

Lecture 4: Neural Networks I

André Martins, Francisco Melo, Mário Figueiredo



Deep Learning Course, Winter 2023-2024

Today's Roadmap

Today's lecture is about **neural networks**:

- From the **perceptron** to the **multi-layer perceptron**
- Feed-forward neural networks
- Activation functions: sigmoid, tanh, relu, ...
- Activation maps: softmax, sparsemax, ...
- Optimization: non-convexity and local minima
- Universal approximation theorem
- Gradient **backpropagation**

Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

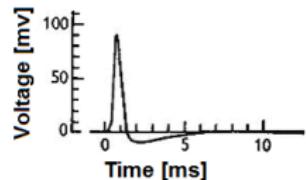
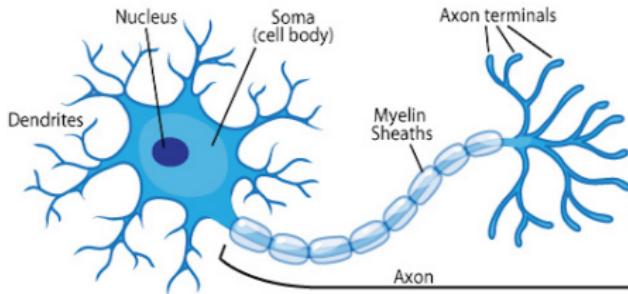
② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

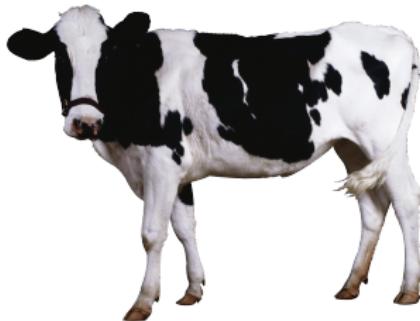
Biological Neuron



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

Word of Caution

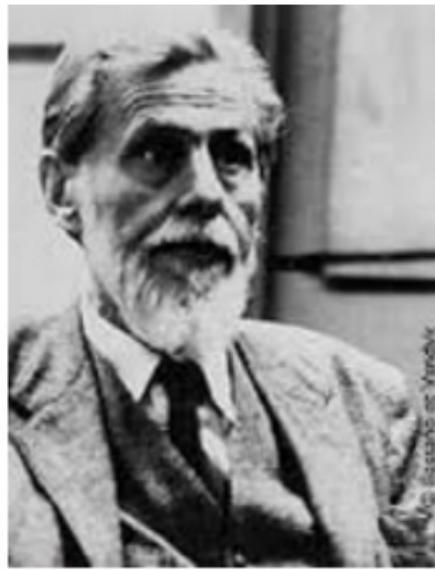
- Artificial neurons are inspired by biological neurons, but...



...beware the **spherical cow**

https://en.wikipedia.org/wiki/Spherical_cow

Warren McCulloch and Walter Pitts



- Earliest computational neuron model, via **threshold logic** (McCulloch and Pitts, 1943).

Artificial Neuron (McCulloch and Pitts, 1943)

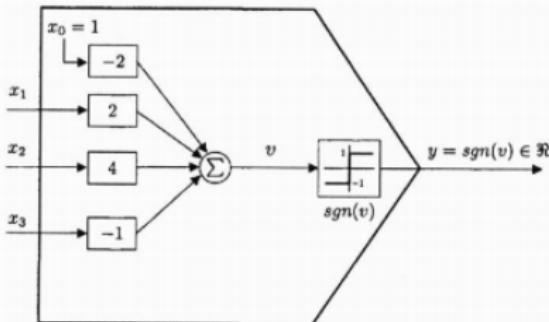


Figure 3.6 Example 3.2: a threshold neural logic for $y = x_2(x_1 + \bar{x}_3)$.

Table 3.6 Truth table for Example 3.2

Neural Inputs			$v = w_a^T x_a$ $= -2 + 2x_1 + 4x_2 - x_3$	$y = sgn(v)$ $= sgn(w_a^T x_a)$
-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):

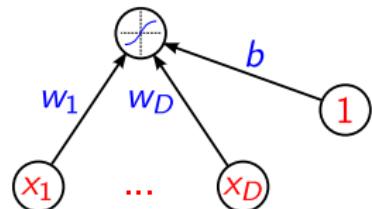
$$z(x) = w^T x + b = \sum_{i=1}^D w_i x_i + b,$$

where w are the **connection weights** and b is a **bias term**.

- **Activation:**

$$h(x) = g(z(x)) = g(w^T x + b),$$

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



Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

Activation Function

Typical choices:

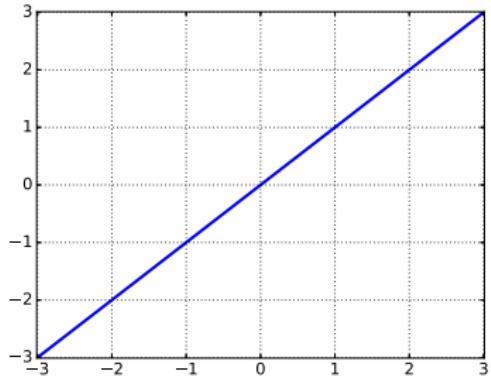
- Linear
- Sigmoid (logistic function)
- Hyperbolic Tangent
- Rectified Linear

Later:

- Softmax
- Sparsemax
- Max-pooling

Linear Activation

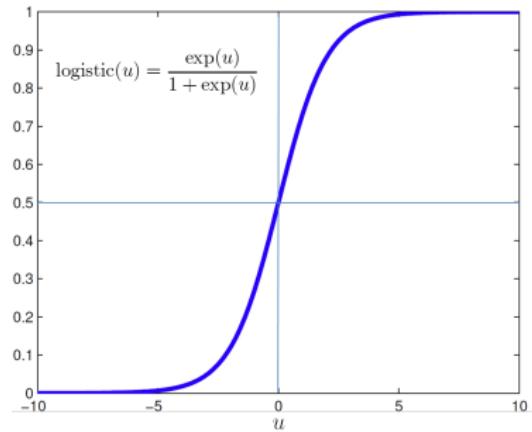
$$g(z) = z$$



- 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

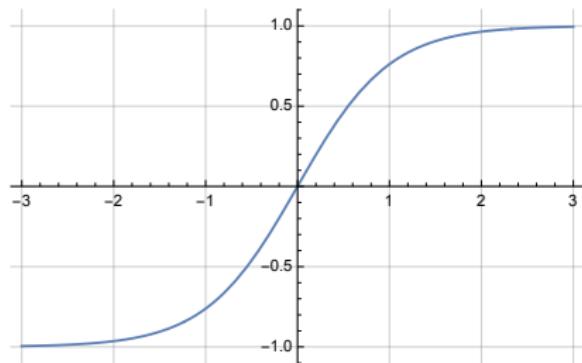
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



- “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

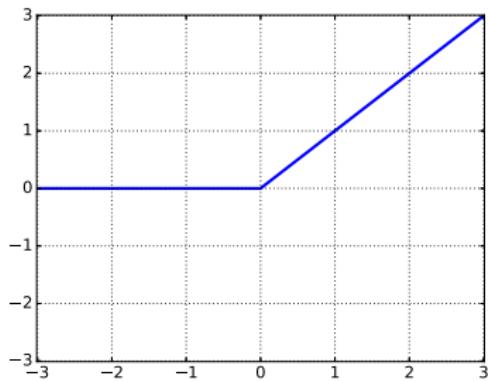
$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



- “Squashes” the neuron pre-activation between -1 and 1
- Related to the sigmoid via $\sigma(z) = \frac{1+\tanh(z/2)}{2}$
- Bounded, strictly increasing
- Combining layers of tanh units increases expressiveness (more later)

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

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



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

Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

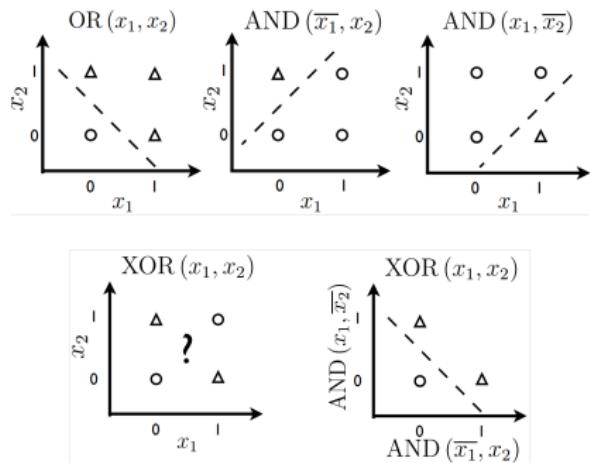
③ Conclusions

Capacity of Single Neuron (Linear Classifier)

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

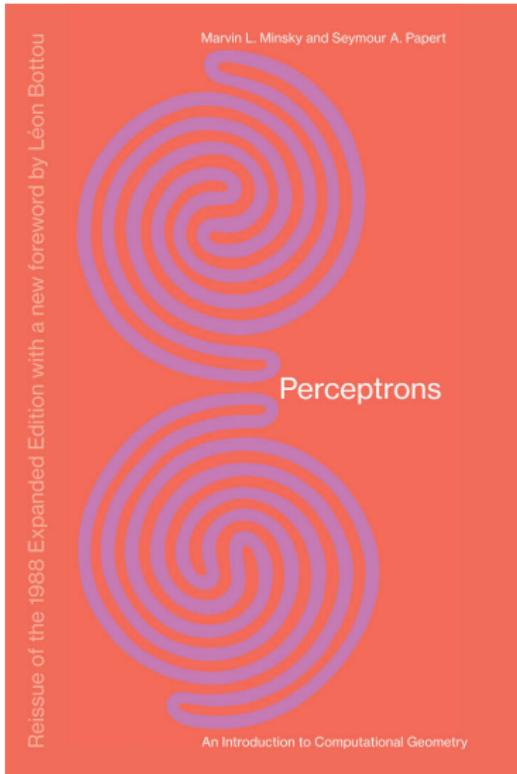
$$p(y = 1|x) = \sigma(w^T x + b).$$

- 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)

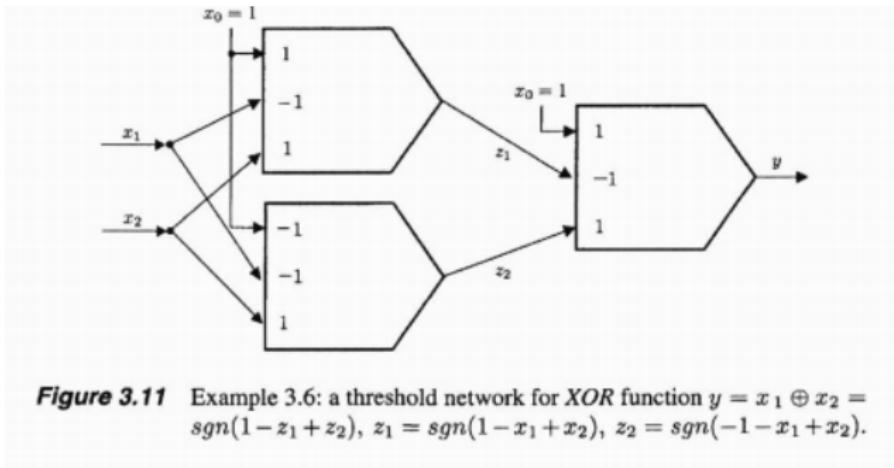
The XOR Affair



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



Solving XOR with Multi-Layer Perceptron

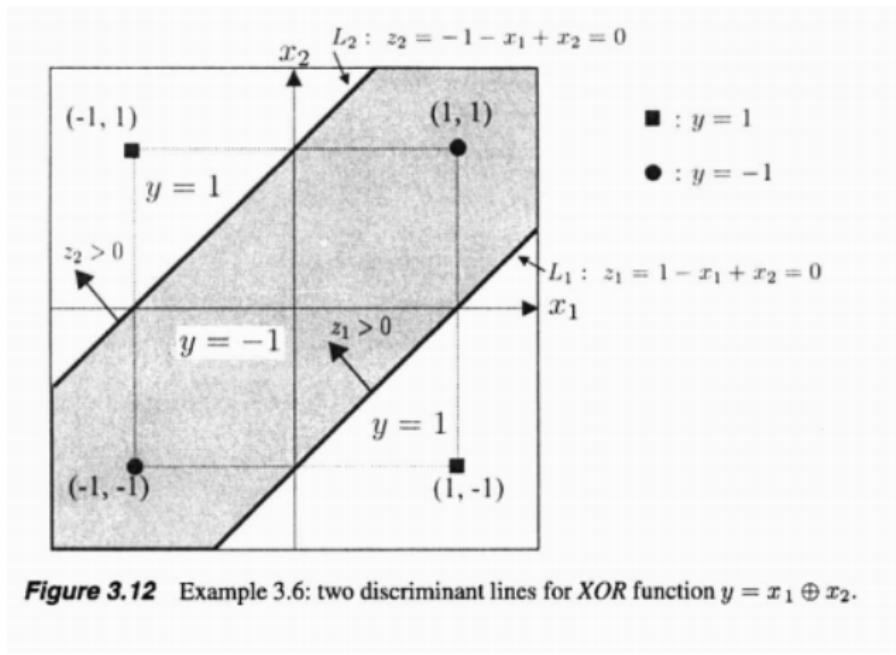


Figure 3.12 Example 3.6: two discriminant lines for XOR function $y = x_1 \oplus x_2$.

Solving XOR with Multi-Layer Perceptron

- This construction was known even by McCulloch and Pitts
- The result by Minsky and Papert (1969) is: to implement arbitrary logic functions, **all hidden units** needs to be connected to **all inputs**
- At the time, there was hope that only “local” neurons were needed

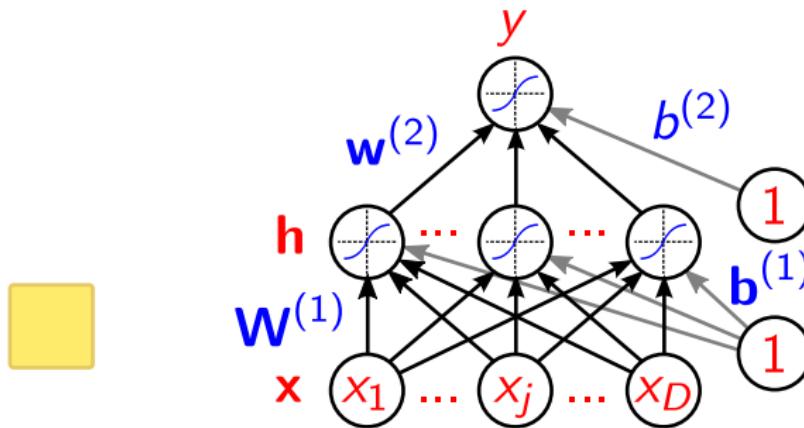
Multi-Layer Neural Network

- **Key idea:** use intermediate layers between the input and output layers.
- Each hidden layer computes a representation of the input and propagates it forward.
- This increases the expressive power of the network, yielding more complex, **non-linear**, functions/classifiers
- Also called **feed-forward neural network**

Single Hidden Layer

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

Include an intermediate layer of K hidden units ($\mathbf{h} \in \mathbb{R}^K$)



Single Hidden Layer

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

- **Hidden layer pre-activation:**

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

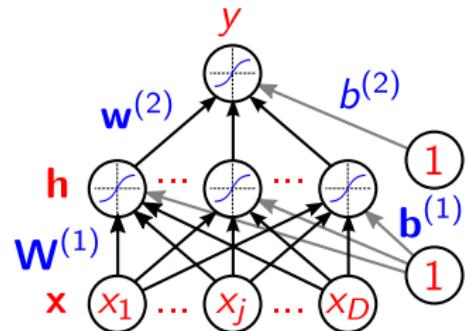
with $\mathbf{W}^{(1)} \in \mathbb{R}^{K \times D}$ and $\mathbf{b}^{(1)} \in \mathbb{R}^K$.

- **Hidden layer activation:**

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

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

- **Output layer activation:** $f(\mathbf{x}) = o(\mathbf{h}(\mathbf{x})^T \mathbf{w}^{(2)} + b^{(2)})$, where $\mathbf{w}^{(2)} \in \mathbb{R}^K$ and $o : \mathbb{R} \rightarrow \mathbb{R}$ if the output activation function.



Single Hidden Layer, Single Output

Overall,

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

Examples:

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

Non-linear in \mathbf{x} and non-linear in $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$

$\mathbf{h}(\mathbf{x})$ is an **internal representation** (not manually engineered features)

Multiple Classes

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

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

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

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:

$$\Delta_{C-1} = \{(p_1, \dots, p_C) : p_i \geq 0, i = 1, \dots, C, \sum_{i=1}^C p_i = 1\}$$

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

$$\mathbf{o}(\mathbf{z}) = \text{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_c \exp(z_c)}, \dots, \frac{\exp(z_C)}{\sum_c \exp(z_c)} \right]$$

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

Multiple ($L \geq 1$) Hidden Layers

- **Hidden layer pre-activation** (define

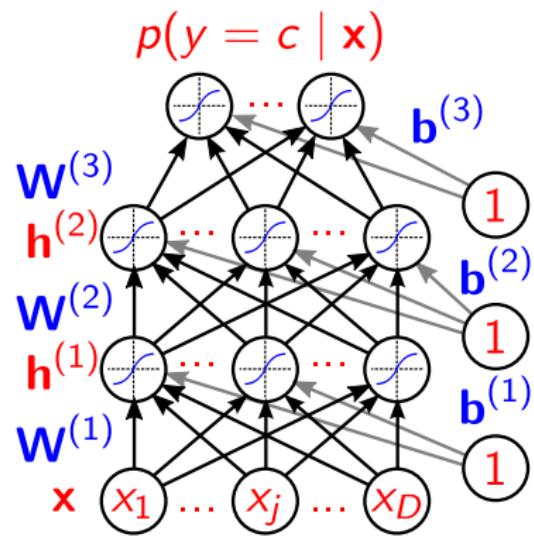
$x^{(0)} = x$ for convenience):

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

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

- **Hidden layer activation:**

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



- **Output layer activation:**

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

Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

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

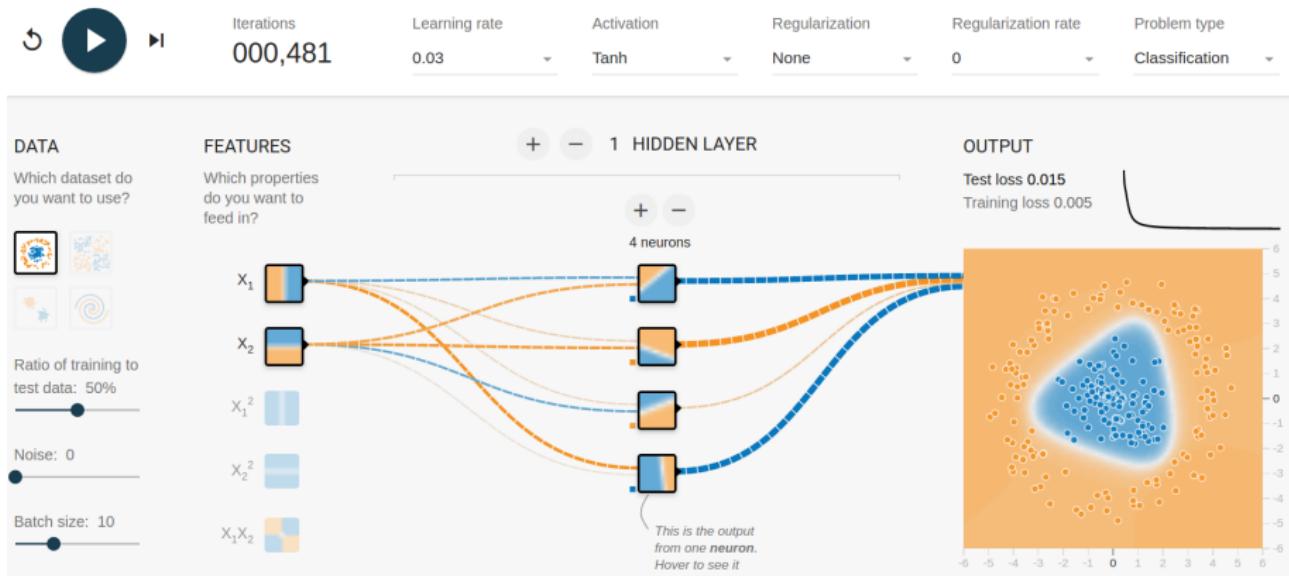
Theorem (Montufar et al. (2014))

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

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

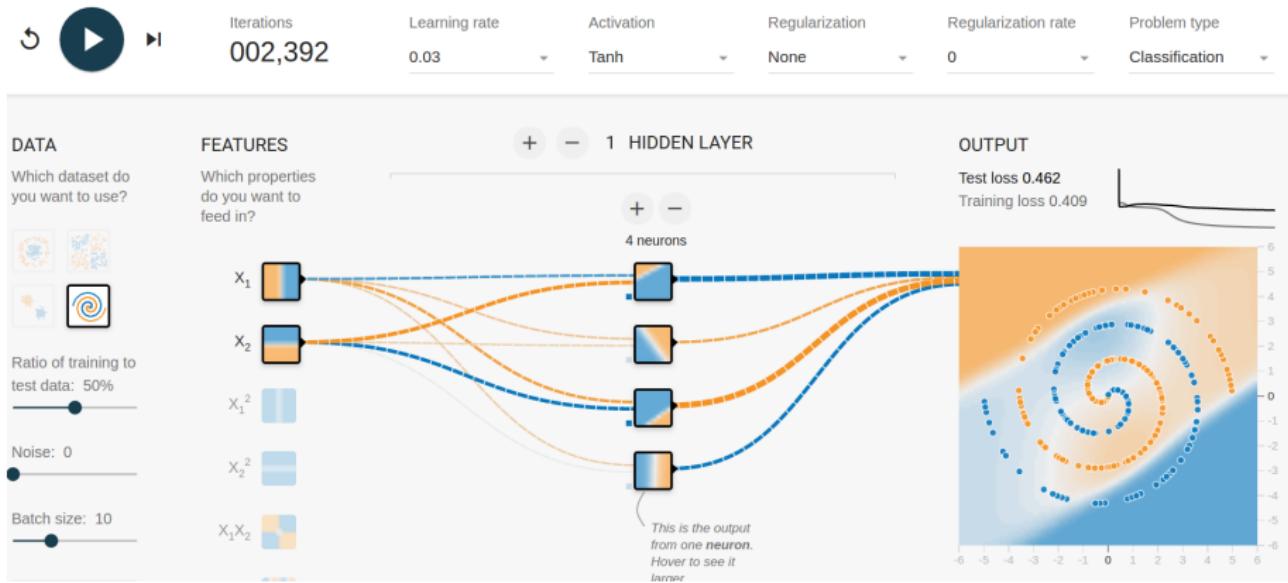
Therefore, for fixed K , deeper networks are exponentially more expressive

“Simple” Target Function, One Hidden Layer



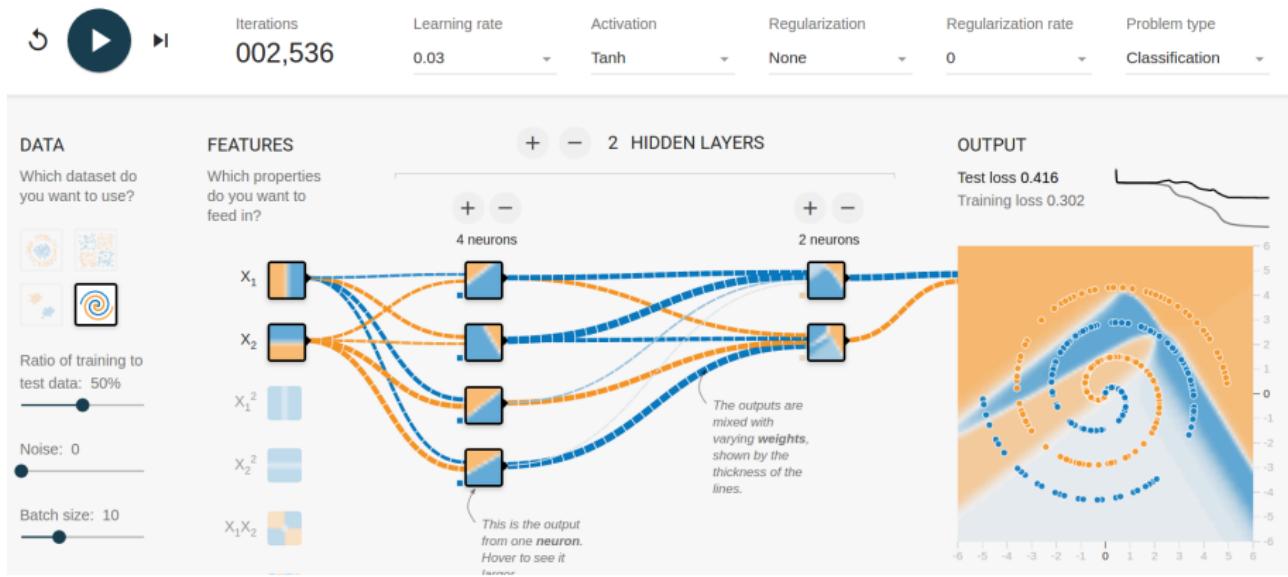
(<http://playground.tensorflow.org>)

Complex Target Function, One Hidden Layer



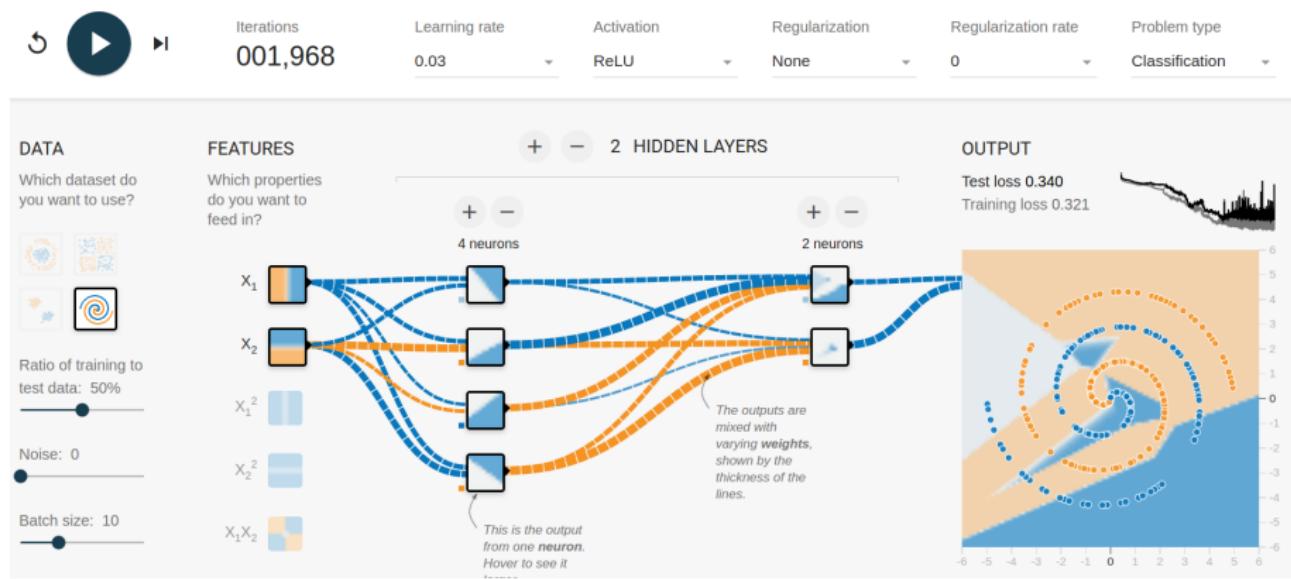
(<http://playground.tensorflow.org>)

Complex Target Function, Two Hidden Layers



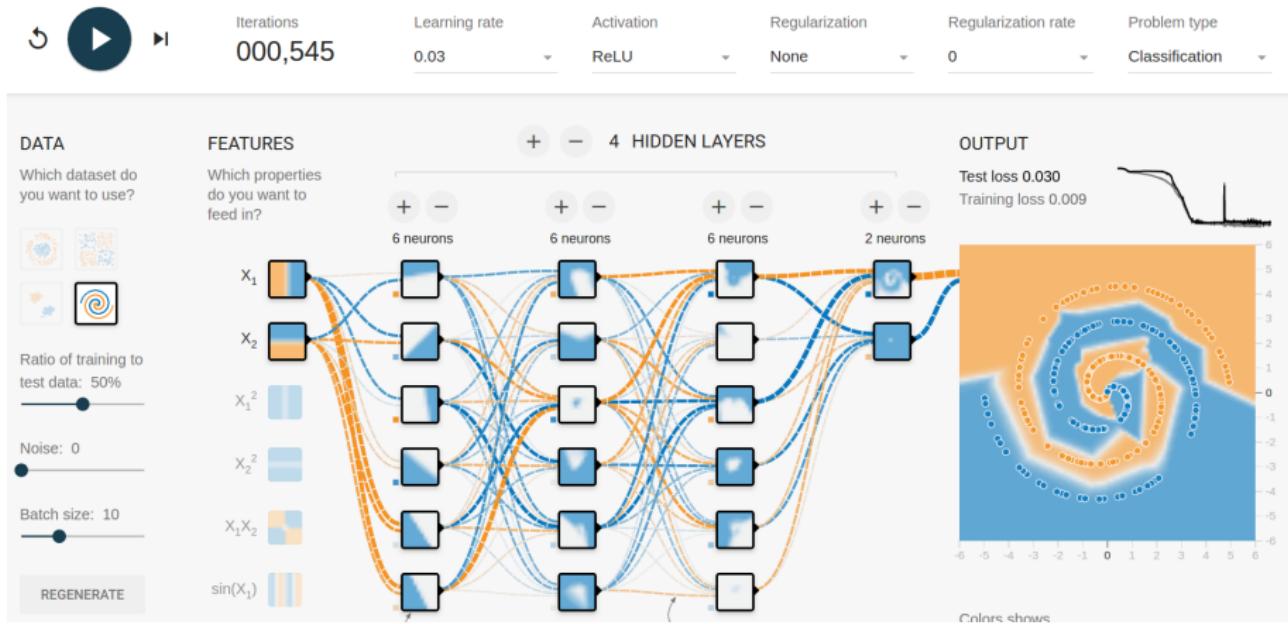
(<http://playground.tensorflow.org>)

Complex Target Function, Two Hidden Layers, ReLU



(<http://playground.tensorflow.org>)

Complex Target Function, Four Hidden Layers, ReLU



Capacity of Neural Networks

Neural networks are excellent function approximators!

The universal approximation theorem is an important result, but:

- We need a learning algorithm to find the parameter values
- ... and, to generalize, we need to control overfitting

Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

Training Neural Networks

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

But to do so, their **parameters**

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

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

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

Empirical Risk Minimization

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

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

- $\{(\mathbf{x}_i, y_i), i = 1, \dots, N\}$ is a training set
- $L(\mathbf{f}(\mathbf{x}_i; \theta), y_i)$ is a **loss function**
- $\Omega(\theta)$ is a **regularizer**
- λ is a **regularization constant** (**hyperparameter** to be tuned)

Recap: Gradient Descent

We can write the objective as:

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

The gradient is:

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

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**: $j \in \{1, \dots, N\}$

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

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

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

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

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

Stochastic Gradient Descent with Mini-Batches

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

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

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

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

The Key Ingredients of SGD

In sum, SGD needs the following ingredients:

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

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

Loss Function

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

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

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 aims at estimating $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta}) \approx \mathbf{y}$
- Minimize the **mean squared error**:

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

- Loss gradient:

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

Negative Log-Likelihood (Cross-Entropy)

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

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

where \mathbf{z} is the **output pre-activation**.

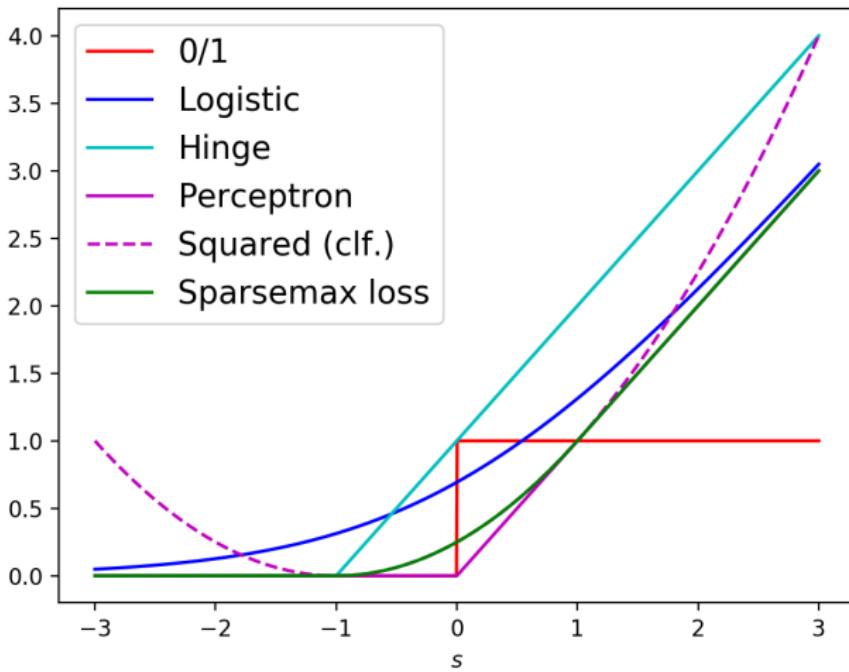
- Loss gradient at output pre-activation:

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

where $1_{(y=c)} = 1$, if $y = c$, and 0 otherwise.

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

Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

Gradient Computation

- Recall that we need to compute

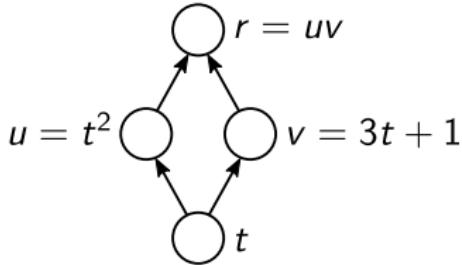
$$\nabla_{\theta} L(f(\mathbf{x}_i; \theta), y_i)$$

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

- This will be done with the **gradient backpropagation algorithm**
- Key idea:** use the chain rule for derivatives!

$$h(x) = f(g(x)) \Rightarrow h'(x) = f'(g(x)) g'(x)$$

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 an output unit in a layer; $q_i(t)$ to the pre-activation in the layer above; and $r(t)$ to the loss function.

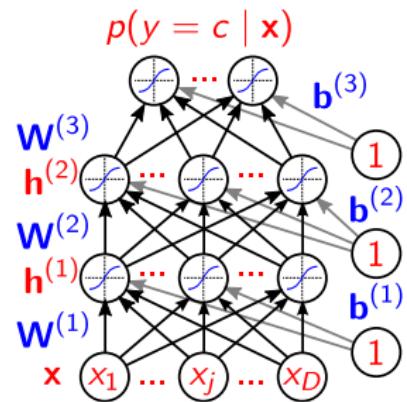
Hidden Layer Gradient

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

Hidden Layer Gradient

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

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

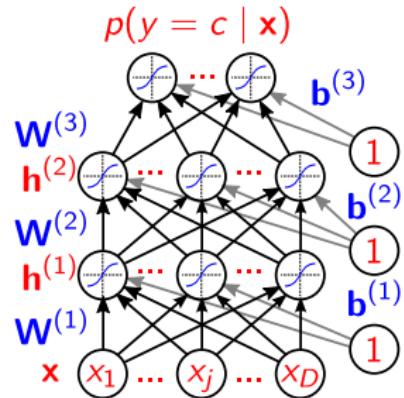


Hence $\nabla_{\mathbf{h}^{(\ell)}} L(\mathbf{f}(\mathbf{x}; \theta), y) = \mathbf{W}^{(\ell+1)^\top} \nabla_{\mathbf{z}^{(\ell+1)}} L(\mathbf{f}(\mathbf{x}; \theta), y)$.

Hidden Layer Gradient (Before Activation)

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

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



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

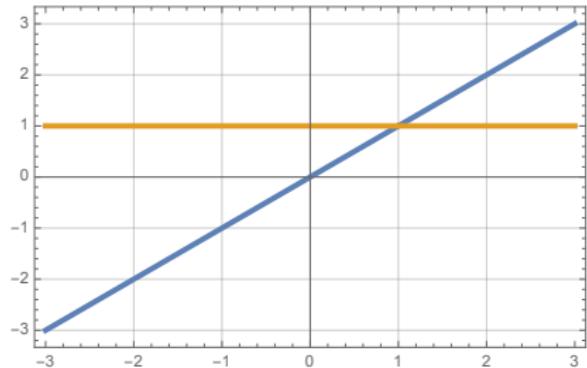
How to compute the derivative of the activation function $\mathbf{g}'(\mathbf{z}^{(\ell)})$?

Linear Activation

$$g(z) = z$$

Derivative:

$$g'(z) = 1$$

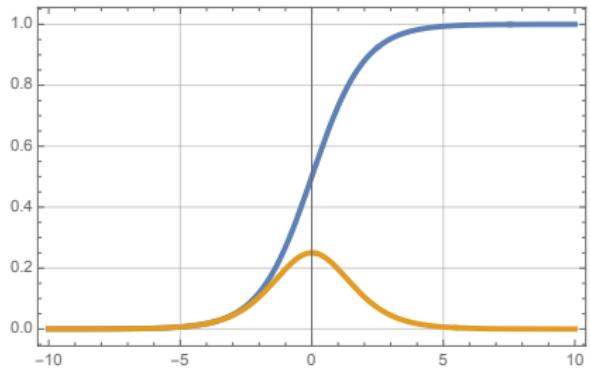


Sigmoid Activation

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

Derivative:

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

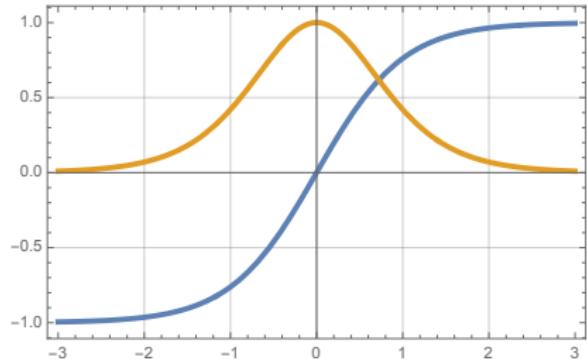


Hyperbolic Tangent Activation

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

Derivative:

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

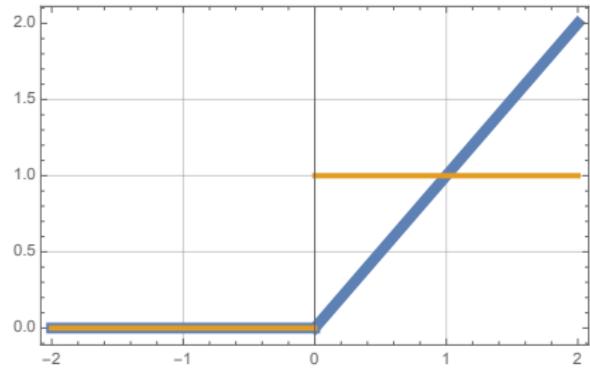


Rectified Linear Unit Activation (Glorot et al., 2011)

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

Derivative (except for $z = 0$):

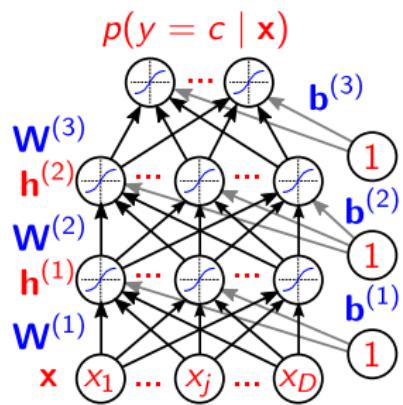
$$g'(z) = 1_{z>0}$$



Parameter Gradient

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

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



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

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

Backpropagation

Compute output gradient (before activation):

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

for ℓ from $L + 1$ to 1 **do**

 Compute gradients of hidden layer parameters:

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

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

 Compute gradient of hidden layer below:

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

 Compute gradient of hidden layer below (before activation):

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

end for

Outline

① Neural Networks

Definition

Activations

Feed-forward Neural Networks

Universal Approximation

② Training Neural Networks

Empirical Risk Minimization

Gradient Backpropagation

③ Conclusions

Conclusions

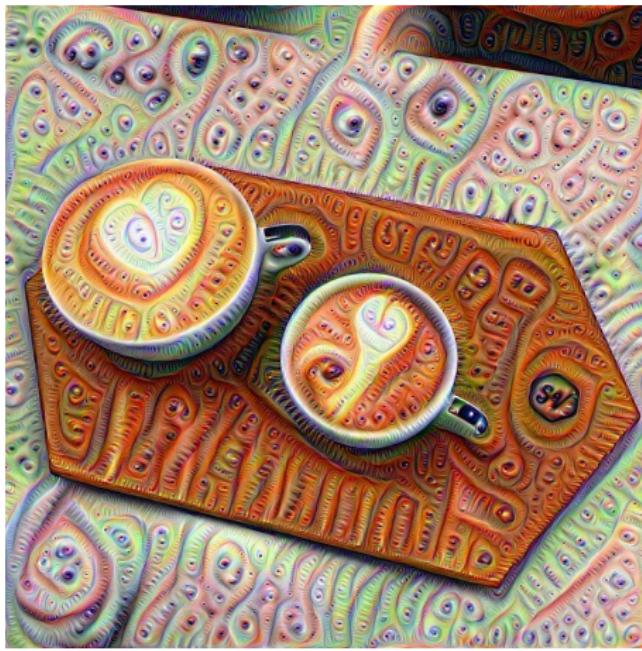
- Multi-layer perceptrons are universal function approximators
- However, they need to be trained
- Stochastic gradient descent is an effective training algorithm
- This is possible with the gradient backpropagation algorithm (an application of the chain rule of derivatives)

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.