# Lecture 3: Neural Networks

André Martins




Deep Structured Learning Course, Fall 2018

# Announcements

Homework 1's deadline is October 10 (next week)

Deadline for project proposal: October 17 (two weeks from now)

- List of project ideas are now in the course webpage!
- You need to turn in a 1-page proposal (use NIPS format) explaining:
  - The problem you propose to solve
  - Which method(s) you are going to use
  - Which evaluation metric and which data you are going to use
- Team size: 2–4 people

# Announcements

Next week: Erick Fonseca will give a guest lecture!

- The lecture will be a practical Pytorch tutorial
- Please bring your laptops if you can!
- Even better: install Pytorch in your computer before the lecture!

# Today's Roadmap

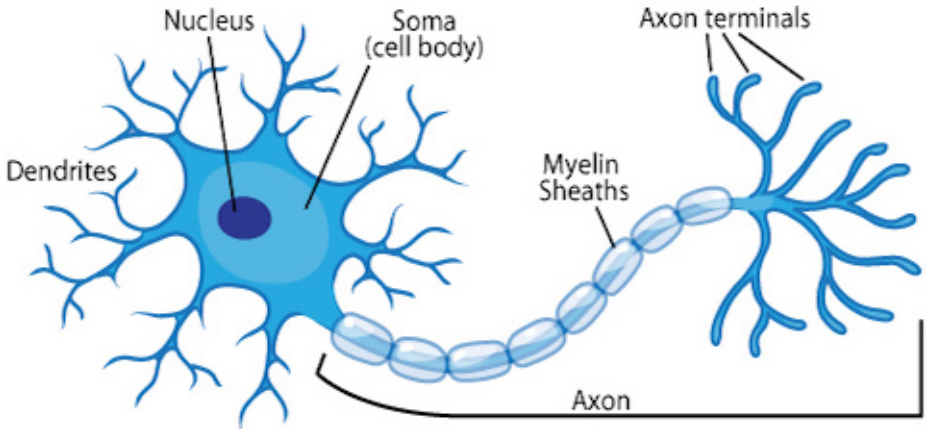
Today's lecture is about neural networks:

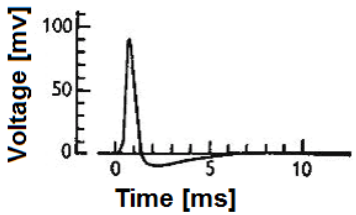
- From perceptron to multi-layer perceptron
- Feed-forward neural networks
- Activation funcions: sigmoid, tanh, relu, ...
- Activation maps: softmax, sparsemax, ...
- Non-convex optimization and local minima
- Universal approximation theorem
- Gradient backpropagation

# Outline

# Outline

# Biological Neuron

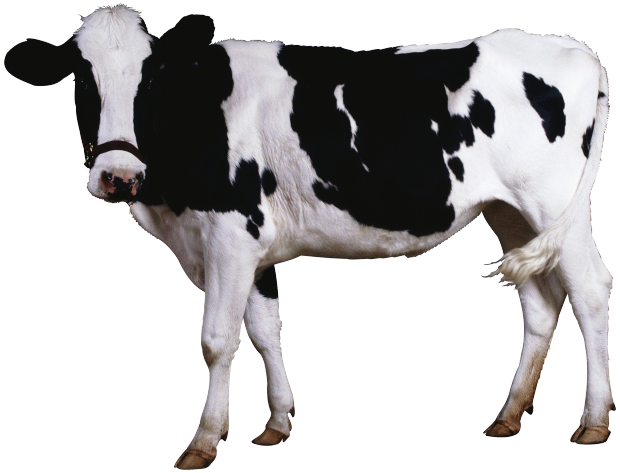


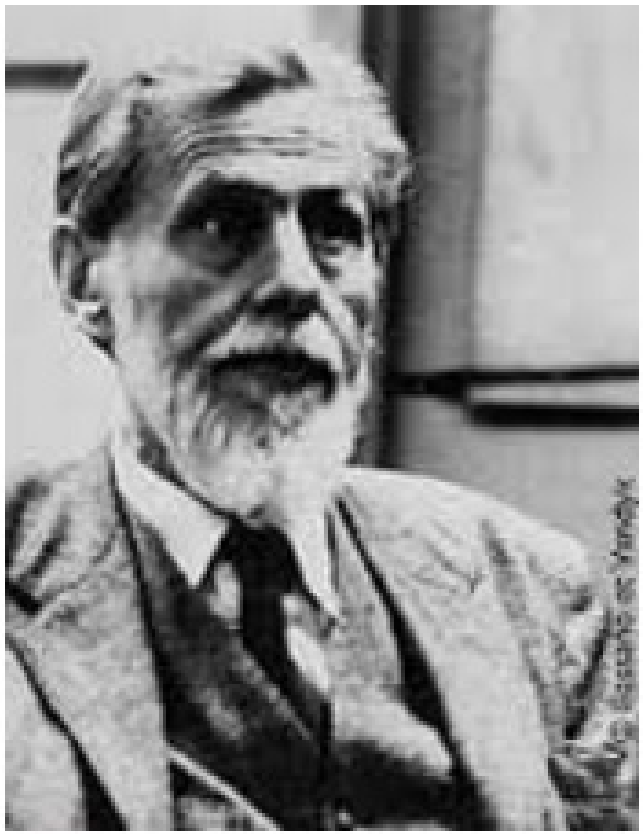

- Three main parts: the main body (soma), dendrites and an axon
- The neuron receives input signals from dendrites, and then outputs its own signals through the axon
- Axons in turn connect to the dendrites of other neurons, using special connections called synapses
- Generate sharp electrical potentials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

# Word of Caution

- Artificial neurons are inspired by biological neurons in nervous systems, but...



# Warren McCulloch and Walter Pitts



- The earliest computational model of a neuron, via threshold logic (**?**).

# Artificial Neuron (?)

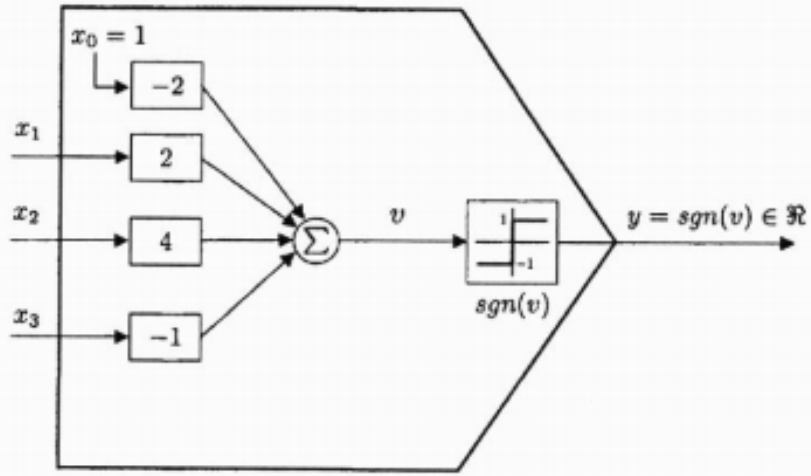



***Figure 3.6*** Example 3.2: a threshold neural logic for $y = x_2(x_1 + \bar{x}_3)$.

***Table 3.6*** **Truth table for Example 3.2**

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

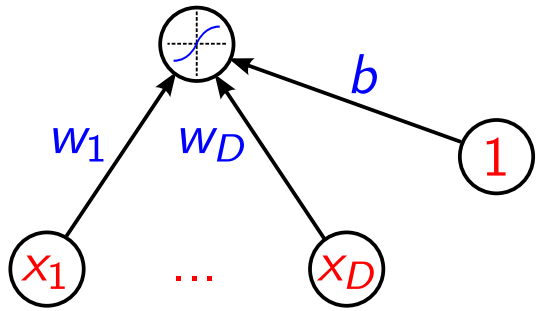
- Later models replaced the hard threshold by more general activations

# 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.




- **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:**

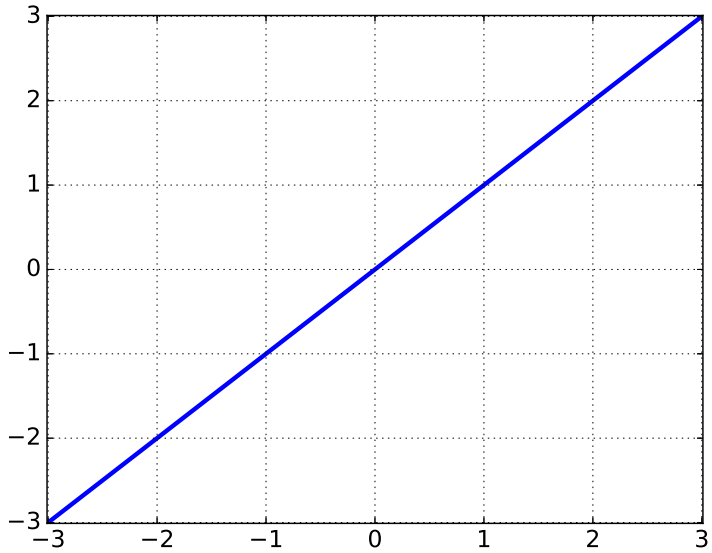
- Linear
- Sigmoid (logistic function)
- Hyperbolic Tangent
- Rectified Linear

**Later:**

- Softmax
- Sparsemax
- Max-pooling

# Linear Activation

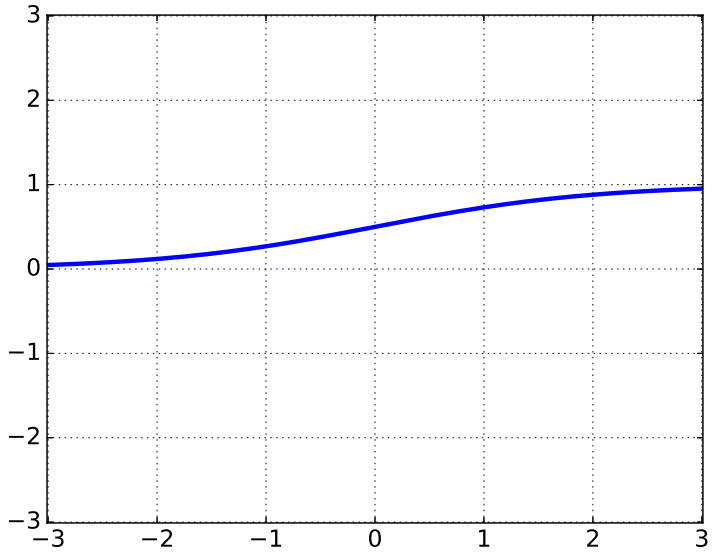
$$\boxed{g(z) = z}$$




- No "squashing" of the input
- Composing layers of linear units is equivalent to a single layer of linear units, so no expressive power added when going multi-layer (more later)
- Still useful to linear-project the input to a lower dimension

# Sigmoid Activation

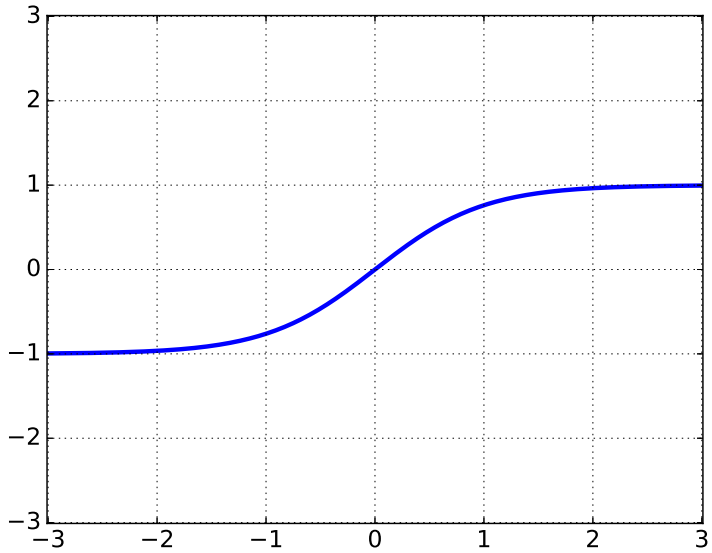
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$




- "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 will increase expressiveness (more later)

# Hyperbolic Tangent Activation

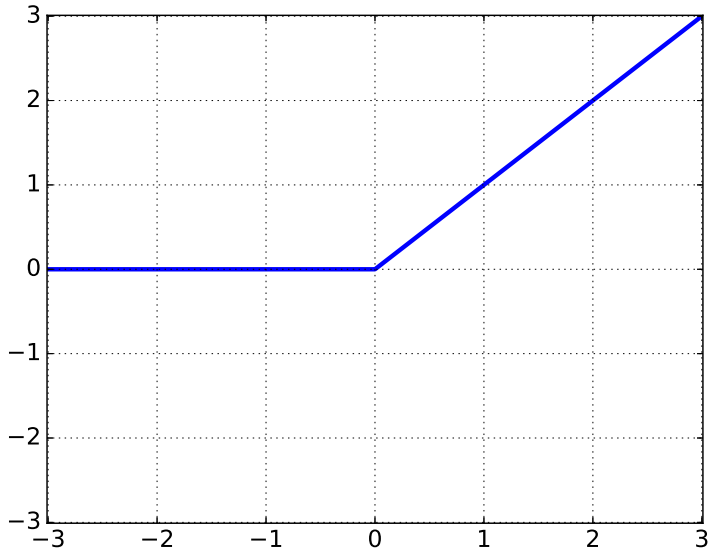
$$g(z) = \textbf{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$




- "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 will increase expressiveness (more later)

# Rectified Linear Unit Activation (?)

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




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

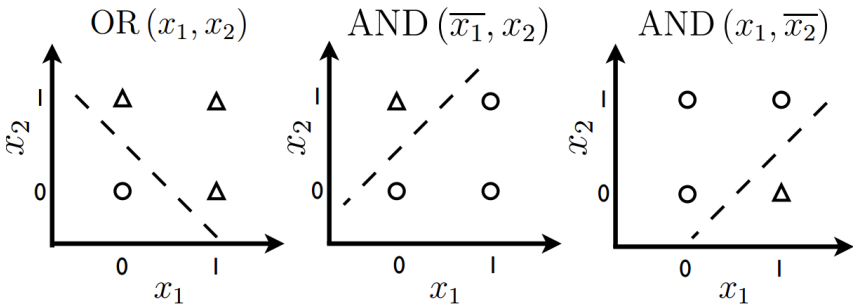
# Outline

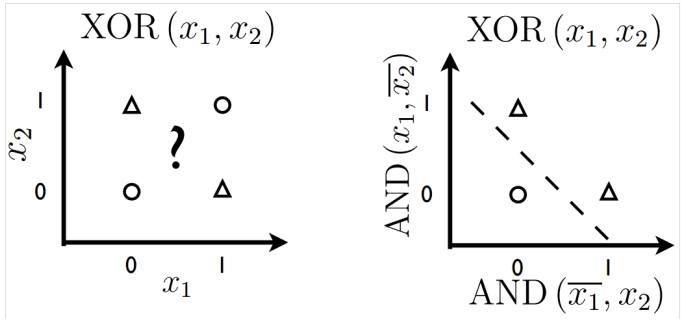
# Capacity of Single Neuron (Linear Classifier)

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

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

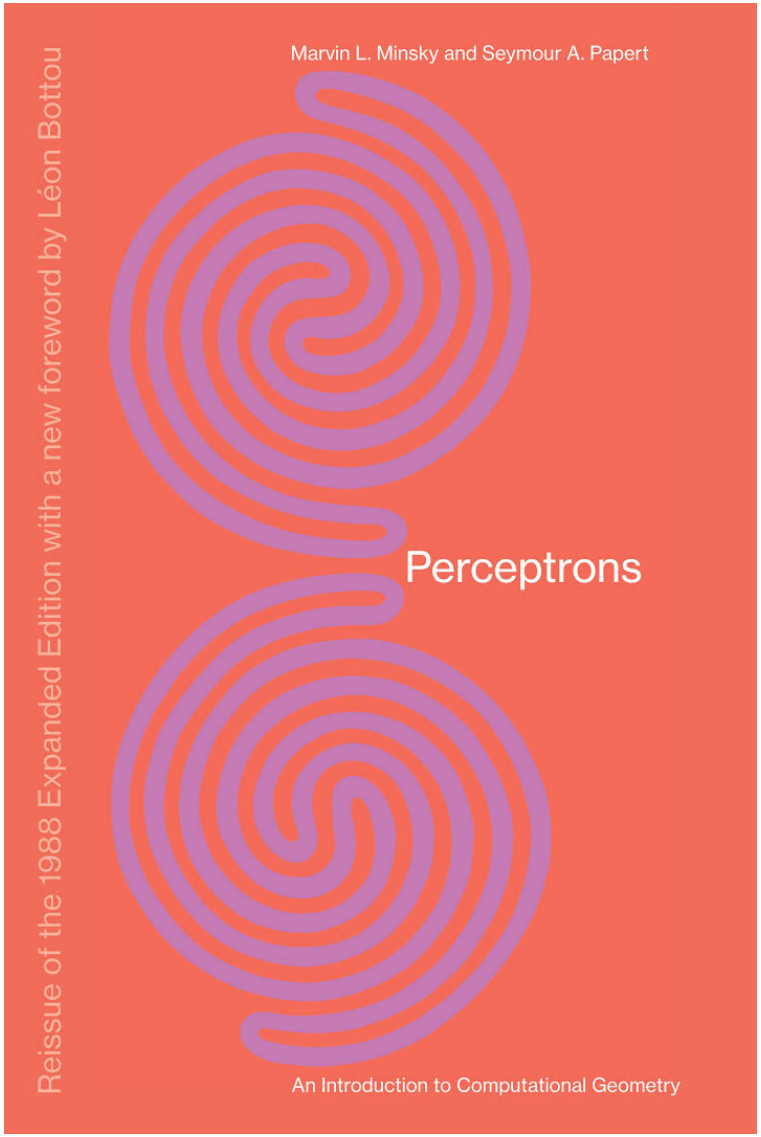
- Can solve linearly separable problems (OR, AND)
- Can't solve non-linearly separable problems (XOR)—unless input is transformed into a better representation




(Slide credit: Hugo Larochelle)

# The XOR Affair




**?**:

- 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

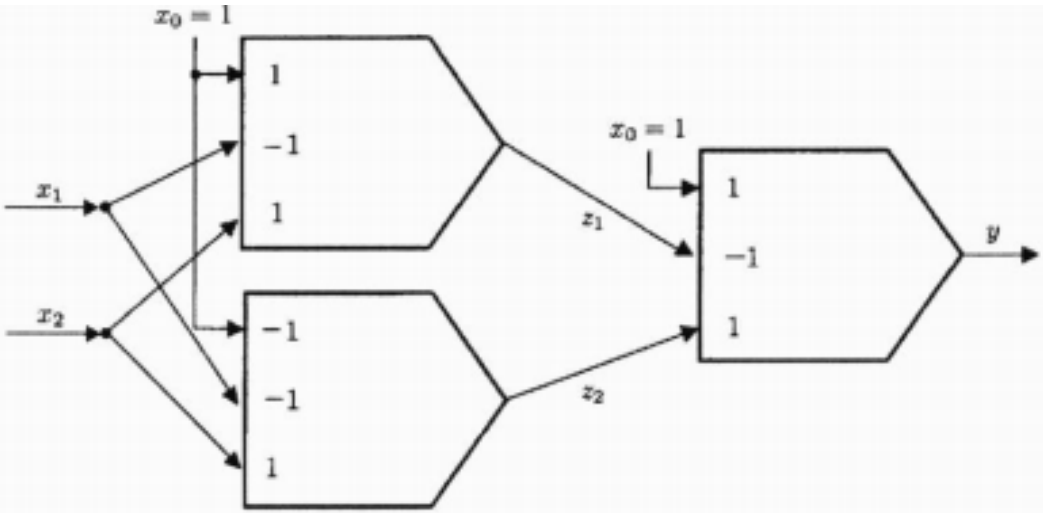



***Figure 3.11*** Example 3.6: a threshold network for *XOR* function $y = x_1 \oplus x_2 = sgn(1 - z_1 + z_2)$, $z_1 = sgn(1 - x_1 + x_2)$, $z_2 = sgn(-1 - x_1 + x_2)$.

# Solving XOR with Multi-Layer Perceptron

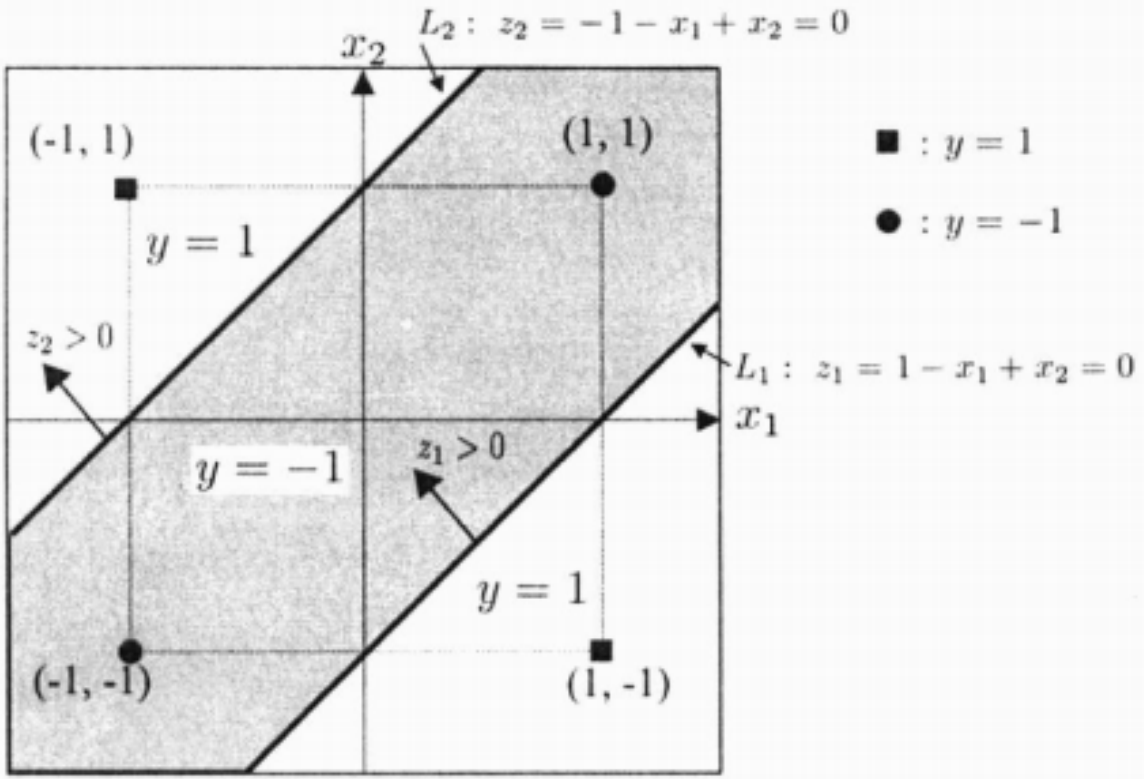



***Figure 3.12*** Example 3.6: two discriminant lines for *XOR* function $y = x_1 \oplus x_2$.

# Solving XOR with Multi-Layer Perceptron

- This construction was known by McCulloch and Pitts themselves!
- The negative result in **?** 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

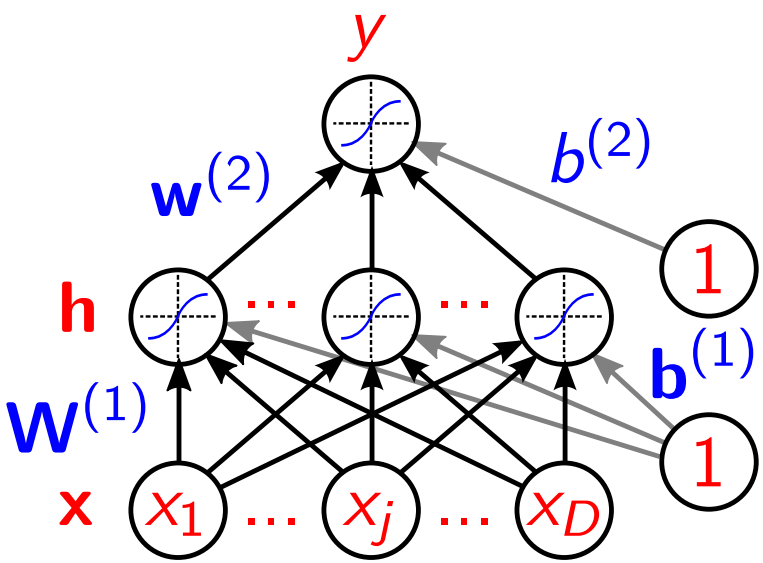
# Multi-Layer Neural Network

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

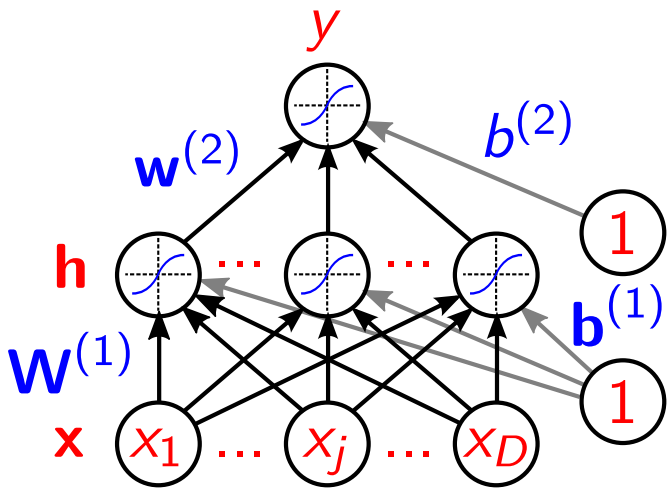
$$\boldsymbol{z}(\boldsymbol{x}) = \mathbf{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

with $\mathbf{W}^{(1)} \in \mathbb{R}^{K \times D}$ 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 vectorwise.




- **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,

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

Examples:

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

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

$\boldsymbol{h}$ is a vector of internal representations (as opposed to manually engineered features)

# Single Hidden Layer

Can we use a similar strategy for multi-class classification? What do we need to change?

# Multiple Classes

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 the one with highest estimated probability

We'll see two activation functions suitable for this:

- Softmax activation
- Sparsemax activation

# Softmax Activation

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

The typical activation function for multi-class classification is **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 say this already in class, when talking about logistic regression!
- Strictly positive, sums to 1
- Resulting distribution has full support: $\textbf{softmax}(\boldsymbol{z}) > \mathbf{0}, \forall \boldsymbol{z}$
- A disadvantage if a *sparse* probability distribution is desired
- Common workaround: threshold and truncate

# Sparsemax Activation (?)

A sparse-friendly alternative is **sparsemax** : $\mathbb{R}^C \to \Delta^{C-1}$, defined as:

$$\boxed{\boldsymbol{o}(\boldsymbol{z}) = \textbf{sparsemax}(\boldsymbol{z}) := \arg\min_{\boldsymbol{p} \in \Delta^{C-1}} \|\boldsymbol{p} - \boldsymbol{z}\|^2.}$$

- In words: Euclidean projection of $\boldsymbol{z}$ onto the probability simplex
- Likely to hit the boundary of the simplex, in which case **sparsemax**$(\boldsymbol{z})$ becomes sparse (hence the name)
- Retains many of the properties of softmax (e.g. differentiability), having in addition the ability of producing sparse distributions
- Projecting onto the simplex amounts to a soft-thresholding operation (next)
- Efficient forward/backward propagation (more later)

# Sparsemax in Closed Form

- Projecting onto the simplex amounts to a soft-thresholding operation:

$$\textbf{sparsemax}_i(\boldsymbol{z}) \;=\; \max\{0, z_i - \tau\}$$

  where $\tau$ is a normalizing constant such that $\sum_j \max\{0, z_j - \tau\} = 1$
- To evaluate the sparsemax, all we need is to compute $\tau$
- Coordinates above the threshold will be shifted by this amount; the others will be truncated to zero

# A Formal Algorithm

**Input:** $\boldsymbol{z} \in \mathbb{R}^K$

Sort $\boldsymbol{z}$ as $z_{(1)} \geq \ldots \geq z_{(K)}$

Find $k(\boldsymbol{z}) := \max \left\{ k \in [K] \mid 1 + k z_{(k)} > \sum_{j \leq k} z_{(j)} \right\}$

Define $\tau(\boldsymbol{z}) = \dfrac{\left(\sum_{j \leq k(\boldsymbol{z})} z_{(j)}\right) - 1}{k(\boldsymbol{z})}$

**Output:** $\boldsymbol{p} \in \Delta^{K-1}$ s.t. $p_i = [z_i - \tau(\boldsymbol{z})]_+$.

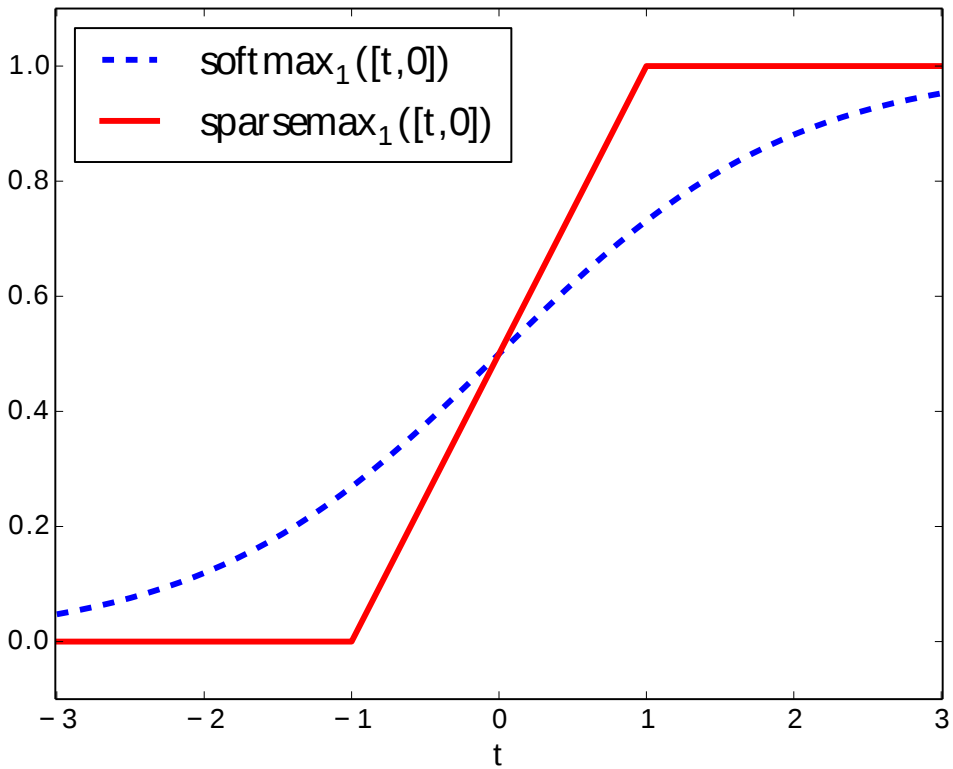
- Time complexity is $O(K \log K)$ due to the sort operation; but $O(K)$ algorithms exist based on linear-time selection.
- Note: evaluating **softmax** costs $O(L)$ too.

# Two Dimensions

- Parametrize $z = (t, 0)$
- The 2D **softmax** is the logistic (sigmoid) function:

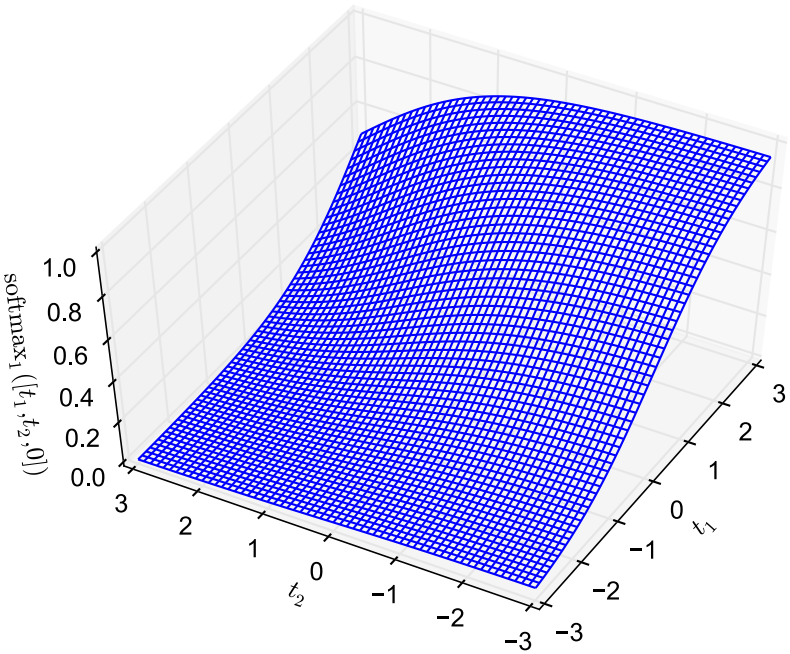
$$\textbf{softmax}_1(z) = (1 + \exp(-t))^{-1}$$

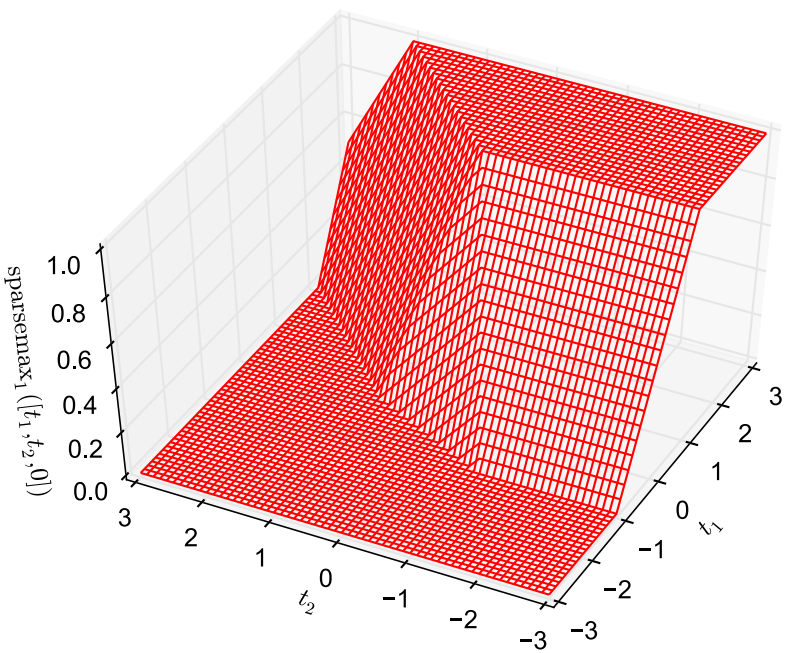
- The 2D **sparsemax** is the "hard" version of the sigmoid:



# Three Dimensions

- Parameterize $z = (t_1, t_2, 0)$ and plot $\textbf{softmax}_1(z)$ and $\textbf{sparsemax}_1(z)$ as a function of $t_1$ and $t_2$
- **sparsemax** is piecewise linear, but asymptotically similar to **softmax**



# Multi-Layer Neural Networks: General Case

In general we can:

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

# Multiple Hidden Layers

Now assume $L \geq 1$ hidden layers:

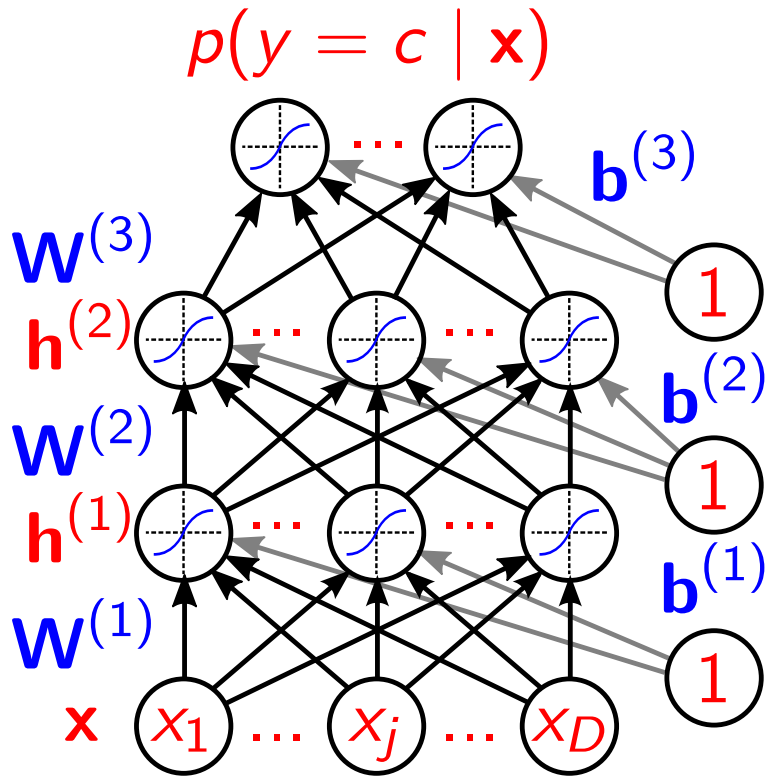
- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

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

  with $\mathbf{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$ 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:**

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

# Outline

# Universal Approximation Theorem

**Theorem (?)**

*A neural network with a single hidden layer and a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units.*

- First proved for the sigmoid case by **?**, then to **tanh** and many other activation functions by **?**
- **Caveat:** may need exponentially many hidden units

# Deeper Networks

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

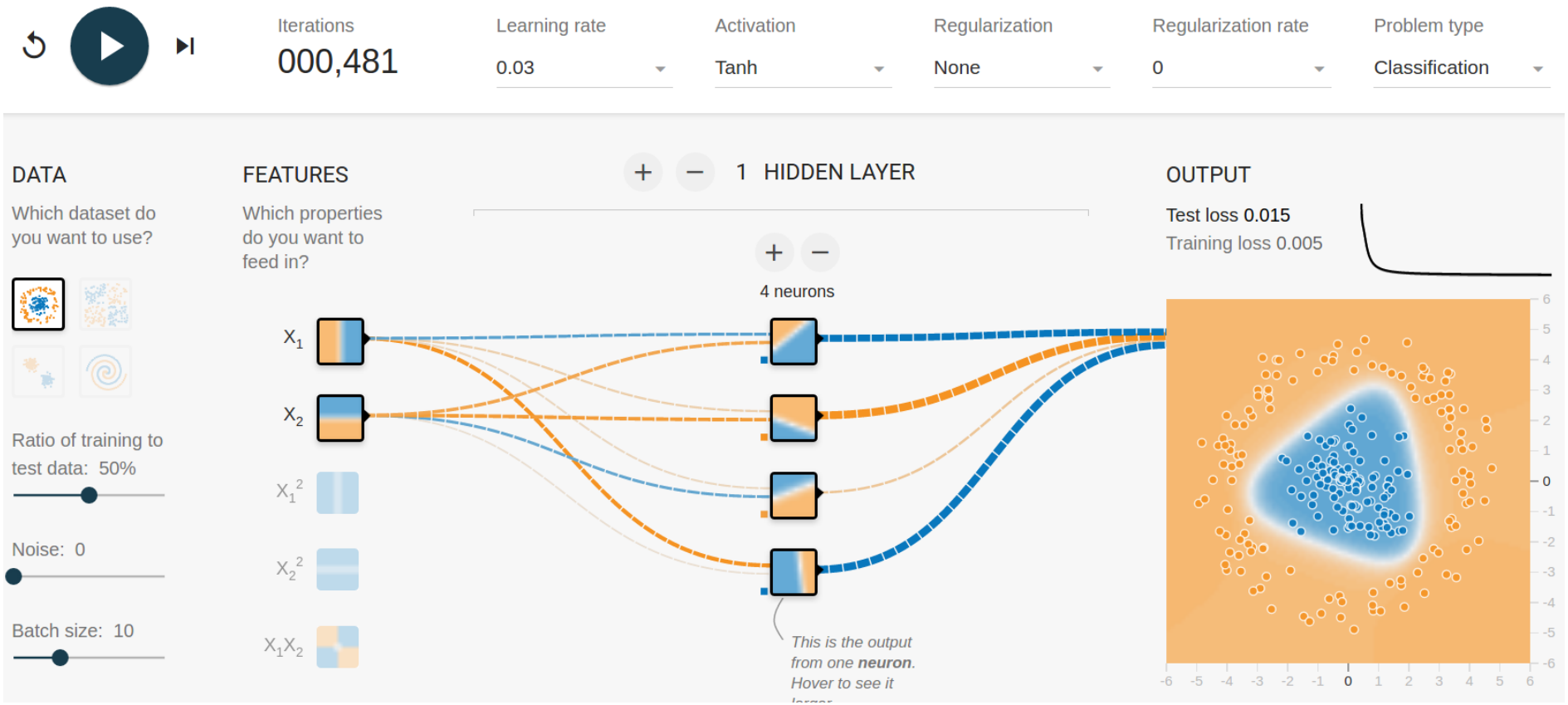
**Theorem (?)**

*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*

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

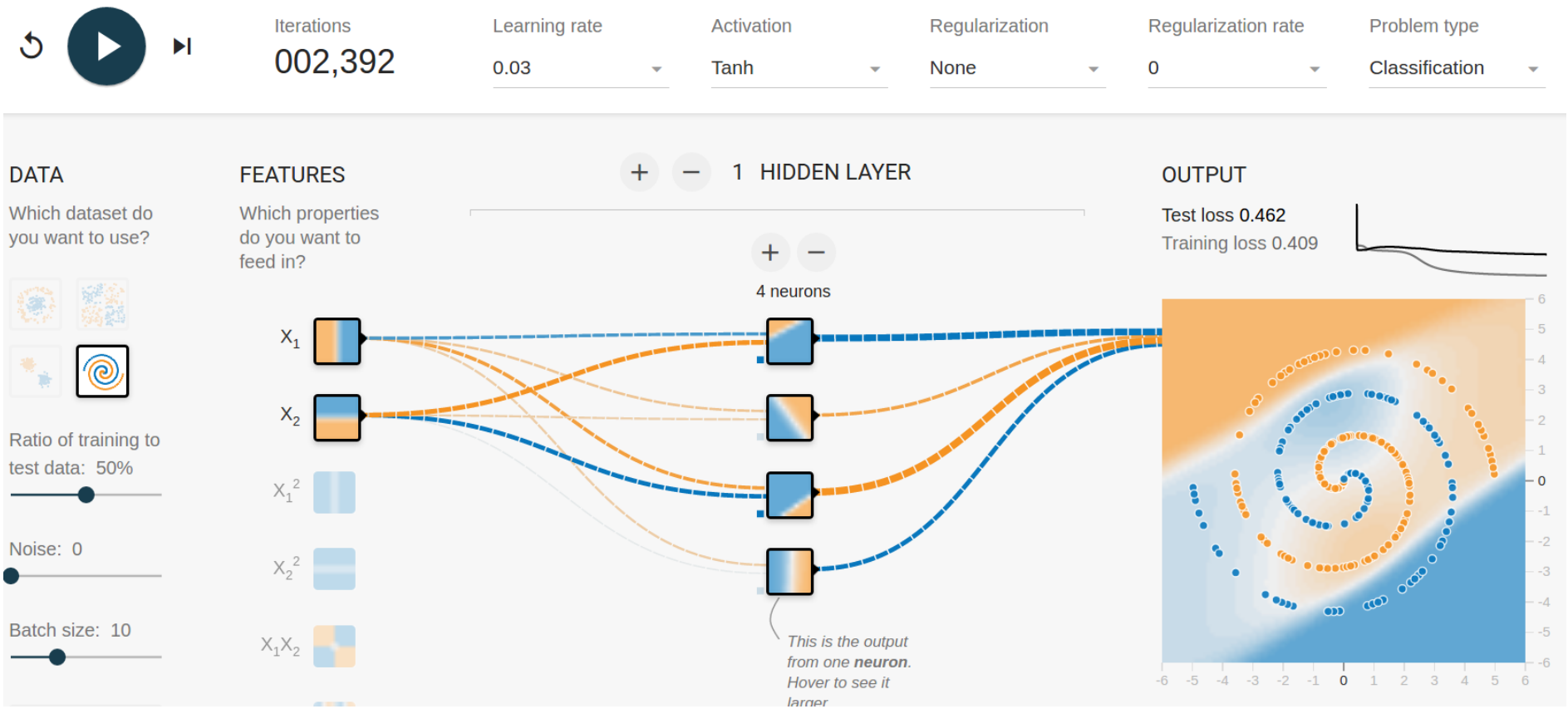
Therefore, for fixed $K$, deeper networks are exponentially more expressive

# "Simple" Target Function, One Hidden Layer



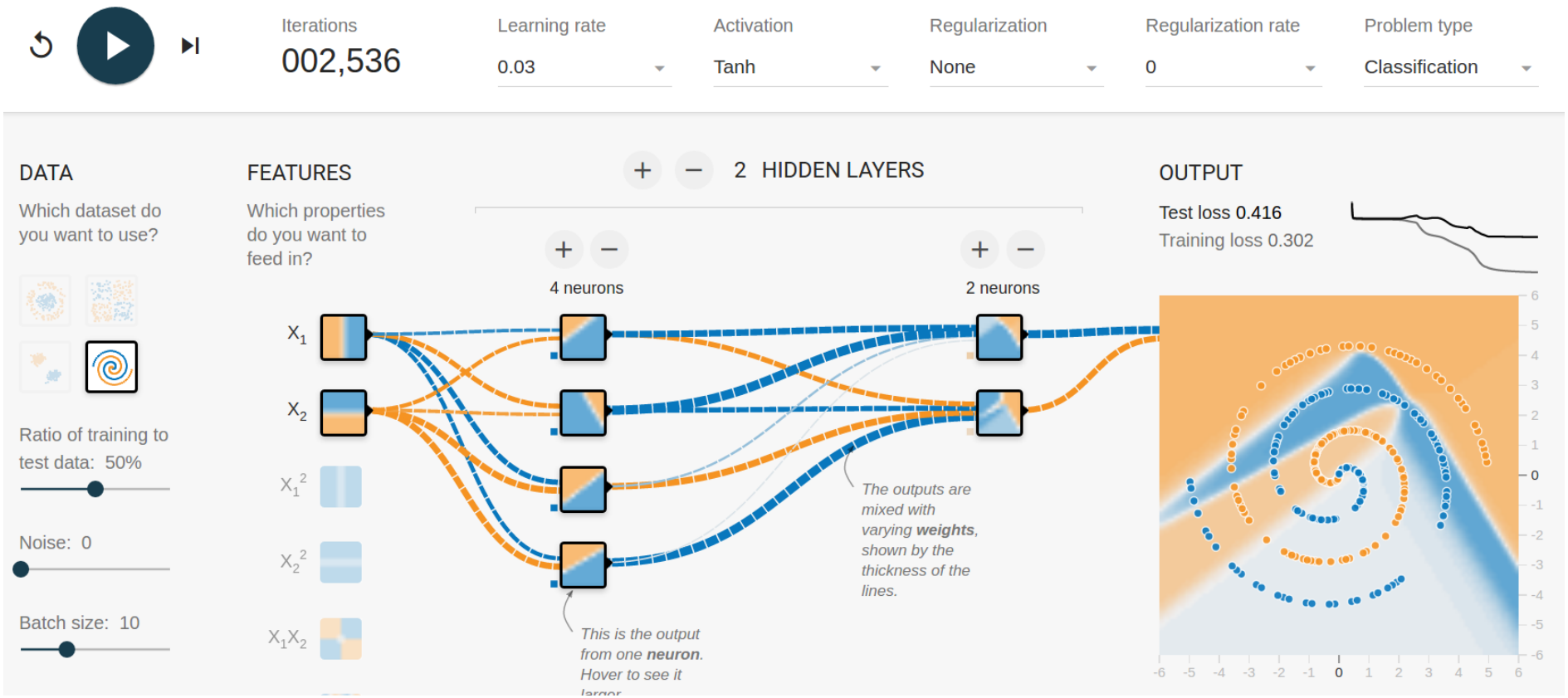

(http://playground.tensorflow.org)

# Complex Target Function, One Hidden Layer



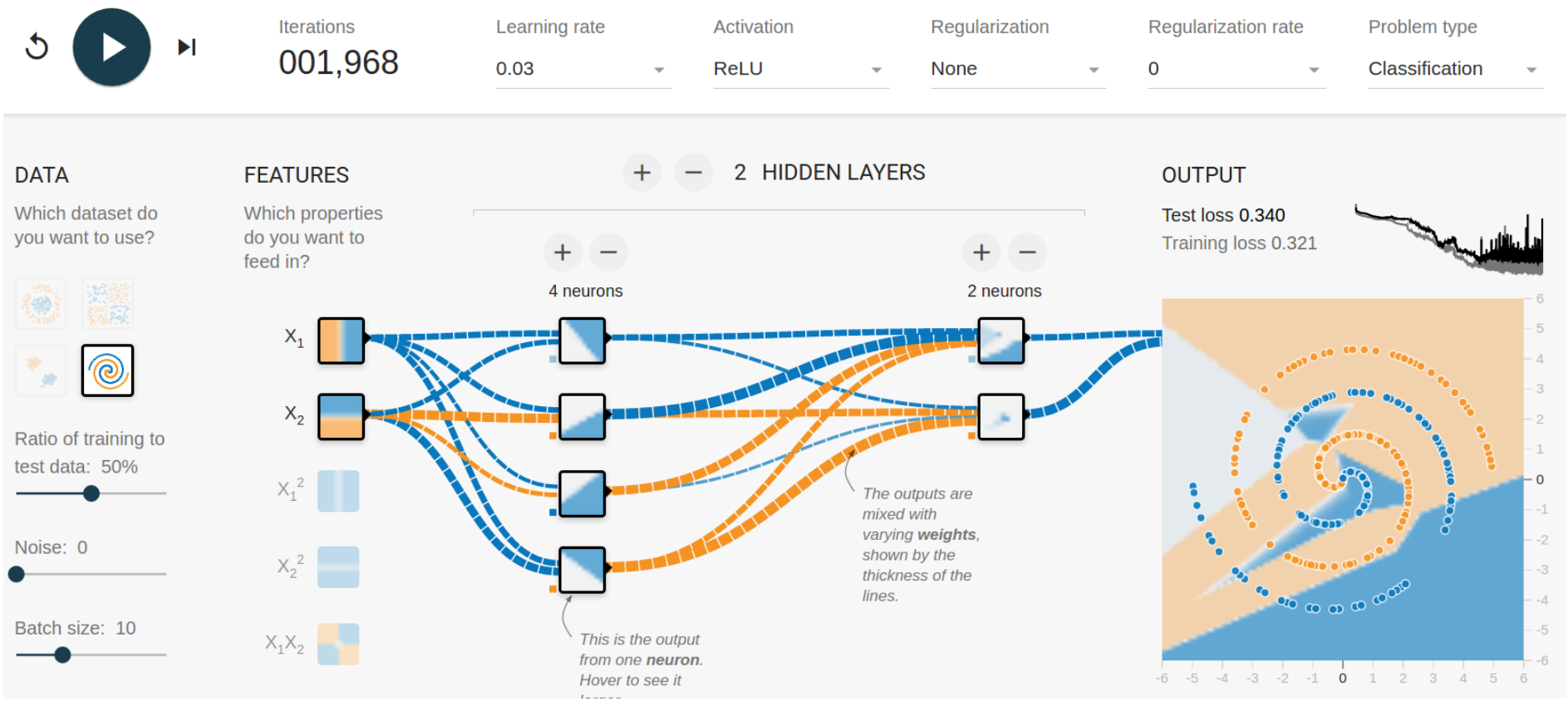

(http://playground.tensorflow.org)

# Complex Target Function, Two Hidden Layers



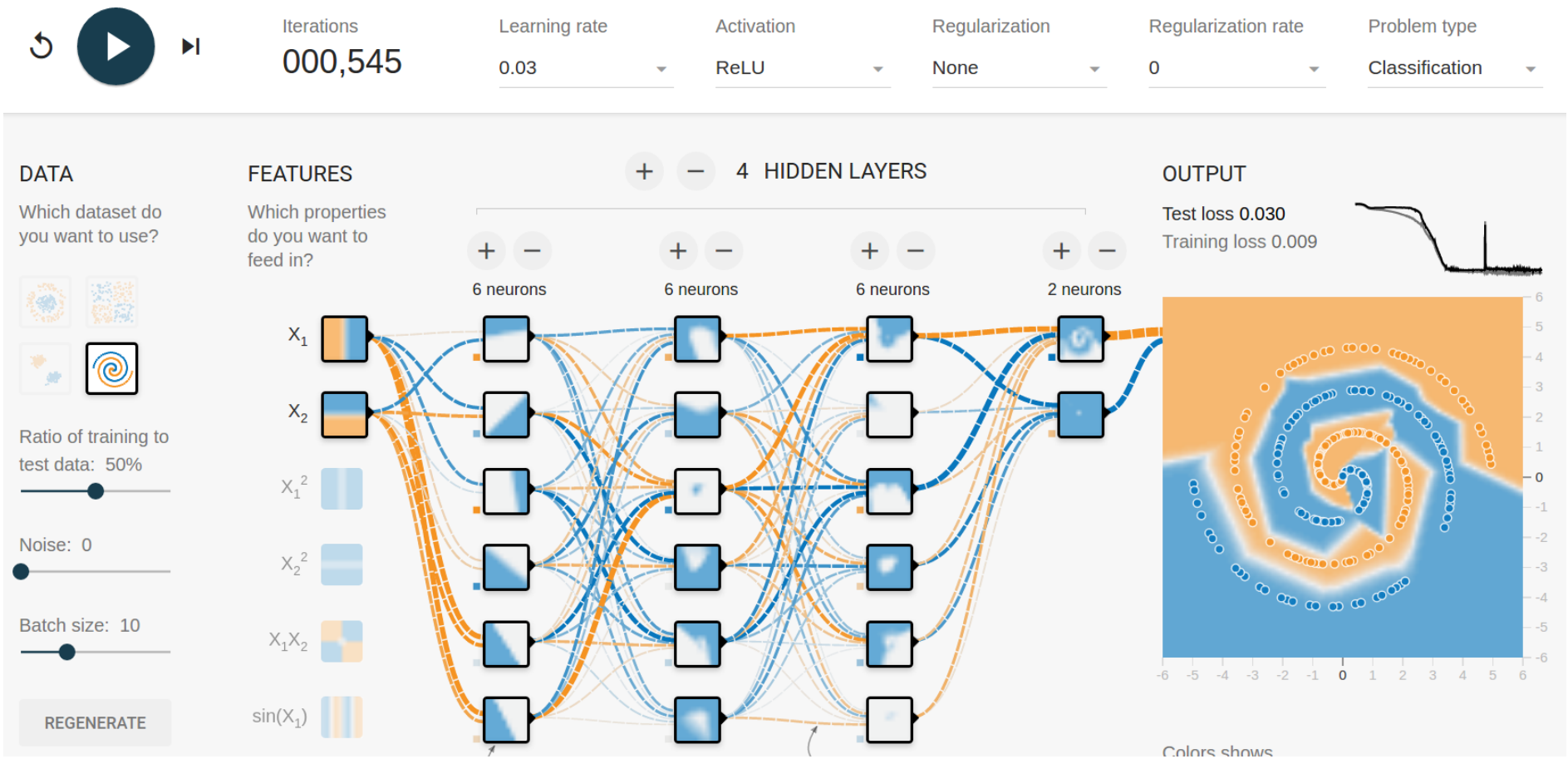

(http://playground.tensorflow.org)

# 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 a nice 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

We've seen that neural networks are very expressive—in theory, they can approximate any function we want

But to do so, their parameters $\boldsymbol{\theta} := \{(\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ need to be set accordingly, for every layer

**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 already when we talked about linear models a few days ago...)

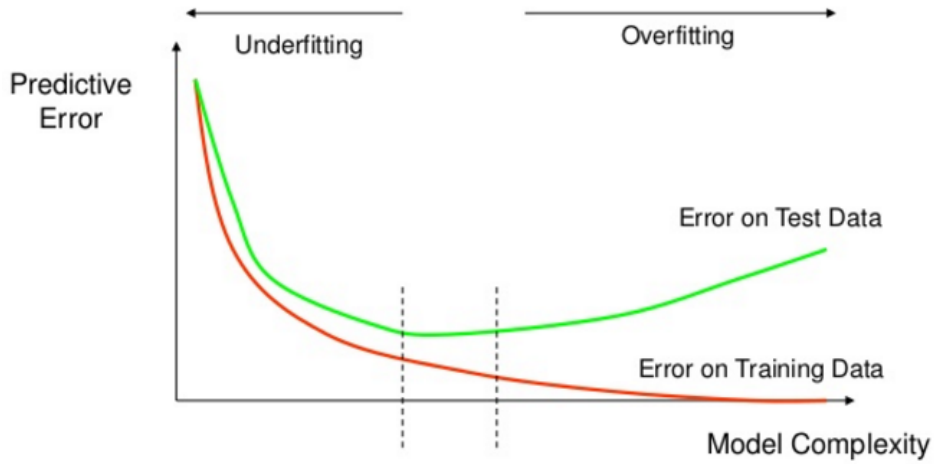
# Outline

# Empirical Risk Minimization

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

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

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



# Recap: Gradient Descent

We can write the objective as:

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

where $\mathcal{L}_i = \lambda\Omega(\boldsymbol{\theta}) + L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$

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 before updating the weights—too slow!

# Recap: Stochastic Gradient Descent

**Key idea:** sample a single training example uniformly (a random $j \in [N]$)

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

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

In practice, we often use a mini-batch instead of a single example!

# 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 $\boldsymbol{\theta} = \{(\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ 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, we need 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
- The neural network estimates $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$
- We 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
- 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):

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

  where $\boldsymbol{z}$ is the output pre-activation.
- Loss gradient at output pre-activation:

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

# Sparsemax Loss (?)

- The natural choice for a sparsemax output layer
- The neural network estimates $f_c(\boldsymbol{x}; \boldsymbol{\theta}) \approx p(y = c \mid \boldsymbol{x})$ as a sparse distribution

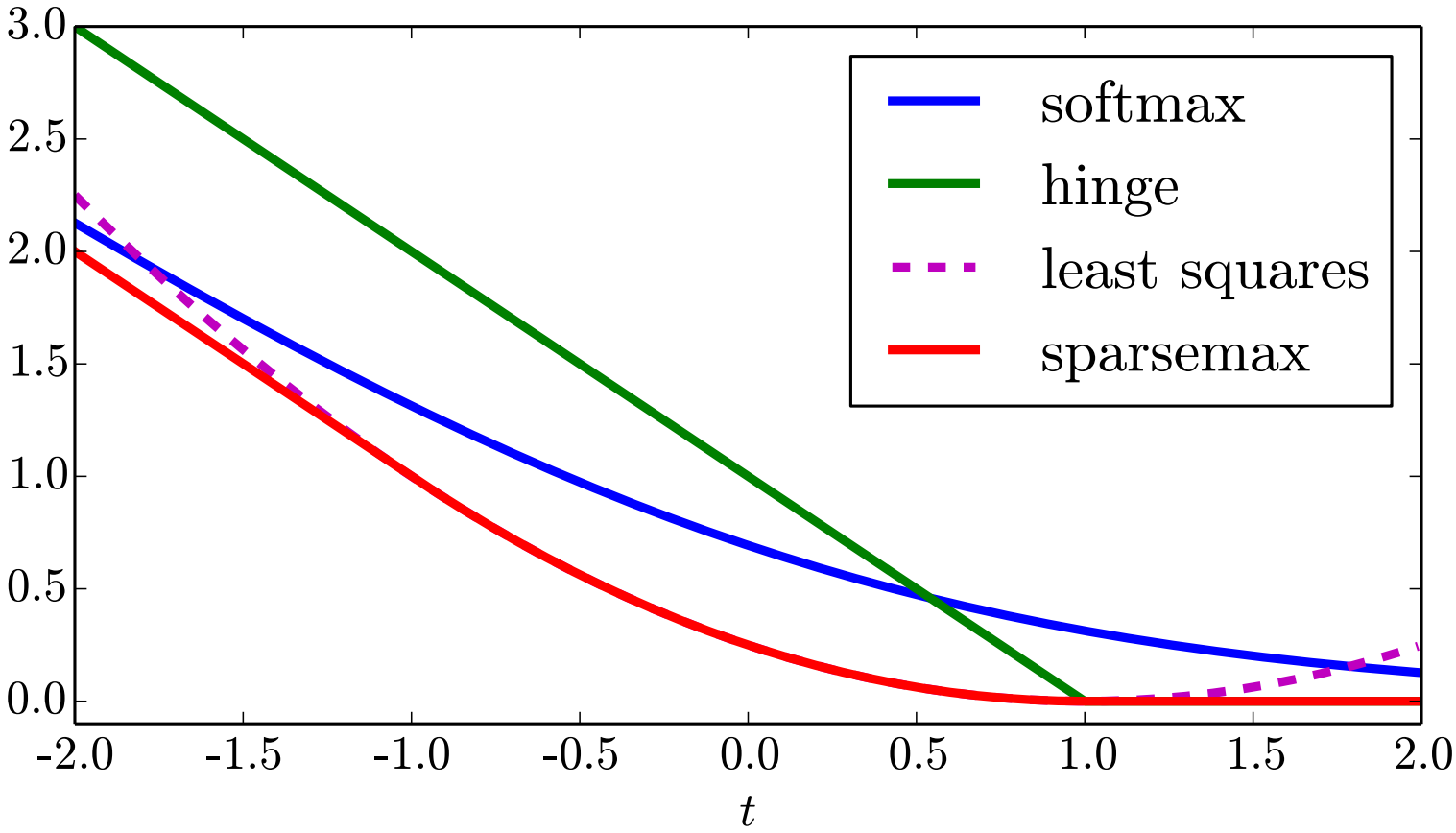
$$L(\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}), y) = -z_c + \frac{1}{2} \sum_{j \in S(\boldsymbol{z})} (z_j^2 - \tau^2(\boldsymbol{z})) + \frac{1}{2},$$

  where $\boldsymbol{z}$ is the output pre-activation, $S(\boldsymbol{z})$ is the support of $p(y \mid \boldsymbol{x})$ and $\tau^2 : \mathbb{R}^K \to \mathbb{R}$ is the square of the threshold function (see **?** for details).
- Loss gradient at output pre-activation:

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

# Classification Losses in Two Dimensions

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



# The Key Ingredients of SGD

In sum, we need 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

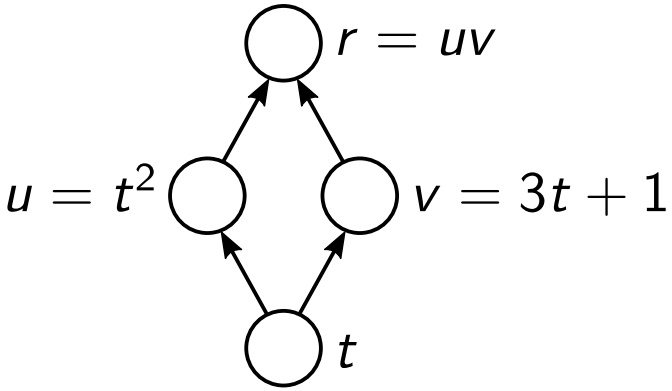
# Gradient Computation

- Recall that we need to compute

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

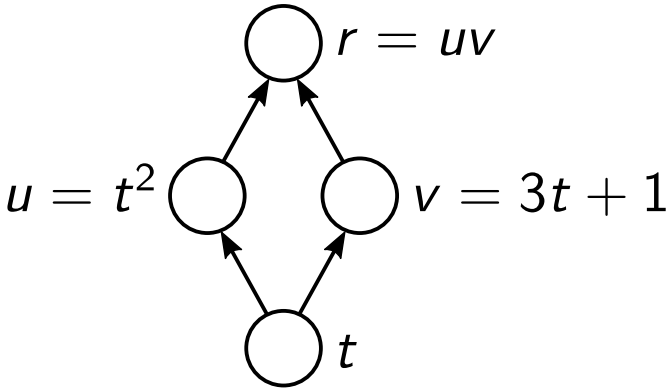
  for $\boldsymbol{\theta} = \{(\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ (the weights and biases at all layers)
- This will be done with the gradient backpropagation algorithm
- **Key idea:** use the chain rule for derivatives!

# Recap: Chain Rule



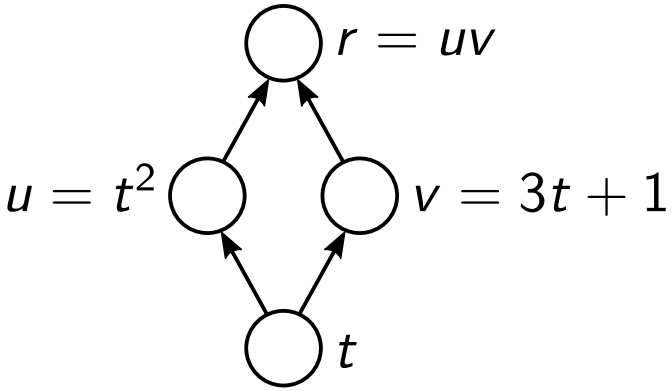

$$\frac{\partial r(t)}{\partial t} = ?$$

# Recap: Chain Rule



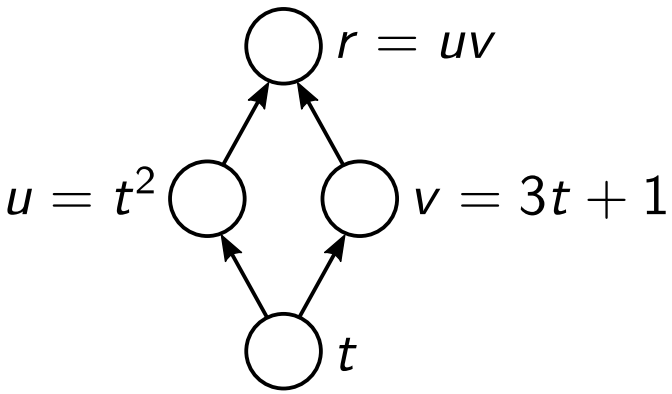

$$\frac{\partial r(t)}{\partial t} = \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t}$$

# 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.

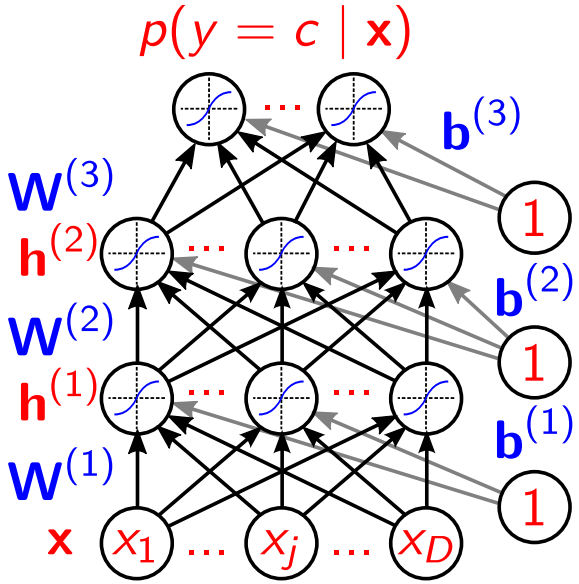
# Hidden Layer Gradient

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

# Hidden Layer Gradient

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

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

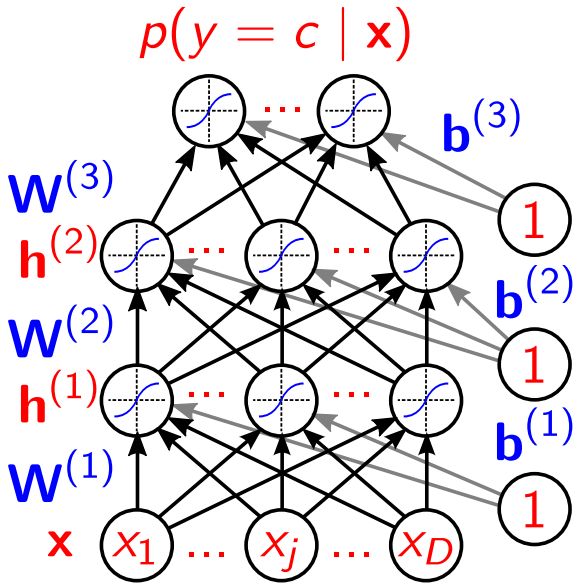



Hence $\nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \boldsymbol{W}^{(\ell+1)^\top} \nabla_{\boldsymbol{z}^{(\ell+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)$.

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

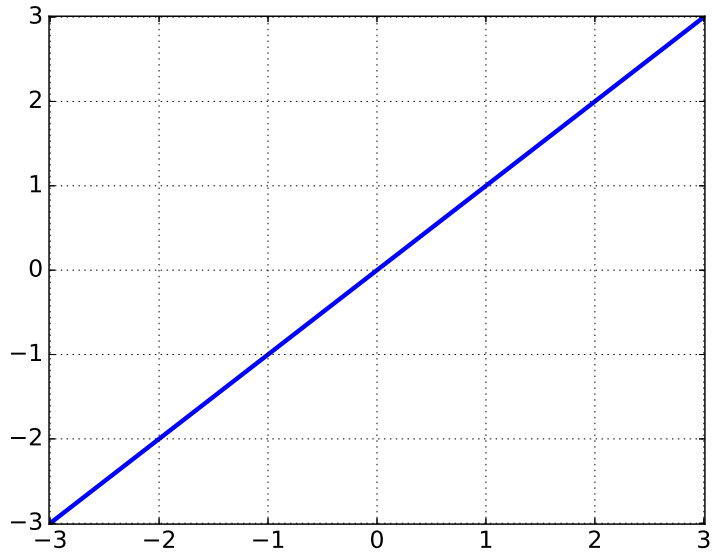
How to compute the derivative of the activation function $\boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$?

# Linear Activation

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

Derivative:

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

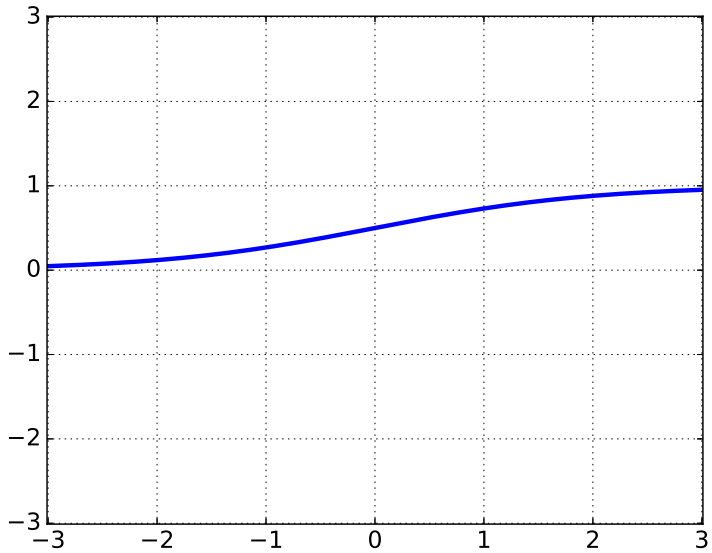


# Sigmoid Activation

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

Derivative:

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

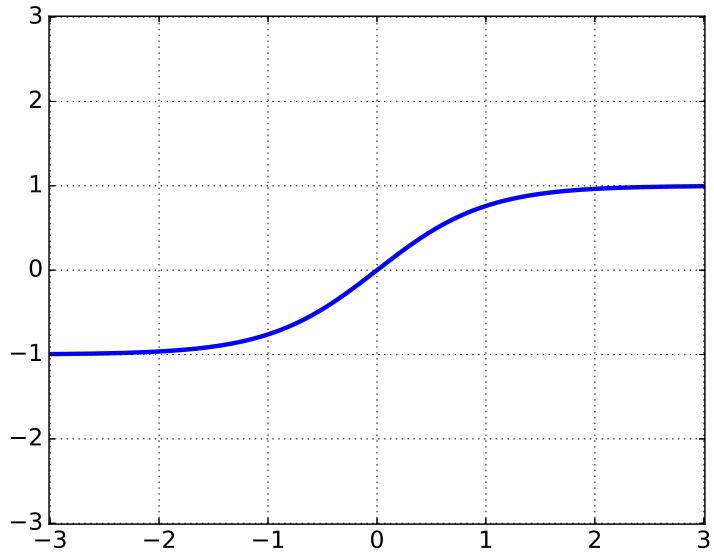


# Hyperbolic Tangent Activation

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

Derivative:

$$g'(z) = 1 - g(z)^2$$

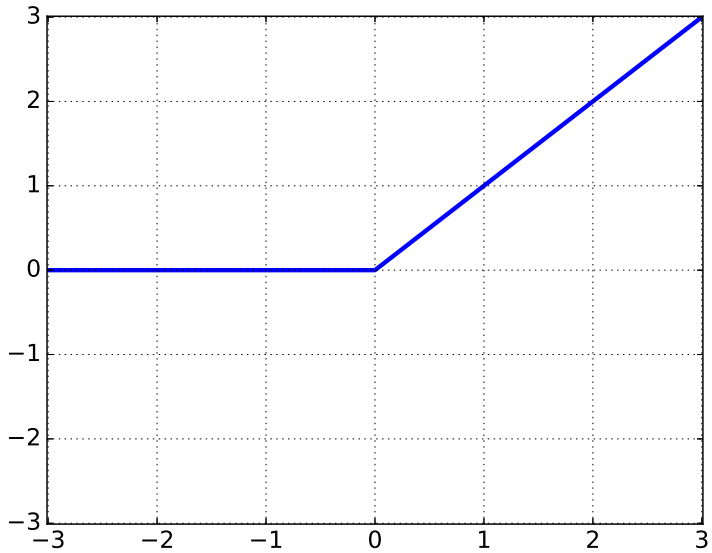


# Rectified Linear Unit Activation (?)

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

Derivative (except for $z = 0$):

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

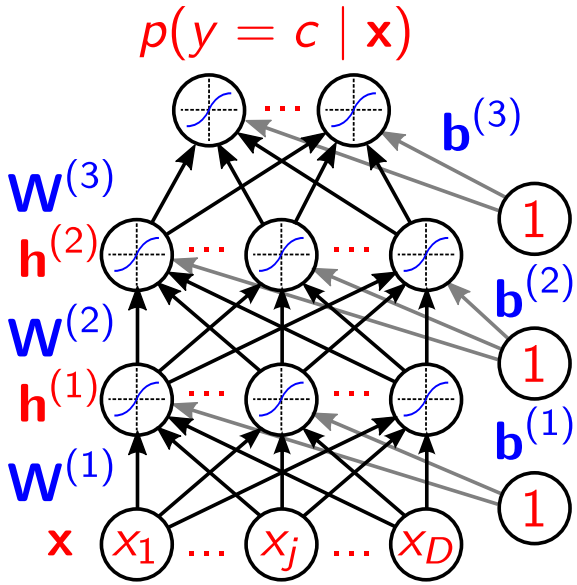


# Parameter Gradient

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

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

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




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

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** $\ell$ from $L+1$ to $1$ **do**

Compute gradients of hidden layer parameters:

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

Compute gradient of hidden layer below:

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

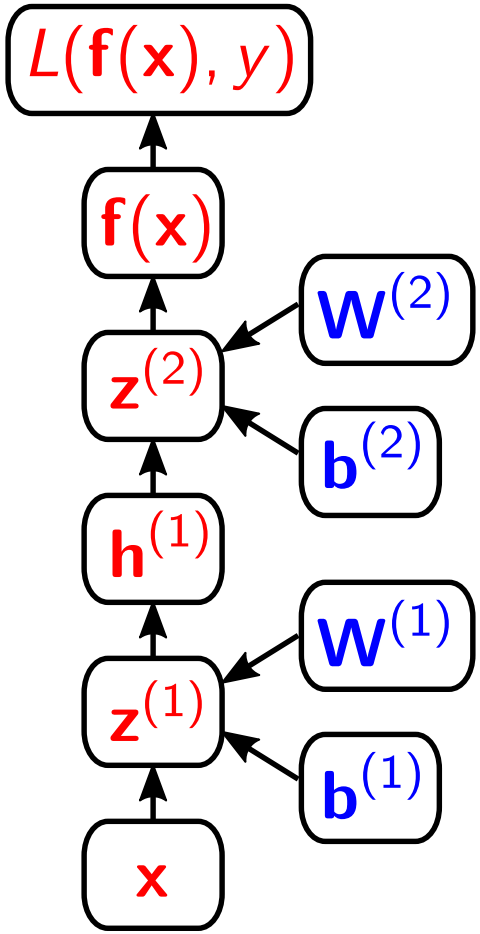
Compute gradient of hidden layer below (before activation):

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

**end for**

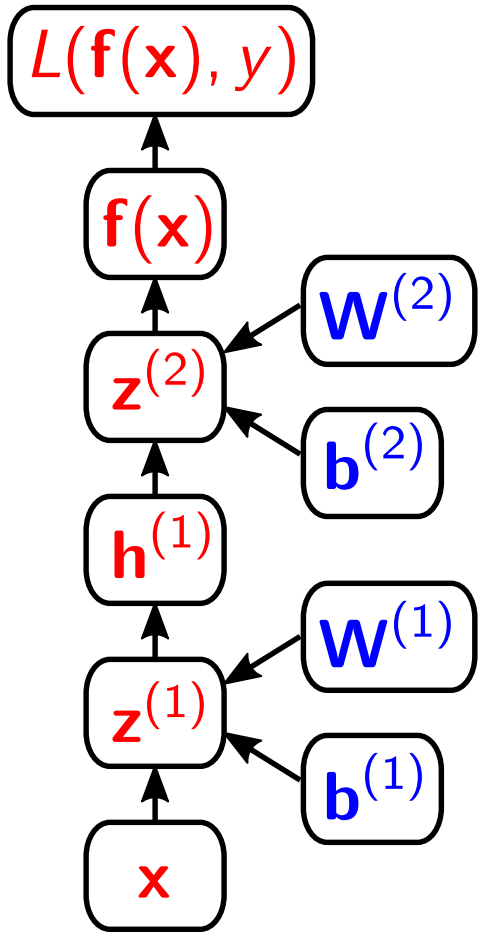
# Computation Graph

- Forward propagation can be represented as a DAG
- Allows to implement forward propagation in a modular way
- Each box can be an object with a `fprop` method, that computes the value of the box given its children
- Calling the `fprop` method of each box in the right order (after a topological sort) yields forward propagation



# Automatic Differentiation

- ... Also allows to implement backpropagation in a modular way
- Each box can also have a `bprop` method, that computes the loss gradient with respect to its children, given the loss gradient with respect to the output
- Can make use of cached computation done during the `fprop` method
- By calling the `bprop` method in reverse order, we get backpropagation (only need to reach the parameters)



# Several Autodiff Strategies

**Symbol-to-number differentiation** (Caffe, Torch, Pytorch, Dynet, ...)

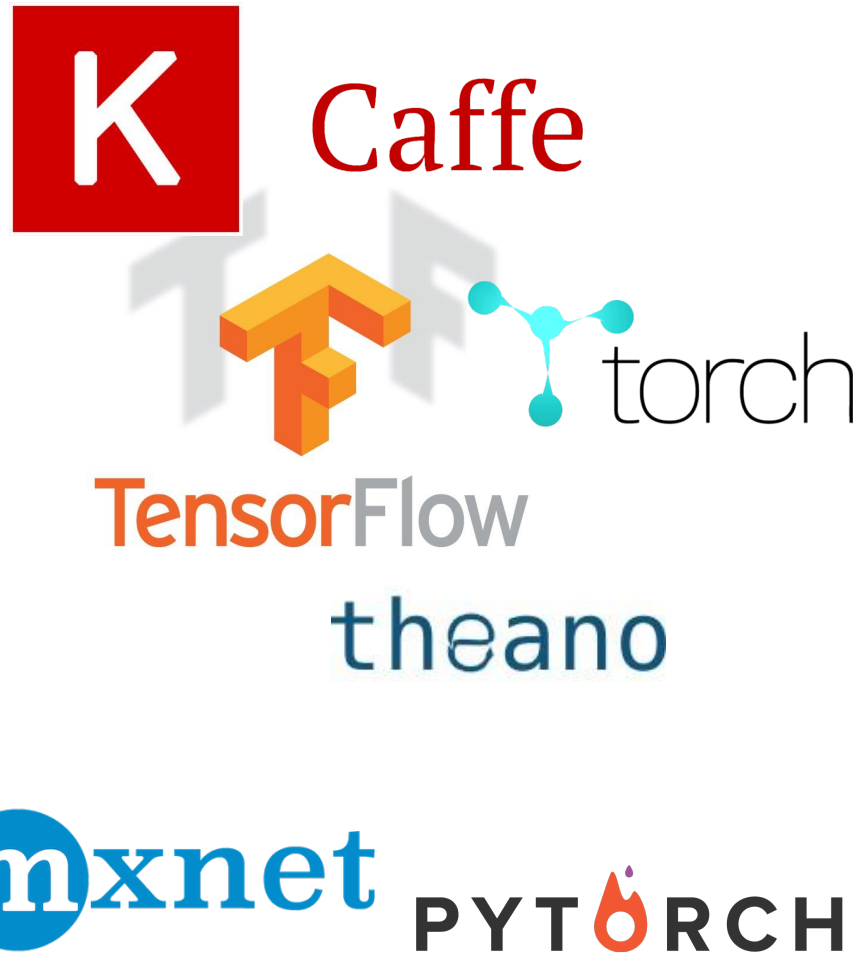
- Take a computational graph and a set of numerical inputs, then return a set of numerical values describing the gradient at those input values
- Advantage: simpler to implement and to debug
- Disadvantage: only works for first-order derivatives

**Symbol-to-symbol differentiation** (Theano, Tensorflow, ...)

- Take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives (i.e. the derivatives are just another computational graph)
- Advantage: generalizes automatically to higher-order derivatives
- Disadvantage: harder to implement and to debug

# Many Software Toolkits for Neural Networks

- Theano
- Tensorflow
- Torch, Pytorch
- MXNet
- Keras
- Caffe
- DyNet
- ...




All implement automatic differentiation.

We will have a Pytorch tutorial next class (guest lecture by Erick Fonseca)

Please bring your laptops!

# Some Theano Code (Logistic Regression)

```python
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400                                   # training sample size
feats = 784                               # number of input variables

# generate a dataset: D = (input_values, target_class)
D = (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))
training_steps = 10000

# Declare Theano symbolic variables
x = T.dmatrix("x")
y = T.dvector("y")

# initialize the weight vector w randomly
#
# this and the following bias variable b
# are shared so they keep their values
# between training iterations (updates)
w = theano.shared(rng.randn(feats), name="w")

# initialize the bias term
b = theano.shared(0., name="b")

print("Initial model:")
print(w.get_value())
print(b.get_value())
```

```python
# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))   # Probability that target = 1
prediction = p_1 > 0.5                    # The prediction thresholded
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1) # Cross-entropy loss function
cost = xent.mean() + 0.01 * (w ** 2).sum()# The cost to minimize
gw, gb = T.grad(cost, [w, b])             # Compute the gradient of the cost
                                          # w.r.t weight vector w and
                                          # bias term b
                                          # (we shall return to this in a
                                          # following section of this tutorial)

# Compile
train = theano.function(
          inputs=[x,y],
          outputs=[prediction, xent],
          updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = theano.function(inputs=[x], outputs=prediction)

# Train
for i in range(training_steps):
    pred, err = train(D[0], D[1])

print("Final model:")
print(w.get_value())
print(b.get_value())
print("target values for D:")
print(D[1])
print("prediction on D:")
print(predict(D[0]))
```

# Some Code in Tensorflow (Linear Regression)

```
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but TensorFlow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables.  We will 'run' this first.
init = tf.global_variables_initializer()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))

# Learns best fit is W: [0.1], b: [0.3]
```

# Some Code in Keras (Multi-Layer Perceptron)

**Multilayer Perceptron (MLP) for multi-class softmax classification:**

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, input_dim=20, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(10, init='uniform'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(X_train, y_train,
          nb_epoch=20,
          batch_size=16)
score = model.evaluate(X_test, y_test, batch_size=16)
```

# Reminder: The Key Ingredients of SGD

In sum, we need 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 – next!

# Regularization

Recall that we're minimizing the following objective function:

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

It remains to define the regularizer and its gradient

We'll talk about:

- $\ell_2$ regularization
- $\ell_1$ regularization
- dropout regularization

# $\ell_2$ Regularization

- Gaussian prior on the weights
- **Note:** only the weights are regularized (not the biases)

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \sum_{\ell} \|\mathbf{W}^{(\ell)}\|^2$$

- Gradient is:

$$\nabla_{\mathbf{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \mathbf{W}^{(\ell)}$$

- This has the effect of a weight decay:

$$\begin{aligned}
\mathbf{W}^{(\ell)} &\leftarrow \mathbf{W}^{(\ell)} - \eta \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}_i(\boldsymbol{\theta}) \\
&= \mathbf{W}^{(\ell)} - \eta(\lambda \nabla_{\mathbf{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) + \nabla_{\mathbf{W}^{(\ell)}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)) \\
&= (1 - \eta\lambda)\mathbf{W}^{(\ell)} - \eta \nabla_{\mathbf{W}^{(\ell)}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)
\end{aligned}$$

# $\ell_1$ Regularization

- Laplacian prior on the weights
- **Note:** only the weights are regularized (not the biases)

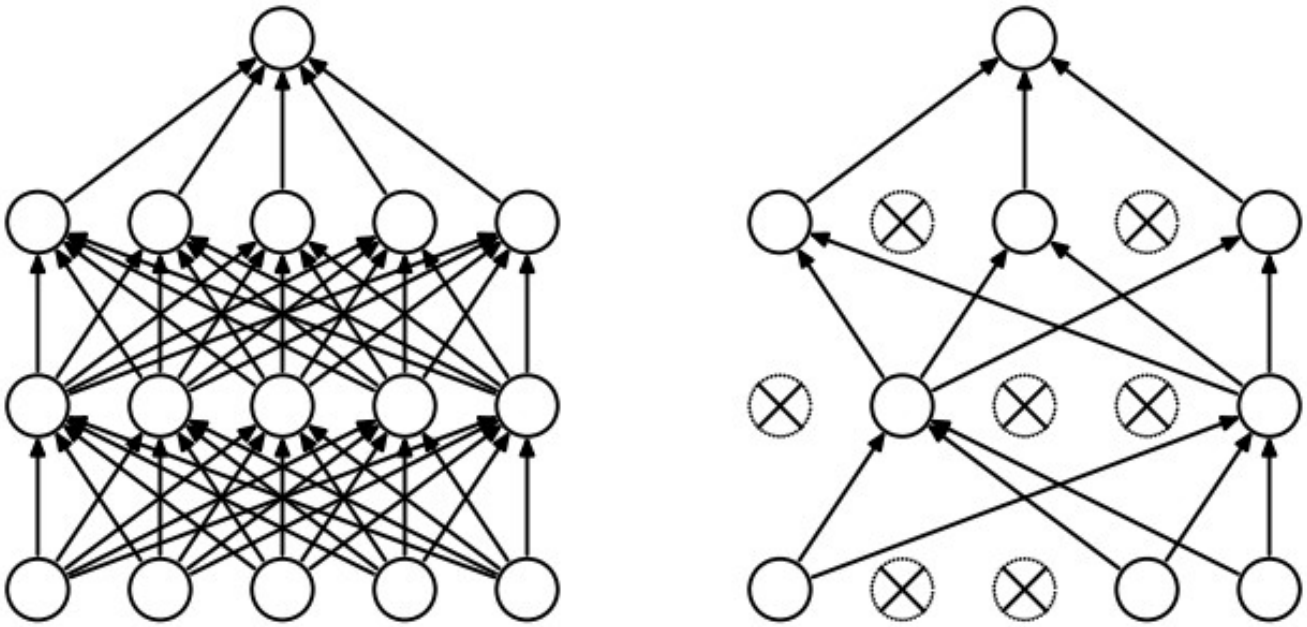
$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_1$$

- Gradient is:

$$\nabla_{\mathbf{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \mathbf{sign}(\mathbf{W}^{(\ell)})$$

- Promotes sparsity of the weights

# Dropout Regularization (?)



(a) Standard Neural Net (b) After applying dropout.

**Idea:** During training, remove some hidden units stochastically

# Dropout Regularization (?)

- Each hidden unit's output is set to 0 with probability $p$ (e.g. $p = 0.5$)
- This prevents hidden units to co-adapt to other units, forcing them to be more generally useful
- At test time, keep all units, but multiply their outputs by $1 - p$
- Shown to be a form of adaptive regularization (**?**)
- Note: many software packages implement another variant, inverted dropout, where at training time the output of the units that were not dropped is divided by $1 - p$ and requires no change at test time

# Implementation of Dropout

- This is usually implemented using random binary masks
- The hidden layer activations become (for $\ell = 1, \ldots, L$):

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

- Beats regular backpropagation on many datasets (**?**)
- Other variants, e.g. DropConnect (**?**), Stochastic Pooling (**?**)

# Backpropagation with Dropout

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** $\ell$ from $L+1$ to $1$ **do**

Compute gradients of hidden layer parameters:

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

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

Compute gradient of hidden layer below:

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

Compute gradient of hidden layer below (before activation):

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

**end for**

# Outline

# Initialization

Initialize all biases to zero

For weights:

- Cannot initialize to zero with **tanh** activation (the gradients would also be zero and we would reach a saddle point)
- Cannot initialize the weights to the same value (need to break the symmetry)
- Random initialization (Gaussian, uniform), sampling around 0 to break symmetry
- For ReLU activations, the mean should be a small positive number
- Variance cannot be too high, otherwise all neuron activations will be saturated

# "Glorot Initialization"

- Recipe from **?**:

$$\mathbf{W}_{i,j}^{(\ell)} \sim U[-t, t], \text{ with } t = \frac{\sqrt{6}}{\sqrt{K^{(\ell)} + K^{(\ell-1)}}}$$

- Works well in practice with **tanh** and sigmoid activations

# Training, Validation, and Test Sets

Split datasets in training, validation, and test partitions.

- Training set serves to train the model
- Validation set serves to tune hyperparameters (learning rate, number of hidden units, regularization coefficient, dropout probability, best epoch, etc.)
- Test set serves to estimate the generalization performance

# Hyperparameter Tuning: Grid Search, Random Search

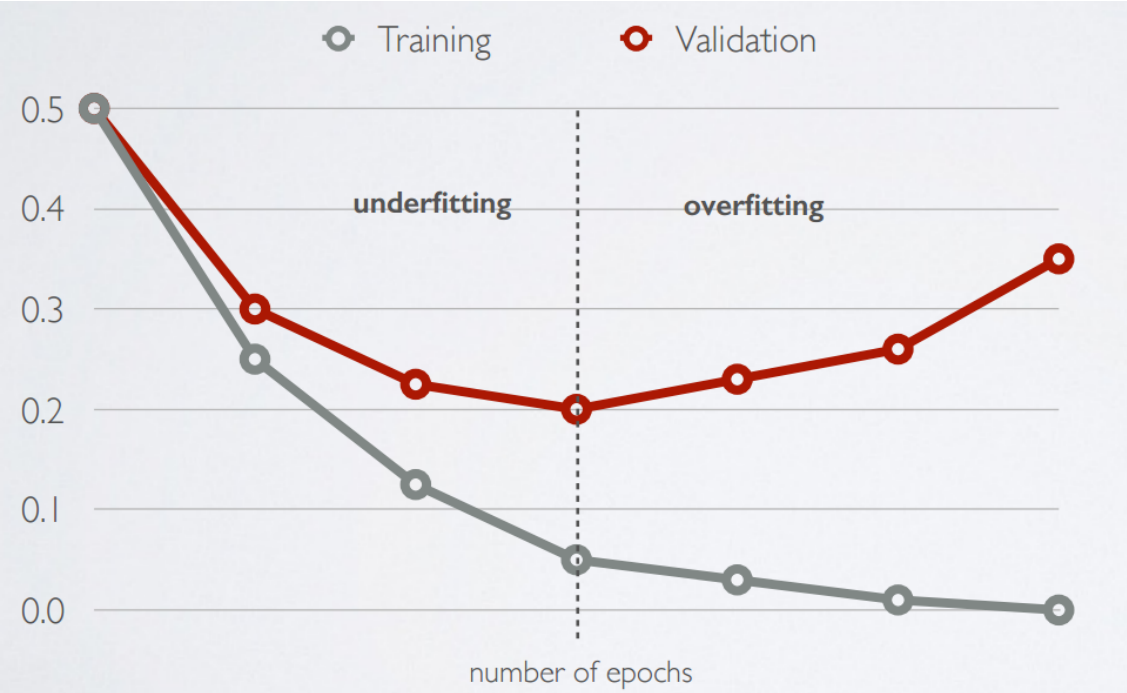
Search for the best configuration of the hyperparameters:

- Grid search: specify a set of values we want to test for each hyperparameter, and try all configurations of these values
- Random search: specify a distribution over the values of each hyper-parameter (e.g. uniform in some range) and sample independently each hyper-parameter to get configurations
- Bayesian optimization and learning to learn (**?**)

We can always go back and fine-tune the grid/distributions if necessary

# Early Stopping

- To select the number of epochs, stop training when validation error increases (with some look ahead)
- One common strategy (with SGD) is to halve the learning rate for every epoch where the validation error increases




(Image credit: Hugo Larochelle)

# Tricks of the Trade

- Normalization of the data
- Decaying the learning rate
- Mini-batches
- Adaptive learning rates
- Gradient checking
- Debugging on a small dataset

# Normalization of the Data

- For each input dimension: subtract the training set mean and divide by the training set standard deviation
- This makes each input dimension have zero mean, unit variance
- This can speed up training (in number of epochs)
- Doesn't work for sparse inputs (destroys sparsity)

# Decaying the Learning Rate

In SGD, as we get closer to a local minimum, it makes sense to take smaller update steps (to avoid diverging)

- Start with a large learning rate (say 0.1)
- Keep it fixed while validation error keeps improving
- Divide by 2 and go back to the previous step

# Mini-Batches

- Instead of updating after a single example, can aggregate a mini-batch of examples (e.g. 50–200 examples) and compute the averaged gradient for the entire mini-batch
- Less noisy than vanilla SGD
- Can leverage matrix-matrix computations (or tensor computations)
- Large computational speed-ups in GPUs (since computation is trivially parallelizable accross the mini-batch and we can exhaust the GPU memory)

# Adaptive Learning Rates

Instead of using the same step size for all parameters, have one learning rate per parameter

- Adagrad (**?**): learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\eta^{(t)} = \eta^{(t-1)} + (\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y))^2, \qquad \bar{\nabla}_{\boldsymbol{\theta}}^{(t)} = \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y)}{\sqrt{\eta^{(t)} + \epsilon}}$$

- RMSprop (**?**): instead of cumulative sum, use exponential moving average

$$\eta^{(t)} = \beta\eta^{(t-1)} + (1-\beta)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y))^2, \qquad \bar{\nabla}_{\boldsymbol{\theta}}^{(t)} = \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y)}{\sqrt{\eta^{(t)} + \epsilon}}$$

- Adam (**?**): combine RMSProp with momentum

# Gradient Checking

- If the training loss is not decreasing even with a very small learning rate, there's likely a bug in the gradient computation
- To debug your implementation of `fprop`/`bprop`, compute the "numeric gradient," a finite difference approximation of the true gradient:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

# Debugging on a Small Dataset

- Extract a small subset of your training set (e.g. 50 examples)
- Monitor your training loss in this set
- You should be able to overfit in this small training set
- If not, see if some units are saturated from the very first iterations (if they are, reduce the initialization variance or properly normalize your inputs)
- If the training error is bouncing up and down, decrease the learning rate

# 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)
- Most current software packages represent a computation graph and implement automatic differentiation
- Dropout regularization is effective to avoid overfitting

# 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.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, volume 9, pages 249–256.

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

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

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

Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In *Proc. of International Conference on Learning Representations*.

Martins, A. F. T. and Astudillo, R. (2016). From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification. In *Proc. of the International Conference on Machine Learning*.

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.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

# References II


Tieleman, T. and Hinton, G. (2012). Rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2).

Wager, S., Wang, S., and Liang, P. S. (2013). Dropout training as adaptive regularization. In *Advances in neural information processing systems*, pages 351–359.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proc. of the International Conference on Machine Learning*, pages 1058–1066.

Zeiler, M. D. and Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*.