BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Applied Deep Learning

Dr. Philippe Blaettchen
Bayes Business School (formerly Cass)

www.bayes.city.ac.uk

Learning objectives of today

Goals: Understand the key concepts underlying neural networks in general and feed-forward networks in particular

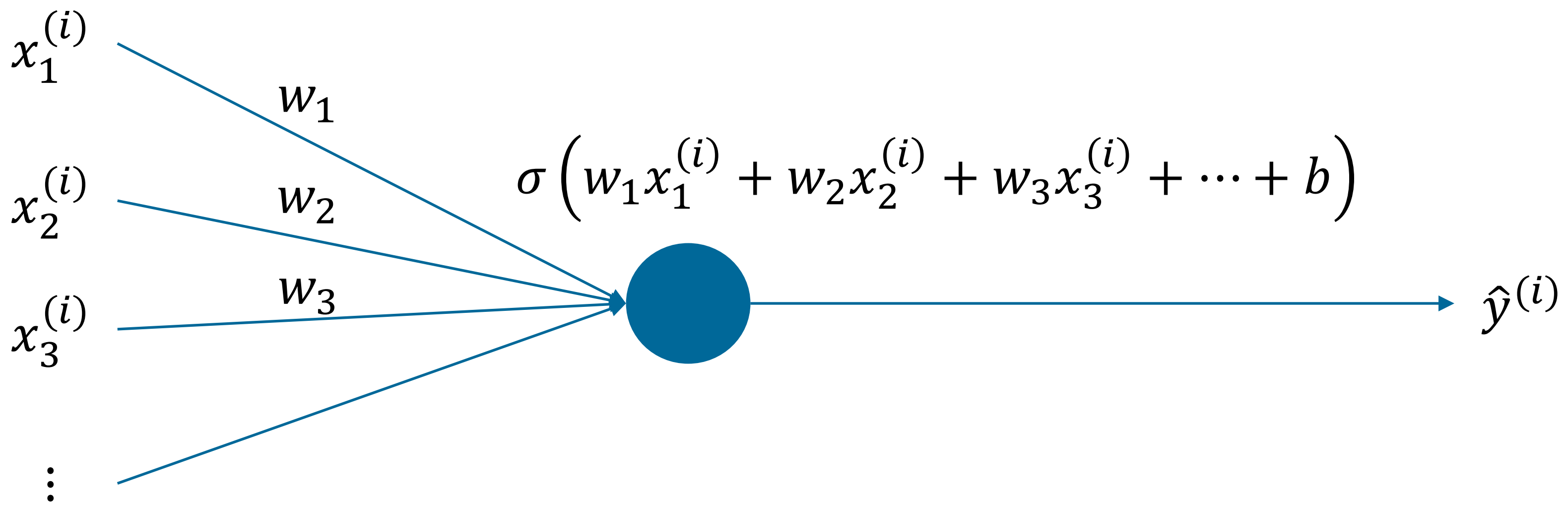
- The components of a neural network and how they interact
- Learning with neural networks using gradient descent
- Parameter initialization
- The reasons for deep (compared to shallow) learning

How will we do this?

- The discussion expands on what we have seen about logistic regression (a very simple neural network with one neuron)
- The components that have already come up are introduced more systematically
- Using the knowledge here, we can start working with arbitrarily complex feed-forward networks

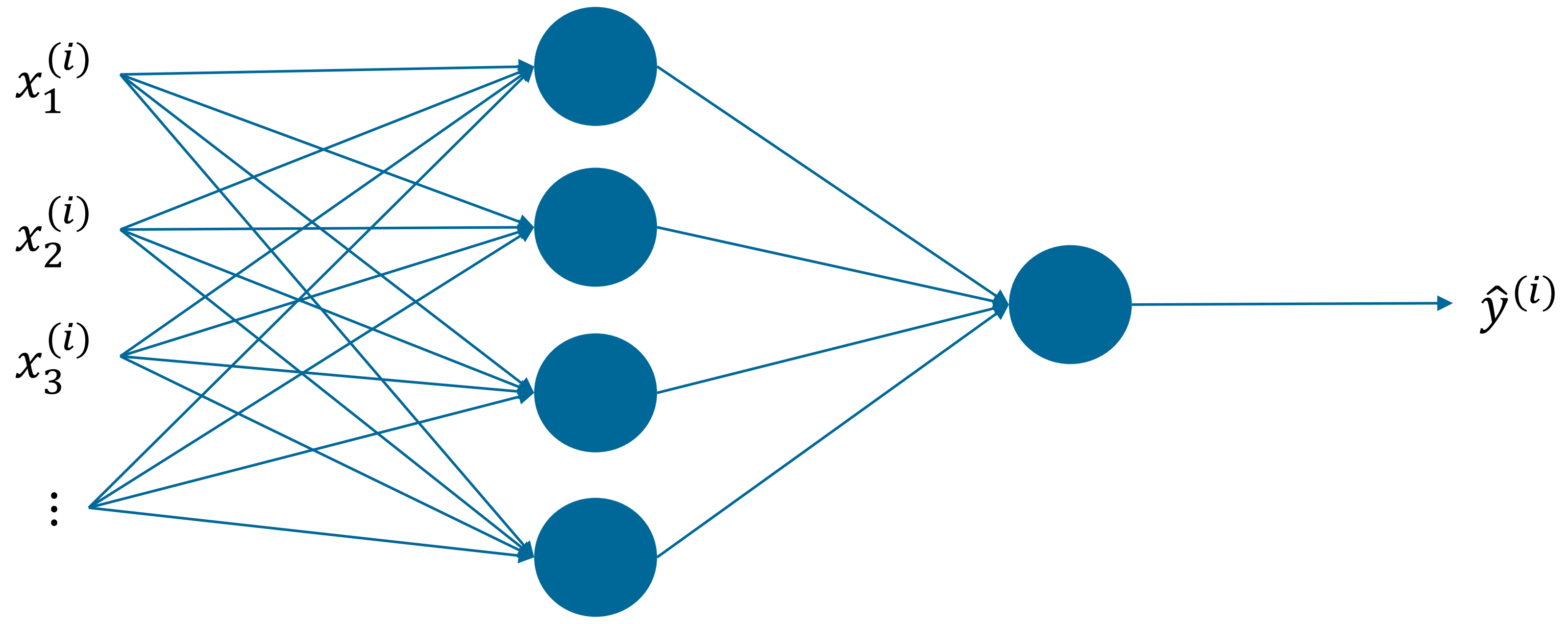
A neural network and its components

Schema of a logistic regression

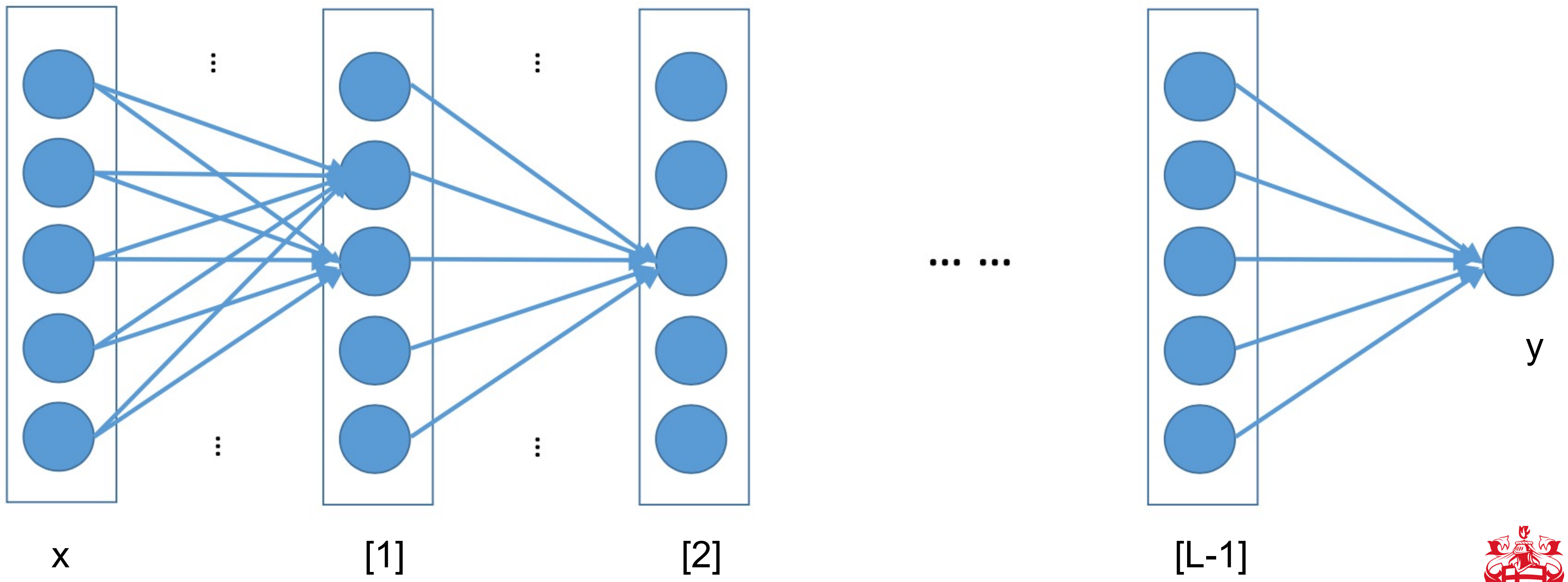

$x_1^{(i)}$
$x_2^{(i)}$
$x_3^{(i)}$
$\vdots$
w_1
w_2
w_3
$\sigma\left(w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)} + \cdots + b\right)$
$\hat{y}^{(i)}$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Putting multiple neurons together

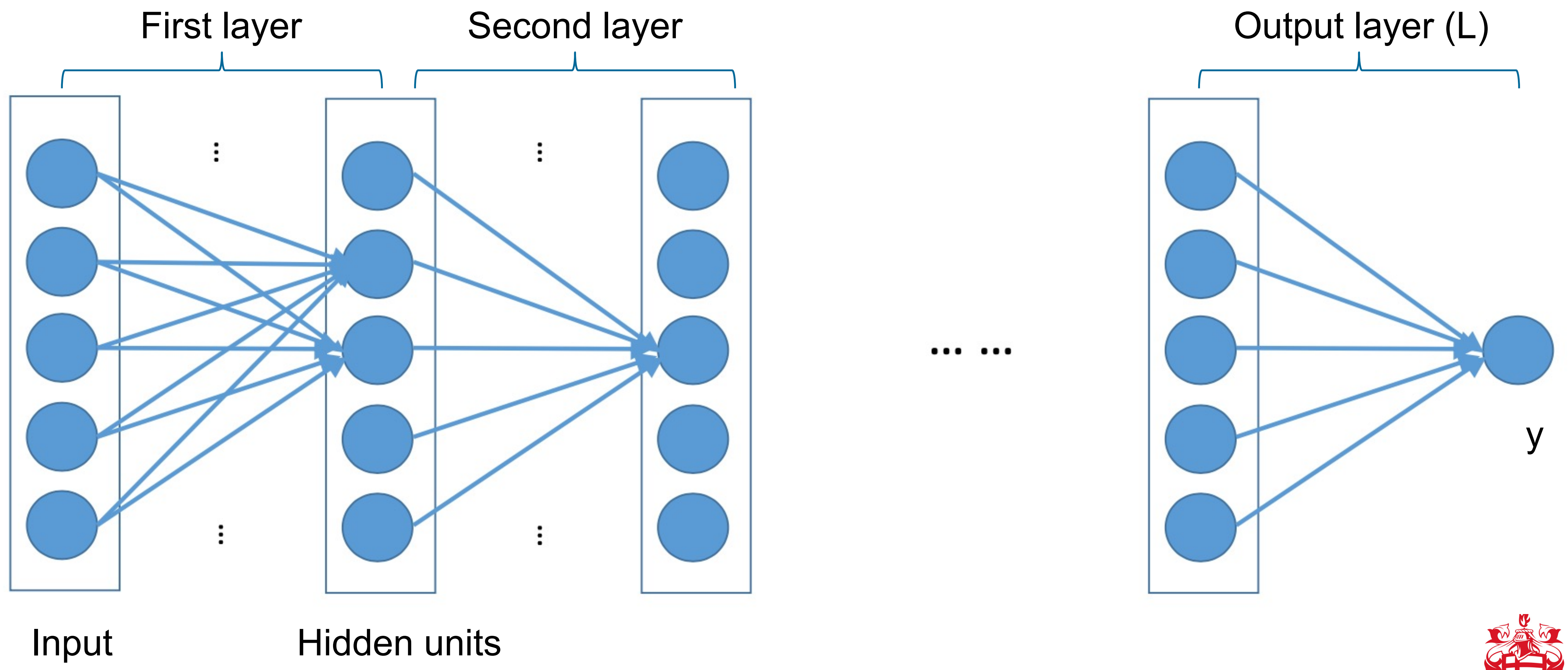


A (deep) neural network



Source: Liang

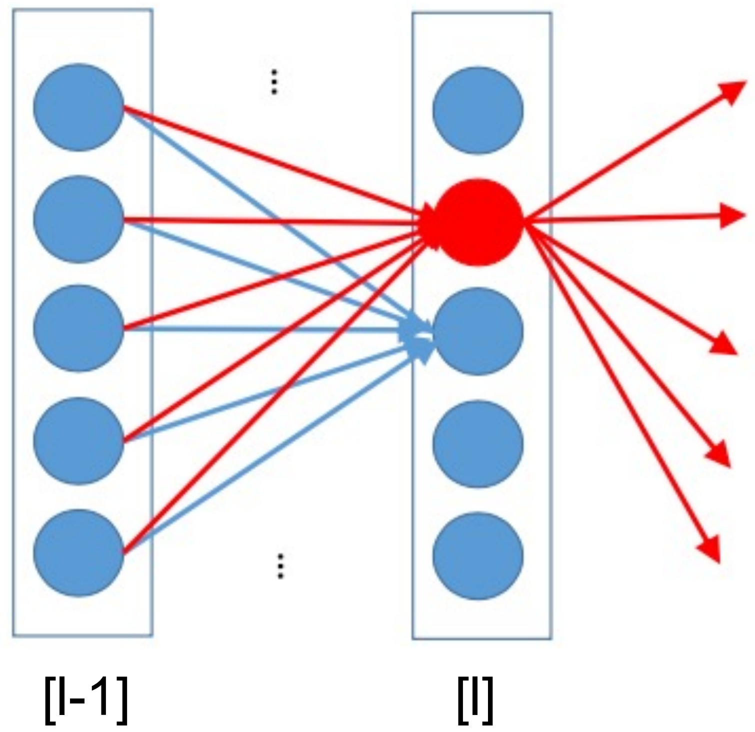
Components



Source: Liang

Hidden units

Hidden layers



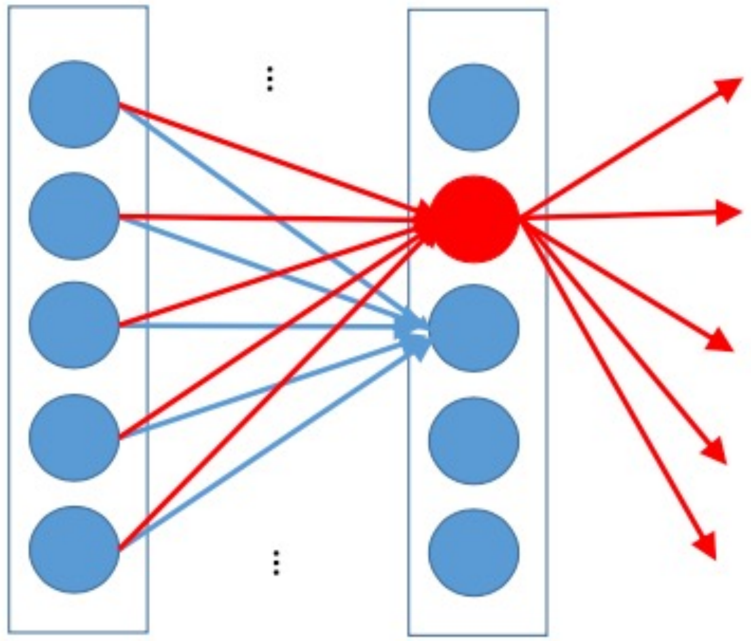
- Input to the neuron: weighted linear combination of the outputs from the previous layer
  - Previous layer / input x are essentially the same from the point of view of the neuron

- Output from the neuron: result of a non-linear function applied to the input

Source: Liang

Hidden layers

$$\text{“}\boldsymbol{x}\text{”} = \boldsymbol{a}^{[l-1]} \qquad z = \boldsymbol{a}^{[\boldsymbol{l-1}]}\boldsymbol{w} + \boldsymbol{b} \qquad f(\mathrm{z})$$



[l-1] [l]

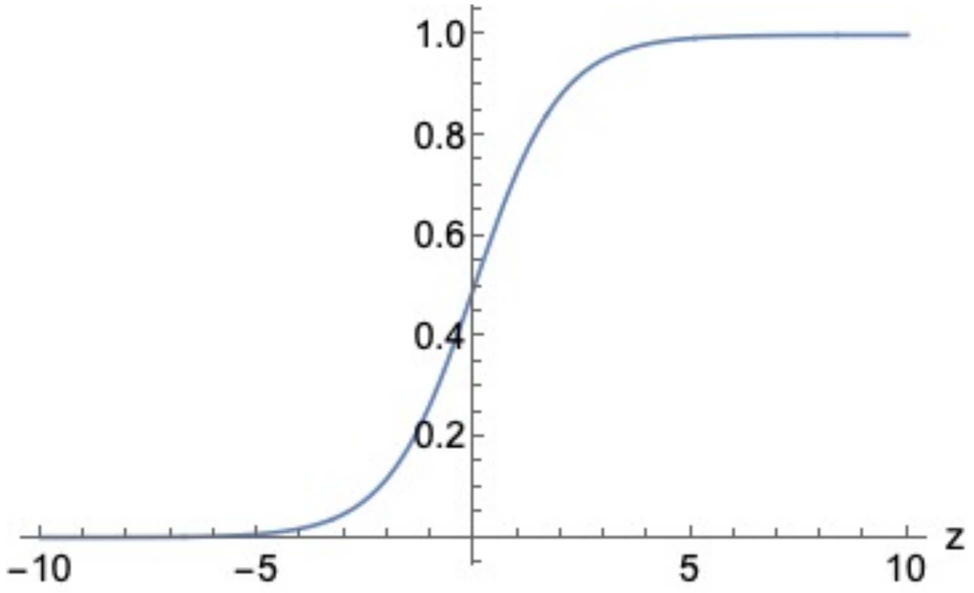
- f is what we call an “activation function”
- There are many activation functions, and new ones are invented all the time
- Many of these functions do just fine, or slightly better than existing ones
- To get a publication, your activation function must perform noticeably better than existing ones

Source: Liang

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

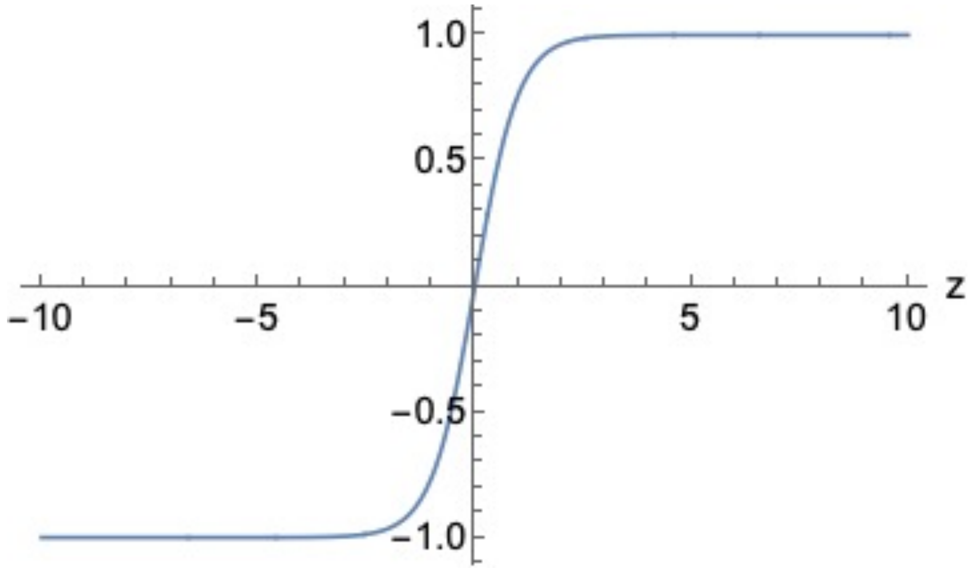
Different activation functions

Logistic (sigmoid) function



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

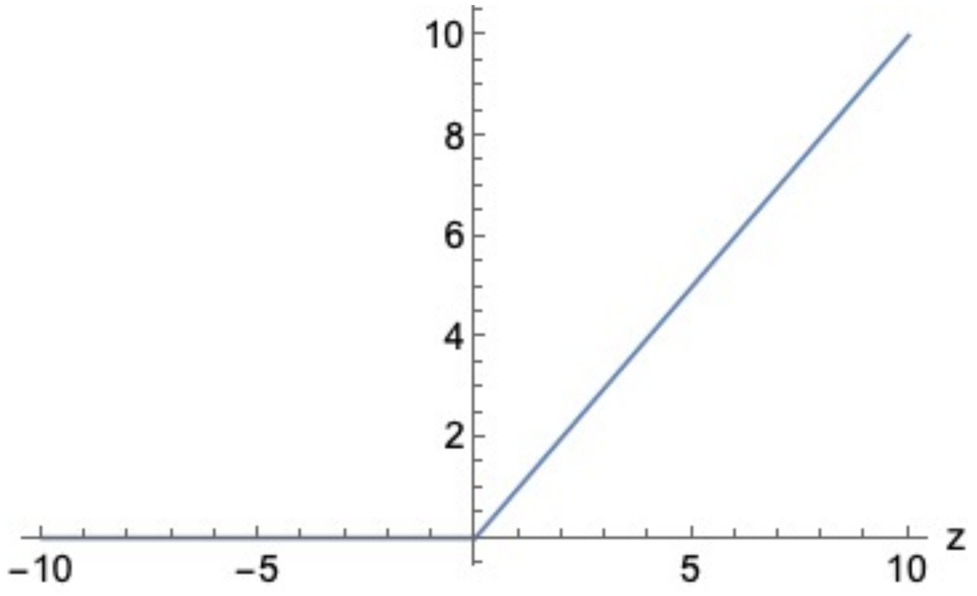
Hyperbolic tangent



$$f(z) = \tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

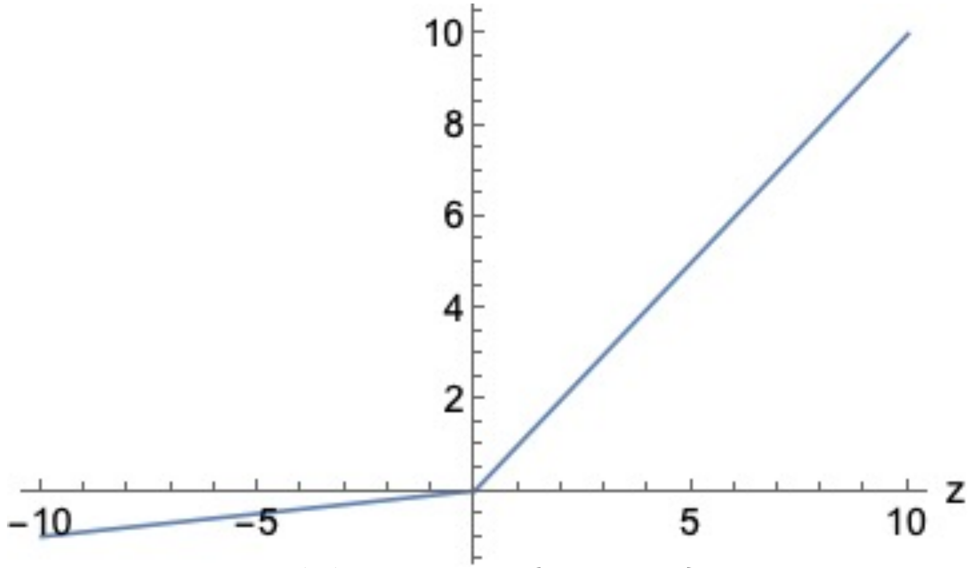
Different activation functions

Rectified Linear Unit (ReLU)



$$f(z) = \max\{0, z\}$$

Leaky ReLU



$$f(z) = \max\{0.1z, z\}$$

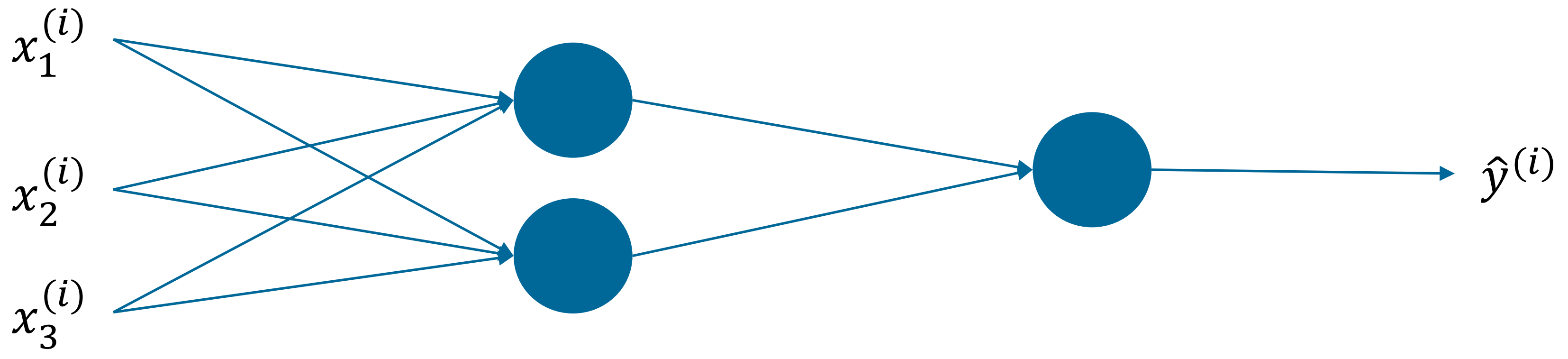
Related:

- Absolute ReLU
- Parametric ReLU

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Why non-linear activation functions?

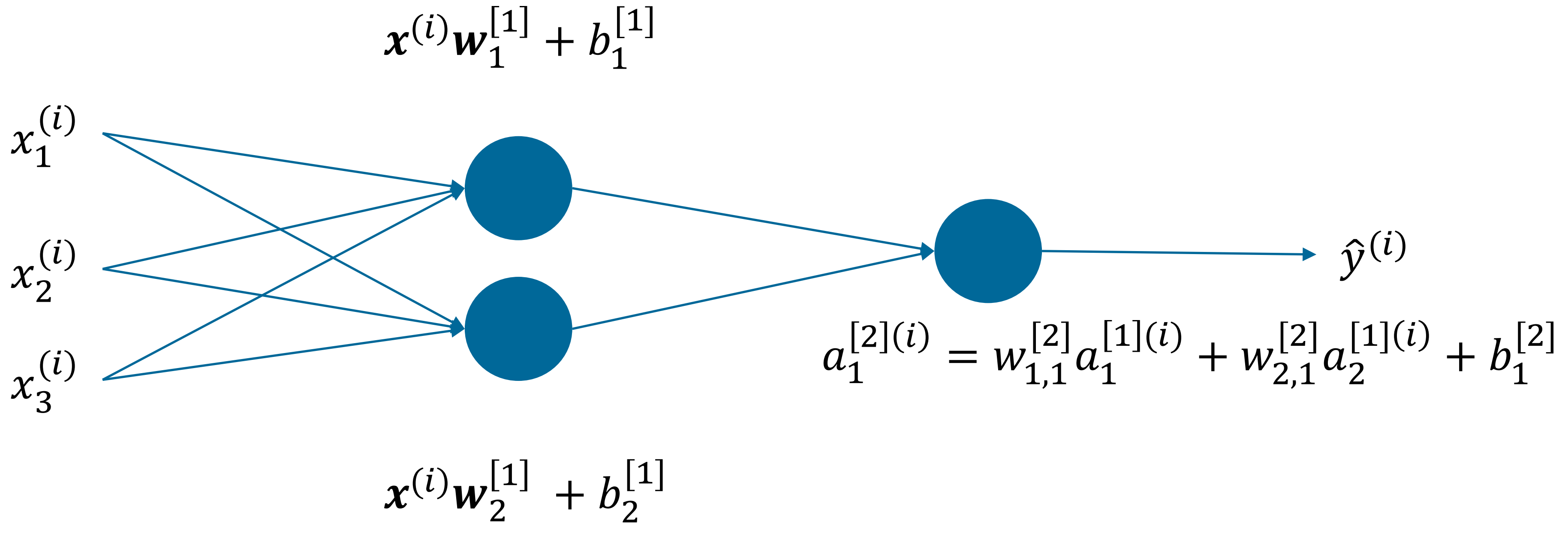
$$a_1^{[1](i)} = w_{1,1}^{[1]} x_1^{(i)} + w_{2,1}^{[1]} x_2^{(i)} + w_{3,1}^{[1]} x_3^{(i)} + b_1^{[1]} = \boldsymbol{x}^{(i)} \boldsymbol{w}_1^{[1]} + b_1^{[1]}$$



$$a_2^{[1](i)} = w_{1,2}^{[1]} x_1^{(i)} + w_{2,2}^{[1]} x_2^{(i)} + w_{3,2}^{[1]} x_3^{(i)} + b_2^{[1]} = \boldsymbol{x}^{(i)} \boldsymbol{w}_2^{[1]} + b_2^{[1]}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{\boldsymbol{a}^{[1](i)} = \boldsymbol{x}^{(i)} \boldsymbol{W}^{[1]} + \boldsymbol{b}^{[1]}}$$

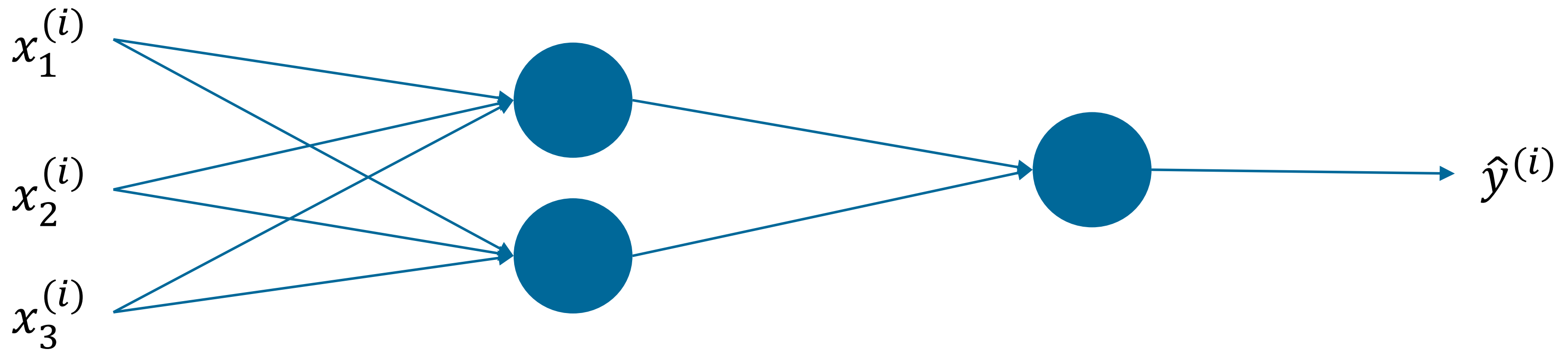
Why non-linear activation functions?


$\boldsymbol{x}^{(i)}\boldsymbol{w}_1^{[1]} + b_1^{[1]}$
$x_1^{(i)}$
$x_2^{(i)}$
$x_3^{(i)}$
$\hat{y}^{(i)}$
$a_1^{[2](i)} = w_{1,1}^{[2]}a_1^{[1](i)} + w_{2,1}^{[2]}a_2^{[1](i)} + b_1^{[2]}$
$\boldsymbol{x}^{(i)}\boldsymbol{w}_2^{[1]} + b_2^{[1]}$

$$\boldsymbol{a}^{[1](i)} = \boldsymbol{x}^{(i)}\boldsymbol{W}^{[1]} + \boldsymbol{b}^{[1]} \qquad \boldsymbol{a}^{[2](i)} = \boldsymbol{a}^{[1](i)}\boldsymbol{W}^{[2]} + \boldsymbol{b}^{[2]}$$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Why non-linear activation functions?



$$\boldsymbol{a}^{[1](i)} = \boldsymbol{x}^{(i)}\boldsymbol{W}^{[1]} + \boldsymbol{b}^{[1]} \qquad \boldsymbol{a}^{[2](i)} = \boldsymbol{a}^{[1](i)}\boldsymbol{W}^{[2]} + \boldsymbol{b}^{[2]}$$

But: $\boldsymbol{a}^{[2](i)} = \boldsymbol{x}^{(i)}\underbrace{\boldsymbol{W}^{[1]}\boldsymbol{W}^{[2]}}_{\widehat{\boldsymbol{W}}} + \underbrace{\boldsymbol{b}^{[1]}\boldsymbol{W}^{[2]} + \boldsymbol{b}^{[2]}}_{\widehat{\boldsymbol{b}}}$

Hence, $\boldsymbol{a}^{[2](i)} = \boldsymbol{x}^{(i)}\widehat{\boldsymbol{W}} + \widehat{\boldsymbol{b}}$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

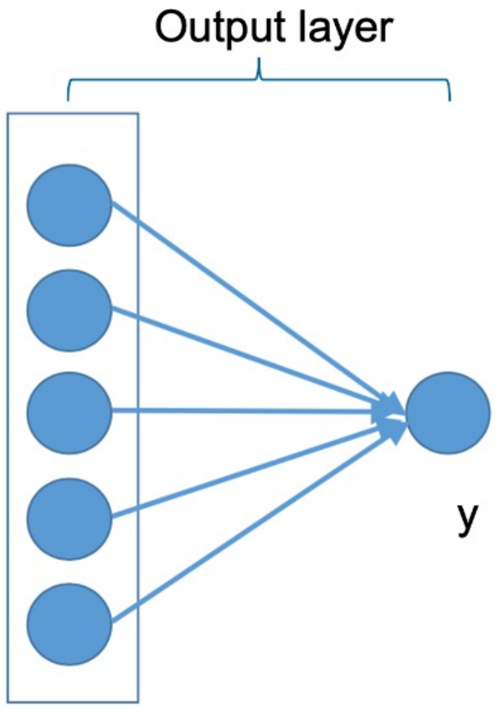
Some additional remarks on activation functions

- Use ReLUs (or their generalizations), unless there is a good reason not to
  - A good reason could be a specific application
- Many hidden units perform similarly to ReLUs
  - Make your life easier by sticking with a ReLU
- Sigmoids are rarely used for hidden layers anymore, but quite important for outputs

Output units

Binary classification

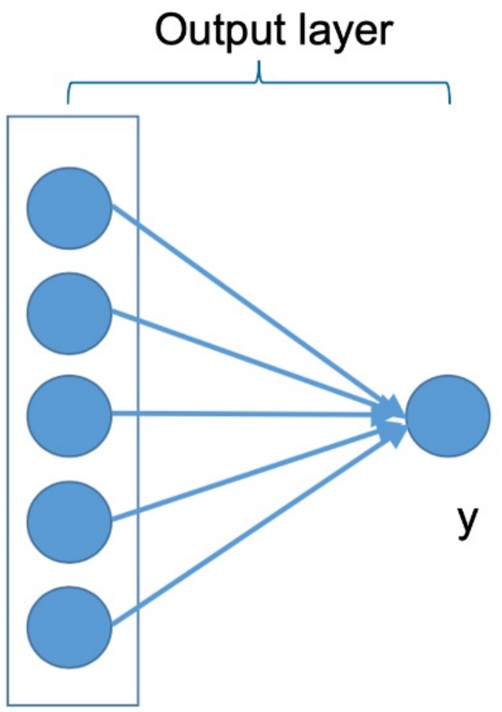
- Input: $\boldsymbol{a}^{[L-1](i)}$
- As usual, we make a linear transformation: $z^{[L](i)} = \boldsymbol{a}^{[L-1](i)}\boldsymbol{w}^{[L]} + b^{[L]}$
- We then use the logistic sigmoid function $\hat{y}^{(i)} = f\left(z^{[L](i)}\right) = \sigma(z^{[L](i)}) = \frac{1}{1+e^{-z^{[L](i)}}}$
- We can interpret the output as the probability of $y^{(i)} = 1$



Source: Liang

Regression

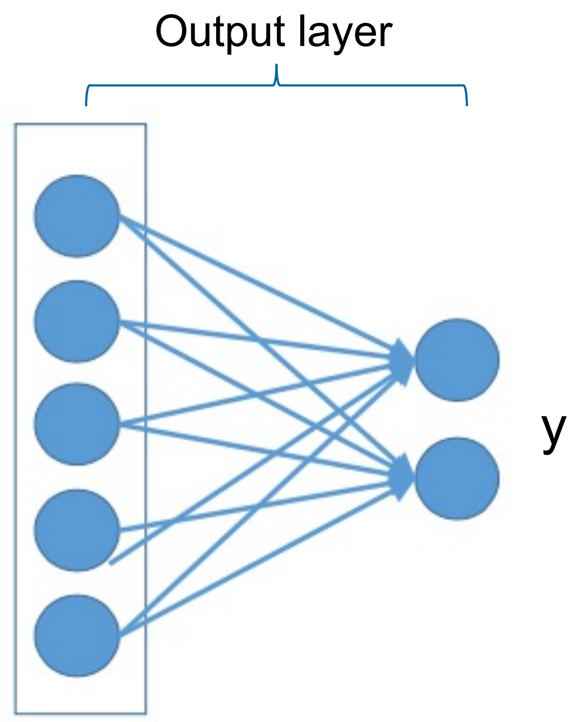
- Input: $\boldsymbol{a}^{[L-1](i)}$
- As usual, we make a linear transformation: $z^{[L](i)} = \boldsymbol{a}^{[L-1](i)}\boldsymbol{w}^{[L]} + b^{[L]}$
- We leave out the non-linearity
- We can interpret the output as our estimate, $\hat{y}^{(i)} = z^{[L](i)}$



Source: Liang

Multi-dimensional regression

- Same as a normal regression, just that we now have multiple output units:
  - $\boldsymbol{z}^{[L](i)} = \boldsymbol{a}^{[L-1](i)}\boldsymbol{W}^{[L]} + \boldsymbol{b}^{[L]}$
  - $\hat{\boldsymbol{y}}^{(i)} = \boldsymbol{z}^{[L](i)}$



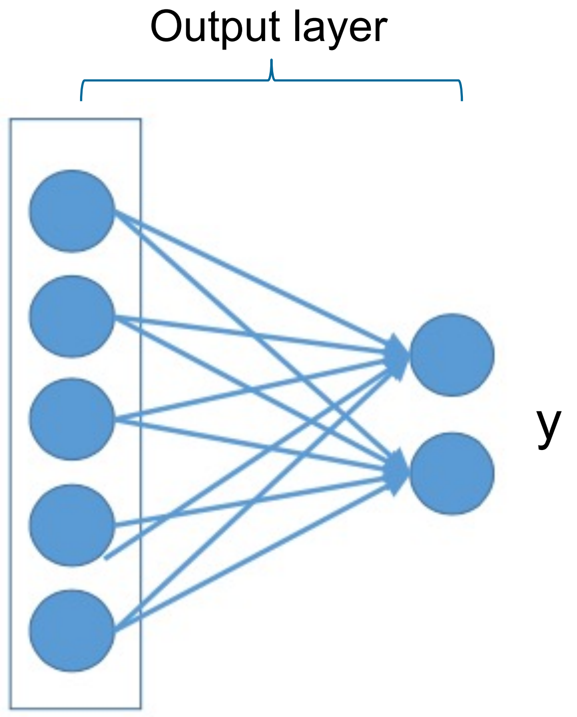
Source: Liang

Multi-class classification

- We again make a linear transformation with a matrix of weights: $\boldsymbol{z}^{[L](i)} = \boldsymbol{a}^{[L-1](i)}\boldsymbol{W}^{[L]} + \boldsymbol{b}^{[L]}$
- Note that $\boldsymbol{z}^{[L](i)} = \begin{pmatrix} z_1^{[L](i)} & z_2^{[L](i)} & \cdots & z_K^{[L](i)} \end{pmatrix}$
- We then use the softmax function on each of the outputs:

$$\hat{y}_k^{(i)} = f\left(\boldsymbol{z}^{[L](i)}\right) = \frac{e^{-z_k^{[L](i)}}}{\sum_{k=1}^{K} e^{-z_k^{[L](i)}}}$$

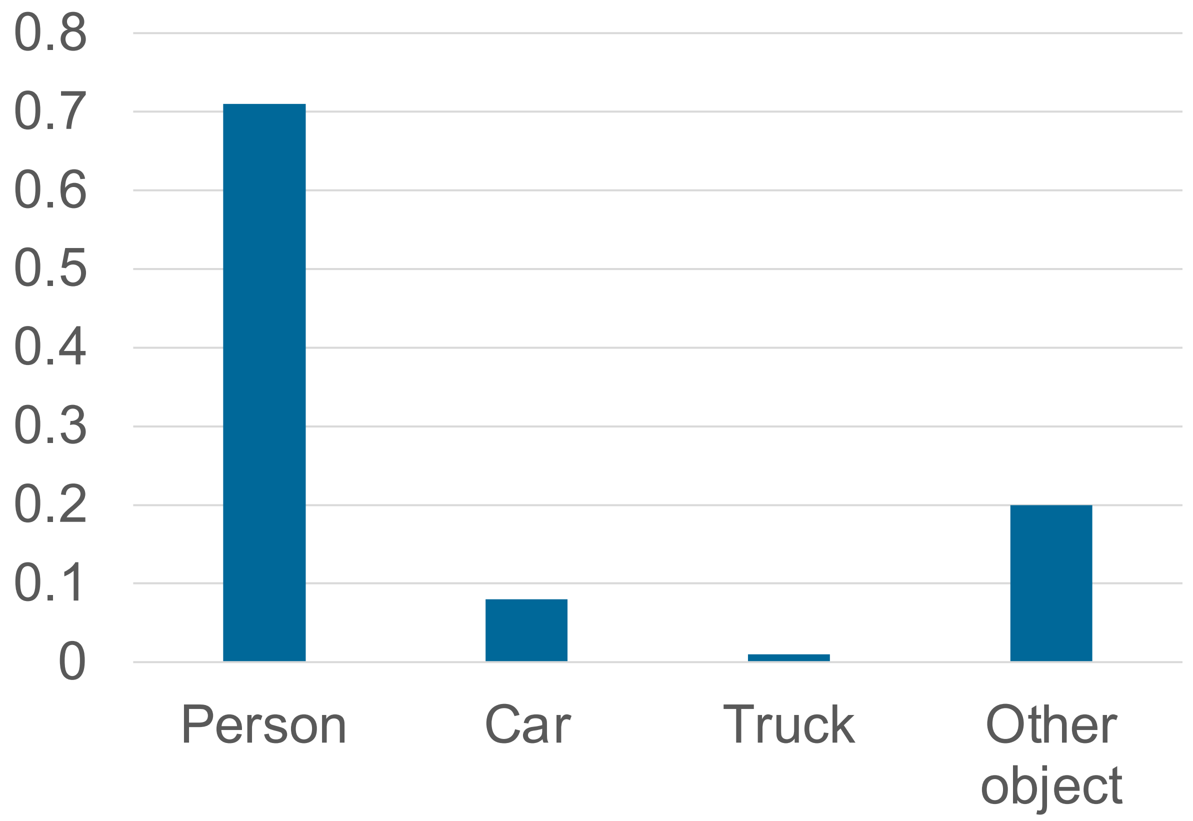
- This implies that $\hat{y}_k^{(i)} \in (0,1)$ and $\sum_{k=1}^{K} \hat{y}_k^{(i)} = 1$
- Hence, we can interpret $\hat{y}_k^{(i)}$ as the probability that $y^{(i)} = k$ ("belongs to class k")



Source: Liang

Softmax output

- E.g., when performing object recognition, we might represent our prediction $\hat{\boldsymbol{y}}^{(i)}$ as



Cost functions

Why a cost function?

- Given an input $\boldsymbol{x}^{(i)}$, our neural network spits out $\hat{y}^{(i)} = f\left(\boldsymbol{x}^{(i)}\right)$

- We “train” our neural network, so that $f\left(\boldsymbol{x}^{(i)}\right) \approx f^{*}\left(\boldsymbol{x}^{(i)}\right)$
  - For example, in supervised learning, $f^{*}\left(\boldsymbol{x}^{(i)}\right) = y^{(i)}$

- Training a model means running an optimization algorithm, where we aim to minimize the distance between $f\left(\boldsymbol{x}^{(i)}\right)$ and $f^{*}\left(\boldsymbol{x}^{(i)}\right)$. You can think of the distance between the desired output and the actual output as your cost

- Of course, we can define arbitrary cost functions (and sometimes specific applications require specific cost functions), but there are some that are typically used because they tend to work well in practice

Cross-entropy

- Recall from logistic regression:

$$J(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} = -\frac{1}{n}\sum_{i=1}^{n}\left[y^{(i)} \ln \hat{y}^{(i)} + \left(1 - y^{(i)}\right)\ln\left(1 - \hat{y}^{(i)}\right)\right]$$

- Generally, to learn parameters $\boldsymbol{\theta}$, we define the cross-entropy

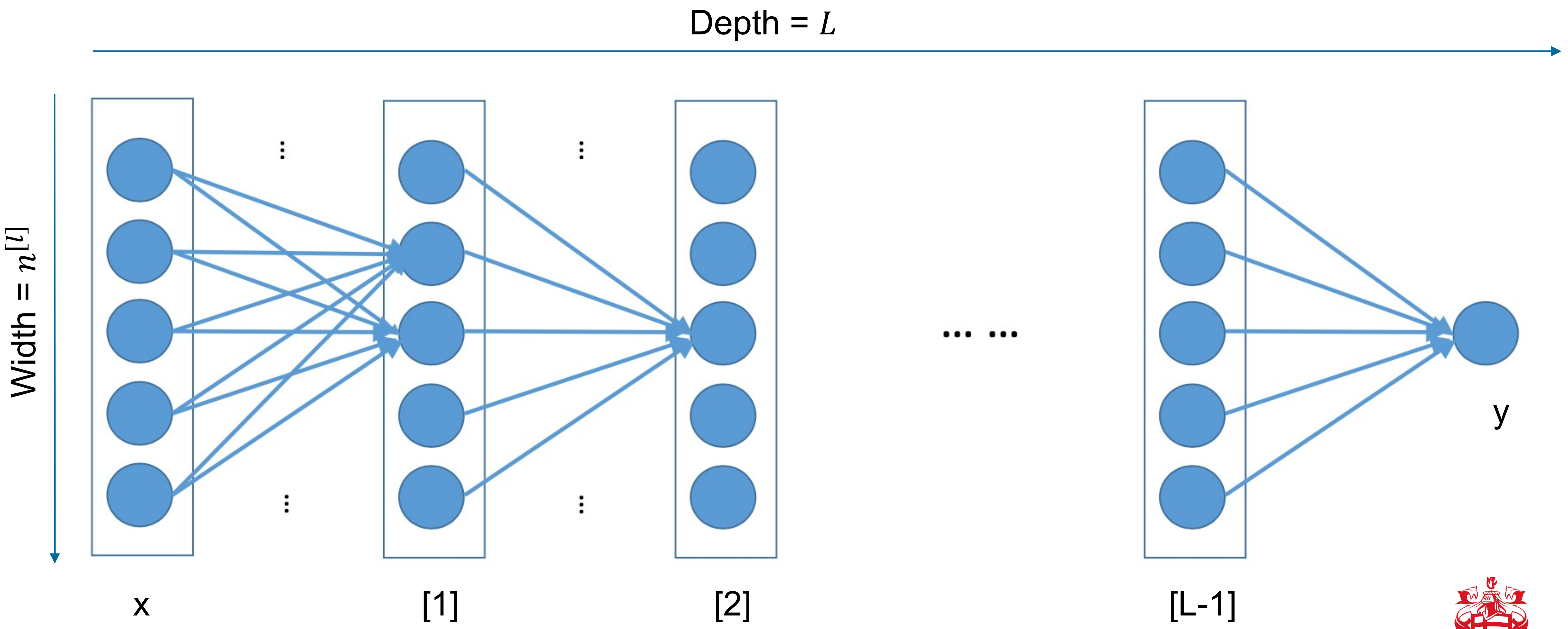
$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} = -\frac{1}{n}\sum_{i=1}^{n} \ln p_{\boldsymbol{\theta}}\left(y^{(i)} \middle| \boldsymbol{x}^{(i)}\right)$$

- Also known as “maximum likelihood estimator”

- Mean square error tends to perform poorly, especially when we have activation functions with e^z

Putting it together – architecture

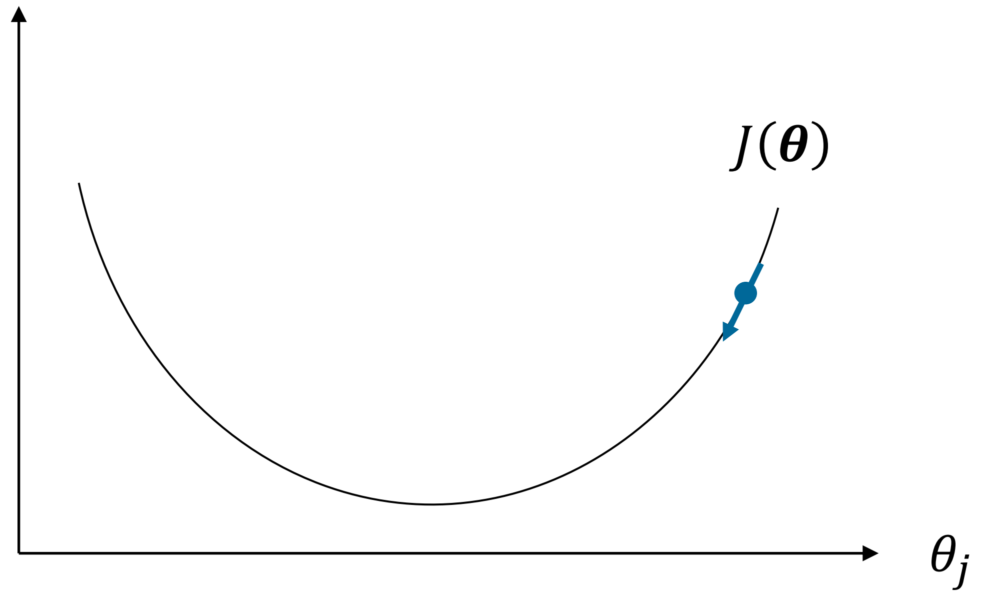
Depth and width of a neural network



Source: Liang

Gradient-based learning in a neural network

Minimizing the cost function through gradient descent



- We need to compute $J(\boldsymbol{\theta})$ for a given vector of parameters $\boldsymbol{\theta}$
- We then need to find the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ **at the current position**
- Using the gradient, we "update" our parameter vector $\boldsymbol{\theta} := \boldsymbol{\theta} - \boldsymbol{\alpha}\, \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
- We repeat the above steps, until we get to the "best" solution

Step 1: Forward propagation through the computational graph

$$\begin{tikzcd}
x \arrow[r] & a^{[1]} \arrow[r] & a^{[2]} \arrow[r] & a^{[3]} \arrow[r] & y \\
& w^{[1]} \arrow[u] & w^{[2]} \arrow[u] & w^{[3]} \arrow[u] &
\end{tikzcd}$$

Step 2: Back-propagation through the computational graph

$$\begin{tikzcd}
x \arrow[r] & a^{[1]} \arrow[r] & a^{[2]} \arrow[r] & a^{[3]} \arrow[r] & y \\
& w^{[1]} \arrow[u] & w^{[2]} \arrow[u] & w^{[3]} \arrow[u] &
\end{tikzcd}$$

- Looking at the first parameter, we can think of the cost function as $J\left(f_3\left(f_2\left(f_1(w^{[1]})\right)\right)\right)$
- We need to find $\frac{\partial J}{\partial w^{[1]}} = J'\left(f_3\left(f_2\left(f_1(w^{[1]})\right)\right)\right) f_3{}'\left(f_2\left(f_1(w^{[1]})\right)\right) f_2{}'\left(f_1(w^{[1]})\right) f_1{}'(w^{[1]})$
- But we also need to find, e.g., $\frac{\partial J}{\partial w^{[2]}} = J'\left(f_3\left(f_2(w^{[2]})\right)\right) f_3{}'\left(f_2(w^{[2]})\right) f_2{}'(w^{[2]})$
- Back-propagation (BackProp) allows us to use the chain rule efficiently (without re-computing everything over and over again)

Step 2: Back-propagation through the computational graph

$$\begin{tikzcd}
x \arrow[r] & a^{[1]} \arrow[rr] \arrow[d] & {} \arrow[d] & a^{[2]} \arrow[rr] \arrow[d] & {} \arrow[d] & a^{[3]} \arrow[rr] \arrow[d] & {} \arrow[d] & y \arrow[dd] \\
 & \frac{da^{[1]}}{dw^{[1]}} \arrow[dd] & \frac{da^{[2]}}{da^{[1]}} \arrow[d] & \frac{da^{[2]}}{dw^{[2]}} \arrow[dd] & \frac{da^{[3]}}{da^{[2]}} \arrow[d] & \frac{da^{[3]}}{dw^{[3]}} \arrow[dd] & \frac{dy}{da^{[3]}} \arrow[d] & \\
 & & \frac{dJ}{da^{[1]}} \arrow[dl] & & \frac{dJ}{da^{[2]}} \arrow[ll] \arrow[dl] & & \frac{dJ}{da^{[3]}} \arrow[ll] \arrow[dl] & \frac{dJ}{dy} \arrow[l] \\
 & \frac{dJ}{dw^{[1]}} & & \frac{dJ}{dw^{[2]}} & & \frac{dJ}{dw^{[3]}} & &
\end{tikzcd}$$

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Forward-propagation algorithm

- $\boldsymbol{A}^{[0]} = \boldsymbol{X}$

- $for\ l = 1, \ldots, L:$
    - $\boldsymbol{Z}^{[l]} = \boldsymbol{A}^{[l-1]}\boldsymbol{W}^{[l]} + \boldsymbol{B}^{[l]}$
    - $\boldsymbol{A}^{[l]} = f\left(\boldsymbol{Z}^{[l]}\right)$

- $\widehat{\boldsymbol{y}} = \boldsymbol{A}^{[L]}$

- $J = J(\widehat{\boldsymbol{y}}, \mathbf{y})$

Back-propagation algorithm

- $\boldsymbol{g} \coloneqq \nabla_{\widehat{\boldsymbol{y}}} J(\widehat{\boldsymbol{y}}, \boldsymbol{y})$

- $for\ l = L, L-1 \ldots, 1:$
  - $\boldsymbol{g} \coloneqq \nabla_{\boldsymbol{Z}^{[l]}} J(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \boldsymbol{g} \odot f'\left(\boldsymbol{Z}^{[l]}\right)$

  - $\nabla_{\boldsymbol{W}^{[l]}} J(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \boldsymbol{A}^{[l-1]^T} \boldsymbol{g}$
  - $\nabla_{\boldsymbol{b}^{[l]}} J(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \boldsymbol{g}$

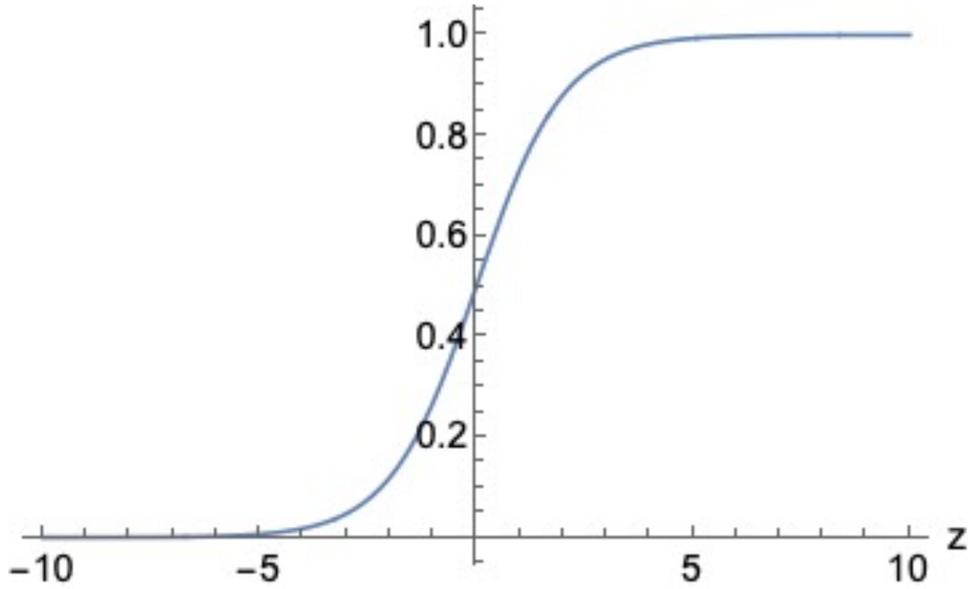
  - $\boldsymbol{g} \coloneqq \nabla_{\boldsymbol{A}^{[l]}} J(\widehat{\boldsymbol{y}}, \boldsymbol{y}) = \boldsymbol{g}\boldsymbol{W}^{[l]^{\boldsymbol{T}}}$

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Derivatives of activation functions

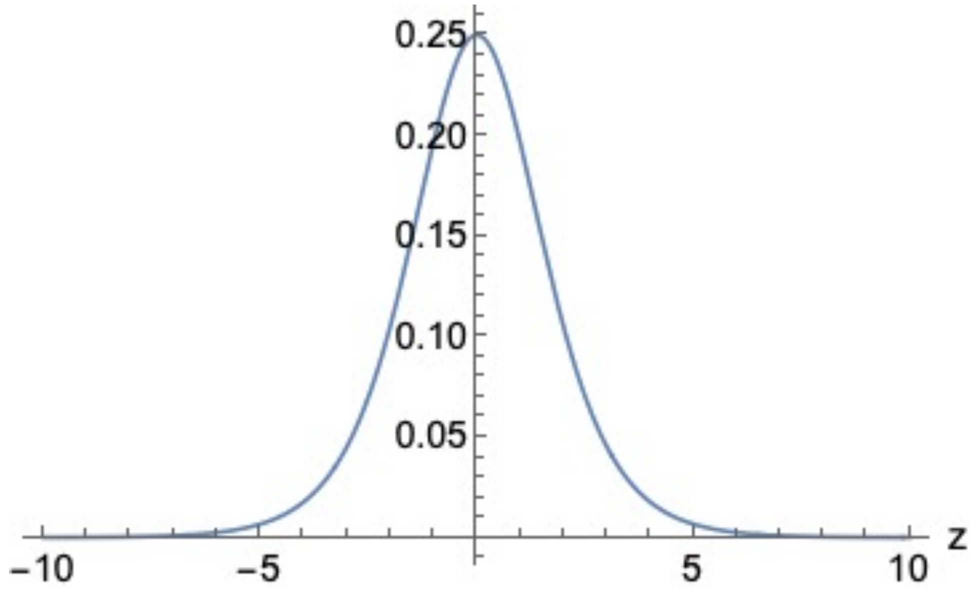
Derivative of the logistic sigmoid function

Logistic (sigmoid) function



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

Derivative

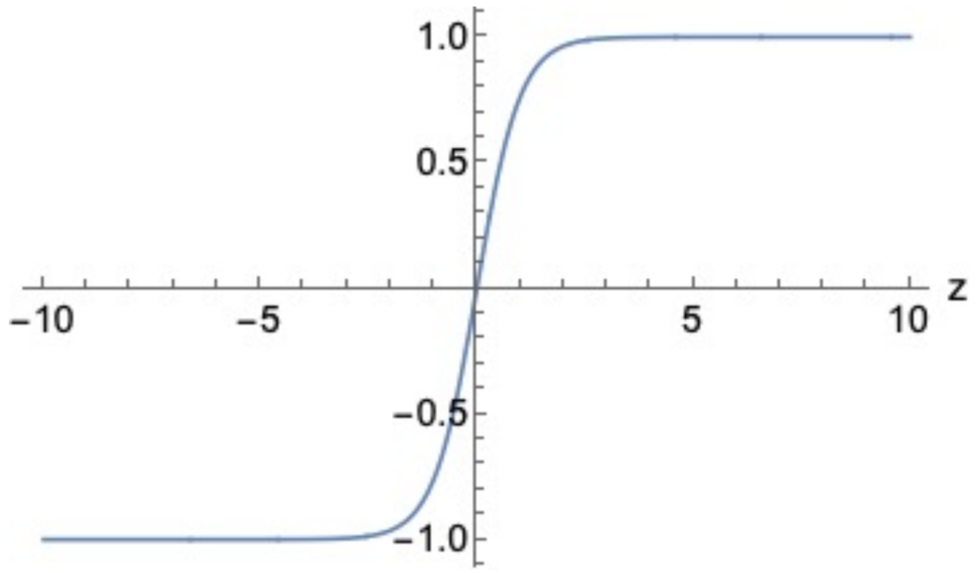


$$f'(z) = \sigma(z)\big(1 - \sigma(z)\big)$$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

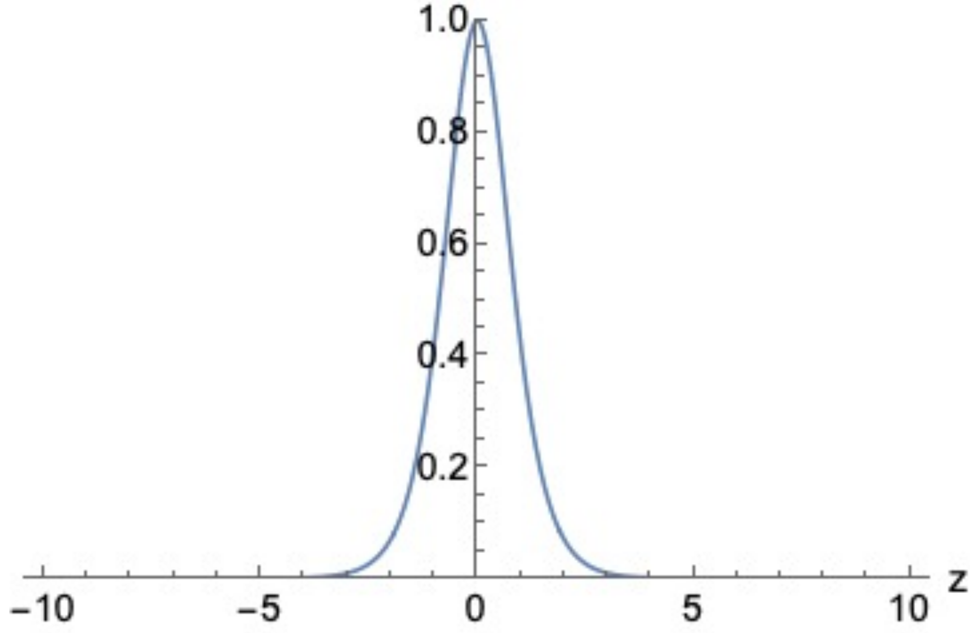
Derivative of the hyperbolic tangent

Hyperbolic tangent



$$f(z) = \tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

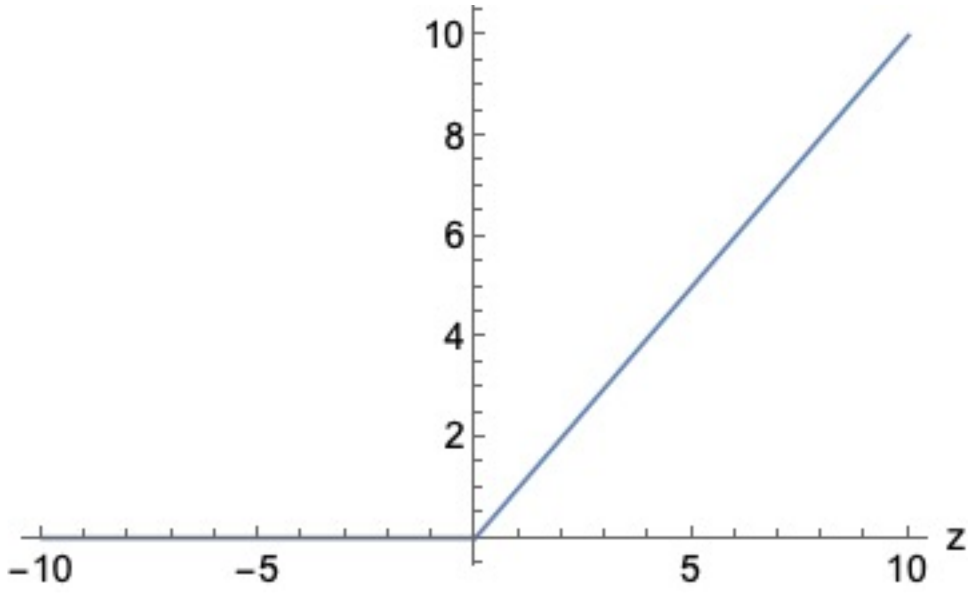
Derivative



$$f'(z) = \text{sech}(z)^2$$

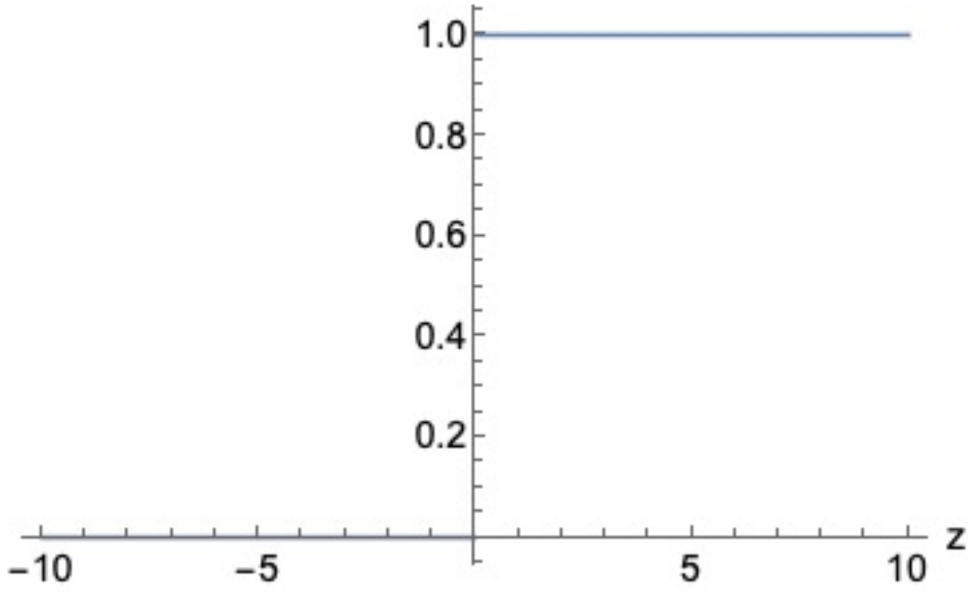
Derivative of the Rectified Linear Unit

Rectified Linear Unit (ReLU)



$$f(z) = \max\{0, z\}$$

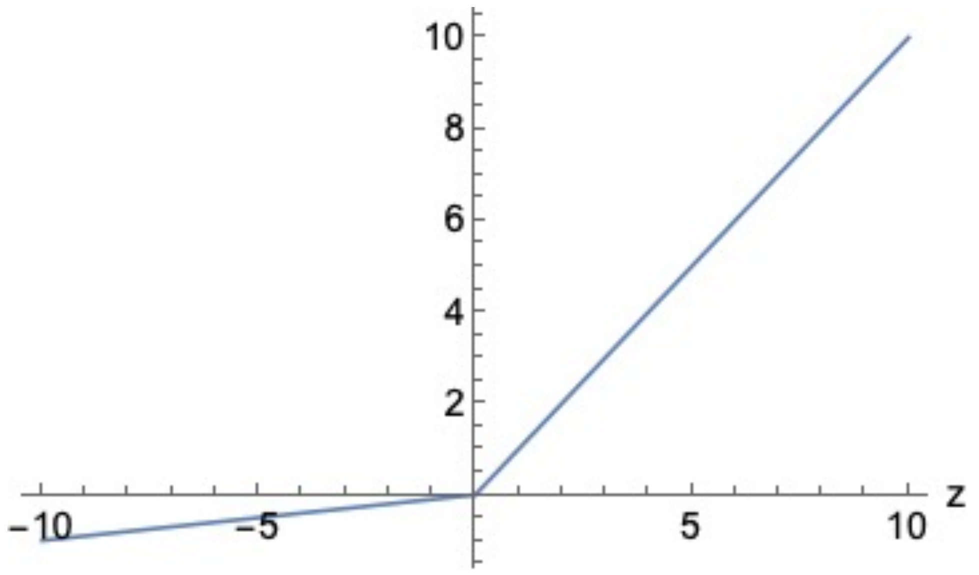
“Derivative”



$$f'(z) = \begin{cases} 0, if\ z < 0 \\ 1, if\ z > 0 \end{cases}$$

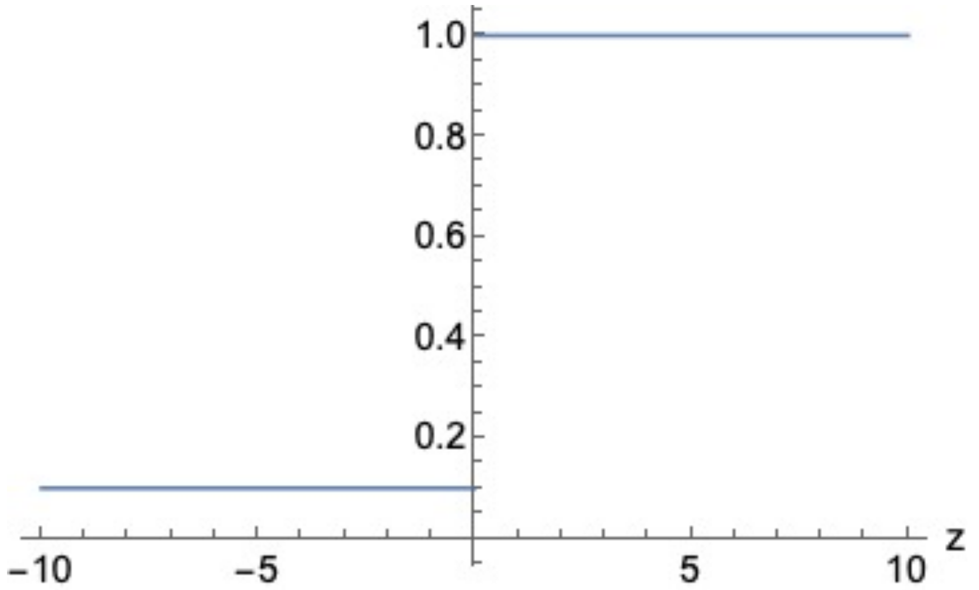
Derivative of the Leaky ReLU

Leaky ReLU



$$f(z) = \max\{0, z\}$$

“Derivative”

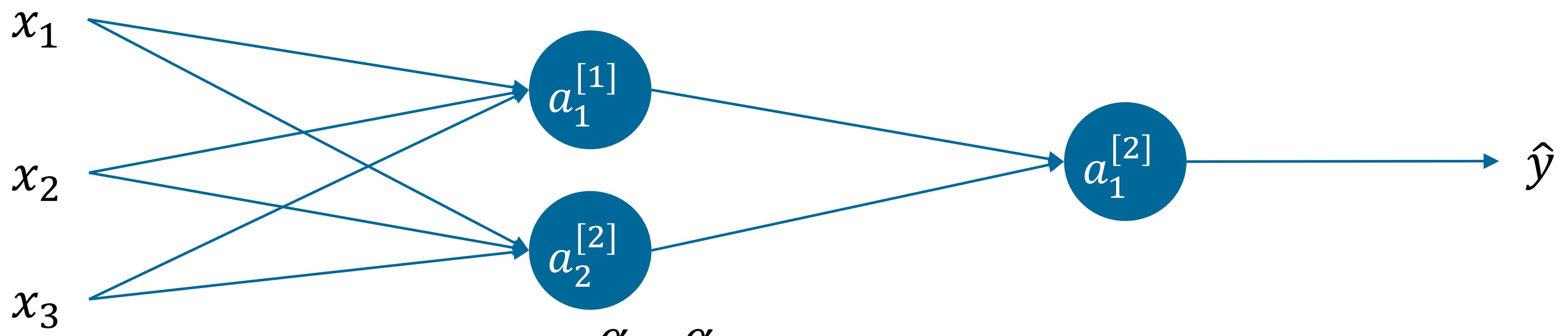


$$f'(z) = \begin{cases} 0.1, if\ z < 0 \\ 1, if\ z > 0 \end{cases}$$

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Parameter initialization

Assume instead that the weights are initialized to the same values



$$W^{[1]} = \begin{bmatrix} \alpha & \alpha \\ \alpha & \alpha \\ \alpha & \alpha \end{bmatrix} \qquad W^{[2]} = \begin{bmatrix} \gamma \\ \gamma \end{bmatrix}$$

$$b^{[1]} = [\beta \quad \beta] \qquad b^{[2]} = [\delta]$$

- We have $a_1^{[1]} = a_2^{[1]} = f(\alpha(x_1 + x_2 + x_3) + \beta)$
- We also have that $\frac{\partial J}{\partial a_1^{[1]}} = \frac{\partial J}{\partial a_2^{[1]}}$ (feel free to verify this!)
- Hence, when we update our parameters, they remain the same.

What to do instead?

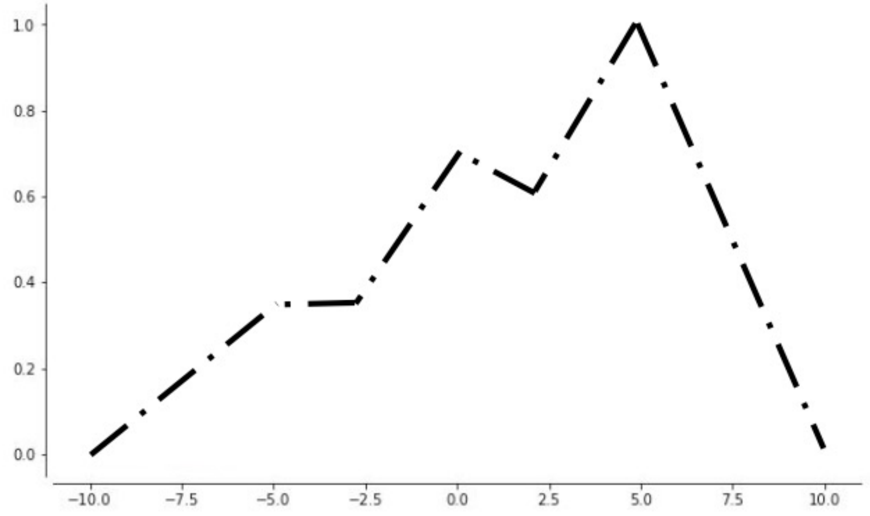
- Initialize weights to small random values (e.g., *np.random.randn(**shape of W**) * 0.01*)
- Bias terms can be initialized randomly, but can also just be initialized to zero (e.g., *np.zeros(**shape of b**)*)

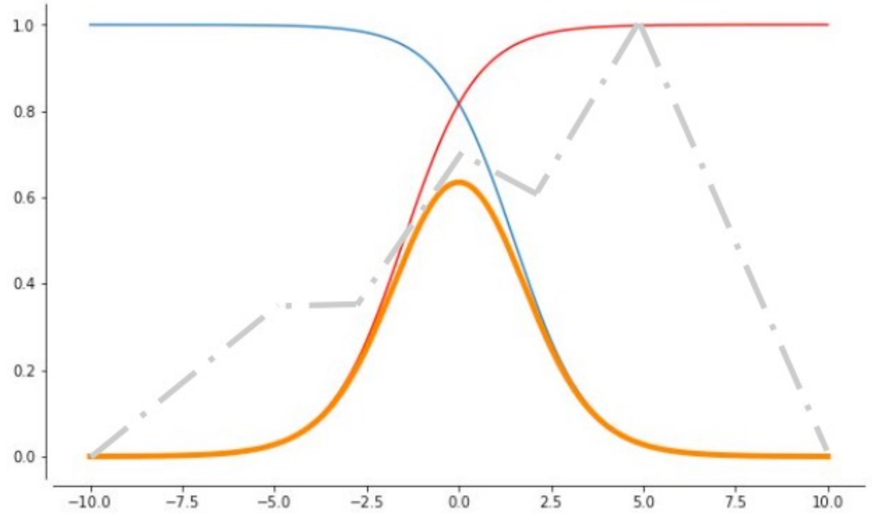
The Universal Approximator Theorem and depth

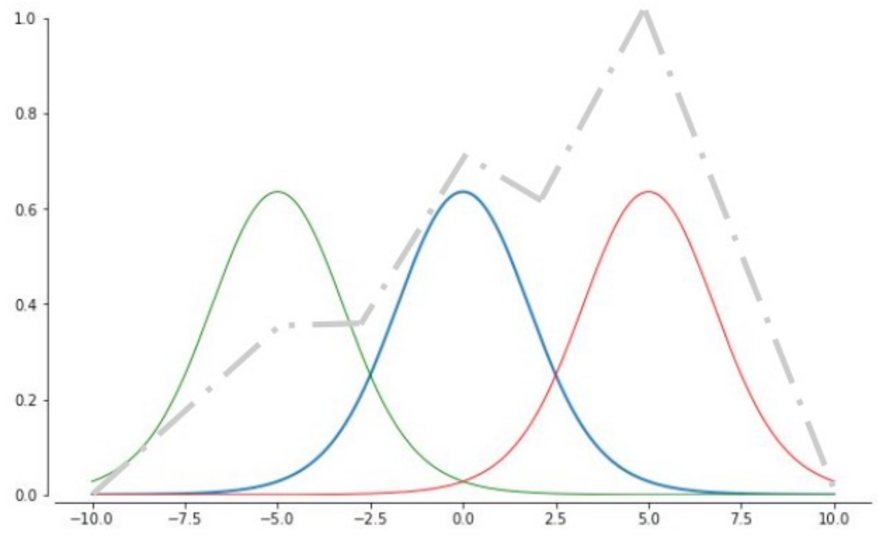
The Universal Approximator Theorem

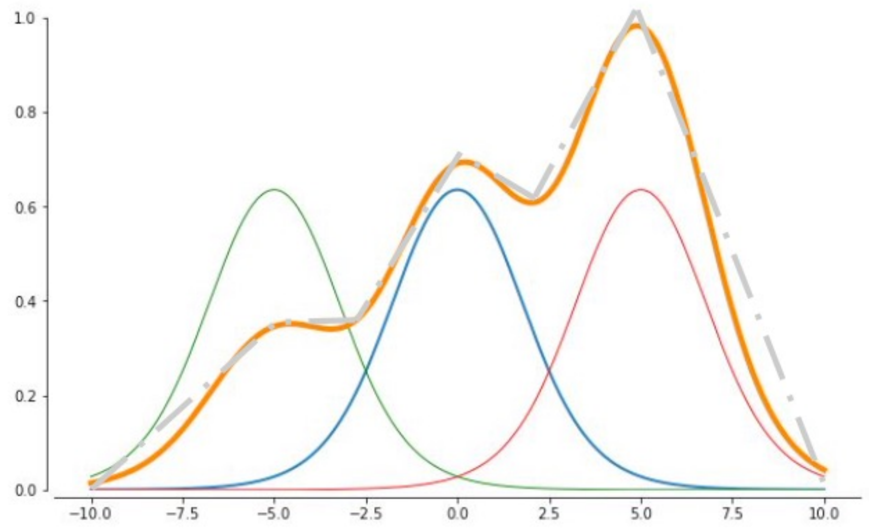
- Original theorem: 1989, extended 1993 to more general activation functions
- Idea: one hidden layer in a neural network is enough to represent an approximation of any function to an arbitrary degree of accuracy

Intuition behind the Universal Approximation Theorem









Source: Czarnecki

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

The Universal Approximator Theorem

- Original theorem: 1989, extended 1993 to more general activation functions

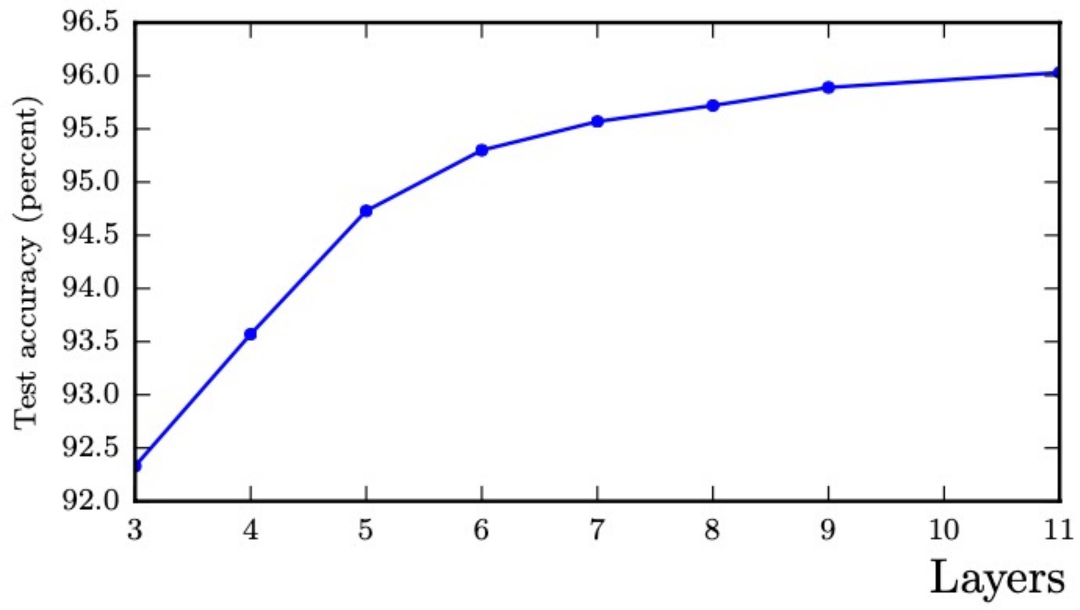
- Idea: one hidden layer in a neural network is enough to represent and approximation of any function to an arbitrary degree of accuracy

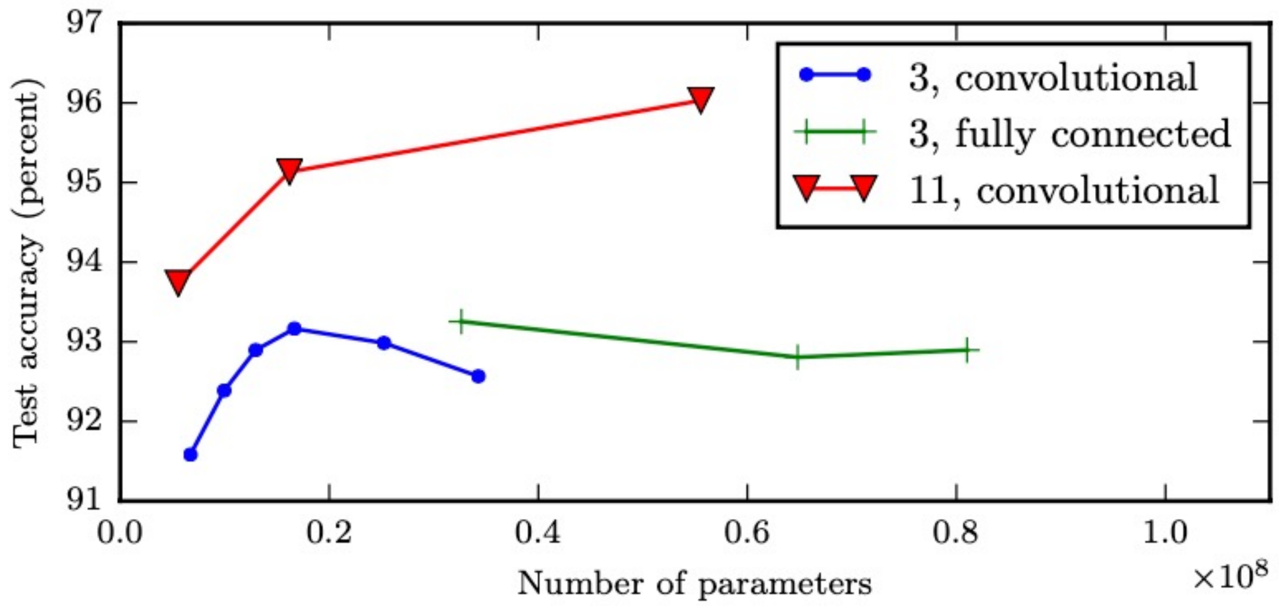
- Just because functions can be represented, that doesn't mean they can be learned
  - Optimization algorithm may not be able to find the right parameters
  - We may end up overfitting (= learning the wrong function)

- In practice, we may prefer a deep network
  - The deep network is better at avoiding overfitting
  - The width needed from a shallow network might be exponentially more than that needed from a deep network

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

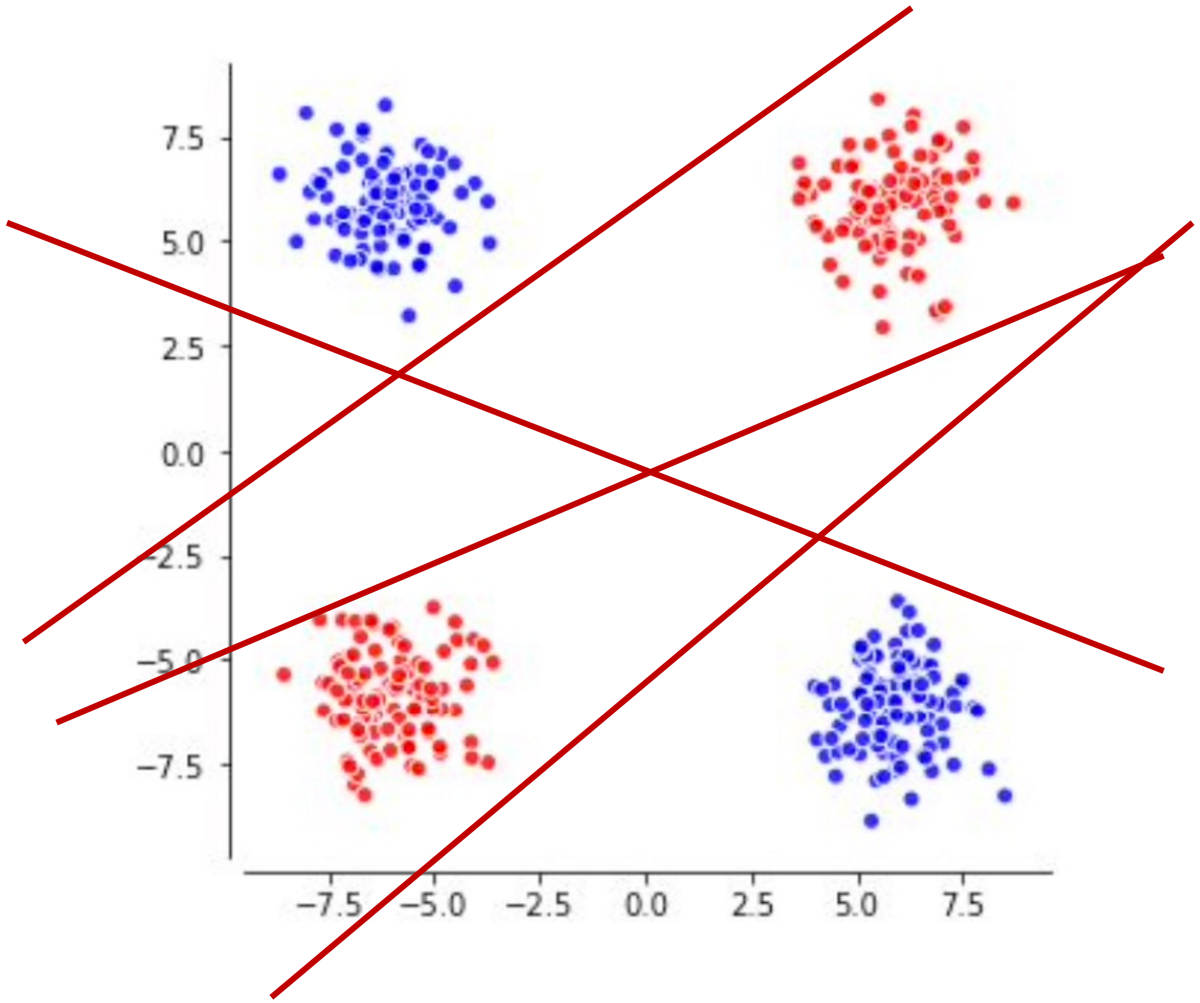
Better generalization / less overfitting with depth





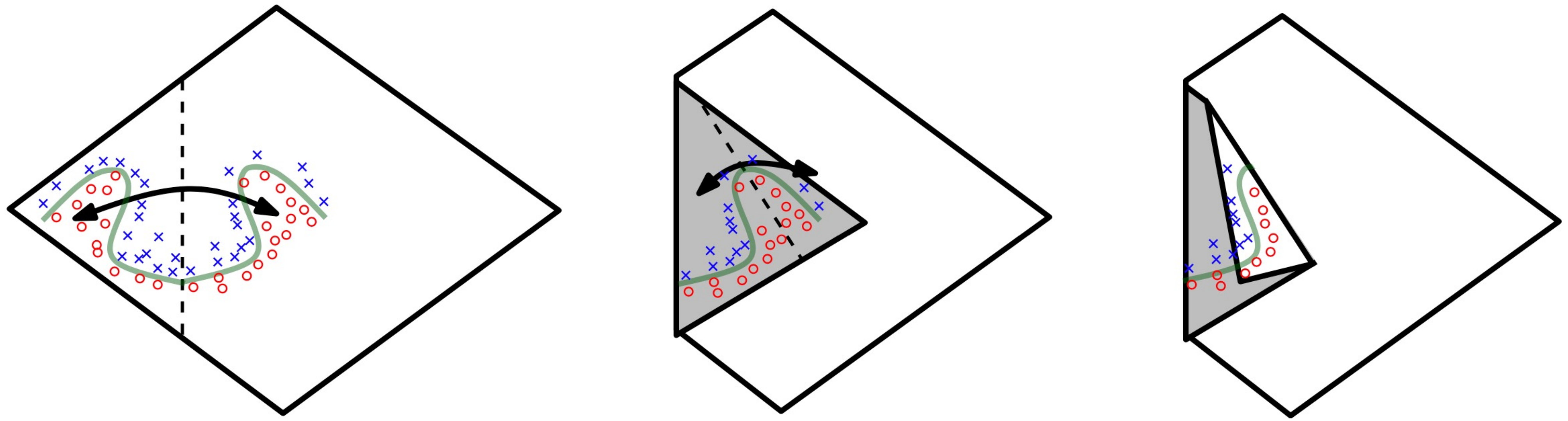
Source: Goodfellow

Reminder – what individual neurons learn



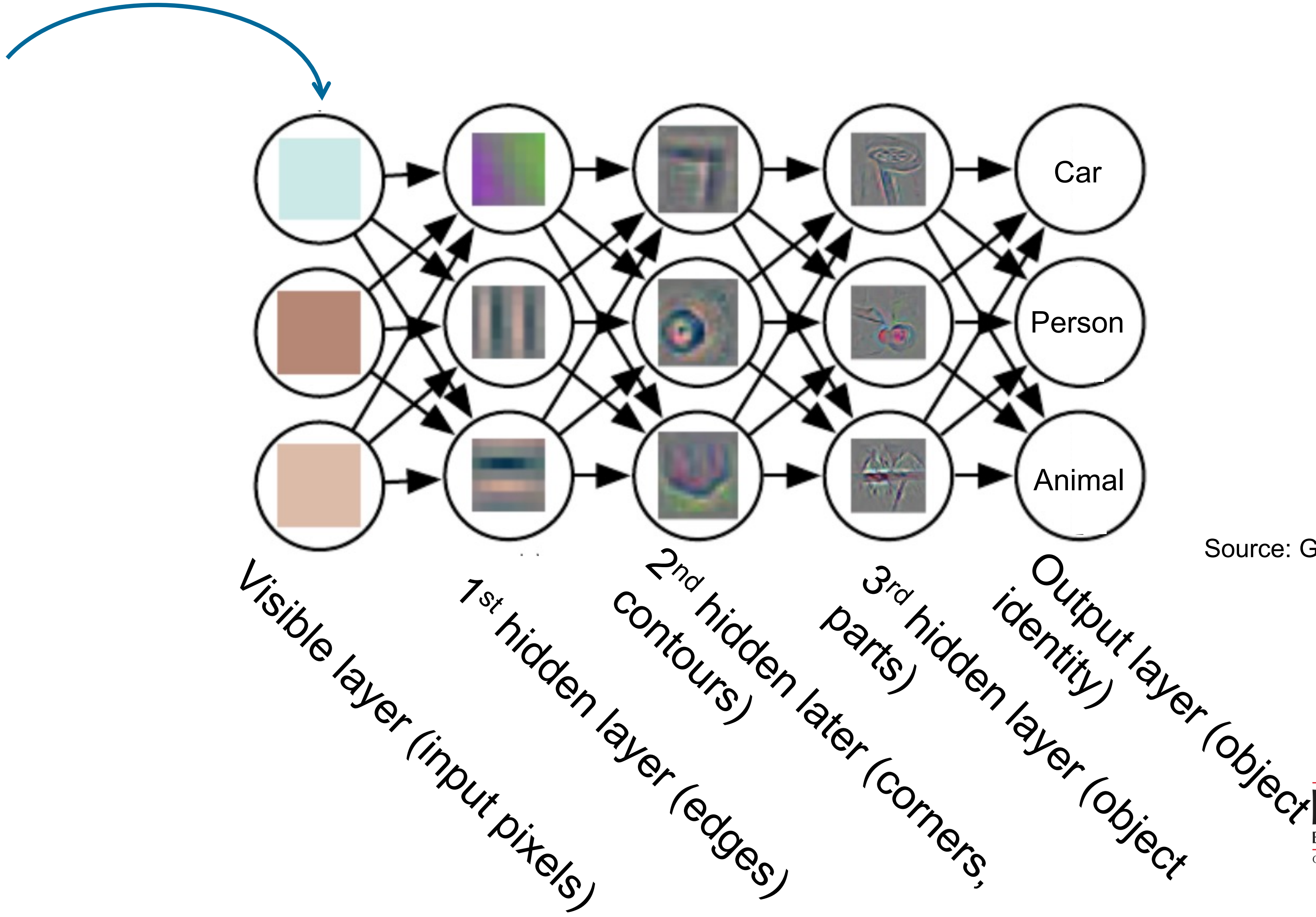
Source: Czarnecki

Representation advantage of depth



Source: Montufar

We see this effect when looking at what activates neurons at each layer


Car
Person
Animal
Visible layer (input pixels)
1st hidden layer (edges)
2nd hidden later (corners, contours)
3rd hidden layer (object parts)
Output layer (object identity)

Source: Goodfellow et al.

See you in class!

Sources

- Czarnecki, 2020, Neural Networks Foundations: https://storage.googleapis.com/deepmind-media/UCLxDeepMind_2020/L2%20-%20UCLxDeepMind%20DL2020.pdf
- DeepLearning.AI, n.d.: deeplearning.ai
- Goodfellow, Bengio, Courville, 2016, The Deep Learning Book: http://www.deeplearningbook.org
- Liang, 2016, Introduction to Deep Learning: https://www.cs.princeton.edu/courses/archive/spring16/cos495/
- Montufar et al., 2014, On the Number of Linear Regions of Deep Neural Networks: https://proceedings.neurips.cc/paper/2014/file/109d2dd3608f669ca17920c511c2a41e-Paper.pdf