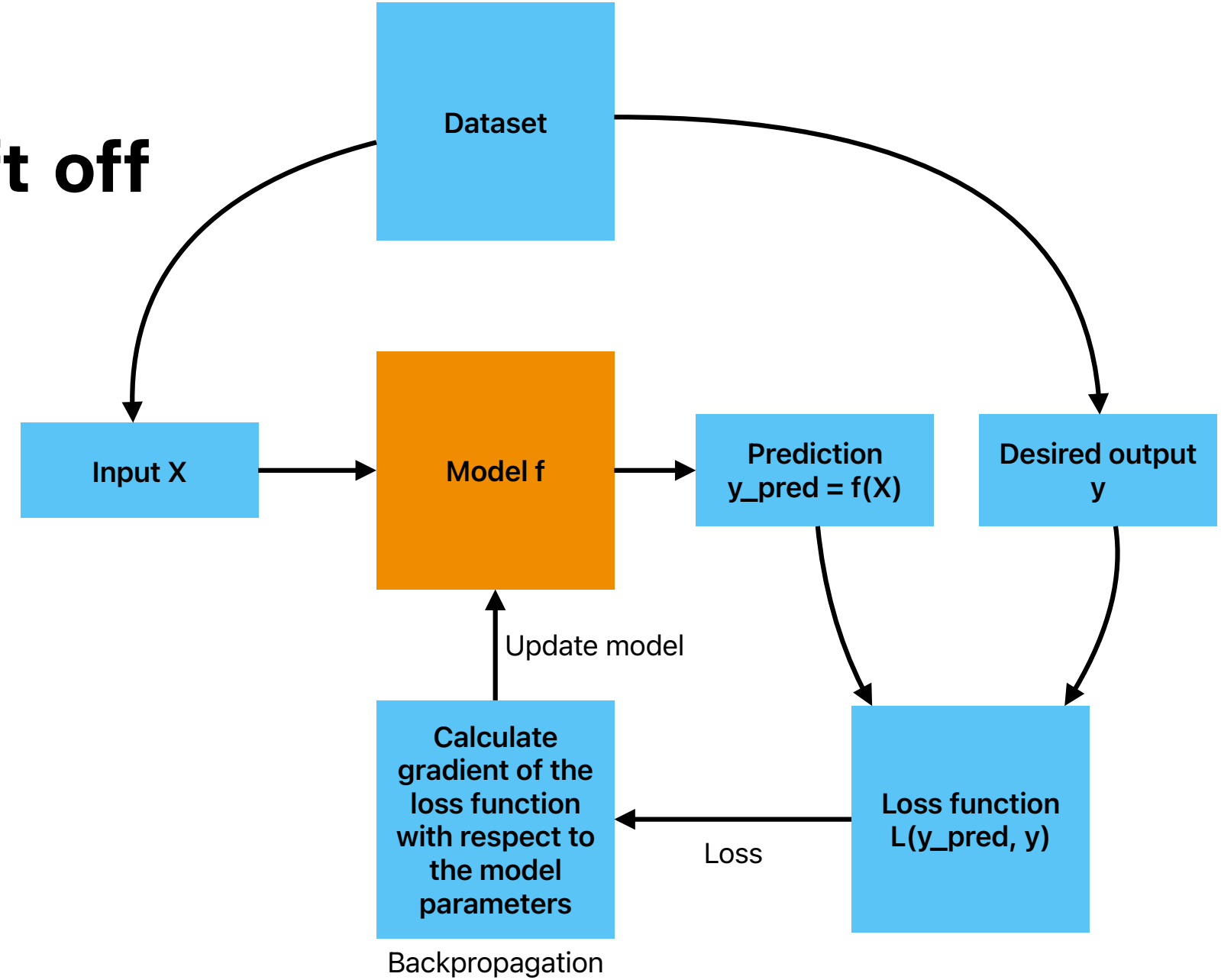
Department of Mathematics
and Computer Science

Advanced Machine Learning

Deep Feedforward Networks

Lukas Galke Poech • Spring 2026

Where we left off

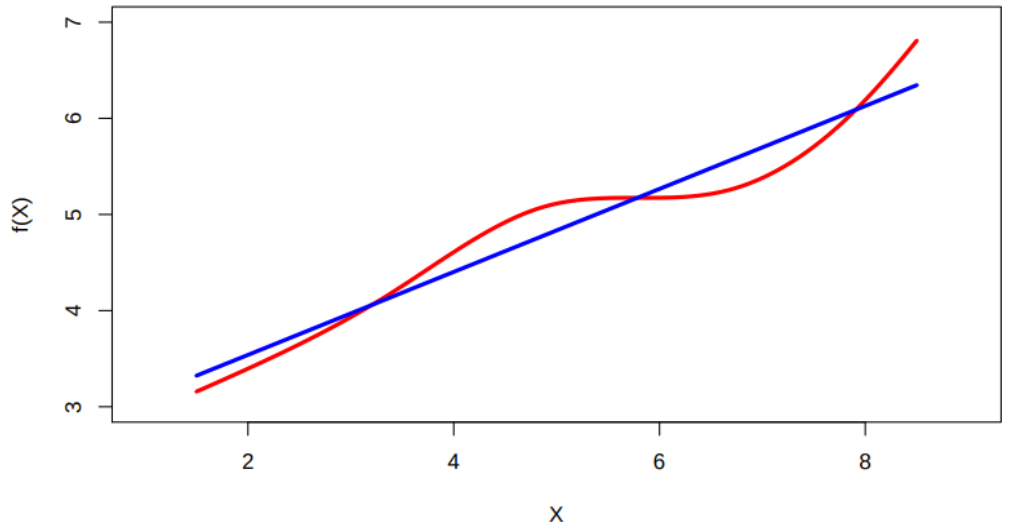


Keywords

- **Single-Layer Networks**
- Deep Neural Networks
- Universal Approximation Theorem
- Activation functions
- Output units

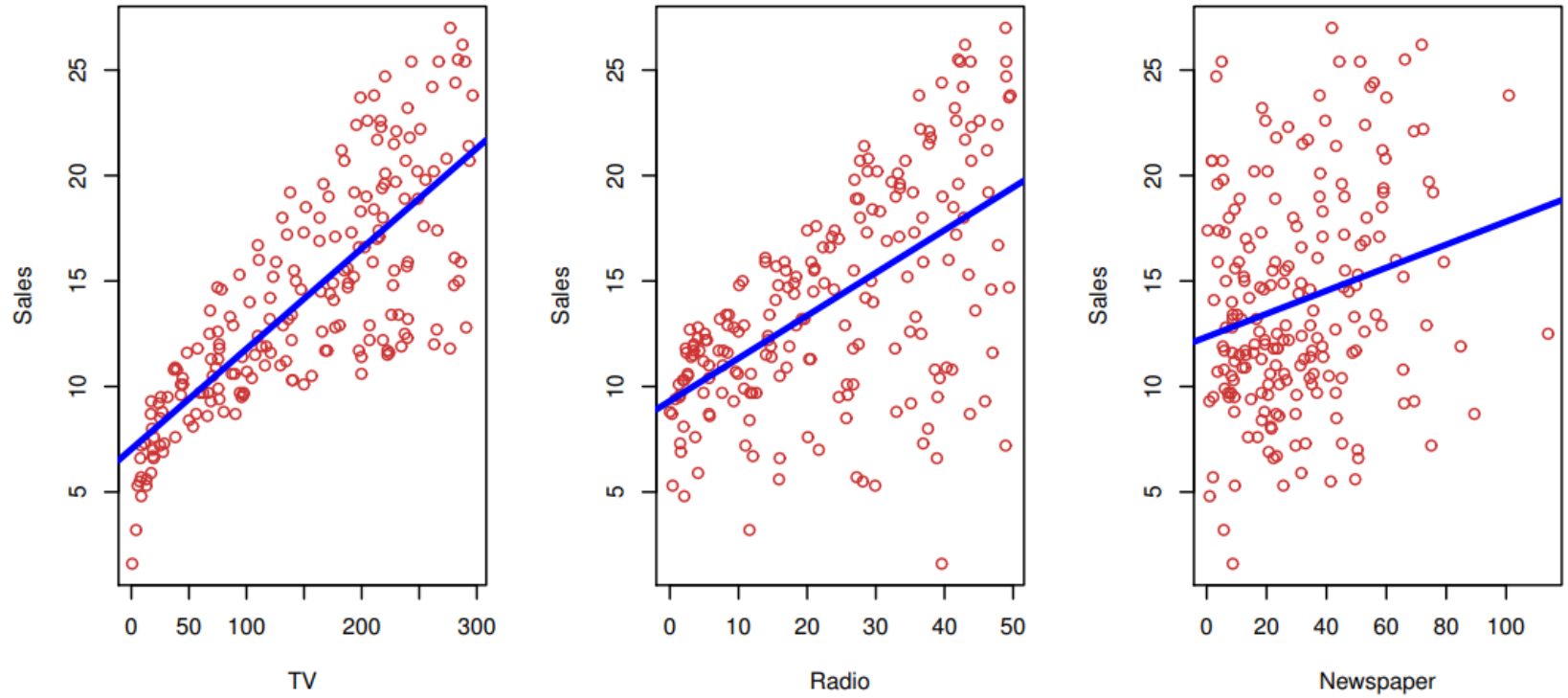
Linear Regression

- Linear regression is a simple approach to supervised learning. It assumes that the dependence of Y on $X_1, X_2, \ldots X_p$ is linear.
- True regression functions are never linear!



- Although it may seem overly simplistic, linear regression is extremely useful both conceptually and practically.

Example: Advertising data



- The Advertising data set consists of the sales of that product in 200 different markets, along with advertising budgets for the product in each of those markets for three different media: TV, radio, and newspaper.

Example: Advertising data

Questions we might ask:

- Is there a relationship between advertising budget and sales?
- How strong is the relationship between advertising budget and sales?
- Which media contribute to sales?
- How accurately can we predict future sales?
- Is the relationship linear?
- Is there synergy among the advertising media?

Simple Linear Regression.

We assume a model

$$Y = \beta_0 + \beta_1 X + \epsilon$$

where β_0 and β_1 are two unknown constants that represent the intercept and slope, also known as coefficients or parameters, and ϵ is the error term.

Given some estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ model coefficients, we **predict** future sales using

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

Estimation of the Parameters by Least Squares

➢ Let $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$ be the prediction of Y for the ith observation.

➢ Then the **residual** is defined as

$$e_i = y_i - \hat{y}_i$$

➢ We define the **residual sum of squares** as

$$\text{RSS} = e_1^2 + e_2^2 + \ldots + e_n^2 =$$
$$\left(y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1\right)^2 + \left(y_2 - \hat{\beta}_0 - \hat{\beta}_1 x_2\right)^2 + \ldots + \left(y_n - \hat{\beta}_0 - \hat{\beta}_1 x_n\right)^2$$

➢ Our task is to find the $\hat{\beta}_0$ and $\hat{\beta}_1$ minimizing the RSS.

Estimation of the Parameters by Least Squares

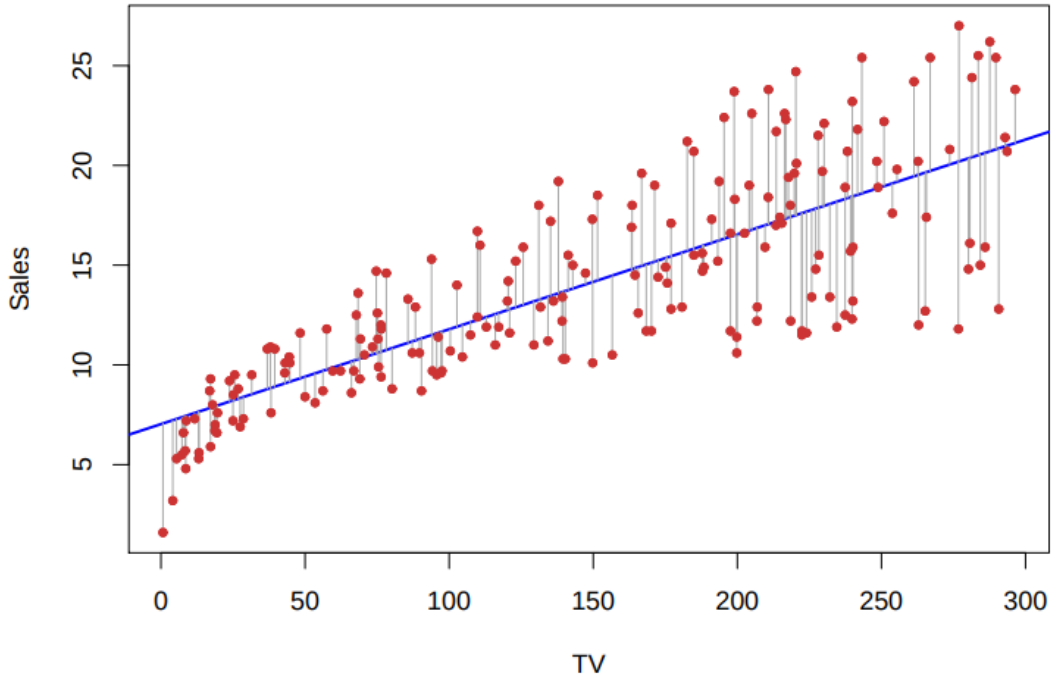
➢ With simple derivation of the RSS formula, we can show that

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} \left(x_i - \bar{x}\right)\left(y_i - \bar{y}\right)}{\sum_{i=1}^{n} \left(x_i - \bar{x}\right)^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

➢ with the sample means $\bar{y} = \frac{1}{\mathrm{n}} \sum_{i=1}^{n} y_i$ and $\bar{x} = \frac{1}{\mathrm{n}} \sum_{i=1}^{n} x_i$

Example: Advertising data



- The least squares fit for the regression of sales onto TV.
- In this case a linear fit captures the essence of the relationship, although it is somewhat deficient in the left of the plot.

Multiple Linear Regression

➢ We can include several features or predictors into our model

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + + \beta_p X_p + \epsilon$$

➢ We interpret β_j as the average effect on Y of a one unit increase in X_j, holding all other predictors fixed. In the advertising example, the model becomes

$$\text{sales} = \beta_0 + \beta_1 \cdot \text{TV} + \beta_2 \cdot \text{radio} + \beta_3 \cdot \text{newspaper} + \epsilon$$

In Matrix Form

We can formulate the entire linear regression also in Matrixform

$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{b} + \boldsymbol{e}$

$\boldsymbol{y} = (y_1, \ldots, y_n) \in \mathbb{R}^n$ is the $n \times 1$ response vector

$\boldsymbol{X} = [1_n, \boldsymbol{X}'] \in \mathbb{R}^{n \times (p+1)}$ is the $n \times (p+1)$ design matrix

1_n is an $n \times 1$ vector of ones

$\boldsymbol{X}' = \left[\boldsymbol{X_1}, \boldsymbol{X_2}, \ldots, \boldsymbol{X_p}\right] \in \mathbb{R}^{n \times p}$ the $n \times p$ predictor Matrix

$b = \left(\beta_0, \beta_1, \ldots, \beta_p\right) \in \mathbb{R}^{p+1}$ is regression coefficient vector

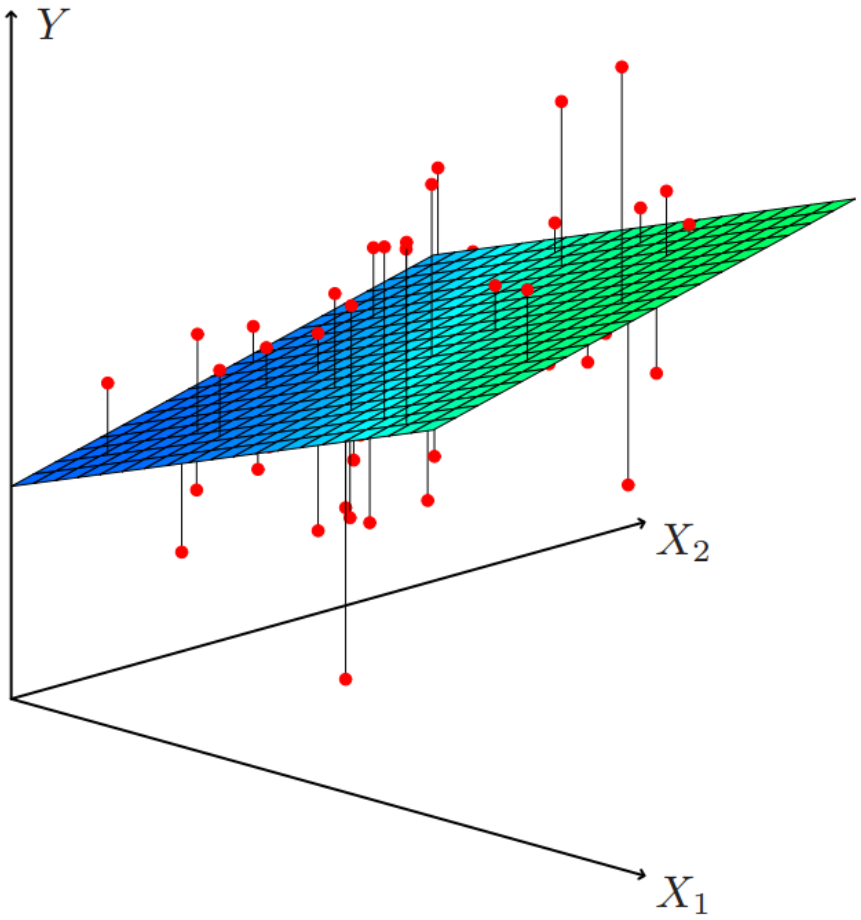
$e = (e_1, e_2, \ldots, e_3) \in \mathbb{R}^n$ is the $n \times 1$ error vector

Example

#	TV	Radio	News	Sales
1	230.1	37.8	69.2	**22.1**
2	44.5	39.3	45.1	**10.4**
3	17.2	45.9	69.3	**9.3**
4	151.5	41.3	58.5	**18.5**
5	180.8	10.8	58.4	**12.9**

$$\begin{pmatrix} 22.1 \\ 10.4 \\ 9.3 \\ 18.5 \\ 12.9 \end{pmatrix} = \begin{pmatrix} 1 & 230.1 & 37.8 & 69.2 \\ 1 & 44.5 & 39.3 & 45.1 \\ 1 & 17.2 & 45.9 & 69.3 \\ 1 & 151.5 & 41.3 & 58.5 \\ 1 & 180.8 & 10.8 & 58.4 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} + \begin{pmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \end{pmatrix}$$

Linear Regression



Generalizations

There exist many different extensions and generalizations to the simple linear regression model:

- **Classification problems**: logistic regression, support vector machines
- **Non-linearity**: kernel smoothing, splines and generalized additive models; nearest neighbor methods.
- **Interactions**: Tree-based methods, bagging, random forests and boosting (these also capture non-linearities)
- **Regularized fitting**: Ridge regression and lasso

Logistic Regression

One type of generalized linear model

- **Logistic regression** models the log-odds of an event as a a linear combination of input variables
- How to get there? Apply sigmoid to the output to ensure it is between 0 and 1, estimating the probability of an event (rather than a quantity)

Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$

Regression $p(X) = \beta_0 + \beta_1 X.$

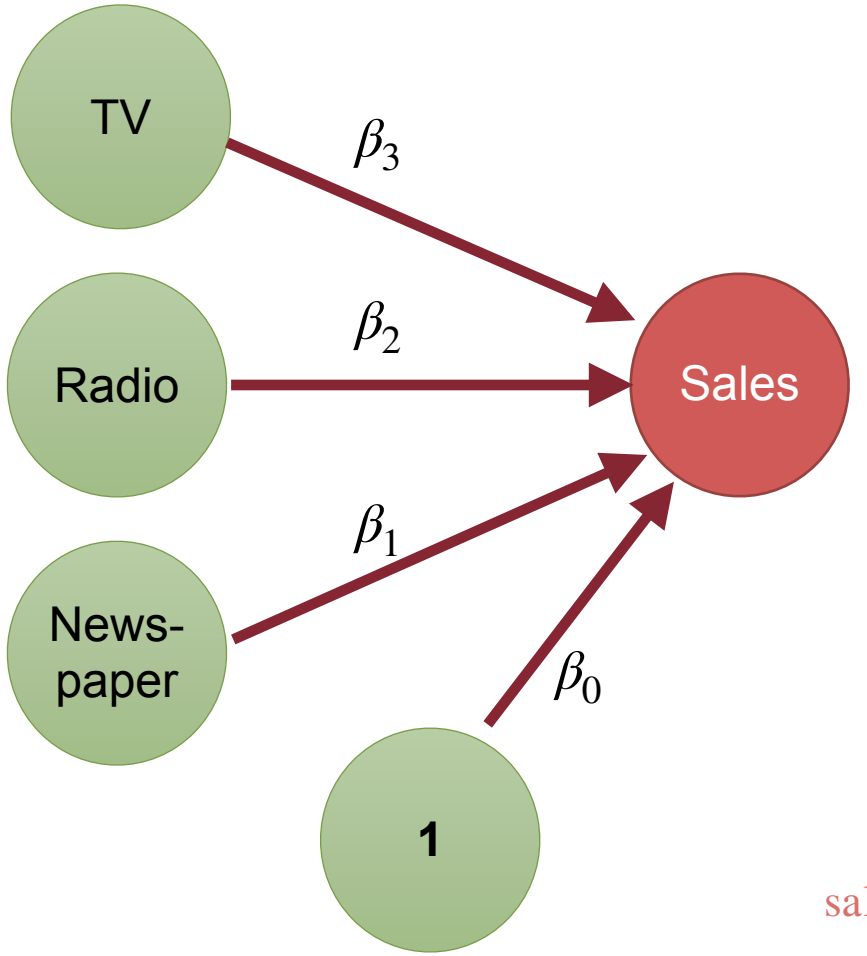
Apply sigmoid to RHS $p(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}}.$

Rewrite to "Odds" $\frac{p(X)}{1-p(X)} = e^{\beta_0+\beta_1 X}.$

Log odds or "Logits" $\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X.$

This terminology has been recycled for the raw outputs of a neural network

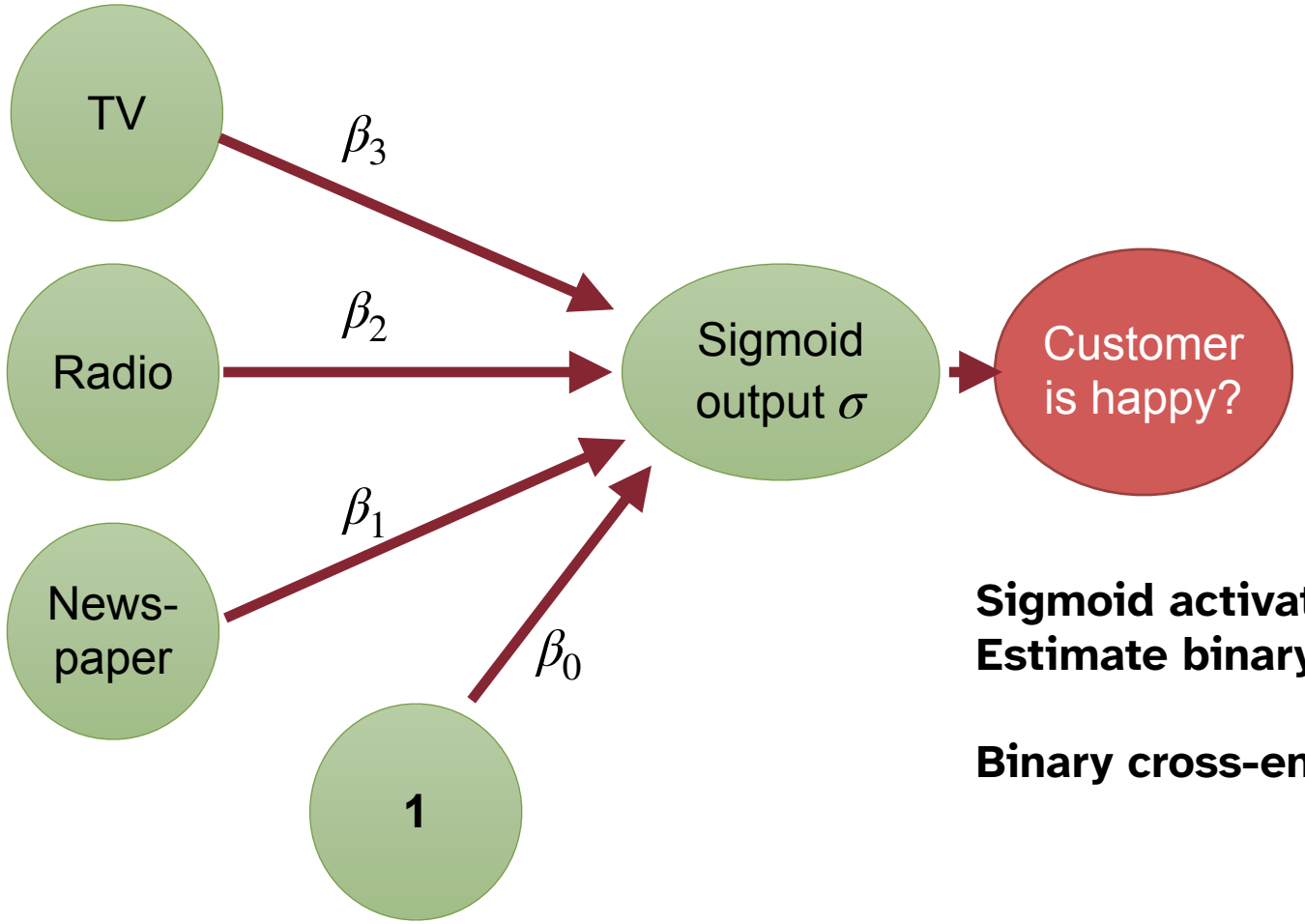
Single-layer neural network for linear regression



Mean-squared error loss

$$\text{sales} = \beta_0 + \beta_1 \cdot \text{TV} + \beta_2 \cdot \text{radio} + \beta_3 \cdot \text{newspaper} + \epsilon$$

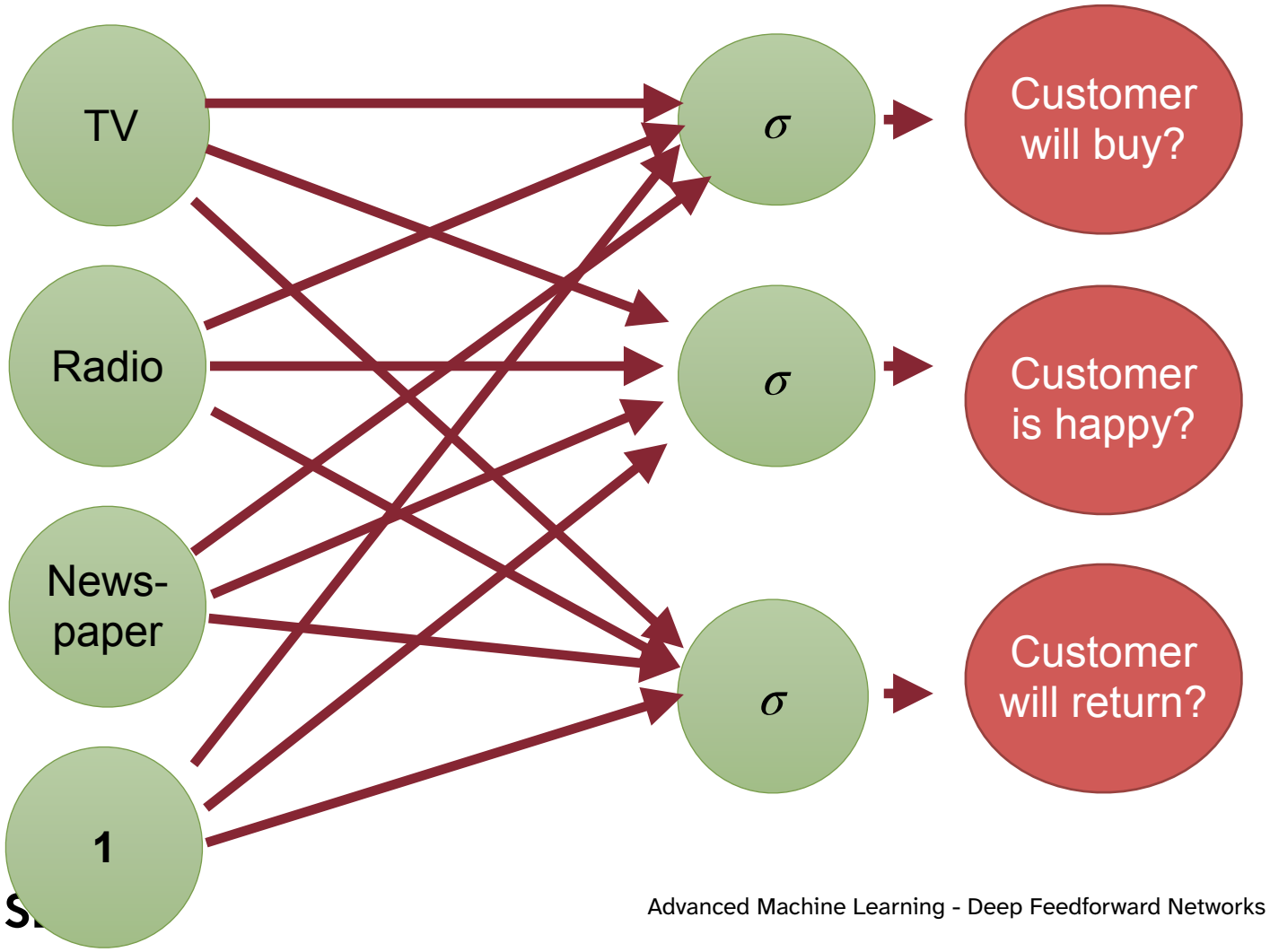
Single-layer neural network for logistic regression



Sigmoid activation,
Estimate binary probability

Binary cross-entropy loss

Single-layer NN doing multiple logistic regressions



Here, all the logistic regressions are independent from each other

Multi-layered networks can find shared factors of variation, similar to the interactions seen before

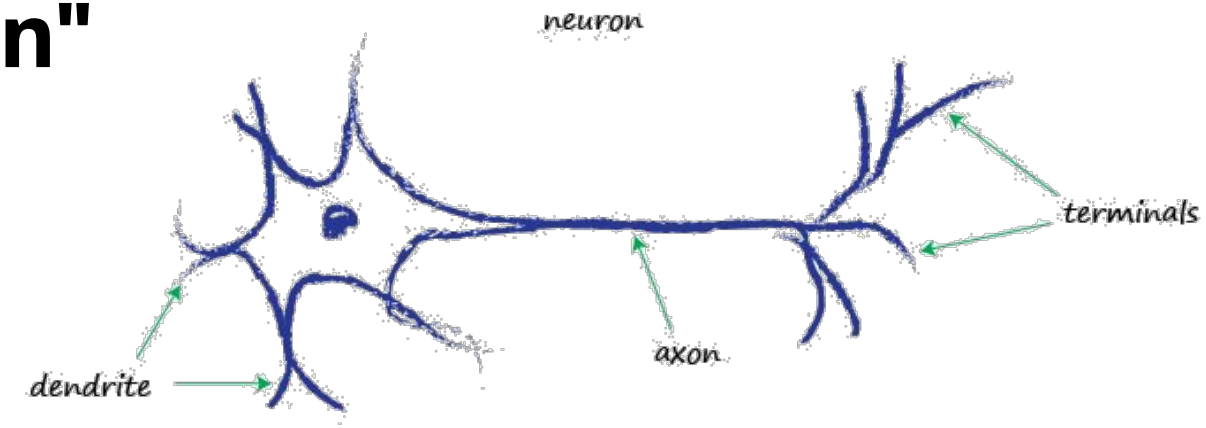
More scalable than modeling explicit interaction for every pair of variables

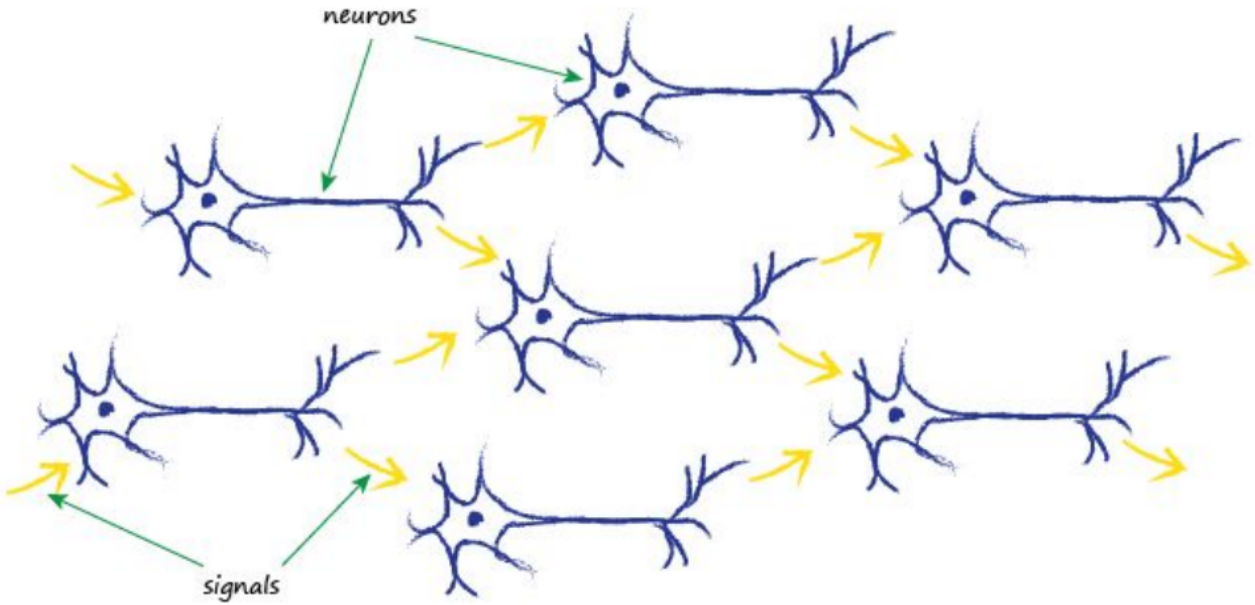
But less interpretable

Keywords

- Single-Layer Networks
- **Deep Neural Networks**
- Universal Approximation Theorem
- Activation functions
- Output units

From Neuron to "Brain"

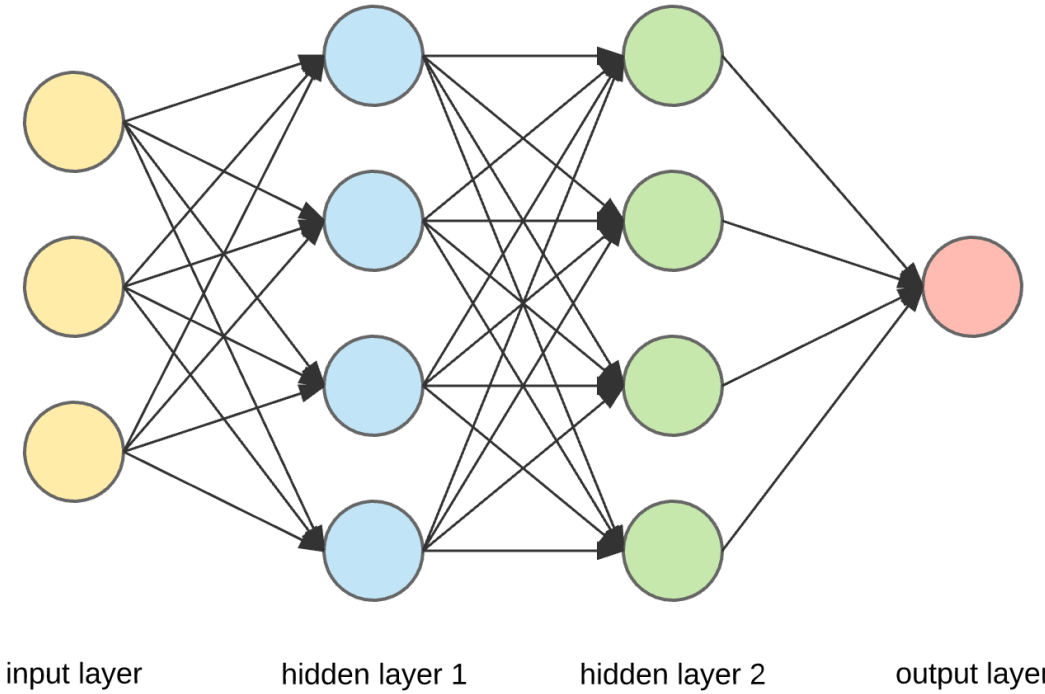




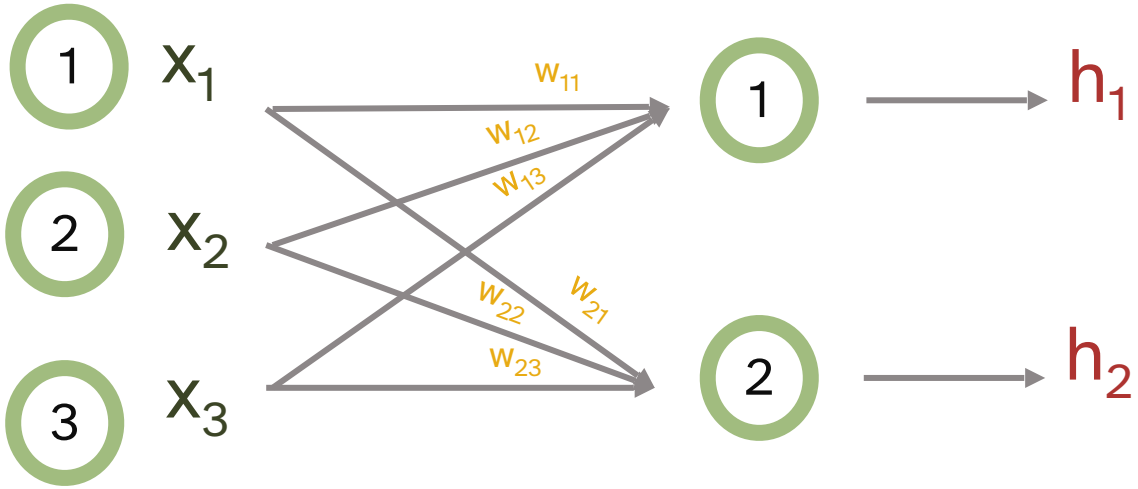
- We now can connect neurons
- The output of one neuron becomes the input of other neurons

In a more structured way

- We normally have more than one node
- Multiple Nodes are arranged in layers
- Each layer receives the generated output from the previous layer

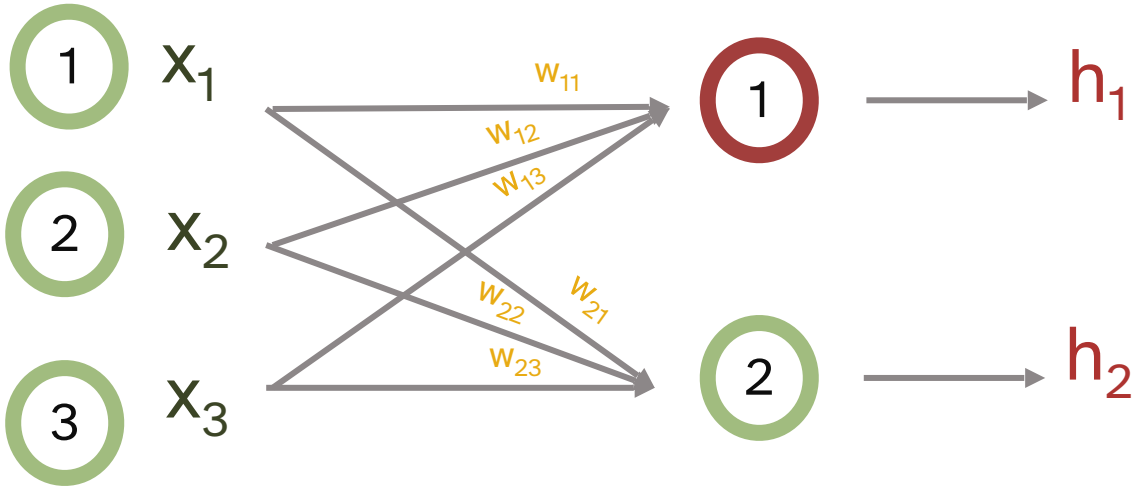


What does such a Network do?



- Remember from before:
  - We have a weighted sum
  - Then we have an "activation function" (which we ignore for now)

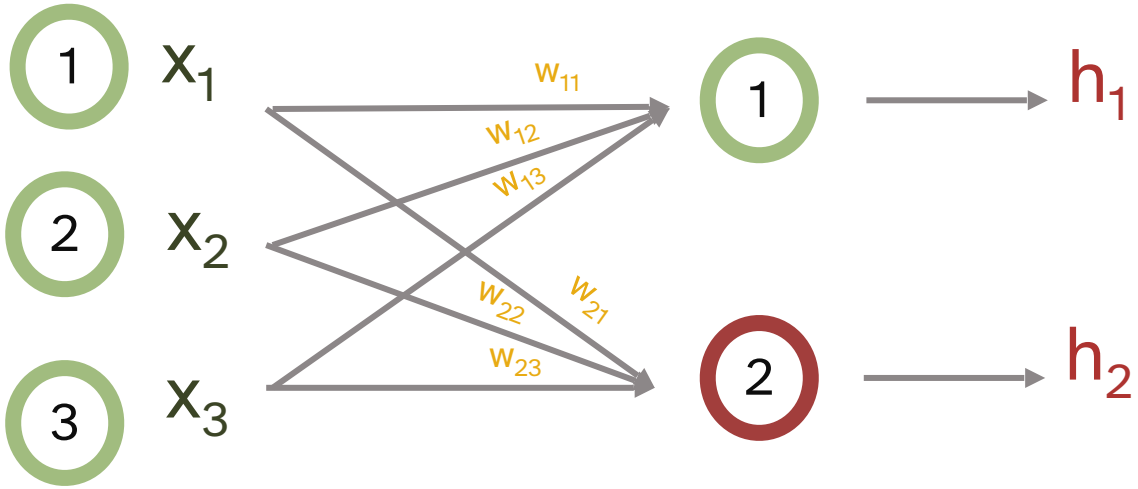
What does such a Network do?



- Remember from before:
  - We have a weighted sum
  - Then we have an "activation function" (which we ignore for now)

- For **Node 1**, we get: $h_1 = w_{11} \cdot x_1 + w_{12} \cdot x_2 + w_{13} \cdot x_3$

What does such a Network do?



- Remember from before:
  - We have a weighted sum
  - Then we have an "activation function" (which we ignore for now)

- For **Node 2**, we get: $h_2 = w_{21} \cdot x_1 + w_{22} \cdot x_2 + w_{23} \cdot x_3$

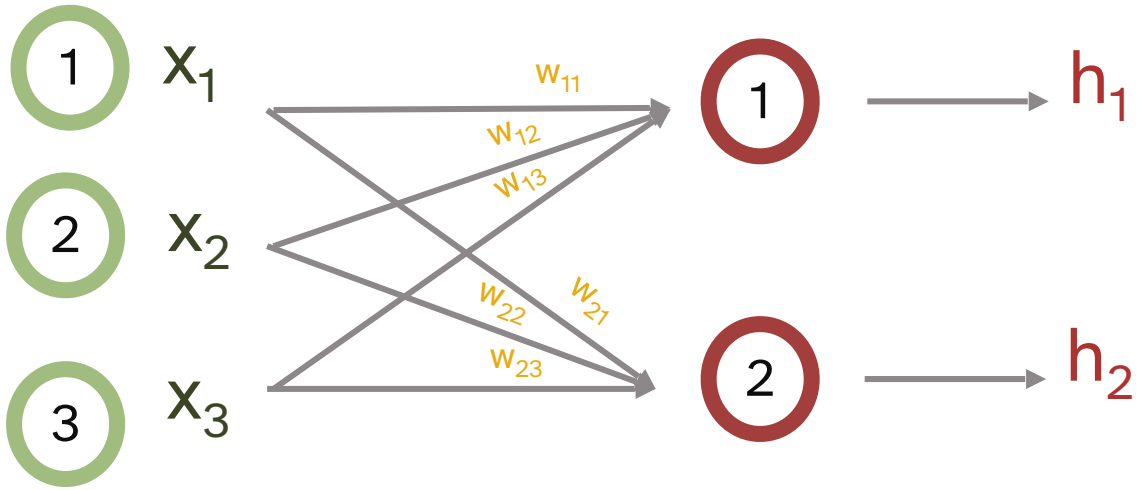
What does such a Network do?

Together:

$$h_1 = w_{11} \cdot x_1 + w_{12} \cdot x_2 + w_{13} \cdot x_3$$

$$h_2 = w_{21} \cdot x_1 + w_{22} \cdot x_2 + w_{23} \cdot x_3$$

Looks familiar, doesn't it?

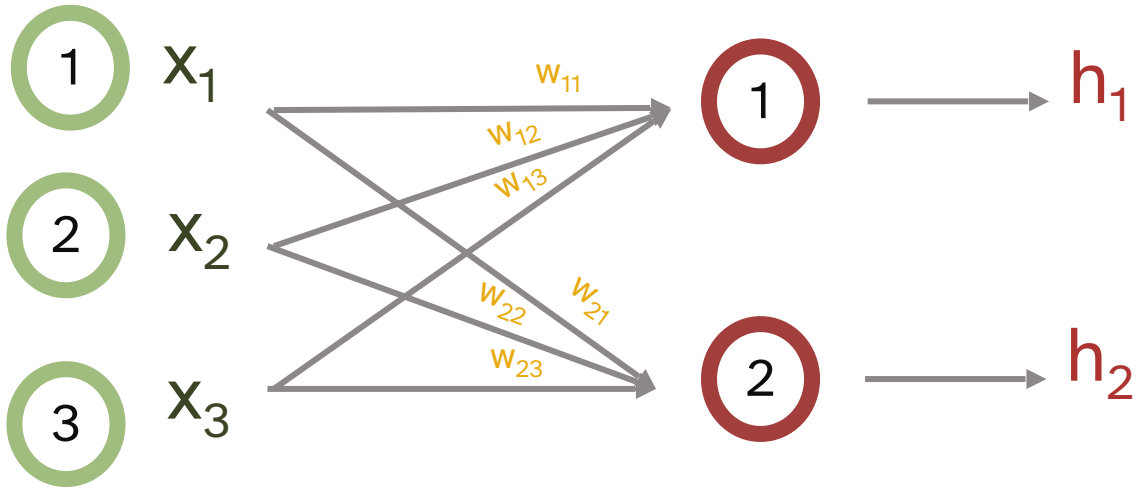


What does such a Network do?

Rewrite:

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

$$h = \boldsymbol{W}\boldsymbol{x}$$

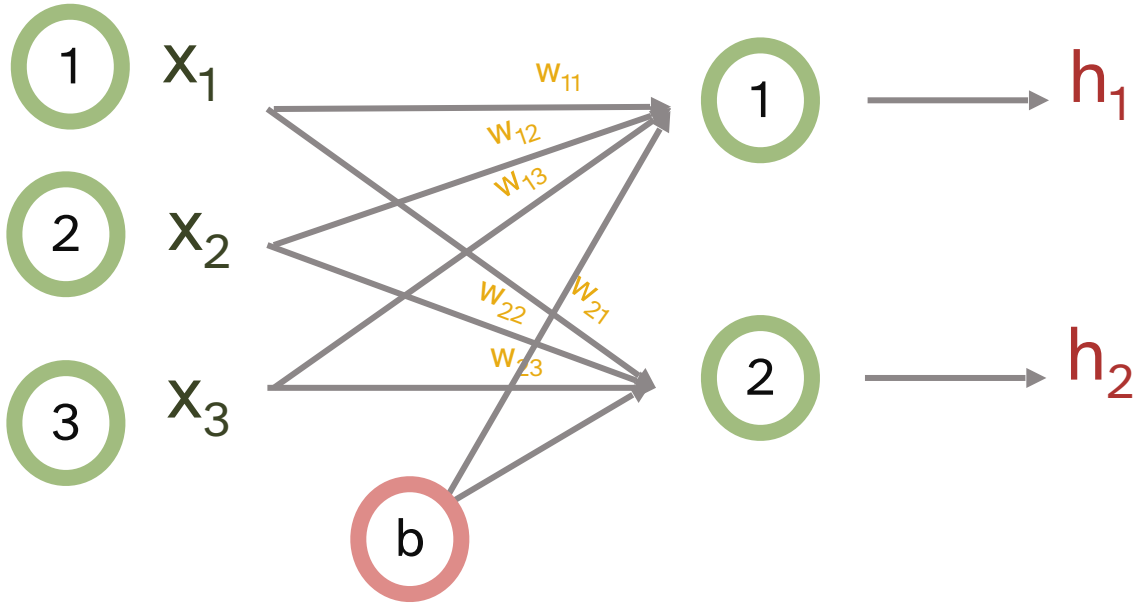


What does such a Network do?

Adding a **Bias**:

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$\boldsymbol{h} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

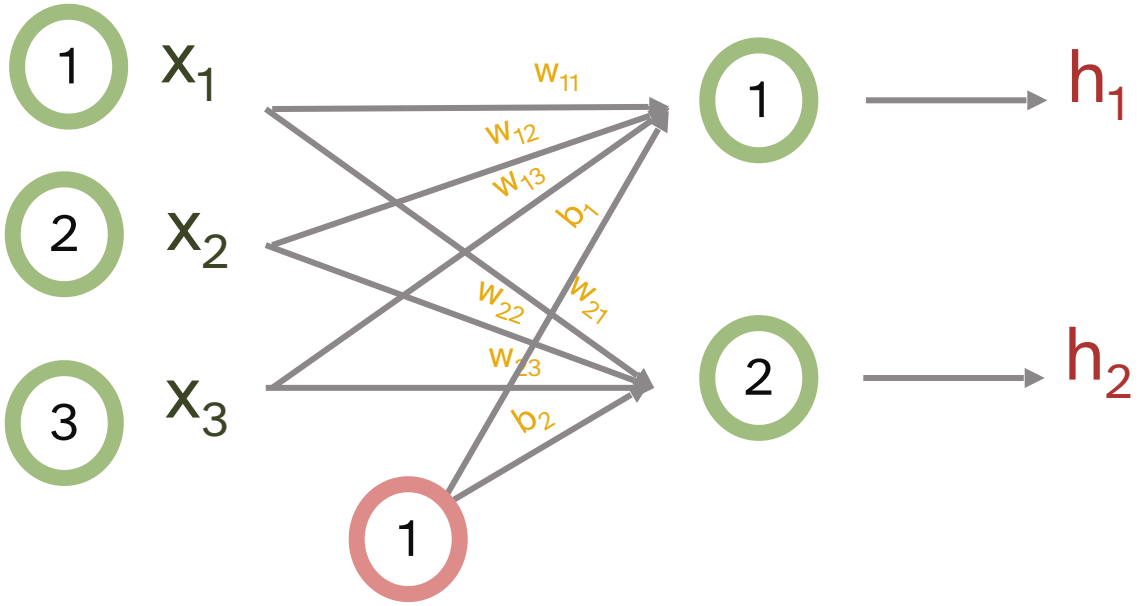


What does such a Network do?

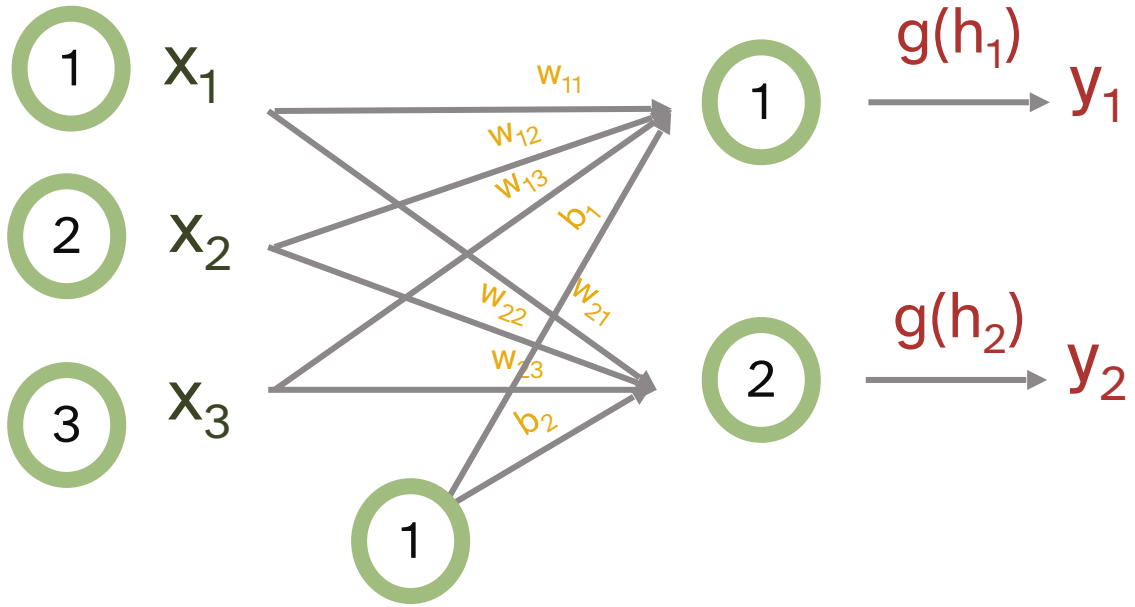
Adding a **Bias** (rewritten):

$$\begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & b_1 \\ w_{21} & w_{22} & w_{23} & b_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix}$$

$$\boldsymbol{h} = \boldsymbol{W}\boldsymbol{x}$$



What does such a Network do?

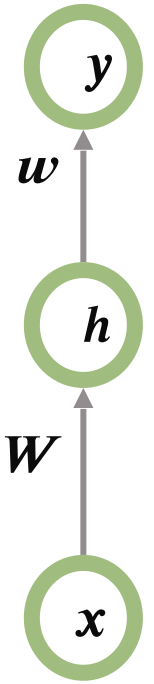


Adding the **activation function:**

$$\boldsymbol{y} = \boldsymbol{g}\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right)$$

More Compact Representation

- We can summarize these networks to their essential parts: We receive the vector $\boldsymbol{x}$ and perform an affine transformation according to the weight matrix
- $\boldsymbol{h}' = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$
- Afterward, the internal representation is component wise feed through an activation function $h_i = g(h_i')$ to generate the input vector $\boldsymbol{h}$ for the next layer
- Please note the slight shift in notation. From now on:
  - $\boldsymbol{x}$ denotes the input
  - $\boldsymbol{W}$ denotes weight matrices, **w** denotes weight vectors
  - $\boldsymbol{h}$ denotes hidden layer "outputs"
  - $\boldsymbol{y}$ is the final output of the network



Composition of Complex Functions

Now let's look at a deeper Network:

E.g., functions $f^{(1)}$, $f^{(2)}$, $f^{(3)}$ connected in a chain to form

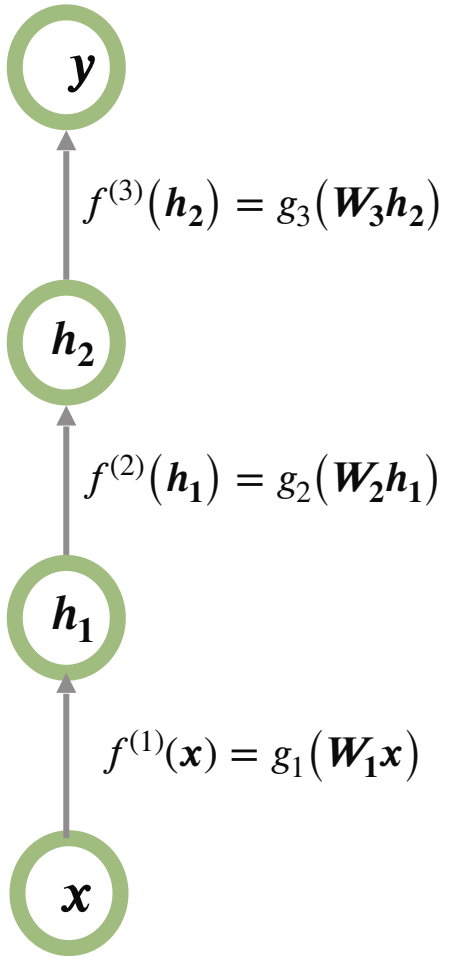
$$f(\boldsymbol{x}) = f^{(3)}\Big(f^{(2)}\big(f^{(1)}(\boldsymbol{x})\big)\Big)$$

$f^{(1)}$ is the first layer, $f^{(2)}$ the second, etc.

The length of the chain is the depth of the model

The name deep learning arises from this terminology

Final layer of a feedforward network, (here $f^{(3)}$) is the output layer



What are Hidden Layers

- Behavior of hidden layers is not directly specified by the data
- Learning algorithm must decide how to use those layers to produce a final value that is close to y
- Training data does not say what individual layers should do
  - This is one of the main tasks: Define the appropriate structure of the network and the hidden layers
- Since the desired output for these layers is not shown, they are called hidden layers

Extending Linear Models

- To represent non-linear functions of $\boldsymbol{x}$
- Apply linear model to transformed input $\phi(\boldsymbol{x})$ with non-linear ϕ

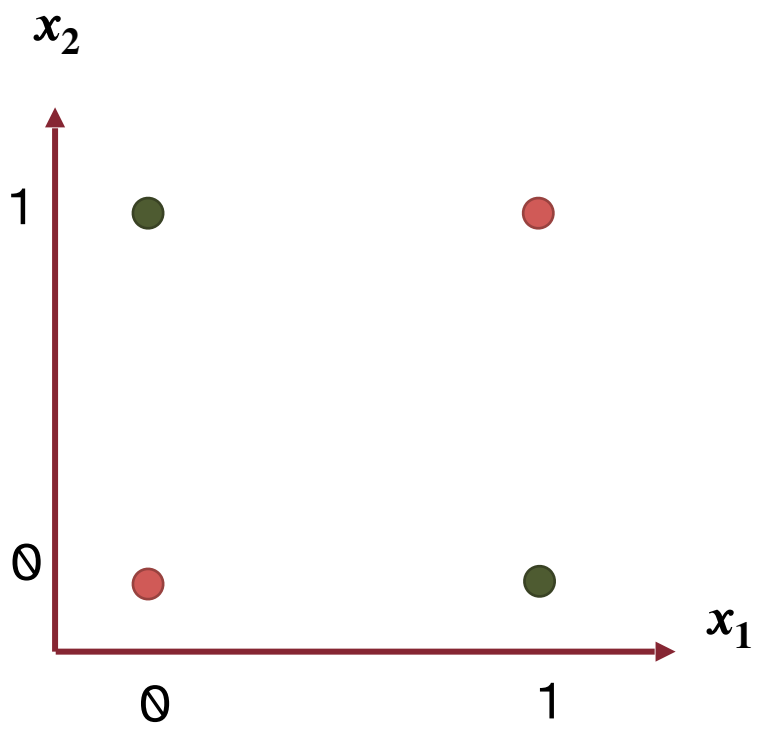
  Equivalently kernel trick of SVM obtains nonlinearity
- Function is nonlinear wrt x but linear wrt $\phi(x)$

  We can think of ϕ as providing a set of features resulting in a new representation for x
- Get ϕ
  - Generic functions: There exists a set of useful kernel functions
  - Manually engineer ϕ: Laborious and not transformable
  - **Deep Learning**: Learn ϕ

On the importance of Activation Functions

- The XOR Problem:
  - XOR is a logical function
  - It takes two inputs, x_1 and x_2

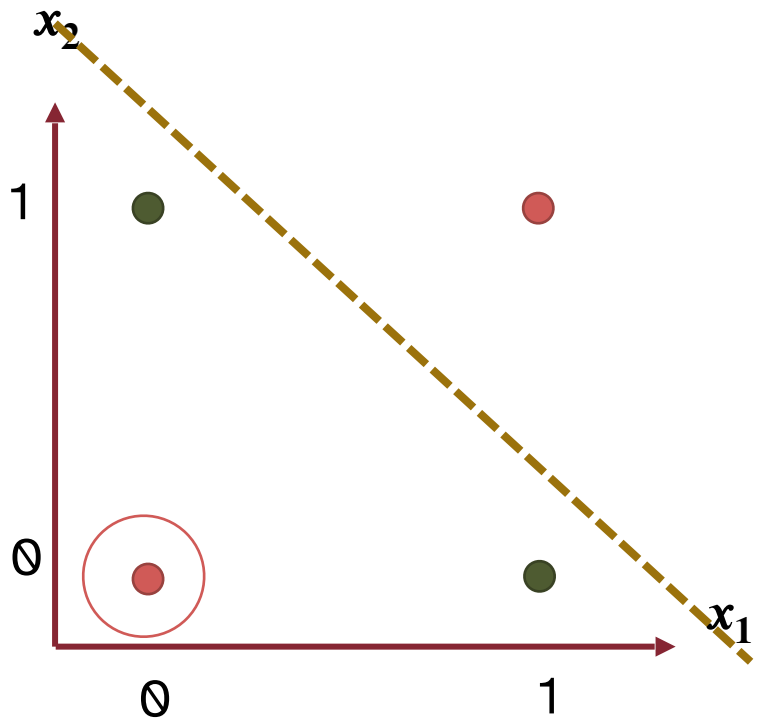
Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0



On the importance of Activation Functions

- The XOR Problem:
  - XOR is a logical function
  - It takes two inputs, x_1 and x_2
- **The problem is not linearly separable**

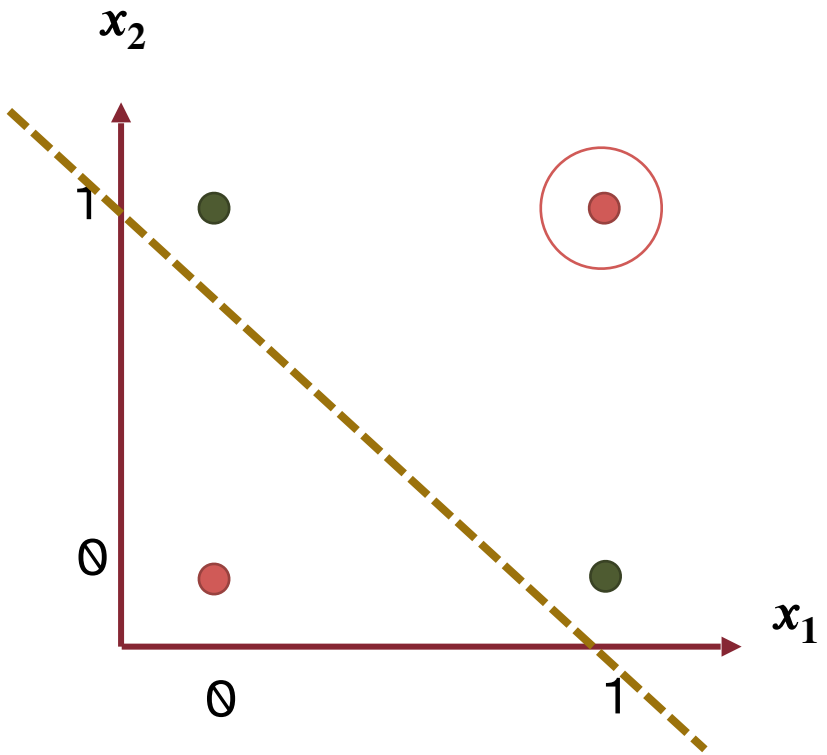
Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0



On the importance of Activation Functions

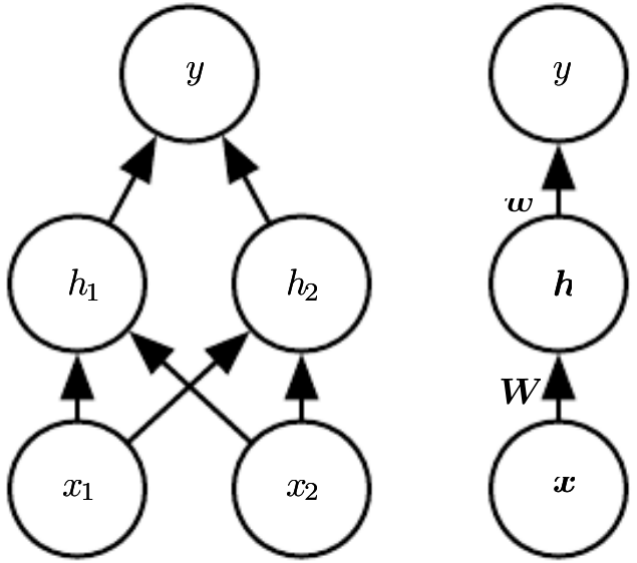
- The XOR Problem:
  - XOR is a logical function
  - It takes two inputs, x_1 and x_2
- **The problem is not linearly separable**

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0



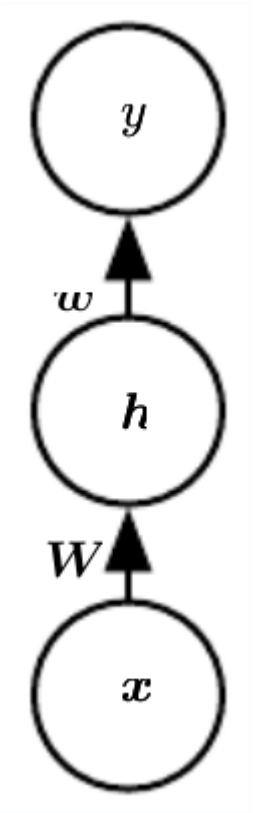
The XOR Problem

- We are trying to approximate the XOR function
  - Our training points: $X = \left\{ [0,0]^T, [1,0]^T, [0,1]^T, [1,1]^T \right\}$
  - As we have seen, linear separation is not possible!
- **Now lets use a FFN**
  - One hidden layer, containing two units
  - Both representations are equivalent
    - Matrix $\boldsymbol{W}$ describes mapping $\boldsymbol{x}$ to $\boldsymbol{h}$
    - Vector $\boldsymbol{w}$ describes mapping $\boldsymbol{h}$ to y
    - (biases omitted)



What is computed

- Layer 1 (hidden layer): vector of hidden units $\boldsymbol{h}$ computed by function $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, c)$ with bias c
- Layer 2 (output layer) computes: $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$
  - $\boldsymbol{w}$ are the weights (it is basically a linear regression)
  - Output is linear regression applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$
- Complete model is $f(\boldsymbol{x}; \boldsymbol{W}, c, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, c), \boldsymbol{w}, b)$



Activation Function

- If we choose both $f^{(1)}$ and $f^{(2)}$ to be linear, the total function will still be linear; which is insufficient (as already seen)

- Therefore, we need an **Activation Function**:
- Remember, our function looks as follows:

$$y = \boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{w} + b$$

- Activation function g is typically chosen to be applied element-wise:

$$h_i = g(x^T W_{:i} + c_i)$$

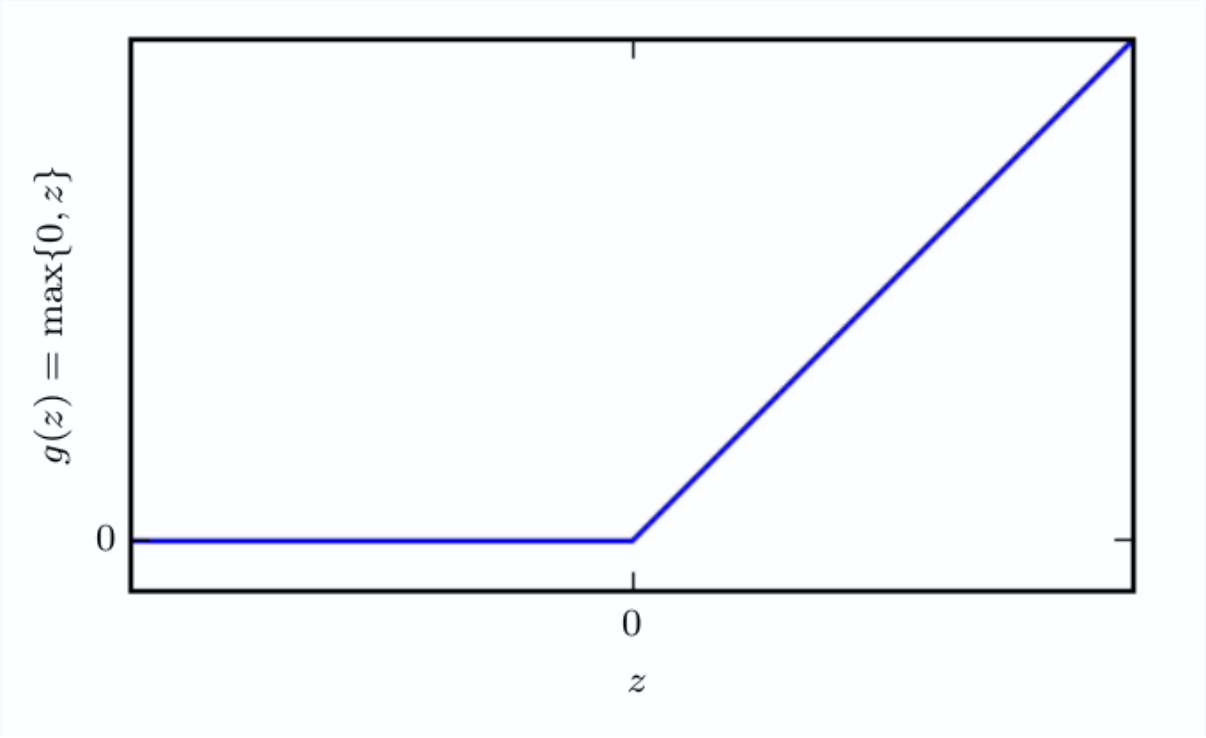
Default Activation Function: ReLU

➢ Activation function:

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

✳ This already yields to a non-linear transformation

- ✳ But still piecewise linear
- ✳ Preserves couple of nice properties making gradient-based learning easy
- ✳ Generalizes well



Back to our XOR Problem

➢ The complete equation is now:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\left\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\right\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

Back to our XOR Problem

➢ The complete equation is now:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Back to our XOR Problem

➢ The complete equation is now:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \qquad \boldsymbol{XW} + \boldsymbol{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Back to our XOR Problem

➢ The complete equation is now:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

$$\boldsymbol{XW} + \boldsymbol{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\max\{0, \boldsymbol{XW} + \boldsymbol{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Back to our XOR Problem

➢ The complete equation is now:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}\} + b$$

Lets use the following weights:

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0 \text{ and } \boldsymbol{X} = \begin{bmatrix} 0101 \\ 0011 \end{bmatrix}^T$$

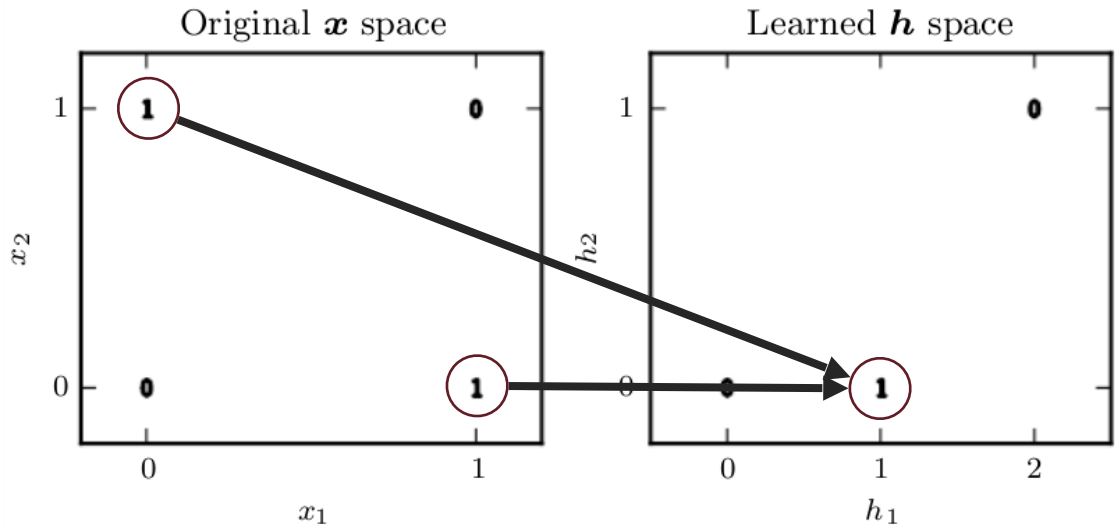
$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

$$\boldsymbol{XW} + \boldsymbol{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\max\{0, \boldsymbol{XW} + \boldsymbol{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

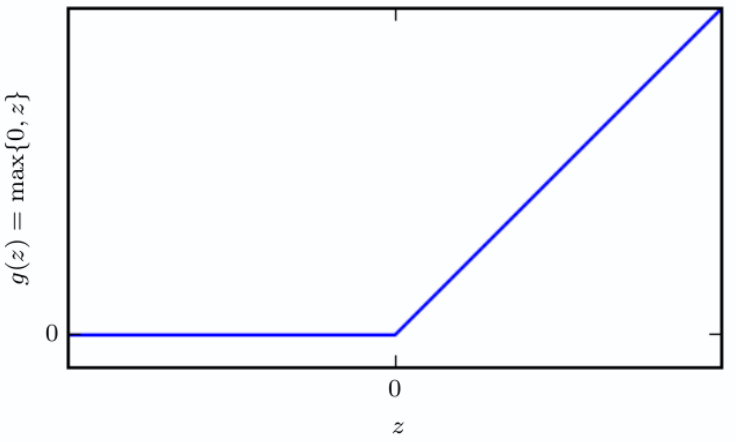
$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \boldsymbol{w} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{1} \\ \mathbf{0} \end{bmatrix}$$

“Learned XOR”



- Two points that must have output 1 have been collapsed into one
- Can be separated in a linear model:
  - For fixed h_2 output has to increase in h_1
- Of course, we “cheated”: We have “guessed” the parameters correctly which lead to the desired result
- In reality: Millions of parameters which we have to actually learn

Interim Summary



- Linear models cannot solve the X-OR (exclusive-or) problem
- Composition of linear models are still Linear, due to associativity.

  In other words, we could rewrite $\mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x}$ as we could set

  $\mathbf{W}' := \mathbf{W}^{(1)} \cdot \mathbf{W}^{(2)}$,

  s.th. $\mathbf{W}'x = \mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x}$ are equivalent.
- Therefore, we need to add **nonlinear activation functions** in-between
- A simple but effective nonlinearity is the rectified linear unit (**ReLU**)

  ReLU(x) = max(0, x), i.e., x if x > 0, else 0

Keywords

- Single-Layer Networks
- Deep Neural Networks
- **Universal Approximation Theorem**
- Activation functions
- Output units

On the Power of Neural Networks

A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. But:

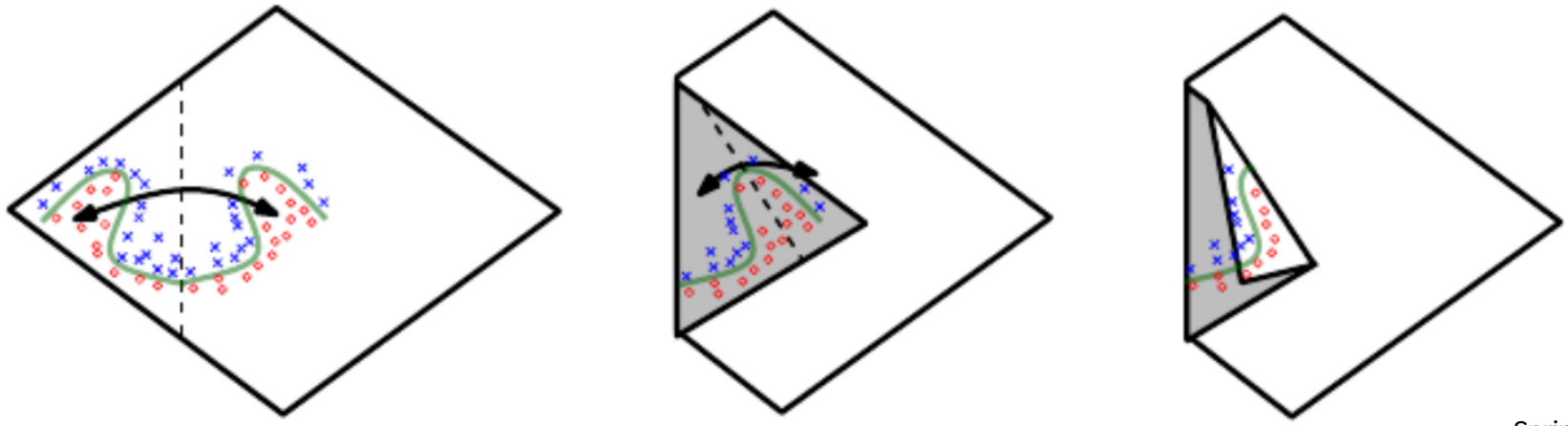
The **universal approximation theorem** states that a feedforward neural network with at least one hidden layer and a nonlinear activation function can approximate any compact, continuous function to an arbitrary degree of accuracy (depending on the hidden layer's width).

Implication of Theorem

- Whatever function we are trying to learn, a sufficiently large MLP will be able to represent it
- However we are not guaranteed that the training algorithm will learn this function
- No Free Lunch: There is no universal procedure for examining a training set of samples and choosing a function that will generalize to points not in training set

Advantage of deeper networks

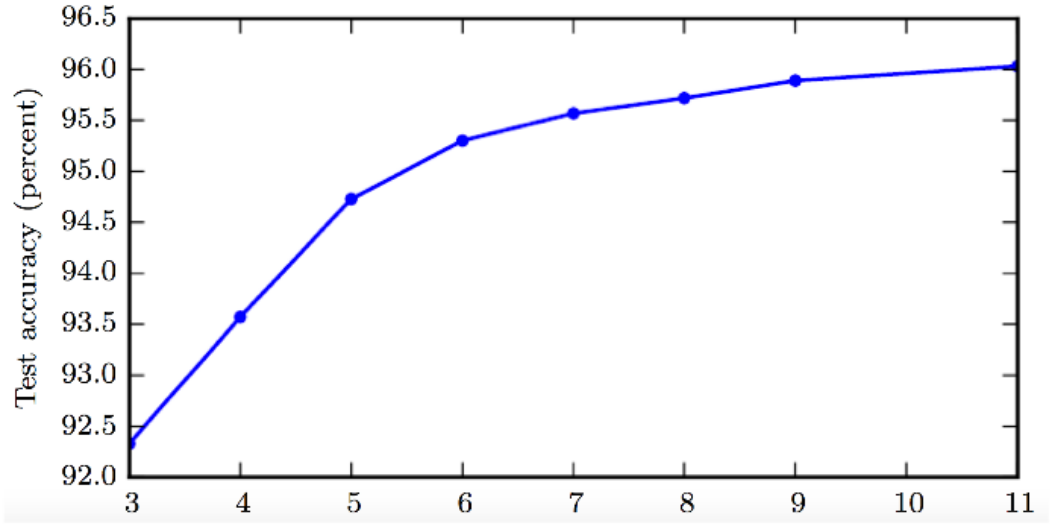
- Absolute value rectification creates mirror images of function computed on top of some hidden unit, wrt the input of that hidden unit.
- Each hidden unit specifies where to fold the input space to create mirror responses.
- By composing these folding operations, we obtain an exponentially large no. of piecewise linear regions which can capture all kinds of repeating patterns



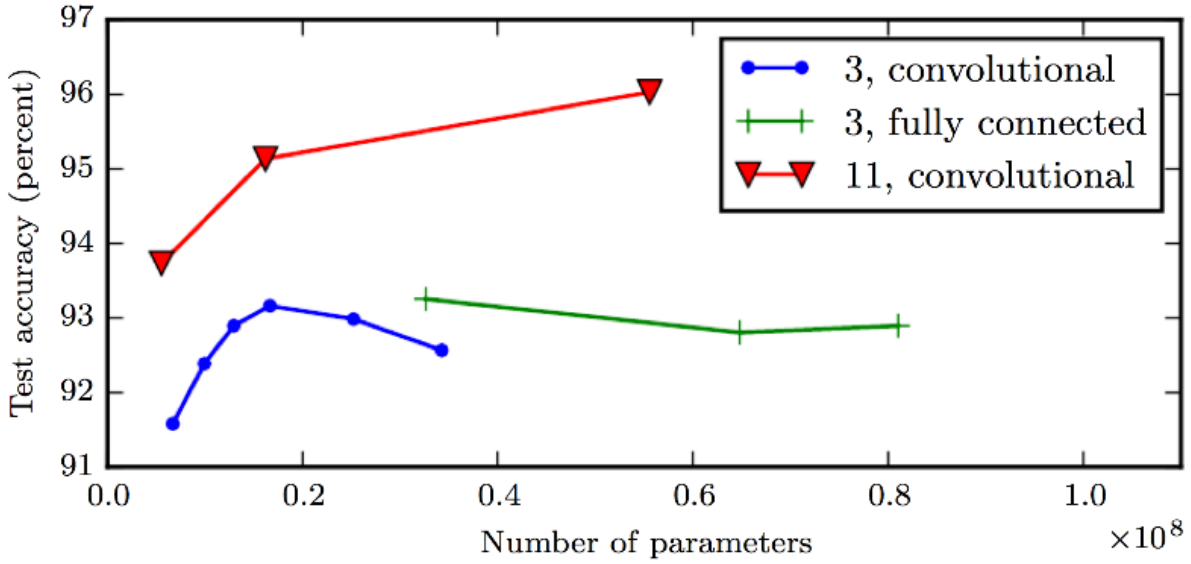
Intuition of Depth

- Any time we choose a ML algorithm we are implicitly stating a set of beliefs about what kind of functions that algorithm should learn
- Choosing a deep model encodes a belief that the function should be a composition several simpler functions
- The learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation.
- Empirically: Deeper Networks perform better ☺

Deeper Networks



Test accuracy consistently increases with depth



Increasing parameters without increasing depth is not as effective

Keywords

- Single-Layer Networks
- Deep Neural Networks
- Universal Approximation Theorem
- **Activation functions**
- Output units

General Construction of a Hidden Unit

- Accepts a vector of inputs $\boldsymbol{x}$ and computes an affine transformation $\boldsymbol{h}' = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$
- Computes an element-wise non-linear function $g(\boldsymbol{h}')$
- Most hidden units are distinguished from each other by the choice of activation function $g(\boldsymbol{h}')$
  - We look at: ReLU, Sigmoid and tanh, and other hidden units

Choice of Hidden Unit

- We now look at how to choose the type of hidden unit in the hidden layers of the model
- Design of hidden units is an active research area that does not have many definitive guiding theoretical principles
- ReLU is an excellent default choice
- But there are many other types of hidden units available
- When to use which kind?
  - Hard to predict in advance which will work best for a problem at hand
  - Design process is trial and error

Logistic Sigmoid

- Prior to introduction of ReLU, most neural networks used logistic sigmoid activation

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

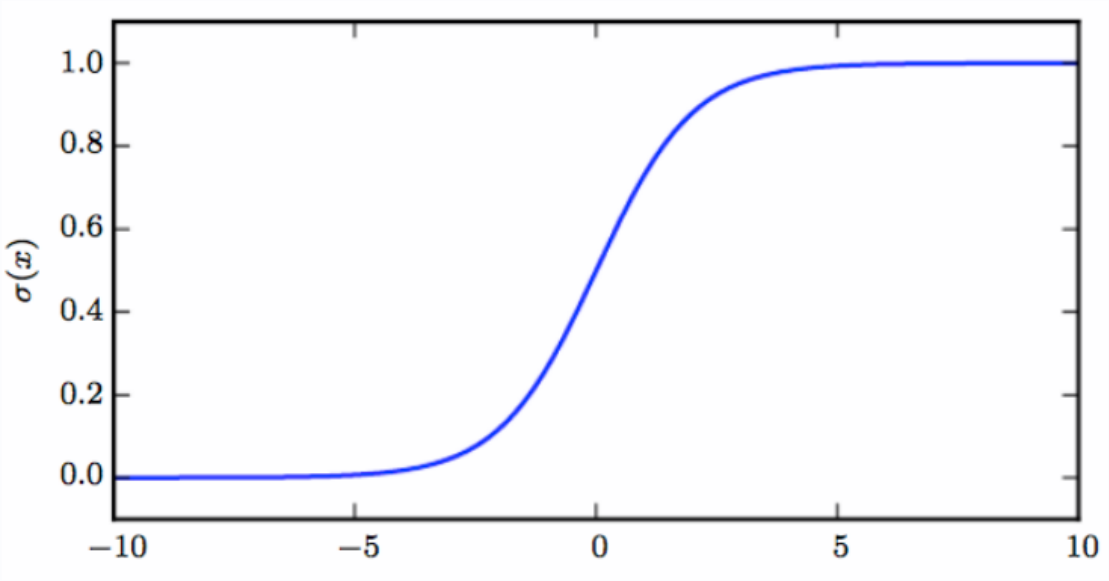
- Or the hyperbolic tangent

$$g(z) = \tanh(z)$$

- These activation functions are closely related because

$$\tanh(z) = 2\sigma(2z) - 1$$

- Sigmoid units are used to predict probability that a binary variable is 1



Other Activation Functions

✳ Cosine

$$\boldsymbol{h} = \cos\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right)$$

- ✳ Feedforward networks on MNIST obtained error rate of less than 1%

✳ Radial Basis

$$h_i = \exp\left(-\frac{1}{\sigma^2}\left\|\boldsymbol{W}_{:j} - \boldsymbol{x}\right\|^2\right)$$

- ✳ Becomes more active as $\boldsymbol{x}$ approaches a template $W_{:,i}$

✳ Softplus

$$g(a) = \zeta(a) = \log\left(1 + e^a\right)$$

- ✳ Smooth version of the rectifier

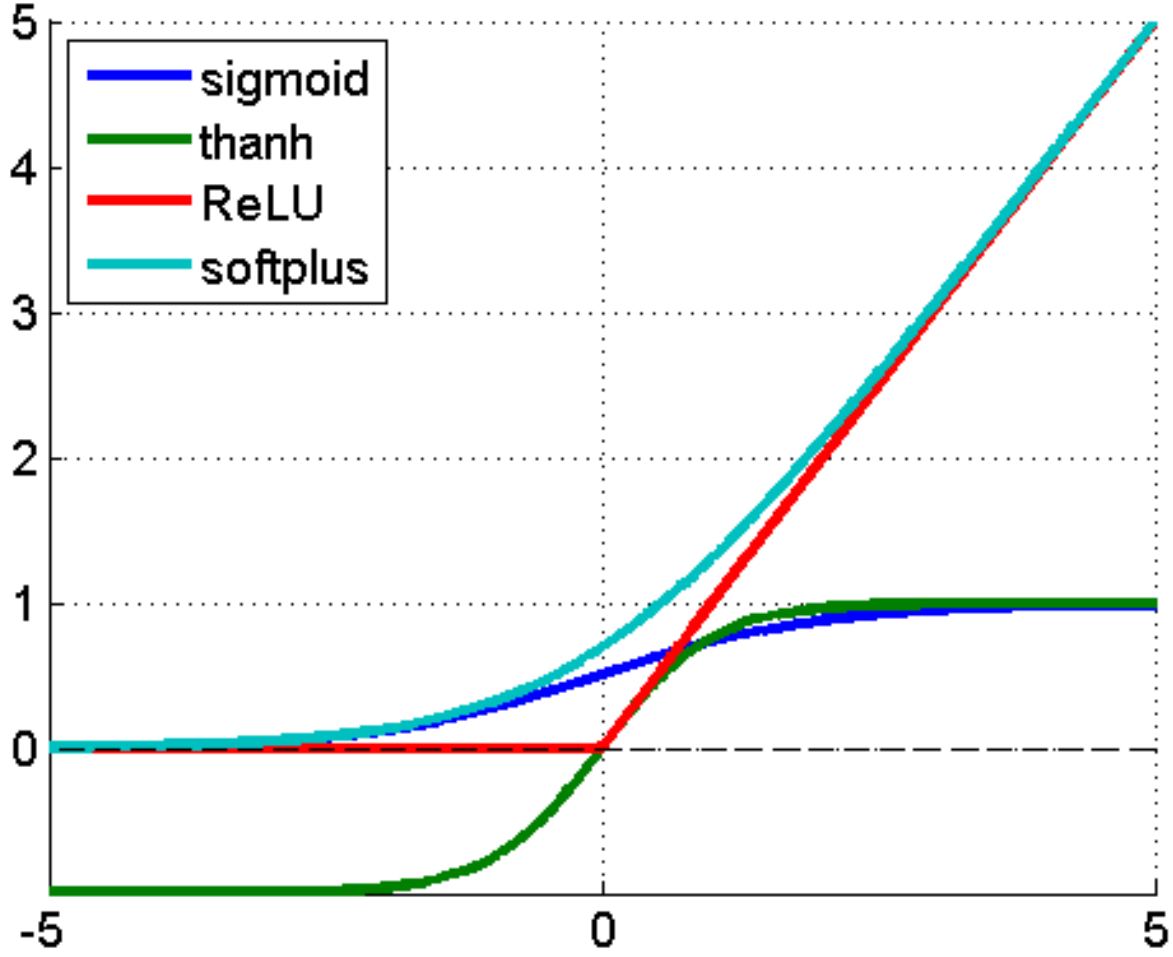
✳ Hard tanh

$$g(a) = \max(-1, \min(1, a))$$

- ✳ Shaped similar to tanh and the rectifier but it is bounded

Sigmoid Saturation

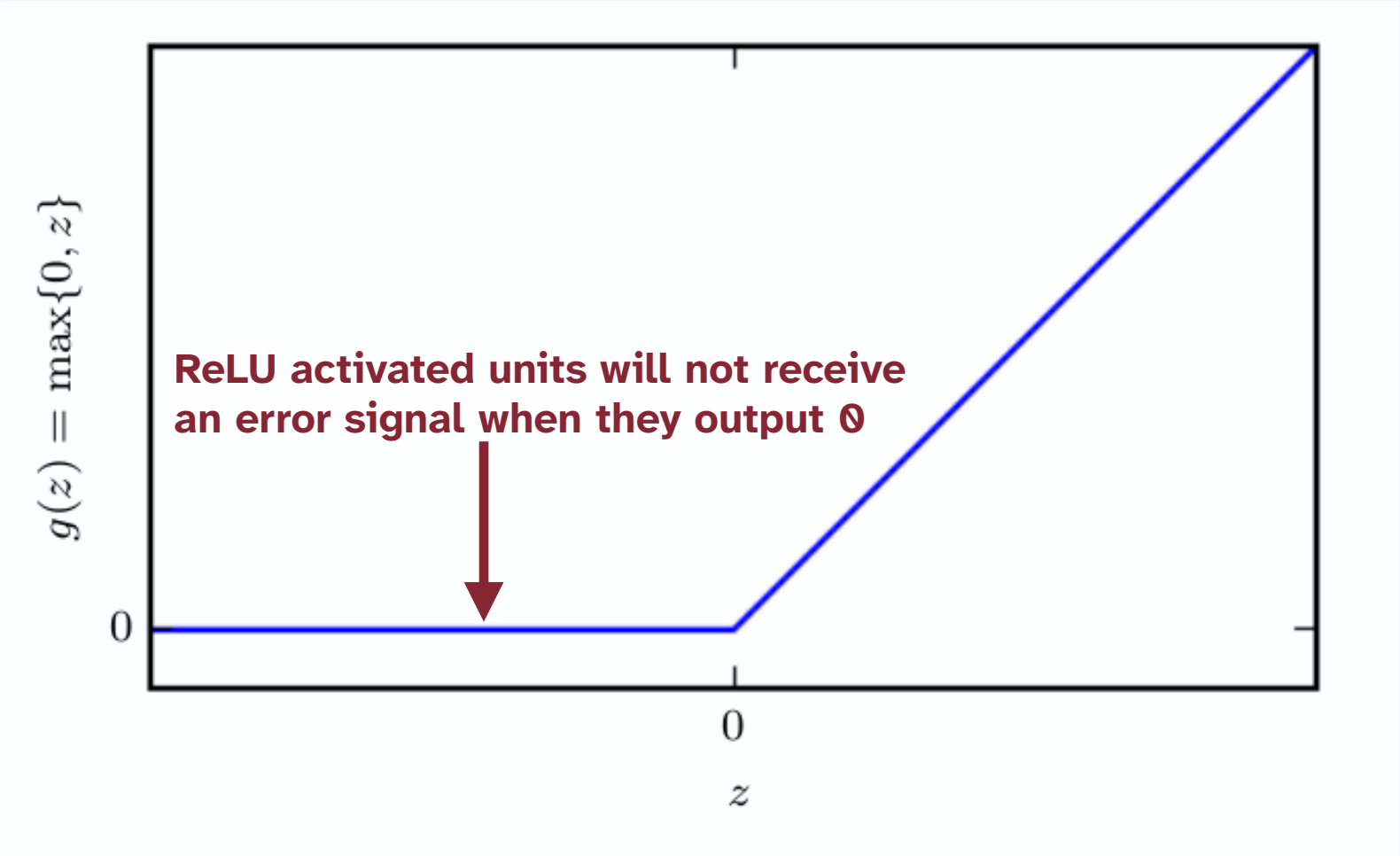
- Sigmoidals saturate
- ReLU and softplus increase for input >0
- As networks get very deep, this linear increase of the ReLU in the positive domain is important, such that the error signals don't get too small in the backward pass. Sigmoid units would learn **much** slower.



But is ReLU the end of the story?

$$\boldsymbol{h} = g\left(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}\right)$$

$$g(x) = \max\{0, x\}$$



Generalizations of ReLU

- Three methods based on using a non-zero slope α_i when $z_i < 0$:

  $h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$

  or equivalently: $g(z, \alpha)_i = z_i$ **if** $z_i > 0$ **else** $\alpha_i \cdot z_i$

- Absolute-value rectification:
  - fixes $\alpha_i = -1$ to obtain $g(z) = |z|$
- Leaky ReLU:
  - fixes α_i to a small value like 0.01
- Parametric ReLU or PReLU
  - treats α_i as a learnable parameter

Dead ReLU's and how to do better: ELU

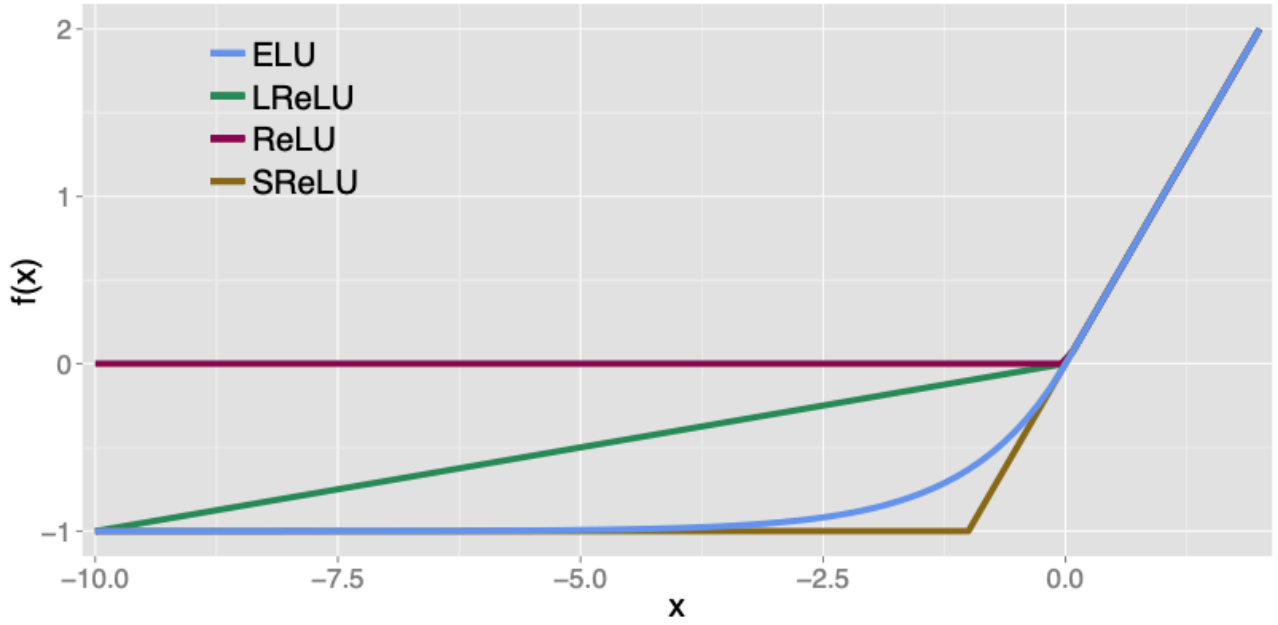
- **ReLU is a good default choice**
- **But**: ReLU's don't receive any gradient (error signal) when they output 0
- It can happen that they output 0 for all possible inputs – then you have **dead ReLU's**
- Variants that mitigate this problem:
  - **LeakyReLU**:

    $h_i = z_i$ if $z_i > 0$ else αz_i

    (with small positive α)
  - **Exponential Linear Unit (ELU)**, better than LeakyReLU

The *exponential linear unit* (ELU) with $0 < \alpha$ is

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

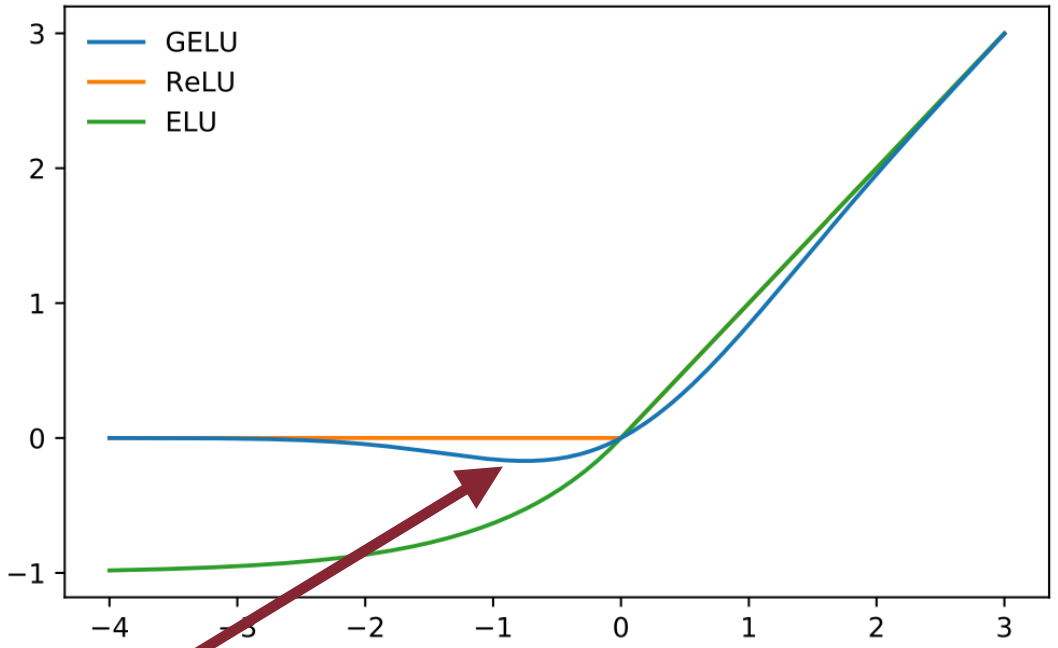


From arXiv:1511.07289v5

Dead ReLU's and how to do even better: GELU

- Popular in Transformers is the Gaussian Error Linear Unit (GELU)
- Exceeds performance of ReLU and ELU

From arXiv:1606.08415v5



Idea: GELU scales activations by how much larger they are than other activations, assuming N(0,1) distribution

$$\mathrm{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \mathrm{erf}(x/\sqrt{2})\right]$$

We can approximate the GELU with

$$0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$$

or

$$x\sigma(1.702x),$$

Leads to a smooth "bump" just before zero

Pattern also seen in other recent activation functions: x multiplied by logistic sigmoid of x

Summary of activation functions

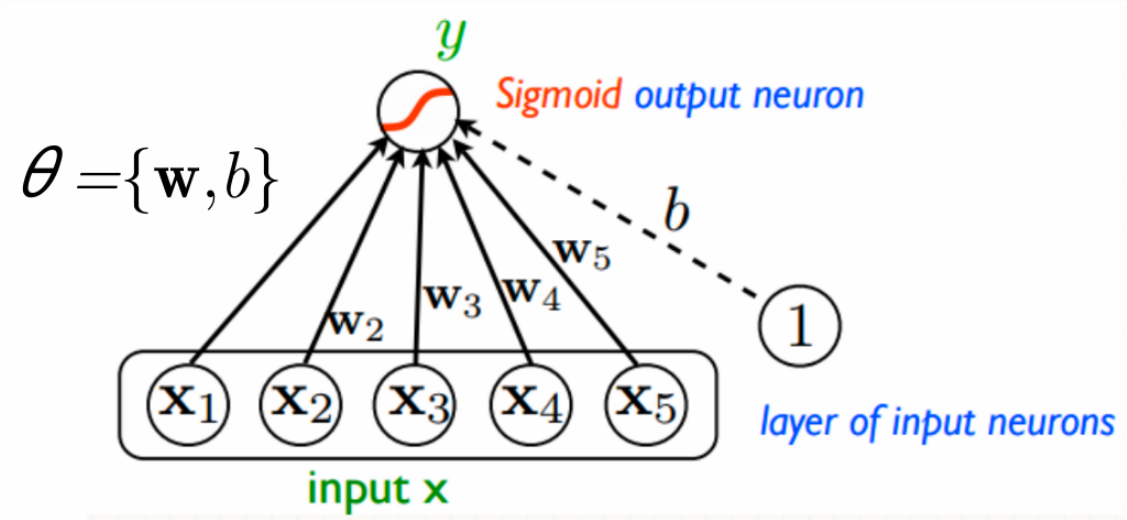
- Sigmoidal functions were used traditionally
- **But** they saturate and make learning difficult – **Sigmoid Saturation**
- **Therefore** ReLU's, which don't saturate on the positive side
- **But** ReLU's don't receive an update when they output 0 – **Dead ReLU's**
- **Therefore** do allow small, downscaled negative values (e.g., **LeakyReLU**)
- **But** we don't want them to become too negative, else they will have difficulty to re-activate
- **Therefore** have them saturate on the negative range (e.g., **ELU**)
- **And** often used recently, introducing a small bump close to zero (e.g., **GELU**)

Keywords

- Single-Layer Networks
- Deep Neural Networks
- Universal Approximation Theorem
- Activation functions
- **Output units**

Output Units

- The choice of cost function is **tightly** coupled with the choice of output unit (later...)
- Most of the time we use cross-entropy between data distribution and model distribution (later ...)
- Choice of how to represent the output then determines the form of the cross-entropy function



Role of an Output Unit

- Any output unit is in principle also usable as a hidden unit
- A feedforward network provides a hidden set of features

$$\boldsymbol{h} = f\left(\boldsymbol{x}; \boldsymbol{\theta}\right)$$

- Role of output layer is to provide some additional transformation from the features to the task that network must perform
- Common Output Units
  - **Linear units**: no non-linearity (used for Regressions)
  - **Sigmoid units** each individual output is between 0 and 1 (used for many-out-of-K classification)
  - **Softmax units** each individual output is between 0 and 1 **and** all outputs together sum up to 1 (e.g., used for one-out-of-K classification)

Linear Units for Regressions

- Linear unit: simple output based on affine transformation with no nonlinearity
- Given features h, a layer of linear output units produces a vector
  $\hat{\boldsymbol{y}} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$
- Linear units are often used to produce mean $\hat{\boldsymbol{y}}$ of a conditional Gaussian distribution
- $P(\boldsymbol{y} \mid \boldsymbol{x}) = N(\boldsymbol{y}; \hat{\boldsymbol{y}}, \sigma^2)$
  We will see later why this is interesting

Most commonly used for regression:
Mean squared error loss (MSELoss)

Sigmoid Units for Binary Classification

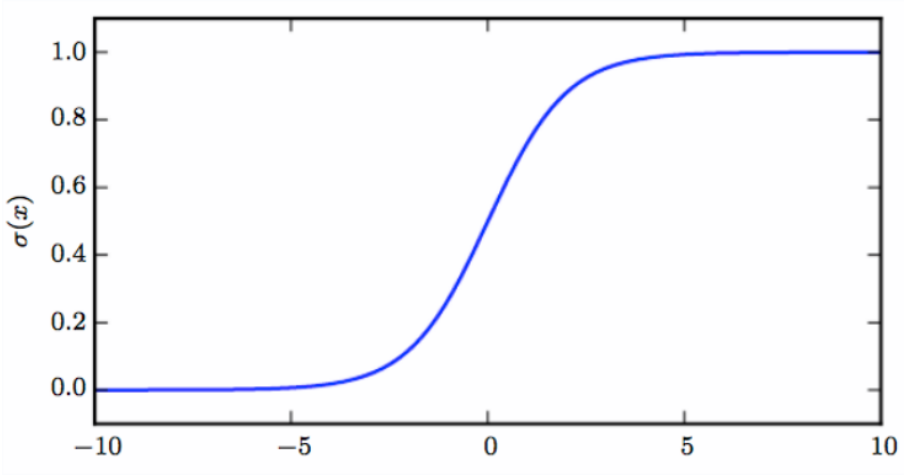
- Task of predicting value of binary variable y
  - Classification problem with two classes
- Maximum likelihood approach is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$
- Neural net needs to predict $p(y = 1 \mid \boldsymbol{x}) \in [0,1]$:

$$P(y = 1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^T \boldsymbol{h} + b\right\}\right\}$$

  - We would define a valid conditional distribution, but cannot train it effectively with gradient descent
  - A gradient of 0: learning algorithm cannot be guided

Binary Cross-Entropy

Sigmoid Units

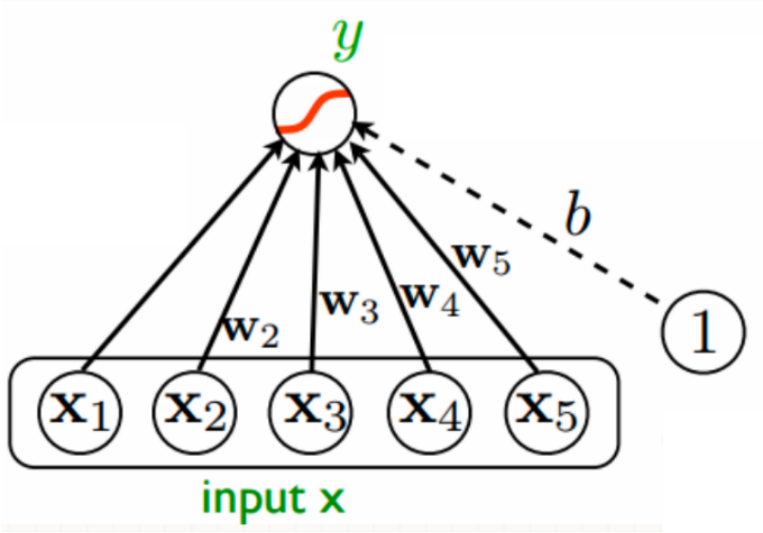


➢ Sigmoid always gives a gradient

$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$

with

$$\sigma(x) = \frac{1}{1 + \exp(-x)} = \frac{e^x}{1 + e^x}$$



➢ Sigmoid Unit has two components:

- ➢ A linear layer to compute

$z = \boldsymbol{w}^T \boldsymbol{h} + b$

- ➢ A sigmoid activation function to convert z into a probability

➢ You will understand these slides later ☺.

Softmax Units for General Classification

➢ Any time we want a probability distribution over a discrete variable with n values we may use the softmax function

➢ Softmax most often used for output of classifiers to represent distribution over n classes

➢ Also, inside the model itself when we wish to choose between one of n options

➢ For the Binary case we have produced a single number

$$\hat{y} = P(y = 1 \mid \boldsymbol{x})$$

➢ Now, we have to produce a vector $\hat{\boldsymbol{y}}$

$$\hat{y}_i = P(y = i \mid \boldsymbol{x})$$

Cross-entropy Loss

➢ You will understand these slides later ☺.

Softmax Defintion

➢ The **softmax function** (aka normalized exponential), is defined as follows:

$$\text{softmax}\left(a_1, \ldots, a_K\right) = \left(\frac{\exp a_1}{\sum_j^K \exp a_j}, \frac{\exp a_2}{\sum_j^K \exp a_j}, \ldots, \frac{\exp a_K}{\sum_j^K \exp a_j} \right)$$

➢ We will learn later more about this function

➢ But it should be visible that it creates a valid distribution
all values are between 0 and 1 and all values sum up to 1

➢ You will understand these slides later ☺.

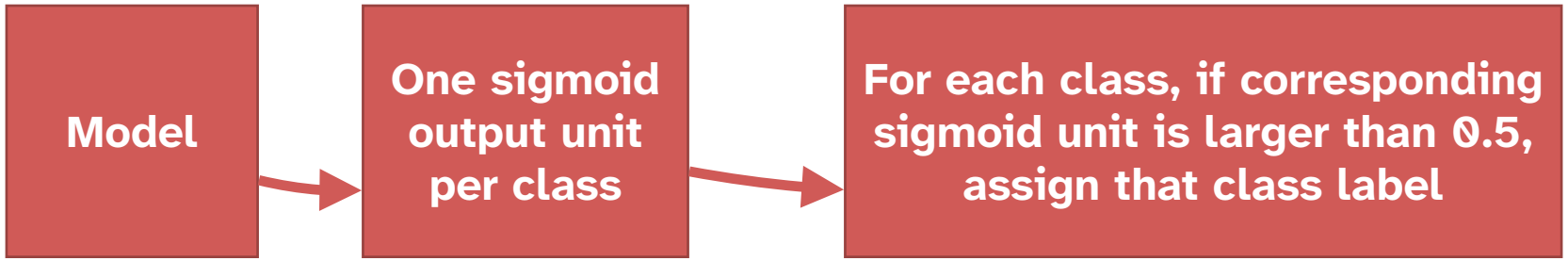
Softmax, even for binary classification

Softmax way often works better in practice!

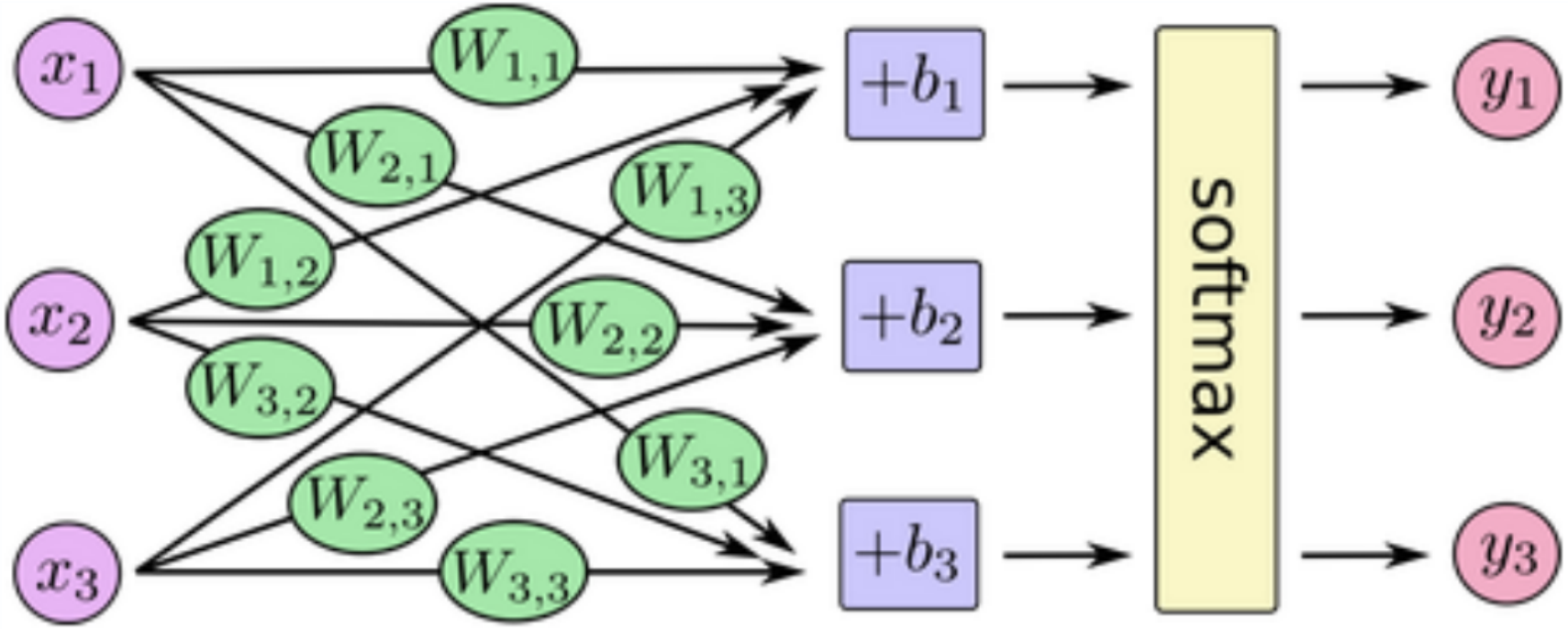
But when do I need sigmoid output units then?

- **Prime example: Multi-label classification**
- Consider the case that you have **M** possible classes.
- And multiple class labels can be assigned to a single example.
- For example, you have a news article, and you need to classify it into topics {Sports, Football, Basketball, Europe, Denmark, Germany, Politics, Economics, Science}

  → **multiple can be correct**
- "A **scientific** investigation of the **football** match between **Denmark** and **Germany**"



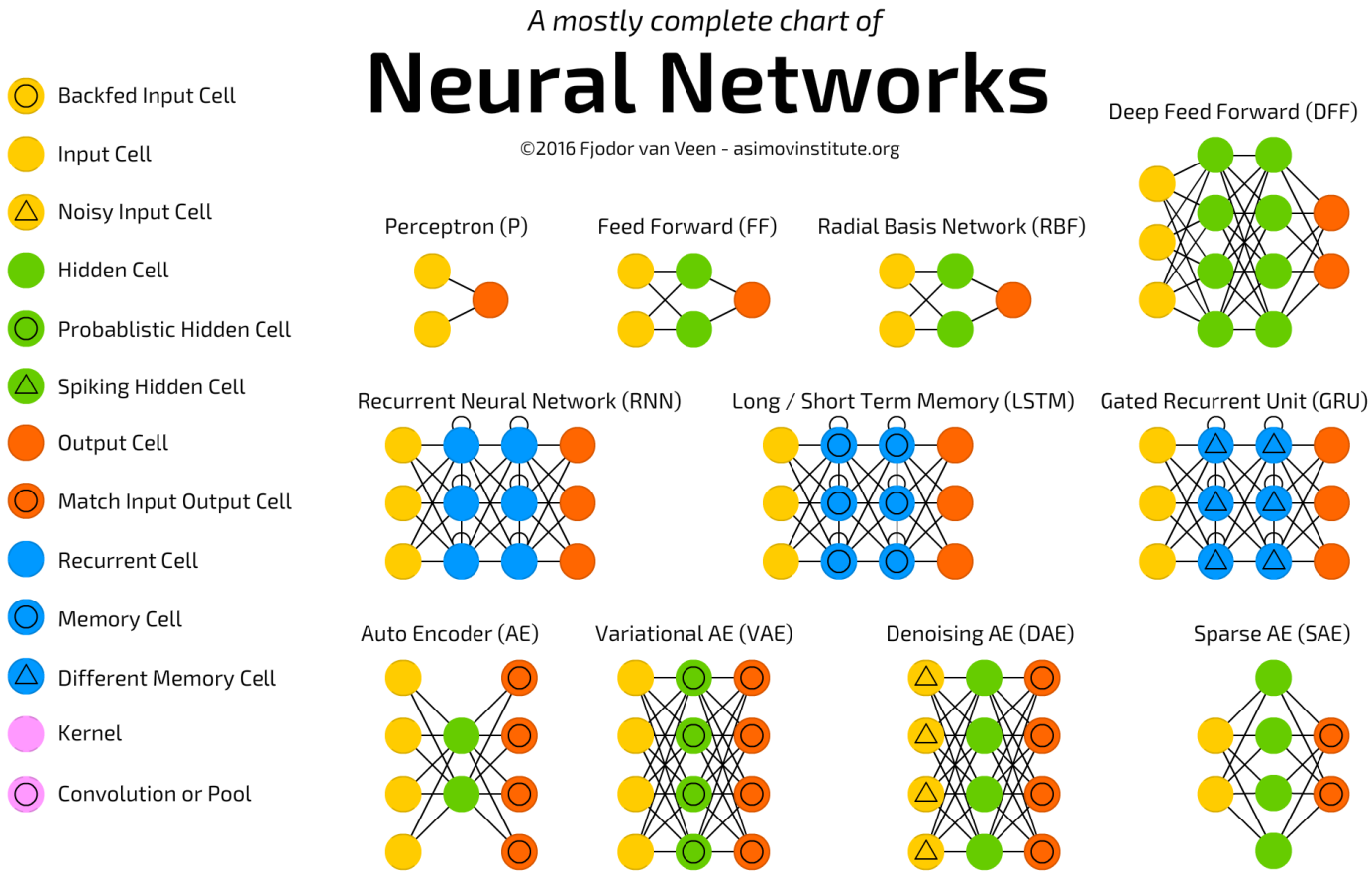
Softmax Regression



Keywords

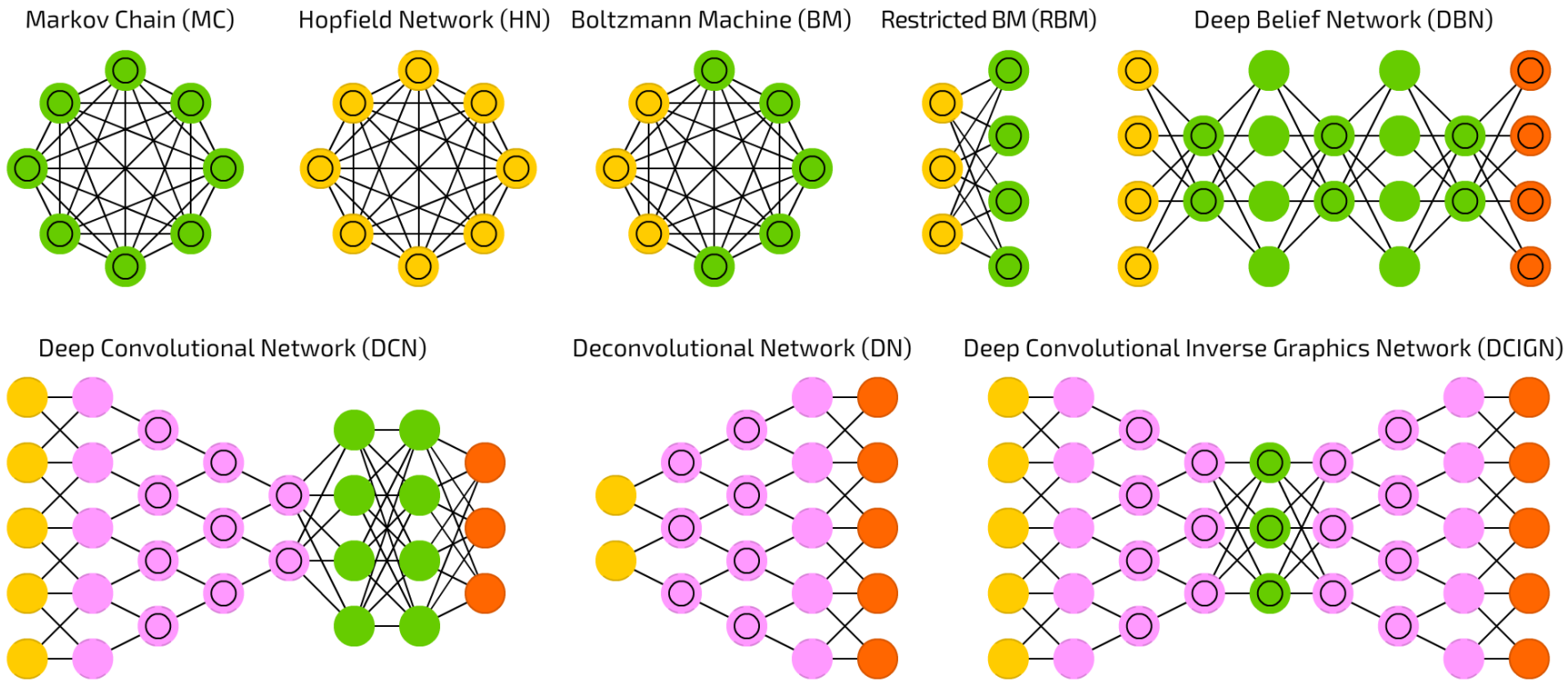
- Single-Layer Networks
- Deep Neural Networks
- Universal Approximation Theorem
- Activation functions
- Output units

A mostly complete chart of Neural Networks



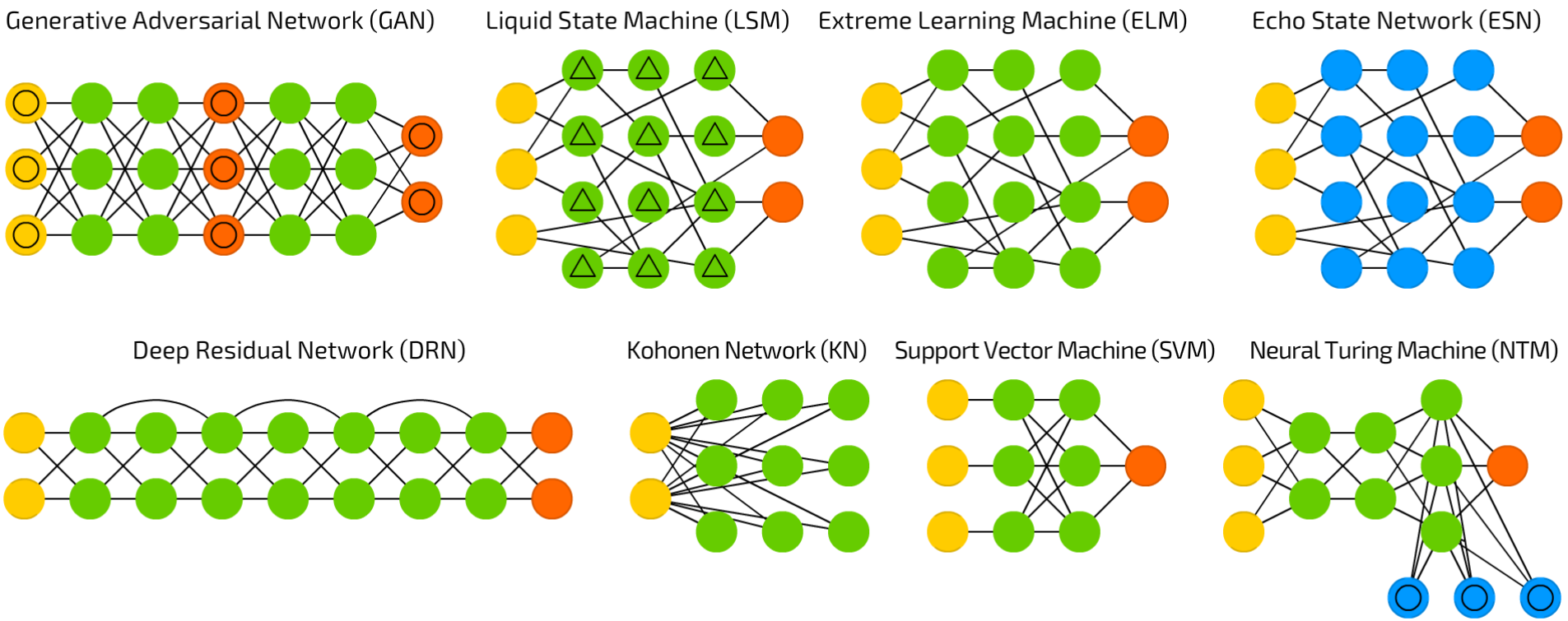
➢ https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

A mostly complete chart of Neural Networks



➢ https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

A mostly complete chart of Neural Networks

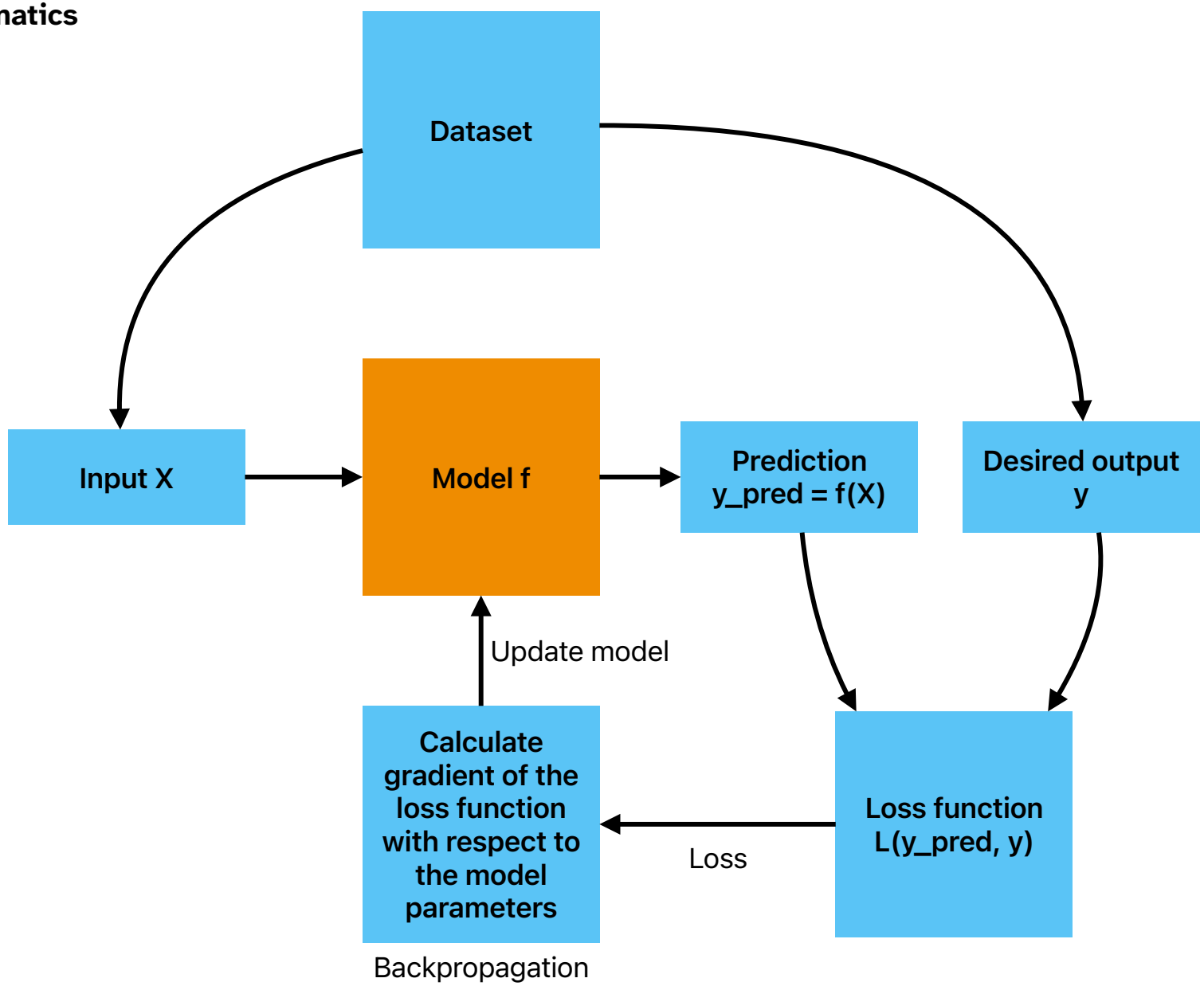


➢ https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

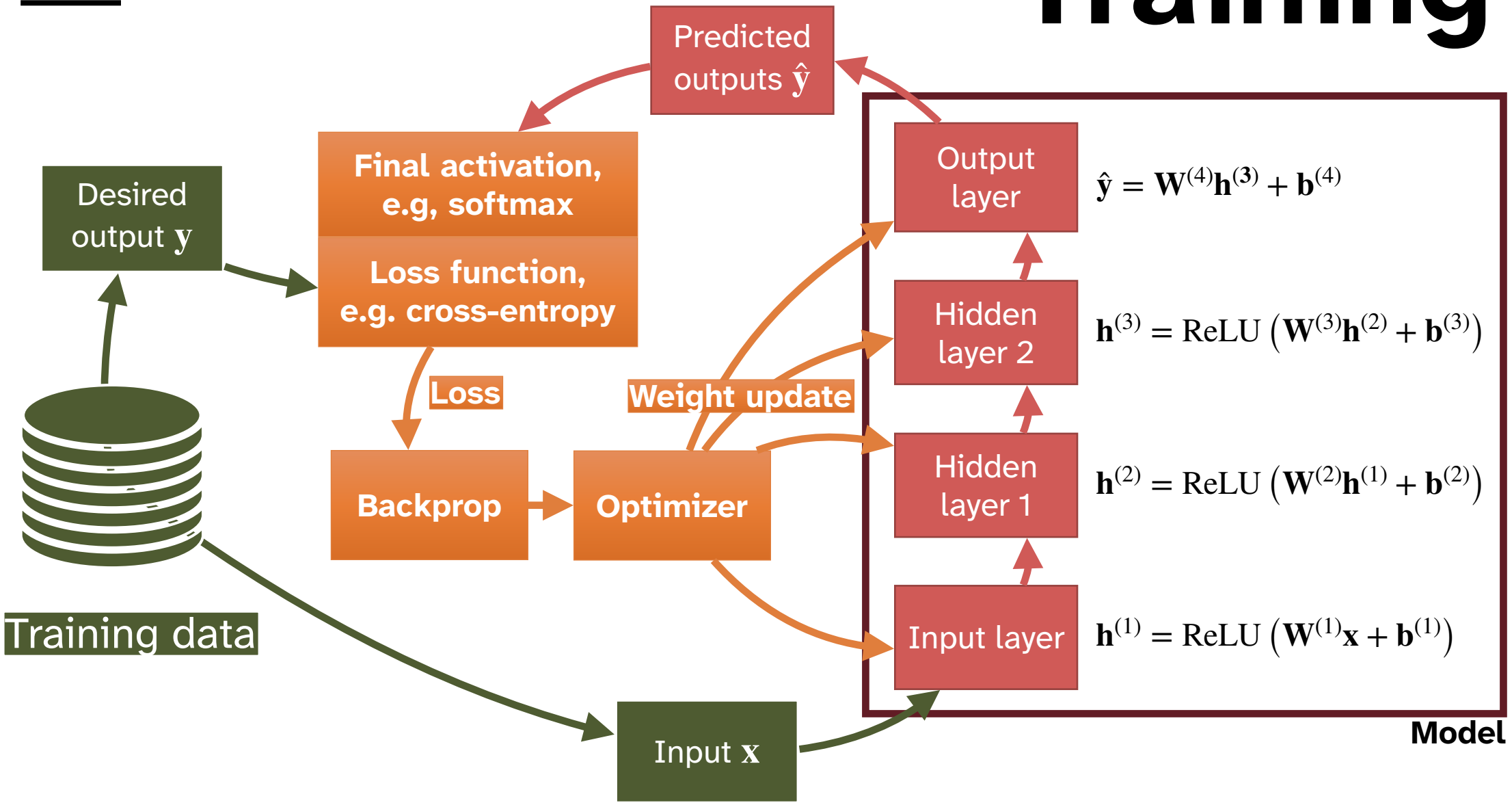
Main Architectural Considerations

1. Choice of depth of network
2. Choice of width of each layer

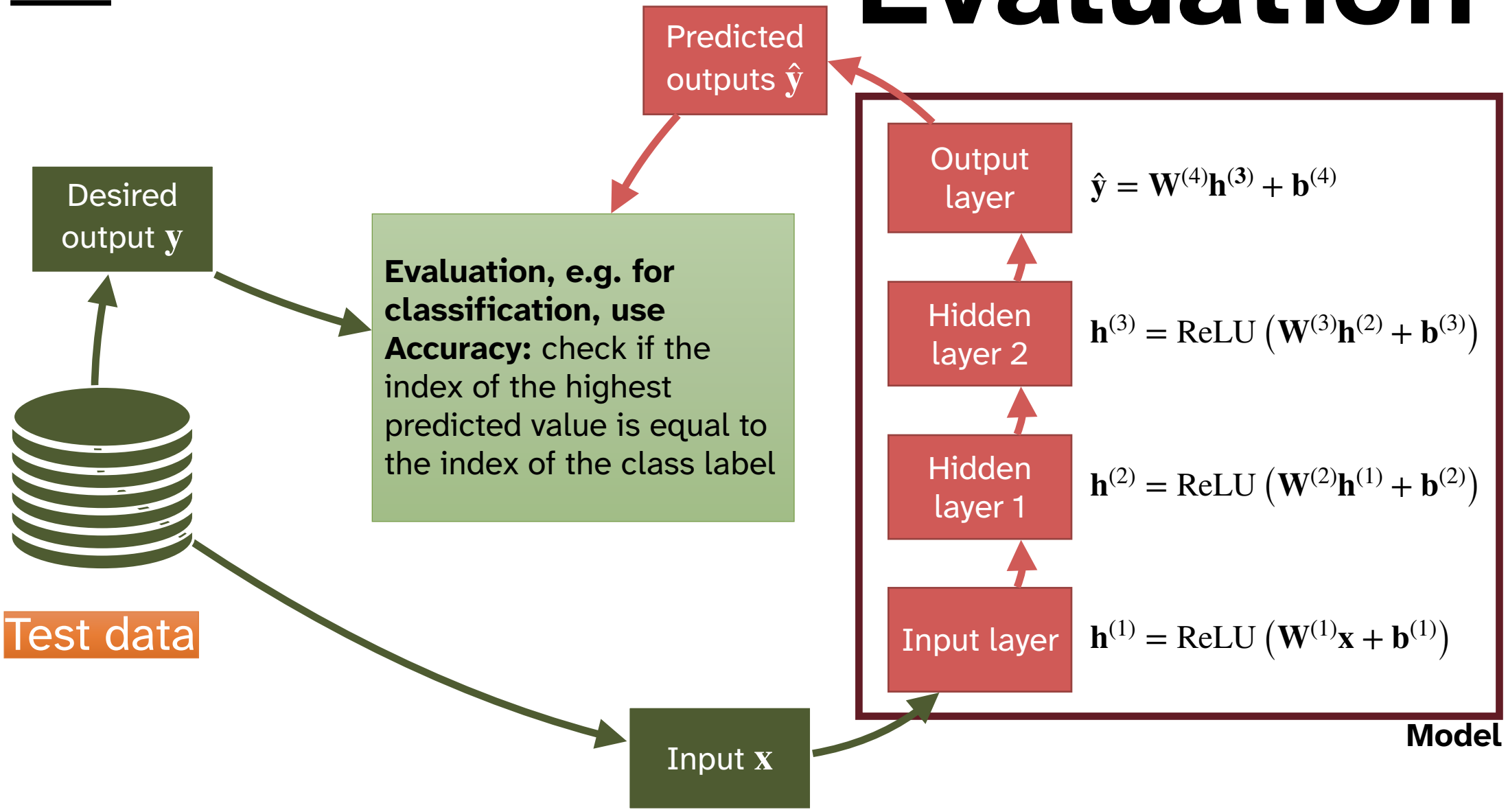
- Deeper networks have
  - Far fewer units in each layer
  - Far fewer parameters
  - Often generalize well to the test set
  - But are often more difficult to optimize
- Ideal network architecture must be found via experimentation guided by validation set error


Dataset
Input X
Model f
Prediction
y_pred = f(X)
Desired output
y
Update model
Calculate
gradient of the
loss function
with respect to
the model
parameters
Loss function
L(y_pred, y)
Loss
Backpropagation

Training


Predicted outputs $\hat{\mathbf{y}}$
Final activation, e.g, softmax
Loss function, e.g. cross-entropy
Desired output $\mathbf{y}$
Loss
Weight update
Backprop
Optimizer
Training data
Input $\mathbf{x}$
Output layer
Hidden layer 2
Hidden layer 1
Input layer
$\hat{\mathbf{y}} = \mathbf{W}^{(4)}\mathbf{h}^{(3)} + \mathbf{b}^{(4)}$
$\mathbf{h}^{(3)} = \mathrm{ReLU}\left(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}\right)$
$\mathbf{h}^{(2)} = \mathrm{ReLU}\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$
$\mathbf{h}^{(1)} = \mathrm{ReLU}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$
Model

Evaluation


Predicted outputs $\hat{\mathbf{y}}$
Desired output $\mathbf{y}$
Evaluation, e.g. for classification, use Accuracy: check if the index of the highest predicted value is equal to the index of the class label
Output layer
$\hat{\mathbf{y}} = \mathbf{W}^{(4)}\mathbf{h}^{(3)} + \mathbf{b}^{(4)}$
Hidden layer 2
$\mathbf{h}^{(3)} = \text{ReLU}\left(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}\right)$
Hidden layer 1
$\mathbf{h}^{(2)} = \text{ReLU}\left(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$
Input layer
$\mathbf{h}^{(1)} = \text{ReLU}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$
Model
Test data
Input $\mathbf{x}$

Training loop (pseudo-PyTorch)


```
# […] model & training_set defined
loss_function = nn.CrossEntropyLoss() # define the loss
function, Softmax activation included
optimizer = optim.SGD(model.parameters(), lr=0.001)  #
Connects your optimizer with parameters
# Inside training loop
For each x with label y in training_set:
  model.zero_grad() # Resets gradient to zero - important!
  y_hat = model(x)  # Feed data into the model
  loss = loss_function(y_hat, y) # Calculate the loss
  loss.backward()  # Back-propagate the error
  optimizer.step()  # Update model weights
```

Coding example

✳ example_xor.py

Take-home

- Linear models – even multi-layered – are not enough to solve a simple XOR problem.
- **Universal Approximation**; already with a few hidden layers and nonlinearities, a neural network model can **approximate any compact function to an arbitrary degree of accuracy**
- **Loss function** and **output units** are tightly coupled and typically determined by the task
- **Hidden units**: Rectified Linear Units are a good choice, yet there are some variants that tackle its shortcomings (dead ReLU's)
- **Depth** and **Width** of a neural network are important for its performance
- **The basic training loop – will be picked up again in the exercises**

Keywords

- Single-Layer Networks
- Deep Neural Networks
- Universal Approximation Theorem
- Activation functions
- Output units

How’s my teaching?

https://www.admonymous.co/lukasgalke