful to linear-project the input to a lower dimension

Sigmoid Activation

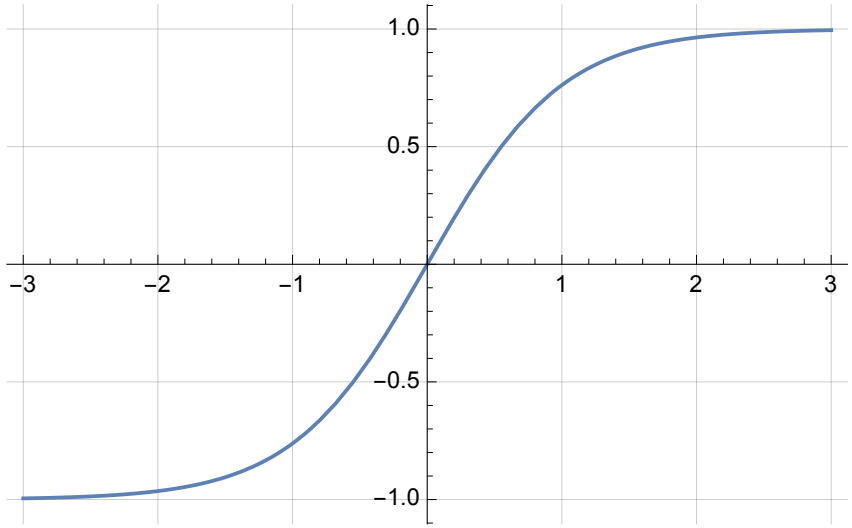
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



- "Squashes" the neuron pre-activation between 0 and 1
- The output can be interpreted as a probability
- Positive, bounded, strictly increasing
- Logistic regression corresponds to a network with a single sigmoid unit
- Combining layers of sigmoid units increases expressiveness (more later)

Hyperbolic Tangent Activation

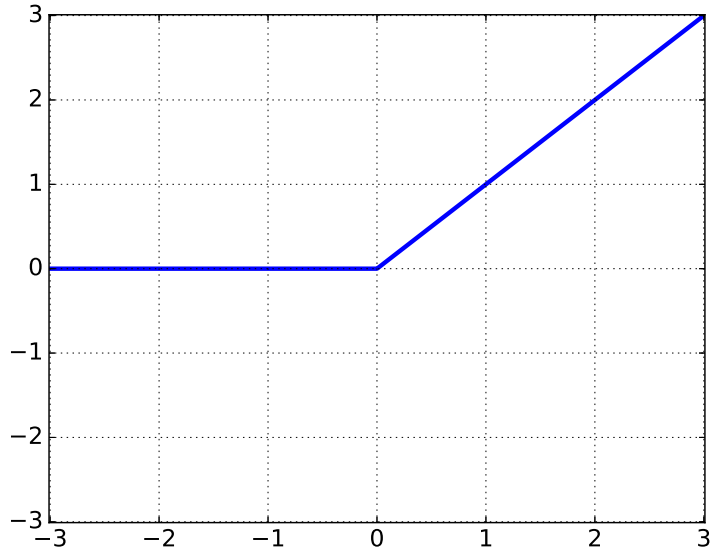
$$g(z) = \textbf{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



- "Squashes" the neuron pre-activation between -1 and 1
- Related to the sigmoid via $\sigma(z) = \frac{1+\textbf{tanh}(z/2)}{2}$
- Can be positive or negative, bounded, strictly increasing
- Combining layers of tanh units increases expressiveness (more later)

Rectified Linear Unit Activation (Glorot et al., 2011)

$$g(z) = \textbf{relu}(z) = \max\{0, z\}$$



- Non-negative, increasing, but not upper bounded
- Not differentiable at 0
- Leads to neurons with sparse activities (biologically more plausible)
- Less prone to vanishing gradients (more later), and historically the first activation that allowed training deep nets without unsupervised pre-training (Glorot et al., 2011)

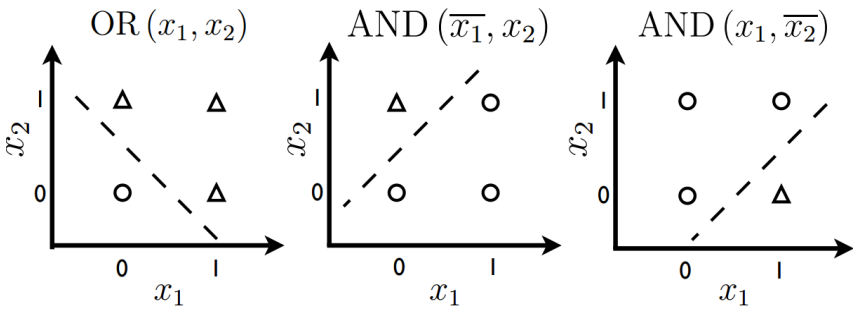
Outline

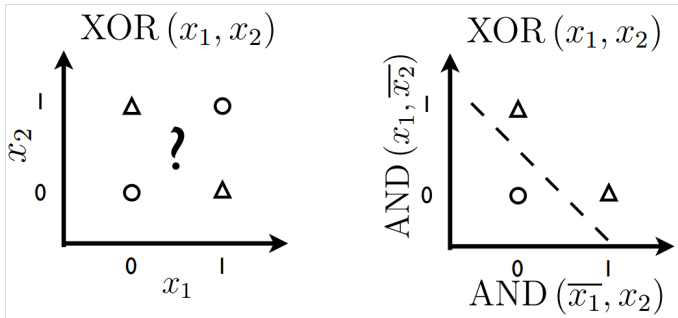
Capacity of Single Neuron (Linear Classifier)

- With a single sigmoid activated neuron we recover logistic regression:

$$p(y = 1|\boldsymbol{x}) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b).$$

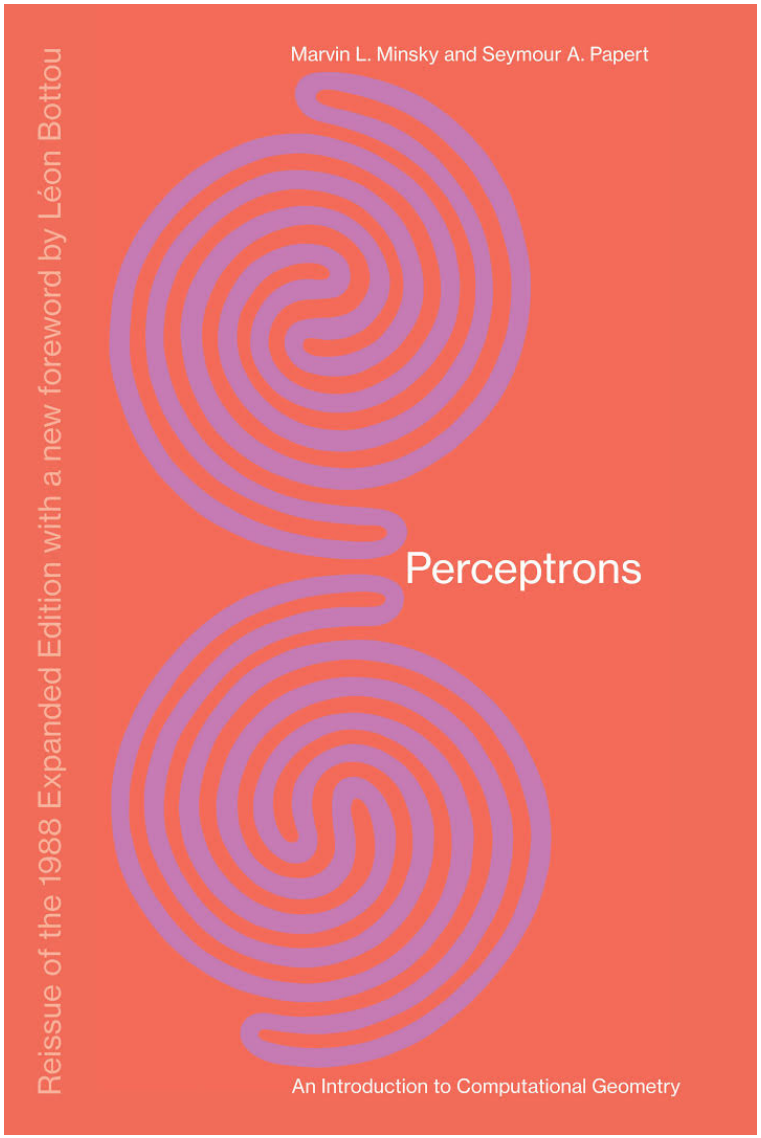
- Can solve linearly separable problems (OR, AND)
- Can't solve non-linearly separable problems (XOR)—unless input is transformed into a better representation





(Slide credit: Hugo Larochelle)

The XOR Affair



Minsky and Papert (1969):

- Misunderstood by many as showing a single perceptron cannot learn XOR (in fact, this was already well-known at the time)
- Fostered an "AI winter" period

Solving XOR with Multi-Layer Perceptron

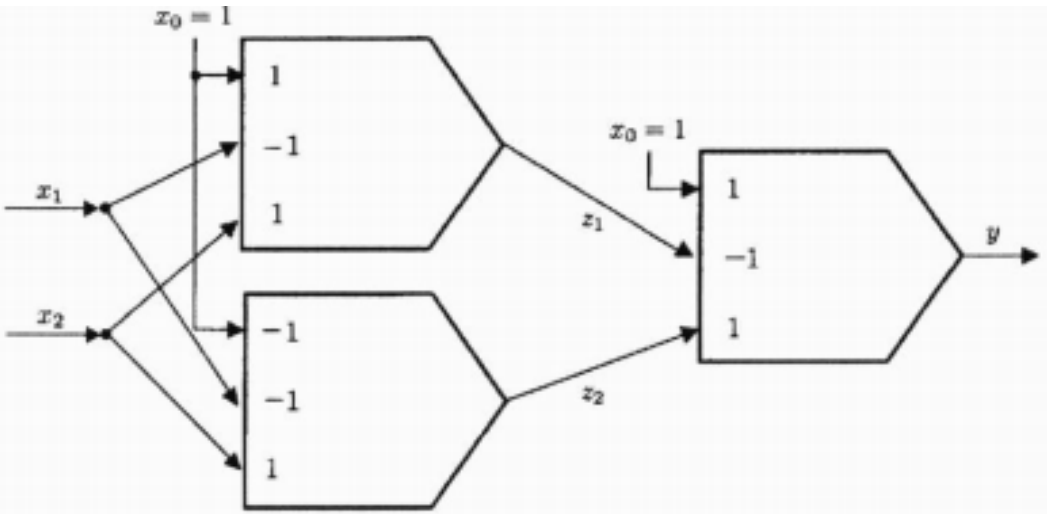


Figure 3.11 Example 3.6: a threshold network for *XOR* function $y = x_1 \oplus x_2 = sgn(1 - z_1 + z_2)$, $z_1 = sgn(1 - x_1 + x_2)$, $z_2 = sgn(-1 - x_1 + x_2)$.

Solving XOR with Multi-Layer Perceptron

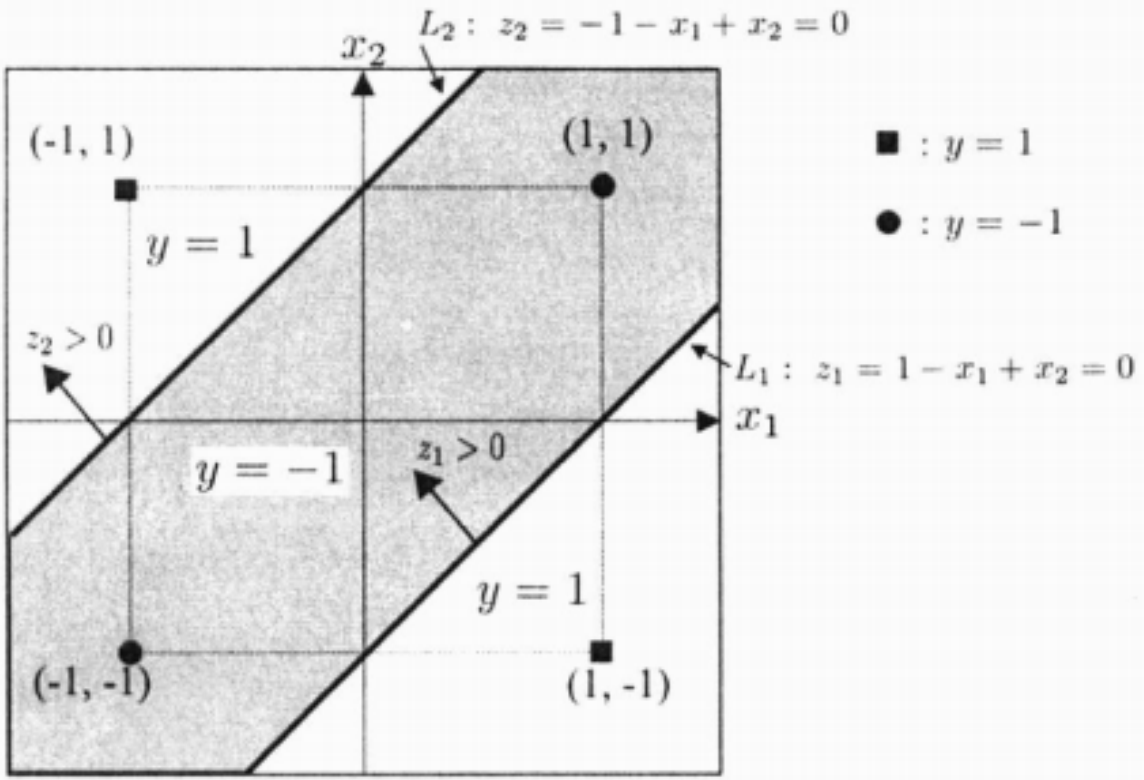


Figure 3.12 Example 3.6: two discriminant lines for *XOR* function $y = x_1 \oplus x_2$.

Solving XOR with Multi-Layer Perceptron

- This construction was known even by McCulloch and Pitts
- The negative result in Minsky and Papert (1969) is that, to learn arbitrary logic functions, each hidden unit needs to be connected to **all inputs**
- At the time, there was some hope that we'd only need "local" neurons

Multi-Layer Neural Network

- **Key idea:** add intermediate layers of artificial neurons before the final output layer

Multi-Layer Neural Network

- **Key idea:** add intermediate layers of artificial neurons before the final output layer

- Each of these hidden units computes some representation of the input and propagates forward that representation

Multi-Layer Neural Network

- **Key idea:** add intermediate layers of artificial neurons before the final output layer
- Each of these hidden units computes some representation of the input and propagates forward that representation
- This increases the expressive power of the network, yielding more complex, non-linear, functions/classifiers

Multi-Layer Neural Network

- **Key idea:** add intermediate layers of artificial neurons before the final output layer
- Each of these hidden units computes some representation of the input and propagates forward that representation
- This increases the expressive power of the network, yielding more complex, non-linear, functions/classifiers
- Similar role as latent variables in probabilistic models, but no need for a probability semantics

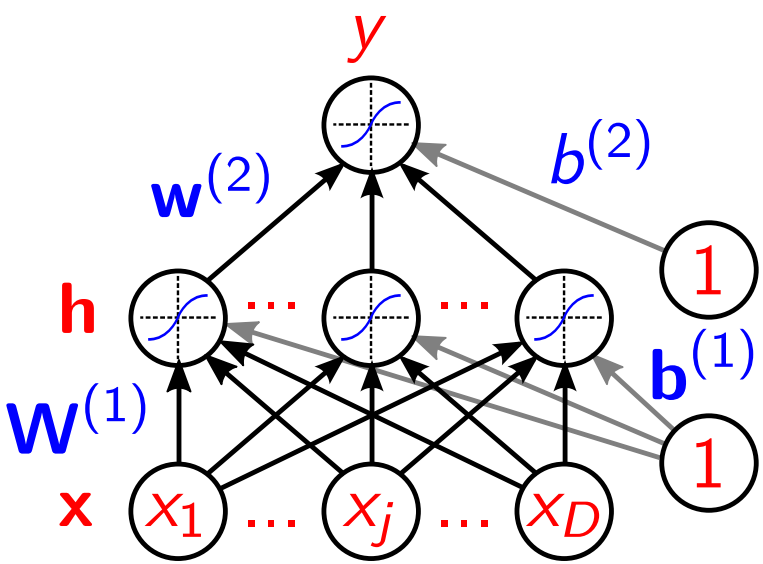
Multi-Layer Neural Network

- **Key idea:** add intermediate layers of artificial neurons before the final output layer
- Each of these hidden units computes some representation of the input and propagates forward that representation
- This increases the expressive power of the network, yielding more complex, non-linear, functions/classifiers
- Similar role as latent variables in probabilistic models, but no need for a probability semantics
- Also called feed-forward neural network

Single Hidden Layer

To start simple, let's assume our task involves several inputs ($\boldsymbol{x} \in \mathbb{R}^D$) but a single output (e.g. $y \in \mathbb{R}$ or $y \in \{0, 1\}$)

Trick: add an intermediate layer of K hidden units ($\boldsymbol{h} \in \mathbb{R}^K$)



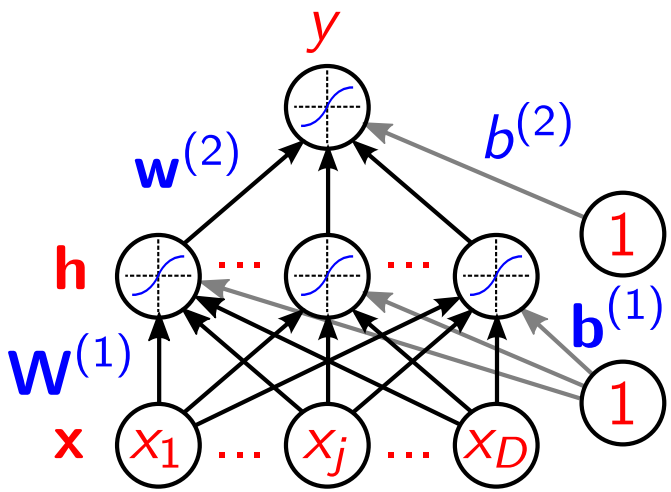
Single Hidden Layer

Assume D inputs ($\boldsymbol{x} \in \mathbb{R}^D$) and K hidden units ($\boldsymbol{h} \in \mathbb{R}^K$)

- **Hidden layer pre-activation:**

$$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(1)} \in \mathbb{R}^{K \times D}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^K$.



Single Hidden Layer

Assume D inputs ($\boldsymbol{x} \in \mathbb{R}^D$) and K hidden units ($\boldsymbol{h} \in \mathbb{R}^K$)

- **Hidden layer pre-activation:**

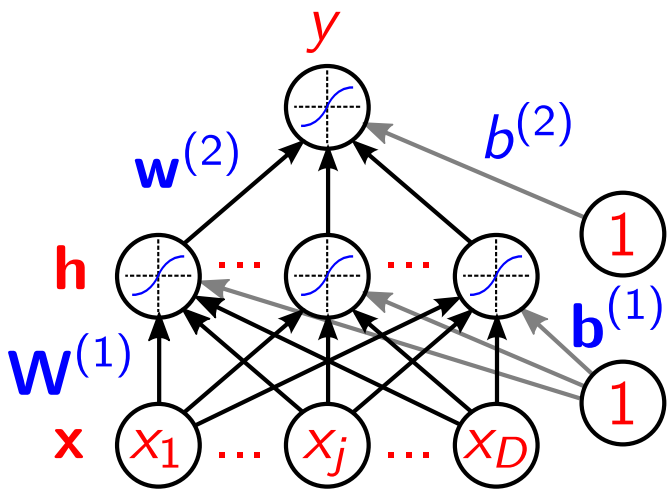
$$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(1)} \in \mathbb{R}^{K \times D}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^K$.

- **Hidden layer activation:**

$$\boldsymbol{h}(\boldsymbol{z}) = \boldsymbol{g}(\boldsymbol{z}(\boldsymbol{x})),$$

where $\boldsymbol{g} : \mathbb{R}^K \to \mathbb{R}^K$ is applied component-wise (component-by-component).



Single Hidden Layer

Assume D inputs ($\boldsymbol{x} \in \mathbb{R}^D$) and K hidden units ($\boldsymbol{h} \in \mathbb{R}^K$)

- **Hidden layer pre-activation:**

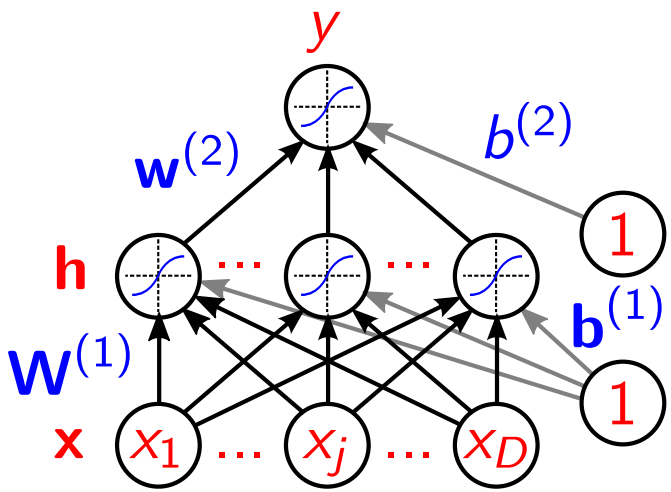
$$\boldsymbol{z}(\boldsymbol{x}) = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(1)} \in \mathbb{R}^{K \times D}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^K$.

- **Hidden layer activation:**

$$\boldsymbol{h}(\boldsymbol{z}) = \boldsymbol{g}(\boldsymbol{z}(\boldsymbol{x})),$$

where $\boldsymbol{g} : \mathbb{R}^K \to \mathbb{R}^K$ is applied component-wise (component-by-component).



- **Output layer activation:**

$$f(\boldsymbol{x}) = o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{h} + b^{(2)}),$$

where $\boldsymbol{w}^{(2)} \in \mathbb{R}^K$ and $o : \mathbb{R} \to \mathbb{R}$ if the output activation function.

Single Hidden Layer

Overall,

$$
\begin{aligned}
f(\boldsymbol{x}) &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{h} + b^{(2)}) \\
&= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)})
\end{aligned}
$$

Single Hidden Layer

Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{h} + b^{(2)}) \\ &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

Examples:

- $o(u) = u$ for regression ($y \in \mathbb{R}$)

Single Hidden Layer

Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{h} + b^{(2)}) \\ &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

Examples:

- $o(u) = u$ for regression ($y \in \mathbb{R}$)
- $o(u) = \sigma(u)$ for binary classification ($y \in \{\pm 1\}$, $f(\boldsymbol{x}) = P(y = 1 \mid \boldsymbol{x})$)

Single Hidden Layer

Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{h} + b^{(2)}) \\ &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

Examples:

- $o(u) = u$ for regression ($y \in \mathbb{R}$)
- $o(u) = \sigma(u)$ for binary classification ($y \in \{\pm 1\}$, $f(\boldsymbol{x}) = P(y = 1 \mid \boldsymbol{x})$)

No longer a linear classifier – non-linear dependency on $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$

Single Hidden Layer

Overall,

$$\begin{aligned} f(\boldsymbol{x}) &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{h} + b^{(2)}) \\ &= o(\boldsymbol{w}^{(2)} \cdot \boldsymbol{g}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) + b^{(2)}) \end{aligned}$$

Examples:

- $o(u) = u$ for regression ($y \in \mathbb{R}$)
- $o(u) = \sigma(u)$ for binary classification ($y \in \{\pm 1\}$, $f(\boldsymbol{x}) = P(y = 1 \mid \boldsymbol{x})$)

No longer a linear classifier – non-linear dependency on $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$

$\boldsymbol{h}$ is a vector of internal representations (not manually engineered features)

Multiple Classes

Can we use a similar strategy for the multi-class case?

For multi-class classification, we need multiple output units (one per class)

Each output estimates the conditional probability $P(y = c \mid \boldsymbol{x})$

Predicted class is (usually) the one with highest estimated probability

We have already seen an activation function suitable for this: softmax

Softmax Activation

Let $\Delta_{C-1} \subseteq \mathbb{R}^C$ be the probability simplex:

$$\Delta_{C-1} = \{(p1, ..., p_C) : \ p_i \geq 0, i = 1, ..., C, \ \sum_{i=1}^{C} p_i = 1\}$$

Softmax Activation

Let $\Delta_{C-1} \subseteq \mathbb{R}^C$ be the probability simplex:

$$\Delta_{C-1} = \{(p1, ..., p_C) : \ p_i \geq 0, i = 1, ..., C, \ \sum_{i=1}^{C} p_i = 1\}$$

Typical activation function for multi-class: **softmax** : $\mathbb{R}^C \to \Delta_{C-1}$:

$$\boxed{\boldsymbol{o}(\boldsymbol{z}) = \textbf{softmax}(\boldsymbol{z}) = \left[\frac{\exp(z_1)}{\sum_c \exp(z_c)}, \ldots, \frac{\exp(z_C)}{\sum_c \exp(z_c)}\right]}$$

Softmax Activation

Let $\Delta_{C-1} \subseteq \mathbb{R}^C$ be the probability simplex:

$$\Delta_{C-1} = \{(p1, ..., p_C) : \ p_i \geq 0, i = 1, ..., C, \ \sum_{i=1}^{C} p_i = 1\}$$

Typical activation function for multi-class: **softmax** : $\mathbb{R}^C \to \Delta_{C-1}$:

$$\boxed{\boldsymbol{o}(\boldsymbol{z}) = \textbf{softmax}(\boldsymbol{z}) = \left[\frac{\exp(z_1)}{\sum_c \exp(z_c)}, \ldots, \frac{\exp(z_C)}{\sum_c \exp(z_c)}\right]}$$

- We saw this previously, when talking about logistic regression!

Softmax Activation

Let $\Delta_{C-1} \subseteq \mathbb{R}^C$ be the probability simplex:

$$\Delta_{C-1} = \{(p1, ..., p_C) : \ p_i \geq 0, i = 1, ..., C, \ \sum_{i=1}^{C} p_i = 1\}$$

Typical activation function for multi-class: **softmax** : $\mathbb{R}^C \to \Delta_{C-1}$:

$$\boxed{\boldsymbol{o}(\boldsymbol{z}) = \textbf{softmax}(\boldsymbol{z}) = \left[\frac{\exp(z_1)}{\sum_c \exp(z_c)}, \ldots, \frac{\exp(z_C)}{\sum_c \exp(z_c)}\right]}$$

- We saw this previously, when talking about logistic regression!
- Strictly positive, sums to 1

Softmax Activation

Let $\Delta_{C-1} \subseteq \mathbb{R}^C$ be the probability simplex:

$$\Delta_{C-1} = \{(p1, ..., p_C) : \ p_i \geq 0, i = 1, ..., C, \ \sum_{i=1}^{C} p_i = 1\}$$

Typical activation function for multi-class: **softmax** : $\mathbb{R}^C \to \Delta_{C-1}$:

$$\boxed{\boldsymbol{o}(\boldsymbol{z}) = \textbf{softmax}(\boldsymbol{z}) = \left[\frac{\exp(z_1)}{\sum_c \exp(z_c)}, \ldots, \frac{\exp(z_C)}{\sum_c \exp(z_c)}\right]}$$

- We saw this previously, when talking about logistic regression!
- Strictly positive, sums to 1
- Resulting distribution has full support: $\textbf{softmax}(\boldsymbol{z}) > \mathbf{0}, \forall \boldsymbol{z}$

Multi-Layer Neural Networks: General Case

In general we can:

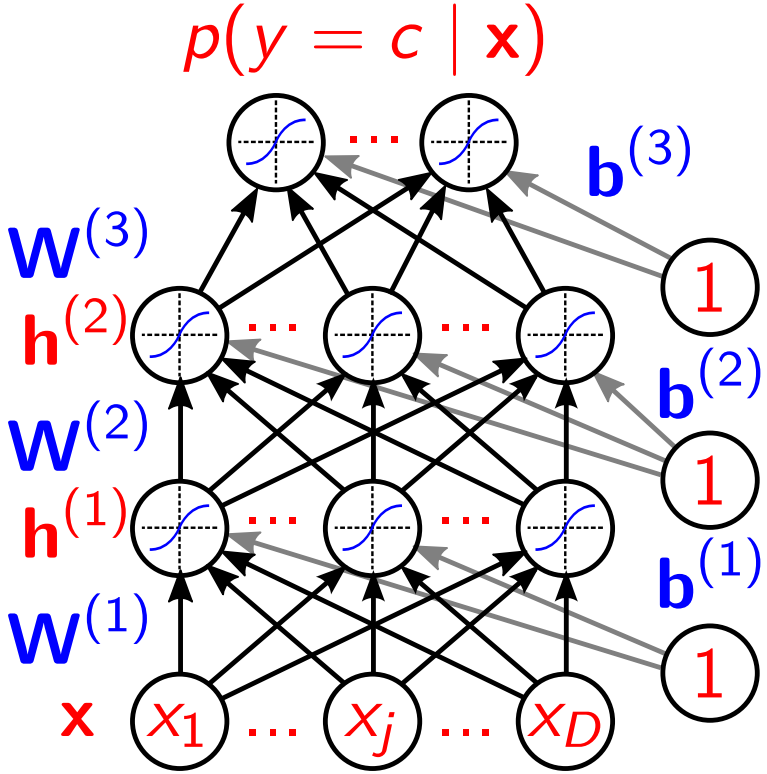
- Have multiple output units (needed for multi-class classification)
- Stack more layers after each other

Multiple ($L \geq 1$) Hidden Layers

- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$ and $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$.



Multiple ($L \geq 1$) Hidden Layers

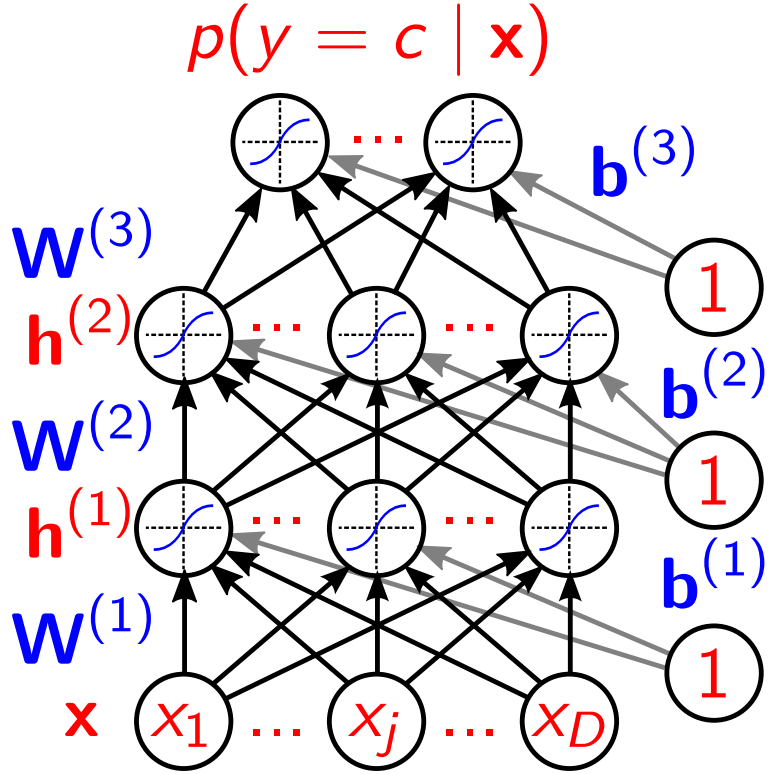
- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$ and $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$.

- **Hidden layer activation:**

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})).$$



Multiple ($L \geq 1$) Hidden Layers

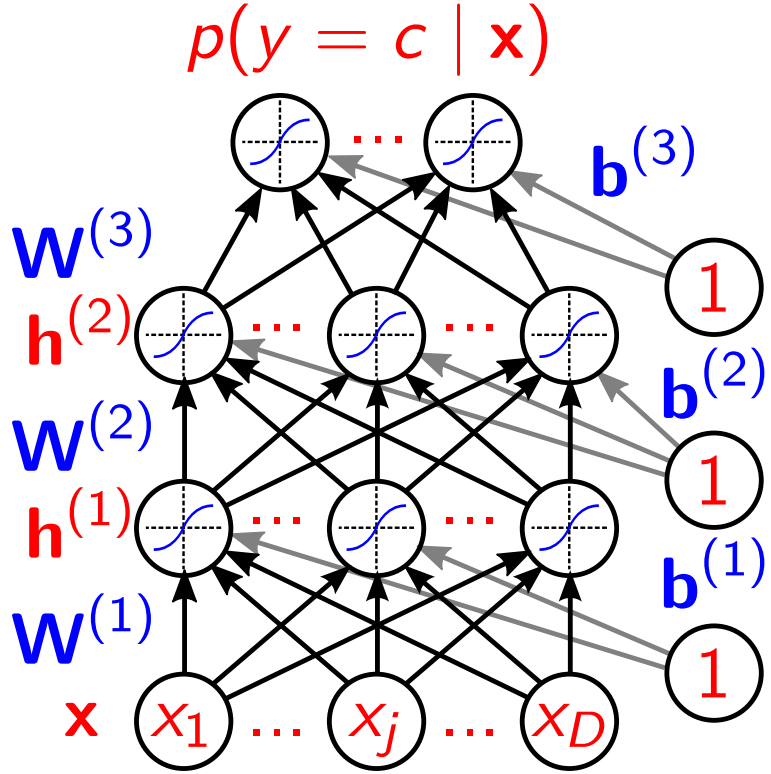
- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(1)},$$

with $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$ and $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$.

- **Hidden layer activation:**

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})).$$



- **Output layer activation:**

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{o}(\boldsymbol{z}^{(L+1)}(\boldsymbol{x})) = \boldsymbol{o}(\boldsymbol{W}^{(L+1)}\boldsymbol{h}^{(L)} + \boldsymbol{b}^{(L+1)}).$$

Outline

Universal Approximation Theorem

Theorem (Hornik et al. (1989))

An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

Universal Approximation Theorem

Theorem (Hornik et al. (1989))

An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

- First proved for the sigmoid case by Cybenko (1989), then to **tanh** and many other activation functions by Hornik et al. (1989)

Universal Approximation Theorem

Theorem (Hornik et al. (1989))

An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

- First proved for the sigmoid case by Cybenko (1989), then to **tanh** and many other activation functions by Hornik et al. (1989)
- **Caveat:** may need exponentially many hidden units

Deeper Networks

- Deeper networks (more layers) can provide more compact approximations

Theorem (Montufar et al. (2014))

The number of linear regions carved out by a deep neural network with D inputs, depth L, and K hidden units per layer with ReLU activations is

$$O\left(\binom{K}{D}^{D(L-1)} K^D\right)$$

Deeper Networks

- Deeper networks (more layers) can provide more compact approximations

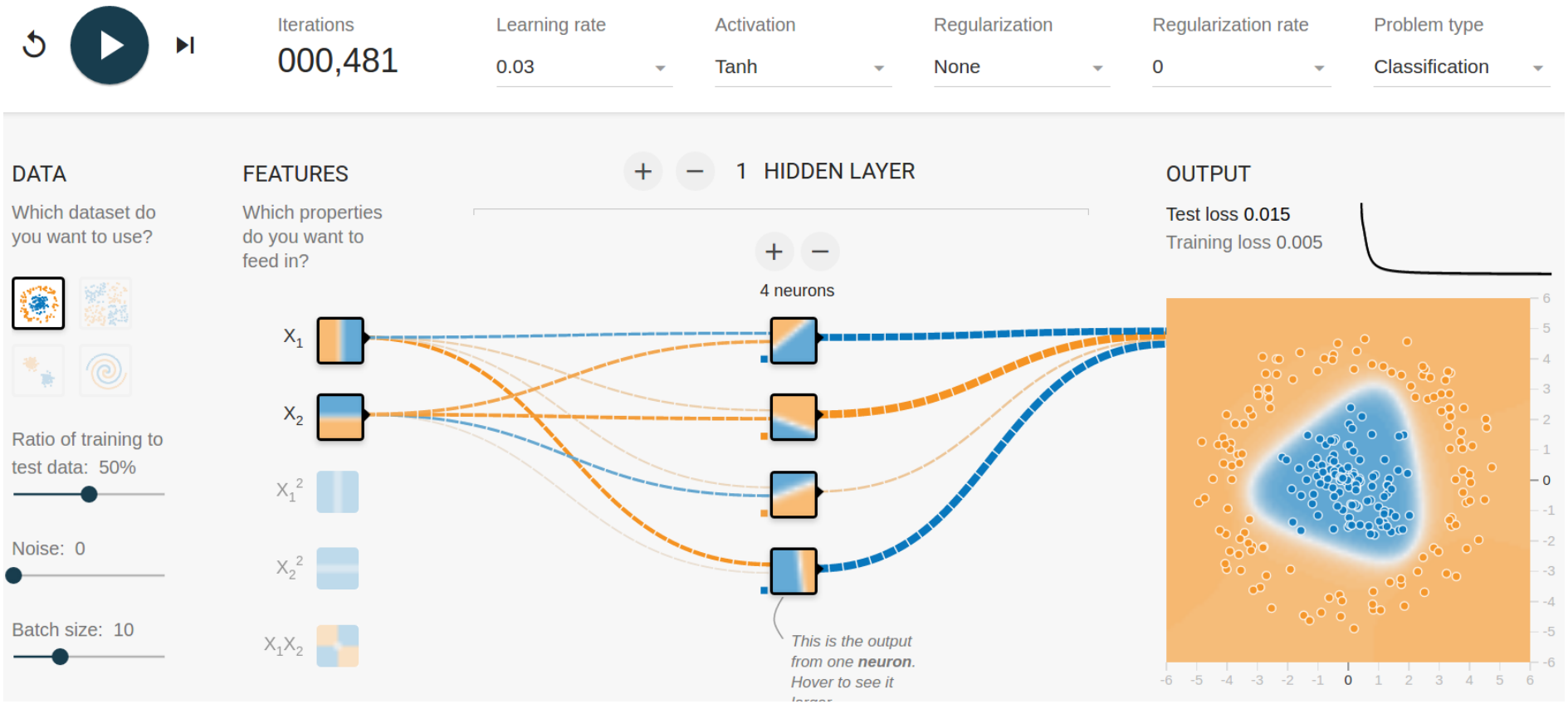
Theorem (Montufar et al. (2014))

The number of linear regions carved out by a deep neural network with D inputs, depth L, and K hidden units per layer with ReLU activations is

$$O\left(\binom{K}{D}^{D(L-1)} K^D\right)$$

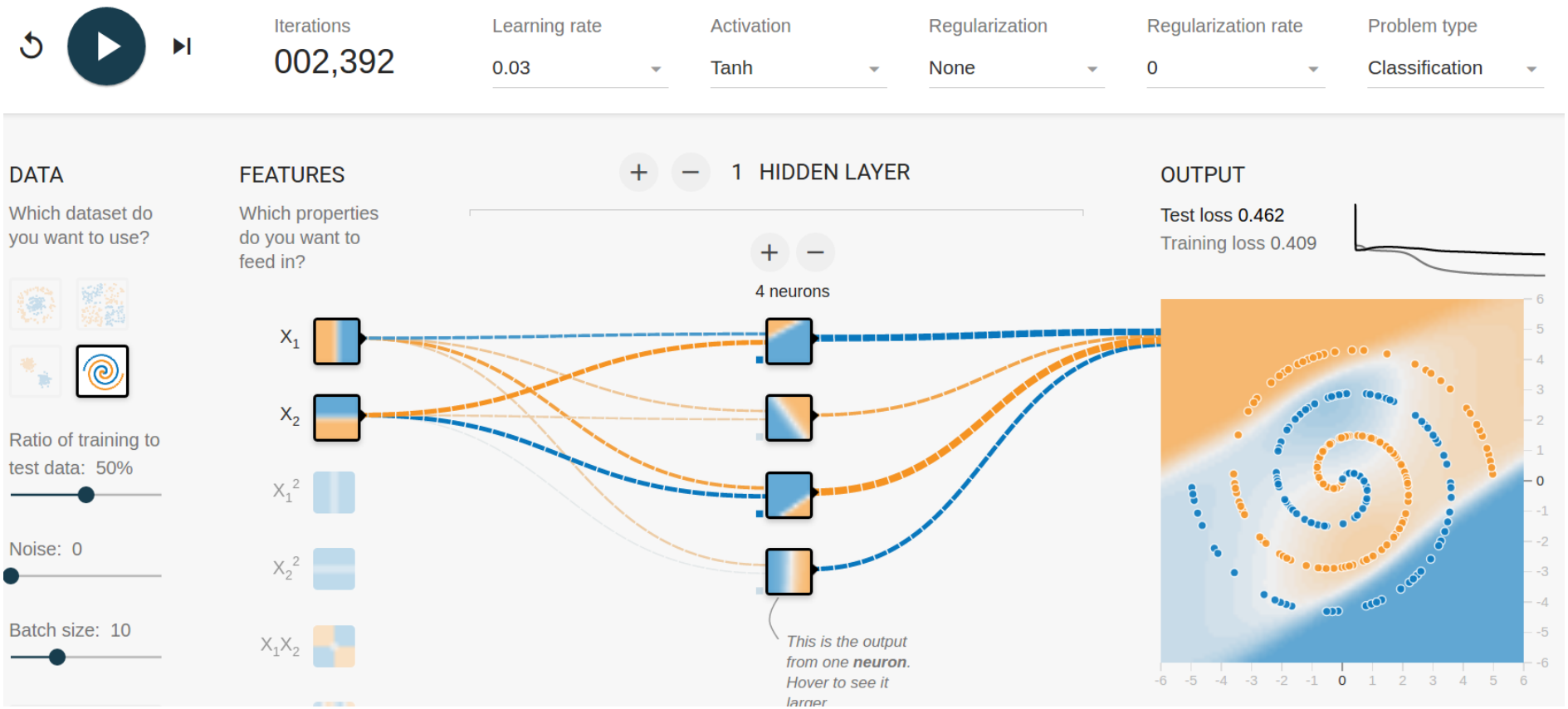
Therefore, for fixed K, deeper networks are exponentially more expressive

"Simple" Target Function, One Hidden Layer



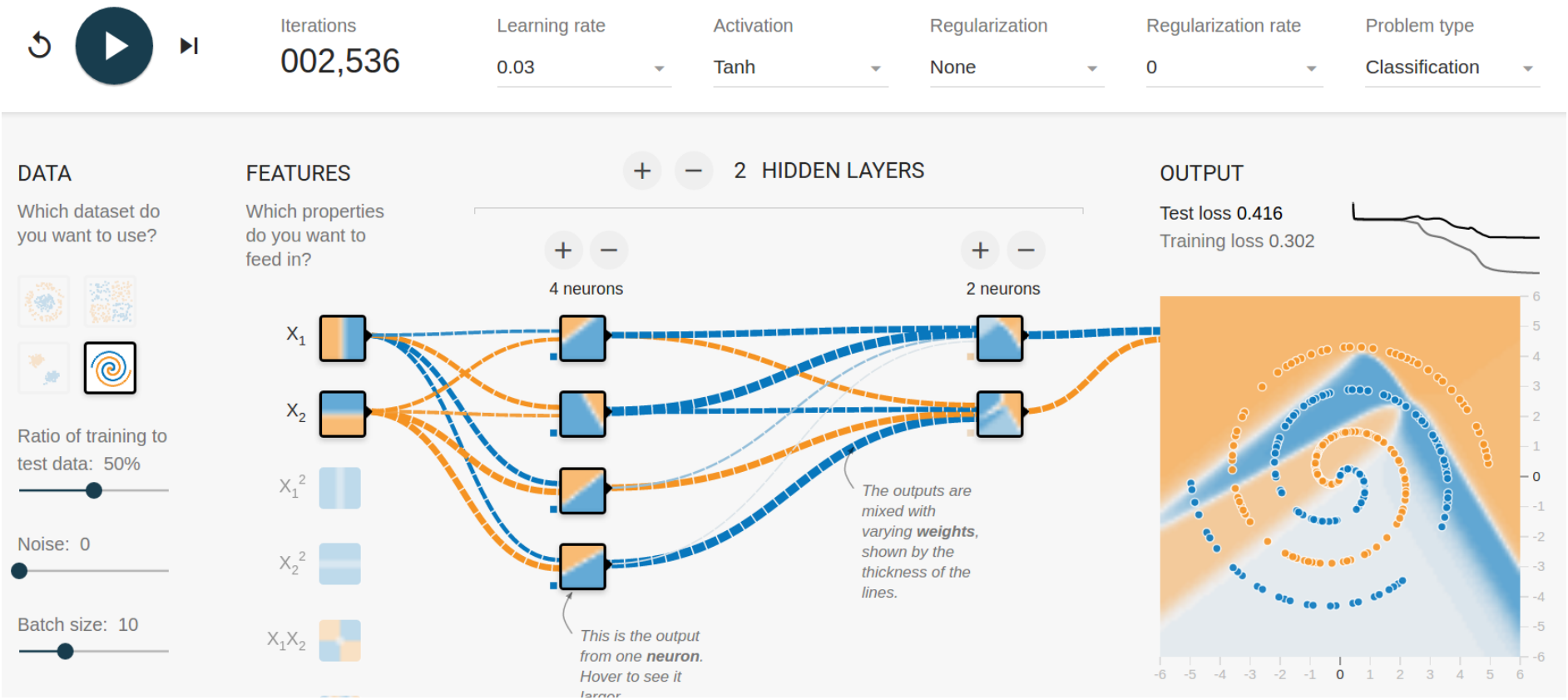
(http://playground.tensorflow.org)

Complex Target Function, One Hidden Layer



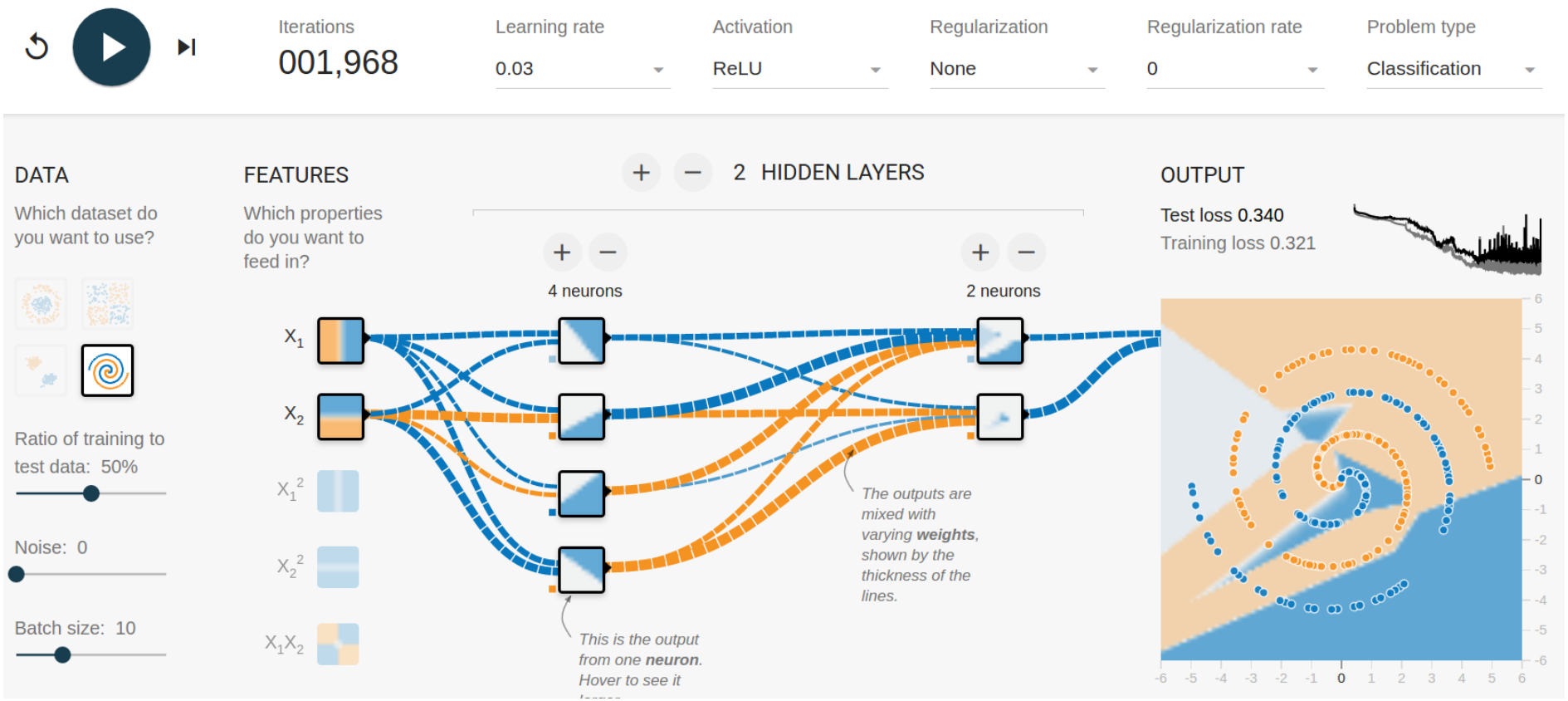
(http://playground.tensorflow.org)

Complex Target Function, Two Hidden Layers



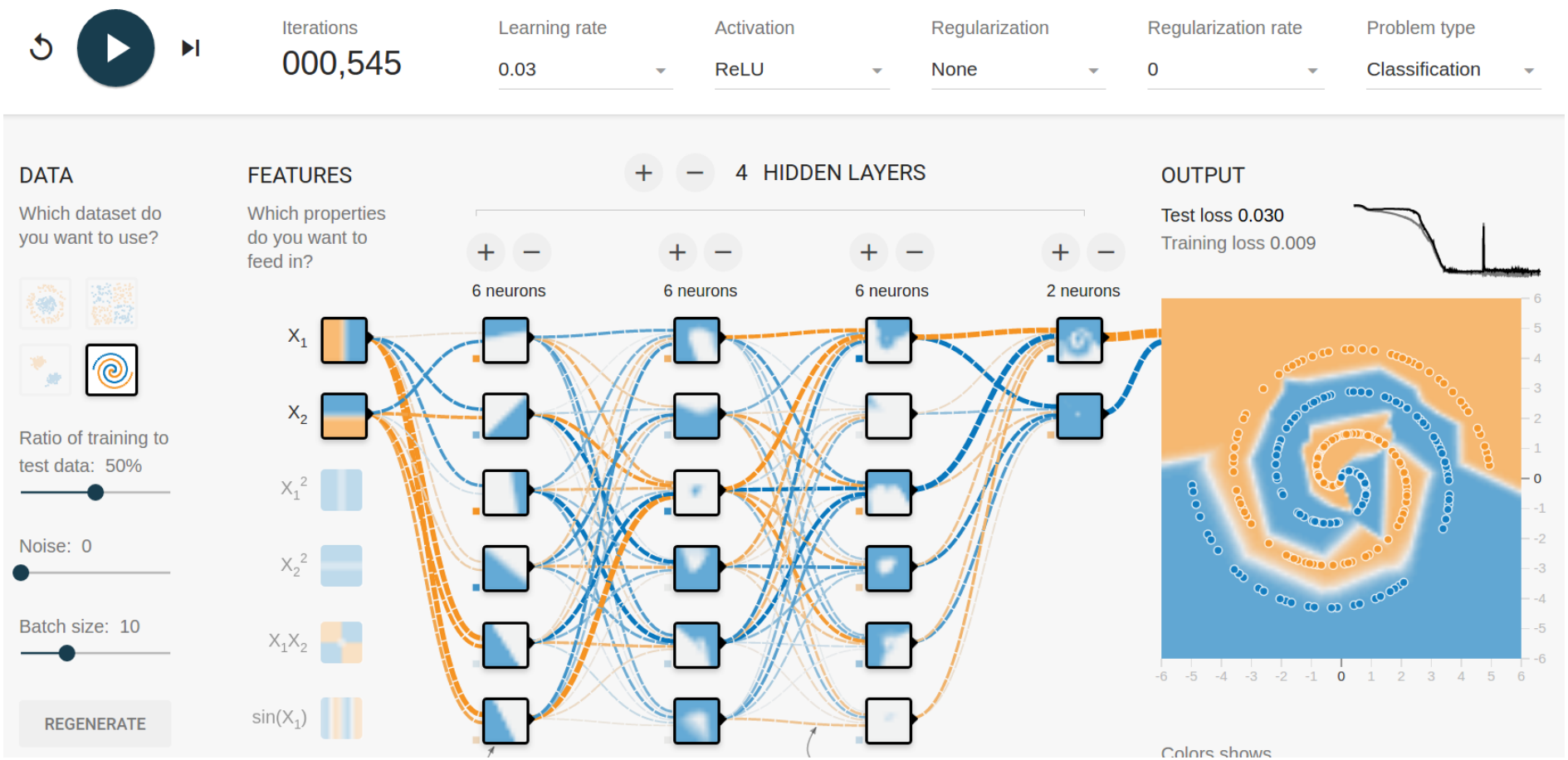
(http://playground.tensorflow.org)

Complex Target Function, Two Hidden Layers, ReLU



(http://playground.tensorflow.org)

Complex Target Function, Four Hidden Layers, ReLU



(http://playground.tensorflow.org)

Capacity of Neural Networks

Neural networks are excellent function approximators!

The universal approximation theorem is an important result, but:

- We need a learning algorithm that finds the necessary parameter values
- ... and if we want to generalize, we need to control overfitting

Outline

Training Neural Networks

Neural networks are expressive: in theory, they approximate any function

Training Neural Networks

Neural networks are expressive: in theory, they approximate any function

But to do so, their parameters

$$\boldsymbol{\theta} := \{(\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}), ..., (\boldsymbol{W}^{(L+1)}, \boldsymbol{b}^{(L+1)})\}$$

need to be set accordingly

Training Neural Networks

Neural networks are expressive: in theory, they approximate any function

But to do so, their parameters

$$\boldsymbol{\theta} := \{(\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}), ..., (\boldsymbol{W}^{(L+1)}, \boldsymbol{b}^{(L+1)})\}$$

need to be set accordingly

Key idea: learn these parameters from data

In other words: learn a function by sampling a few points and their values

Training Neural Networks

Neural networks are expressive: in theory, they approximate any function

But to do so, their parameters

$$\boldsymbol{\theta} := \{(\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}), ..., (\boldsymbol{W}^{(L+1)}, \boldsymbol{b}^{(L+1)})\}$$

need to be set accordingly

Key idea: learn these parameters from data

In other words: learn a function by sampling a few points and their values

(We've seen this when we talked about linear models a few days ago...)

Outline

Empirical Risk Minimization

Goal: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ that minimize the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda \Omega(\boldsymbol{\theta}) + \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

Empirical Risk Minimization

Goal: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ that minimize the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda \Omega(\boldsymbol{\theta}) + \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

- $L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$ is a loss function

Empirical Risk Minimization

Goal: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ that minimize the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda \Omega(\boldsymbol{\theta}) + \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

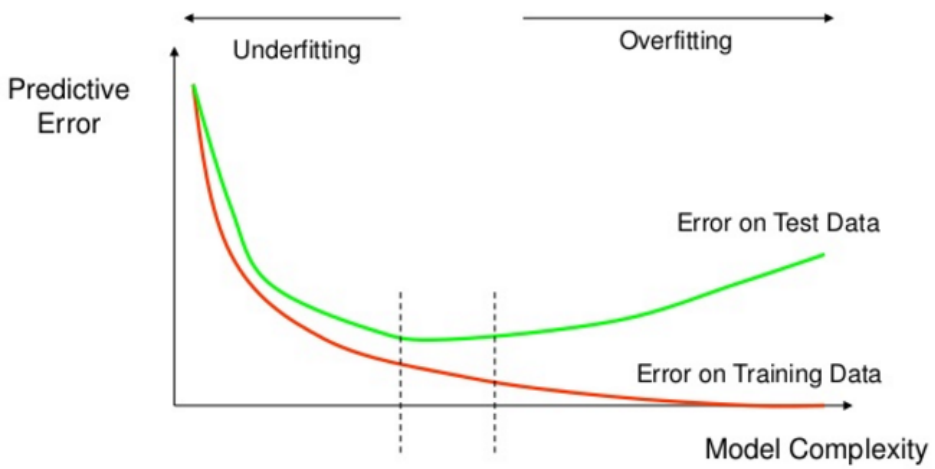
- $L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$ is a loss function
- $\Omega(\boldsymbol{\theta})$ is a regularizer

Empirical Risk Minimization

Goal: choose parameters $\boldsymbol{\theta} := \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ that minimize the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda \Omega(\boldsymbol{\theta}) + \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

- $L(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$ is a loss function
- $\Omega(\boldsymbol{\theta})$ is a regularizer
- λ is a regularization constant (an hyperparameter that needs to be tuned)



Recap: Gradient Descent

We can write the objective as:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &:= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) \\ &:= \frac{1}{N}\sum_{i=1}^{N} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})}\end{aligned}$$

Recap: Gradient Descent

We can write the objective as:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &:= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) \\ &:= \frac{1}{N}\sum_{i=1}^{N} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})} = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i(\boldsymbol{\theta})\end{aligned}$$

Recap: Gradient Descent

We can write the objective as:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &:= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) \\ &:= \frac{1}{N}\sum_{i=1}^{N} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\boldsymbol{\theta})\end{aligned}$$

The gradient is:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{N}\sum_{i=1}^{N}\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta})$$

Recap: Gradient Descent

We can write the objective as:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &:= \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{i=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i) \\ &:= \frac{1}{N}\sum_{i=1}^{N} \underbrace{\lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)}_{\mathcal{L}_i(\boldsymbol{\theta})} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\boldsymbol{\theta})\end{aligned}$$

The gradient is:

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) := \frac{1}{N}\sum_{i=1}^{N}\nabla_{\boldsymbol{\theta}}\mathcal{L}_i(\boldsymbol{\theta})$$

Requires a full pass over the data to update the weights—too slow!

Recap: Stochastic Gradient Descent

Sample a single training example uniformly at random: $j \in \{1, ..., N\}$

Recap: Stochastic Gradient Descent

Sample a single training example uniformly at random: $j \in \{1, ..., N\}$

This way we get a noisy but unbiased estimate of the gradient:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \;\; := \;\; \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) \;\; \approx \;\; \nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta})$$

Recap: Stochastic Gradient Descent

Sample a single training example uniformly at random: $j \in \{1, ..., N\}$

This way we get a noisy but unbiased estimate of the gradient:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &:= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta}) \\ &= \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_j; \boldsymbol{\theta}), y_j). \end{aligned}$$

Recap: Stochastic Gradient Descent

Sample a single training example uniformly at random: $j \in \{1, ..., N\}$

This way we get a noisy but unbiased estimate of the gradient:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &:= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta}) \\ &= \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_j; \boldsymbol{\theta}), y_j). \end{aligned}$$

The weights $\boldsymbol{\theta} = \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ are then updated as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta})$$

Recap: Stochastic Gradient Descent

Sample a single training example uniformly at random: $j \in \{1, ..., N\}$

This way we get a noisy but unbiased estimate of the gradient:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &:= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta}) \\ &= \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_j; \boldsymbol{\theta}), y_j). \end{aligned}$$

The weights $\boldsymbol{\theta} = \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ are then updated as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_j(\boldsymbol{\theta})$$

In practice, use mini-batch instead of a single sample

Stochastic Gradient Descent with Mini-Batches

With a mini-batch $\{j_1, \ldots, j_B\}$ ($B \ll N$) we get a less noisy, still unbiased estimate of the gradient:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \quad := \quad \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) \quad \approx \quad \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{j_i}(\boldsymbol{\theta})$$

Stochastic Gradient Descent with Mini-Batches

With a mini-batch $\{j_1, \ldots, j_B\}$ ($B \ll N$) we get a less noisy, still unbiased estimate of the gradient:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &:= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) \approx \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{j_i}(\boldsymbol{\theta}) \\ &= \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) + \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_{j_i}; \boldsymbol{\theta}), y_{j_i}). \end{aligned}$$

Stochastic Gradient Descent with Mini-Batches

With a mini-batch $\{j_1, \ldots, j_B\}$ ($B \ll N$) we get a less noisy, still unbiased estimate of the gradient:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &:= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) \approx \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{j_i}(\boldsymbol{\theta}) \\ &= \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) + \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_{j_i}; \boldsymbol{\theta}), y_{j_i}). \end{aligned}$$

The weights $\boldsymbol{\theta} = \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ are then updated as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{j_i}(\boldsymbol{\theta})$$

The Key Ingredients of SGD

In sum, SGD needs the following ingredients:

- The loss function $L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- A procedure for computing the gradients $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- The regularizer $\Omega(\boldsymbol{\theta})$ and its gradient.

The Key Ingredients of SGD

In sum, SGD needs the following ingredients:

- The loss function $L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- A procedure for computing the gradients $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- The regularizer $\Omega(\boldsymbol{\theta})$ and its gradient.

Let's see them one at the time...

Loss Function

Should match as much as possible the metric we want to optimize at test time

Should be well-behaved (continuous, maybe smooth) to be amenable to optimization (this rules out the 0/1 loss)

Some examples:

- Squared loss for regression
- Negative log-likelihood (cross-entropy) for multi-class classification
- Sparsemax loss for multi-class and multi-label classification

Squared Loss

- The common choice for regression/reconstruction problems

Squared Loss

- The common choice for regression/reconstruction problems
- The neural network aims at estimating $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$

Squared Loss

- The common choice for regression/reconstruction problems
- The neural network aims at estimating $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$
- Minimize the mean squared error:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) \quad = \quad \frac{1}{2}\|\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) - \boldsymbol{y}\|^2$$

Squared Loss

- The common choice for regression/reconstruction problems
- The neural network aims at estimating $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$
- Minimize the mean squared error:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) \quad = \quad \frac{1}{2}\|\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) - \boldsymbol{y}\|^2$$

- Loss gradient:

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta},\boldsymbol{y}))}{\partial f_c(\boldsymbol{x};\boldsymbol{\theta})} = f_c(\boldsymbol{x};\boldsymbol{\theta}) - y_c$$

Negative Log-Likelihood (Cross-Entropy)

- The common choice for a softmax output layer

Negative Log-Likelihood (Cross-Entropy)

- The common choice for a softmax output layer
- The neural network estimates $f_c(\boldsymbol{x}; \boldsymbol{\theta}) \approx P(y = c \mid \boldsymbol{x})$

Negative Log-Likelihood (Cross-Entropy)

- The common choice for a softmax output layer
- The neural network estimates $f_c(\boldsymbol{x}; \boldsymbol{\theta}) \approx P(y = c \mid \boldsymbol{x})$
- We minimize the negative log-likelihood (also called cross-entropy):

$$\begin{aligned} L(\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}), y) &= -\sum_c 1_{(y=c)} \log f_c(\boldsymbol{x}; \boldsymbol{\theta}) \\ &= -\log f_y(\boldsymbol{x}; \boldsymbol{\theta}) \\ &= -\log \mathbf{softmax}_y(\boldsymbol{z}(\boldsymbol{x})) \end{aligned}$$

where $\boldsymbol{z}$ is the output pre-activation.

Negative Log-Likelihood (Cross-Entropy)

- The common choice for a softmax output layer
- The neural network estimates $f_c(\boldsymbol{x}; \boldsymbol{\theta}) \approx P(y = c \mid \boldsymbol{x})$
- We minimize the negative log-likelihood (also called cross-entropy):

$$\begin{aligned} L(\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}), y) &= -\sum_c 1_{(y=c)} \log f_c(\boldsymbol{x}; \boldsymbol{\theta}) \\ &= -\log f_y(\boldsymbol{x}; \boldsymbol{\theta}) \\ &= -\log \textbf{softmax}_y(\boldsymbol{z}(\boldsymbol{x})) \end{aligned}$$

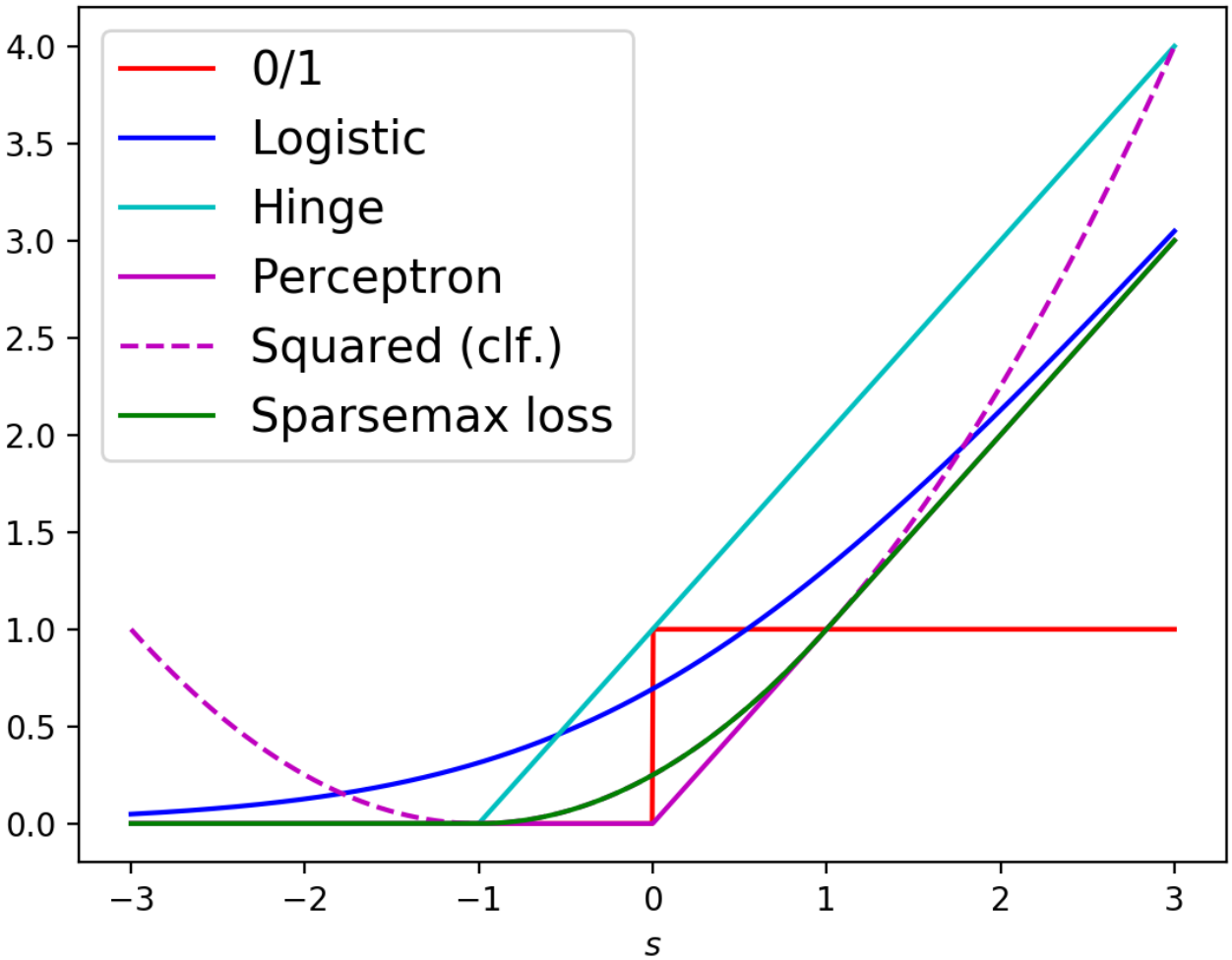
  where $\boldsymbol{z}$ is the output pre-activation.

- Loss gradient at output pre-activation:

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}, y))}{\partial z_c} = \textbf{softmax}_c(\boldsymbol{z}(\boldsymbol{x})) - 1_{(y=c)}$$

Classification Losses

- Let the correct label be $y = 1$ and define $s = z_2 - z_1$:



The Key Ingredients of SGD

In sum, SGD needs the following ingredients:

- The loss function $L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$;
- A procedure for computing the gradients $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$: next
- The regularizer $\Omega(\boldsymbol{\theta})$ and its gradient.

Outline

Gradient Computation

- Recall that we need to compute

$$\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

for $\boldsymbol{\theta} = \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ (the weights and biases at all layers)

Gradient Computation

- Recall that we need to compute

$$\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

for $\boldsymbol{\theta} = \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ (the weights and biases at all layers)

- This will be done with the gradient backpropagation algorithm

Gradient Computation

- Recall that we need to compute

$$\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

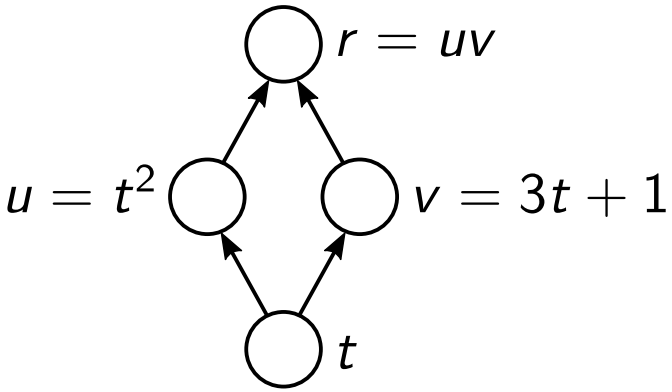
for $\boldsymbol{\theta} = \{(\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ (the weights and biases at all layers)

- This will be done with the gradient backpropagation algorithm

- **Key idea:** use the chain rule for derivatives!

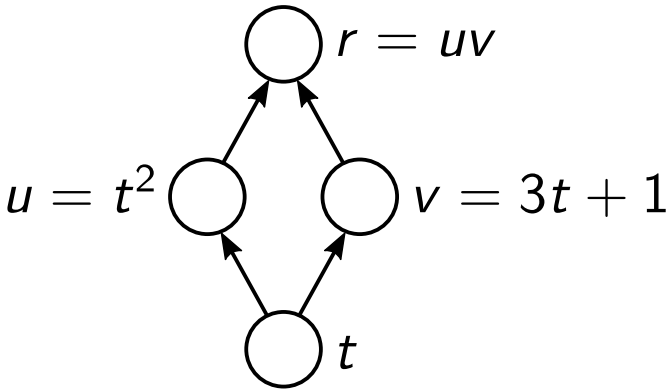
$$h(x) = f\big(g(x)\big) \quad \Rightarrow \quad h'(x) = f'(g(x))\, g'(x)$$

Recap: Chain Rule



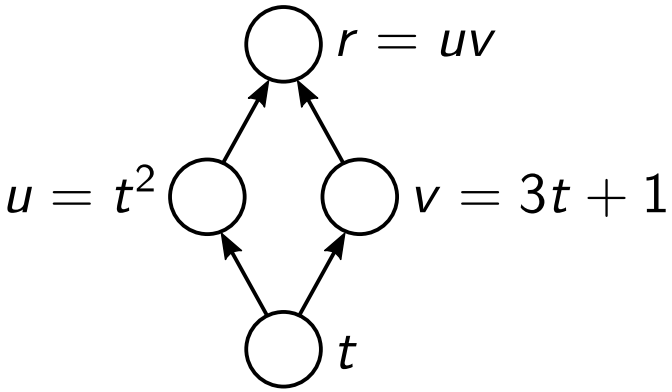
$$\frac{\partial r(t)}{\partial t} = ?$$

Recap: Chain Rule



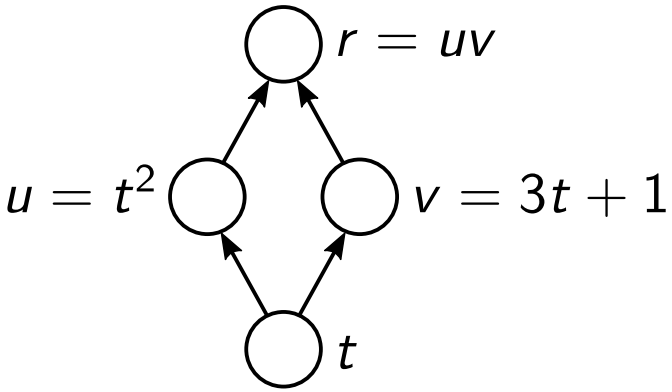
$$\frac{\partial r(t)}{\partial t} = \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t}$$

Recap: Chain Rule



$$\begin{aligned}\frac{\partial r(t)}{\partial t} &= \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t} \\ &= 2tv + 3u \\ &= 2t(3t+1) + 3t^2 = 9t^2 + 2t.\end{aligned}$$

Recap: Chain Rule



$$\begin{aligned}\frac{\partial r(t)}{\partial t} &= \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t} \\ &= 2tv + 3u \\ &= 2t(3t+1) + 3t^2 = 9t^2 + 2t.\end{aligned}$$

- If a function $r(t)$ can be written as a function of intermediate results $q_i(t)$, then we have:

$$\frac{\partial r(t)}{\partial t} = \sum_i \frac{\partial r(t)}{\partial q_i(t)}\frac{\partial q_i(t)}{\partial t}$$

- We can invoke it by setting t to a output unit in a layer; $q_i(t)$ to the pre-activation in the layer above; and $r(t)$ to the loss function.

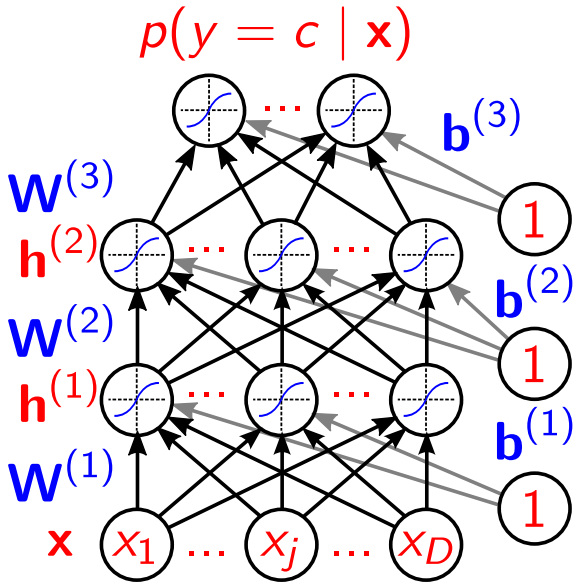
Hidden Layer Gradient

Main message: gradient backpropagation is just the chain rule of derivatives!

Hidden Layer Gradient

(Recap: $\boldsymbol{z}^{(\ell+1)} = \boldsymbol{W}^{(\ell+1)}\boldsymbol{h}^{(\ell)} + \boldsymbol{b}^{(\ell+1)}$)

$$\begin{aligned}\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial h_j^{(\ell)}} &= \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \frac{\partial z_i^{(\ell+1)}}{\partial h_j^{(\ell)}} \\ &= \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \boldsymbol{W}_{i,j}^{(\ell+1)}\end{aligned}$$

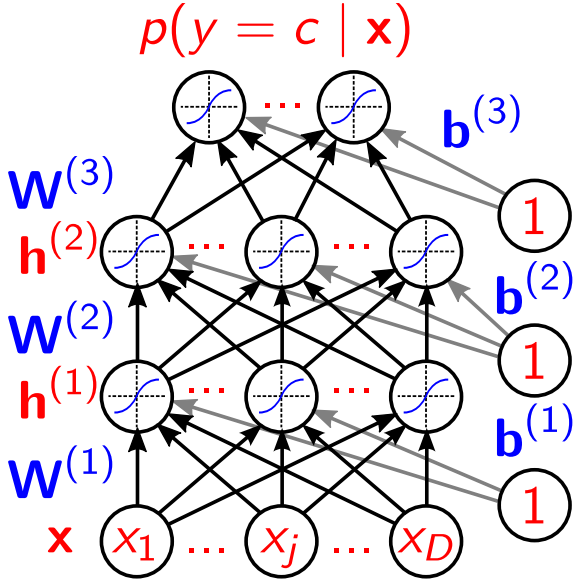


Hence $\nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \boldsymbol{W}^{(\ell+1)^\top} \nabla_{\boldsymbol{z}^{(\ell+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)$.

Hidden Layer Gradient (Before Activation)

(Recap: $h_j^{(\ell)} = g(z_j^{(\ell)})$, where $g : \mathbb{R} \to \mathbb{R}$ is the activation function)

$$\begin{aligned}\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z_j^{(\ell)}} &= \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial z_j^{(\ell)}} \\ &= \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} g'(z_j^{(\ell)})\end{aligned}$$



Hence $\nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$.

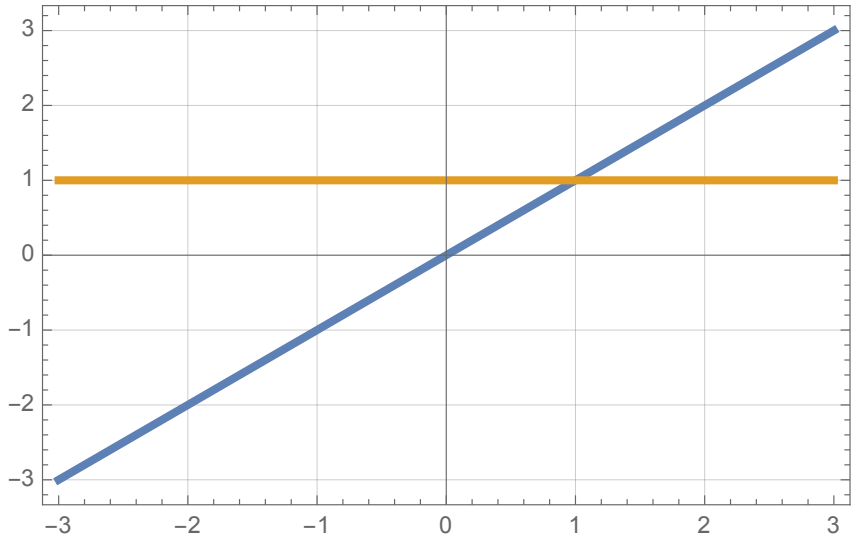
How to compute the derivative of the activation function $\boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$?

Linear Activation

$$\boxed{g(z) = z}$$

Derivative:

$$\boxed{g'(z) = 1}$$

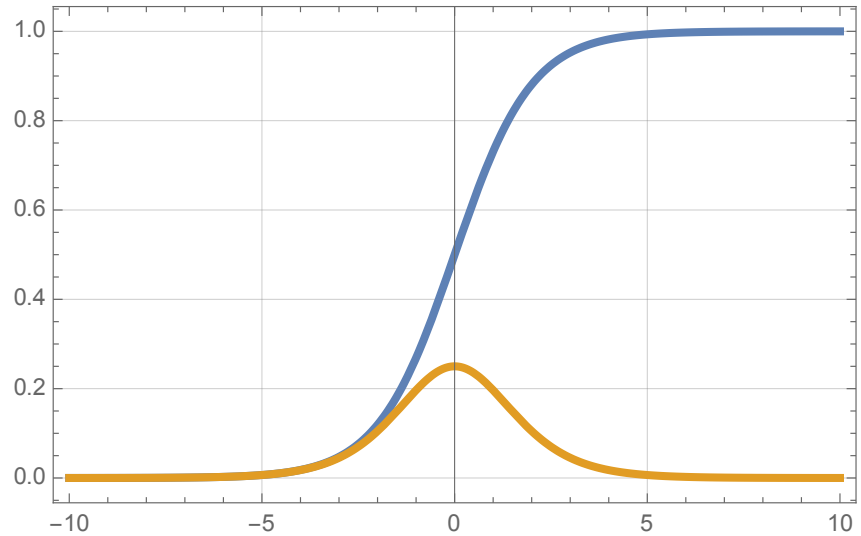


Sigmoid Activation

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

Derivative:

$$\boxed{g'(z) = g(z)(1 - g(z))}$$

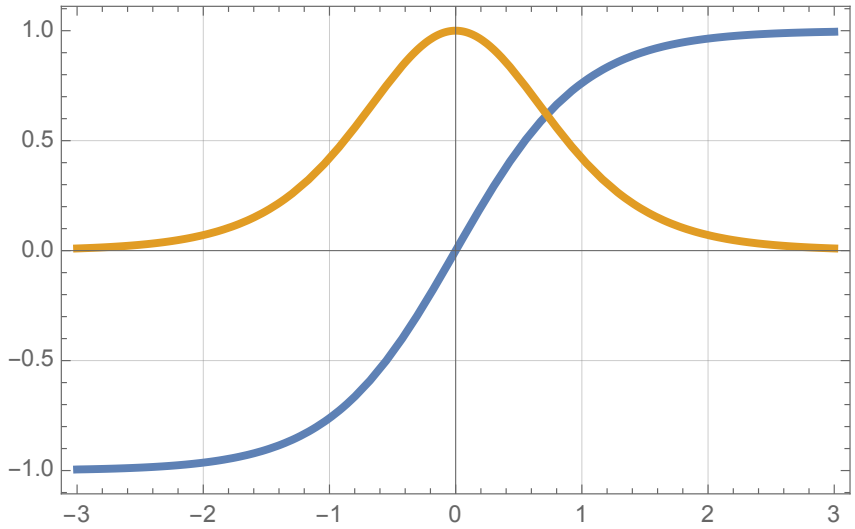


Hyperbolic Tangent Activation

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

Derivative:

$$g'(z) = 1 - g(z)^2 = \text{sech}^2(x)$$

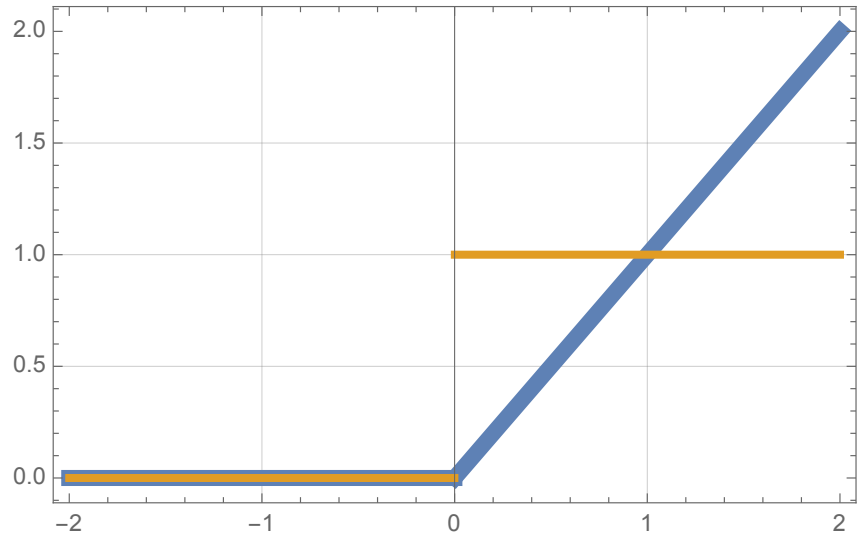


Rectified Linear Unit Activation (Glorot et al., 2011)

$$\boxed{g(z) = \textbf{relu}(z) = \max\{0, z\}}$$

Derivative (except for $z = 0$):

$$\boxed{g'(z) = 1_{z>0}}$$

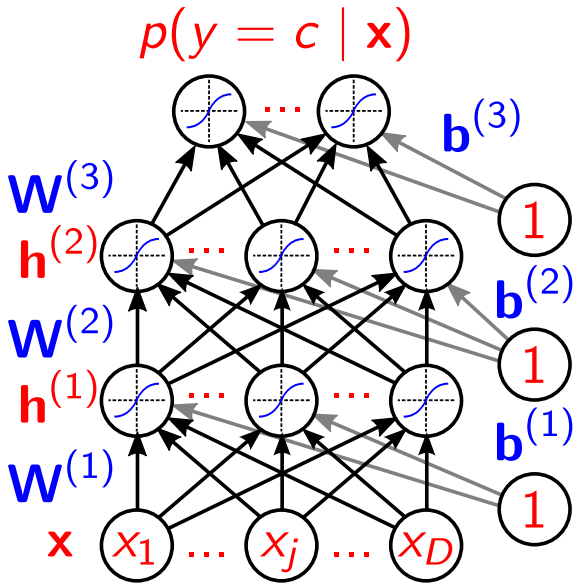


Parameter Gradient

(Recap: $\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$)

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial \boldsymbol{W}_{i,j}^{(\ell)}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial \boldsymbol{W}_{i,j}^{(\ell)}}$$

$$= \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell)}} h_j^{(\ell-1)}$$



Hence $\nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)\boldsymbol{h}^{(\ell-1)\top}$

Similarly, $\nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)$

Backpropagation

Compute output gradient (before activation):

$$\nabla_{\boldsymbol{z}^{(L+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -(\mathbf{1}_y - \boldsymbol{f}(\boldsymbol{x}))$$

for ℓ from $L+1$ to 1 **do**

Compute gradients of hidden layer parameters:

$$\begin{aligned} \nabla_{\boldsymbol{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)\ \boldsymbol{h}^{(\ell-1)\top} \\ \nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \end{aligned}$$

Compute gradient of hidden layer below:

$$\nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \boldsymbol{W}^{(\ell)\top} \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Compute gradient of hidden layer below (before activation):

$$\nabla_{\boldsymbol{z}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell-1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell-1)})$$

end for

Outline

Conclusions

- Multi-layer perceptrons are universal function approximators
- However, they need to be trained
- Stochastic gradient descent is an effective training algorithm
- This is possible with the gradient backpropagation algorithm (an application of the chain rule of derivatives)

Next class:

- Most current software packages represent a computation graph and implement automatic differentiation
- Dropout regularization is effective to avoid overfitting
- Tricks of the trade

Thank you!

Questions?

References I

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Minsky, M. and Papert, S. (1969). Perceptrons.

Montufar, G. F., Pascanu, R., Cho, K., and Bengio, Y. (2014). On the number of linear regions of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2924–2932.