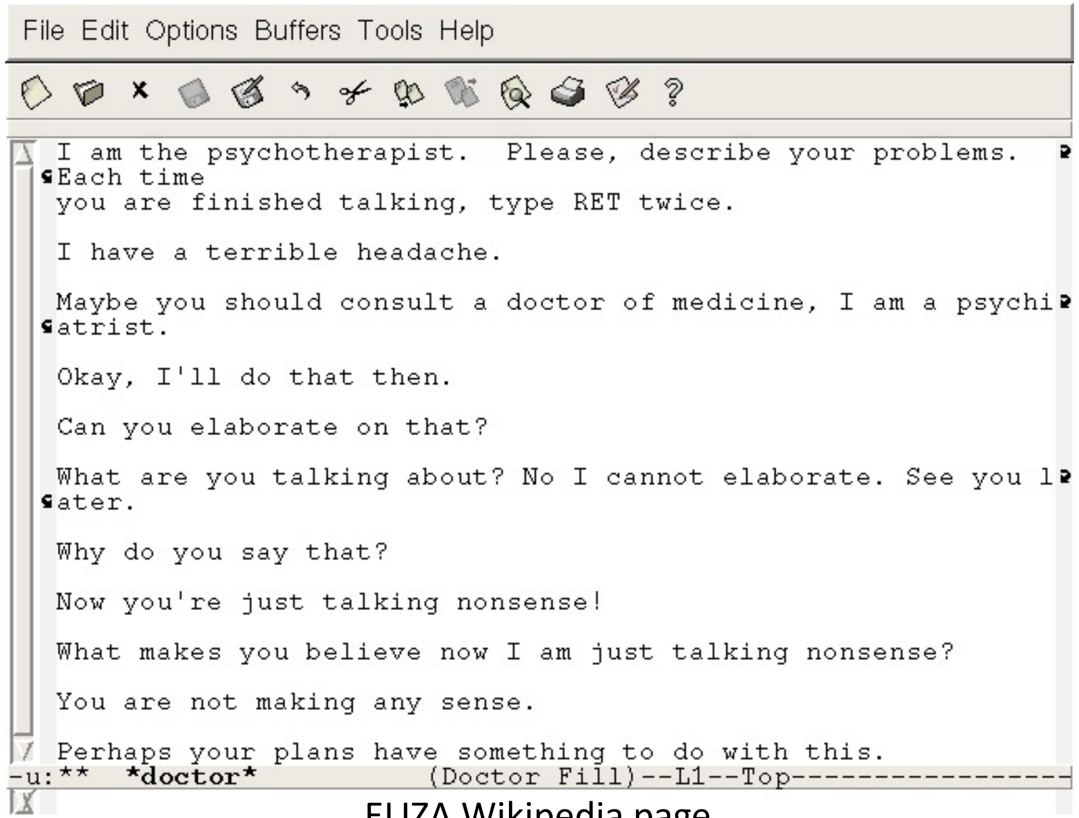
Perceptron and Multi-Layer Perceptron (MLP)

History of ANNs

- Interest in ANNs came in waves :
  - 1960s : people thought we would soon be conversing with truly intelligent machines.
    - ELIZA, Weisenbaum 1964 :
  - When this wasn’t the case,

  funding went elsewhere…



ELIZA Wikipedia page

History of ANNs

- 1980s : revival of interest, new architectures invented, better training techniques, but progress was slow…
- by the 90s, other machine learning techniques were invented (SVMs) => seemed to offer better results and stronger theoretical foundations.

History of ANNs

- Present day : another wave of interest, but this one seems to be different :
  - Now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
  - Tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a « reasonable » amount of time.
  - Training algorithms have been improved.
  - Possible to create ANNs on a massive scale (100 Trillion params in GPT4)
  - ANNs seem to have entered a virtuous circle of funding and progress.

Biological Neurons

- Biological neurons receive short electrical impulses called signals from other neurons via synapses.
- When a neuron receives a sufficient number of signals from other neurons, it fires its own signals.

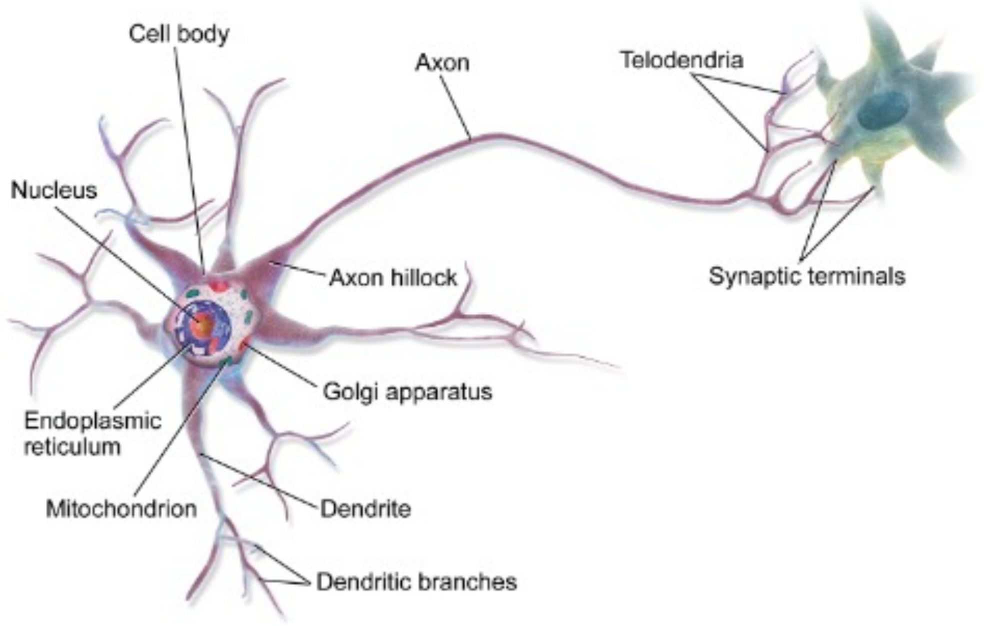
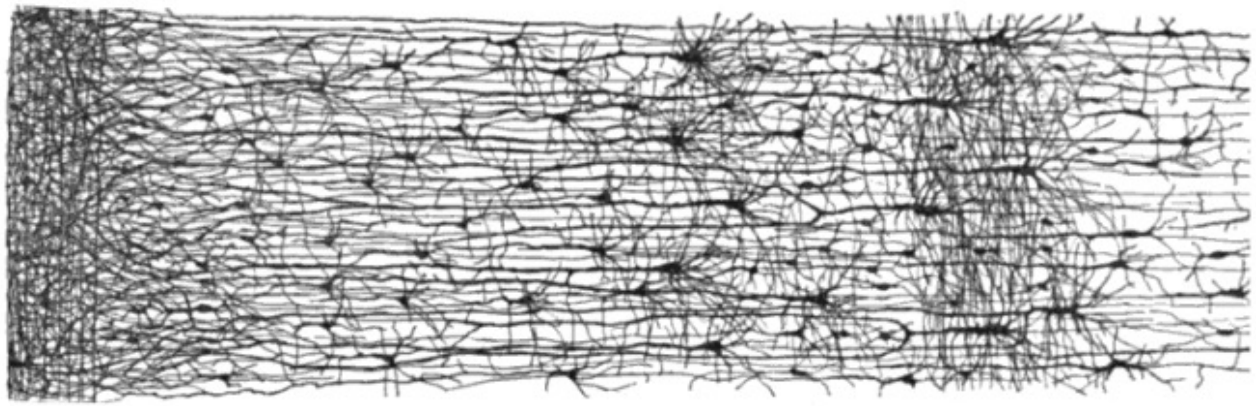


Image by Bruce Blaus (Creative Commons 3.0).
Reproduced from https://en.wikipedia.org/wiki/Neuron

Biological Neurons

- Organized in a vast network of billions of neurons, each neuron is typically connected to thousands of other neurons.
- Research suggests that neurons are often organized in consecutive layers

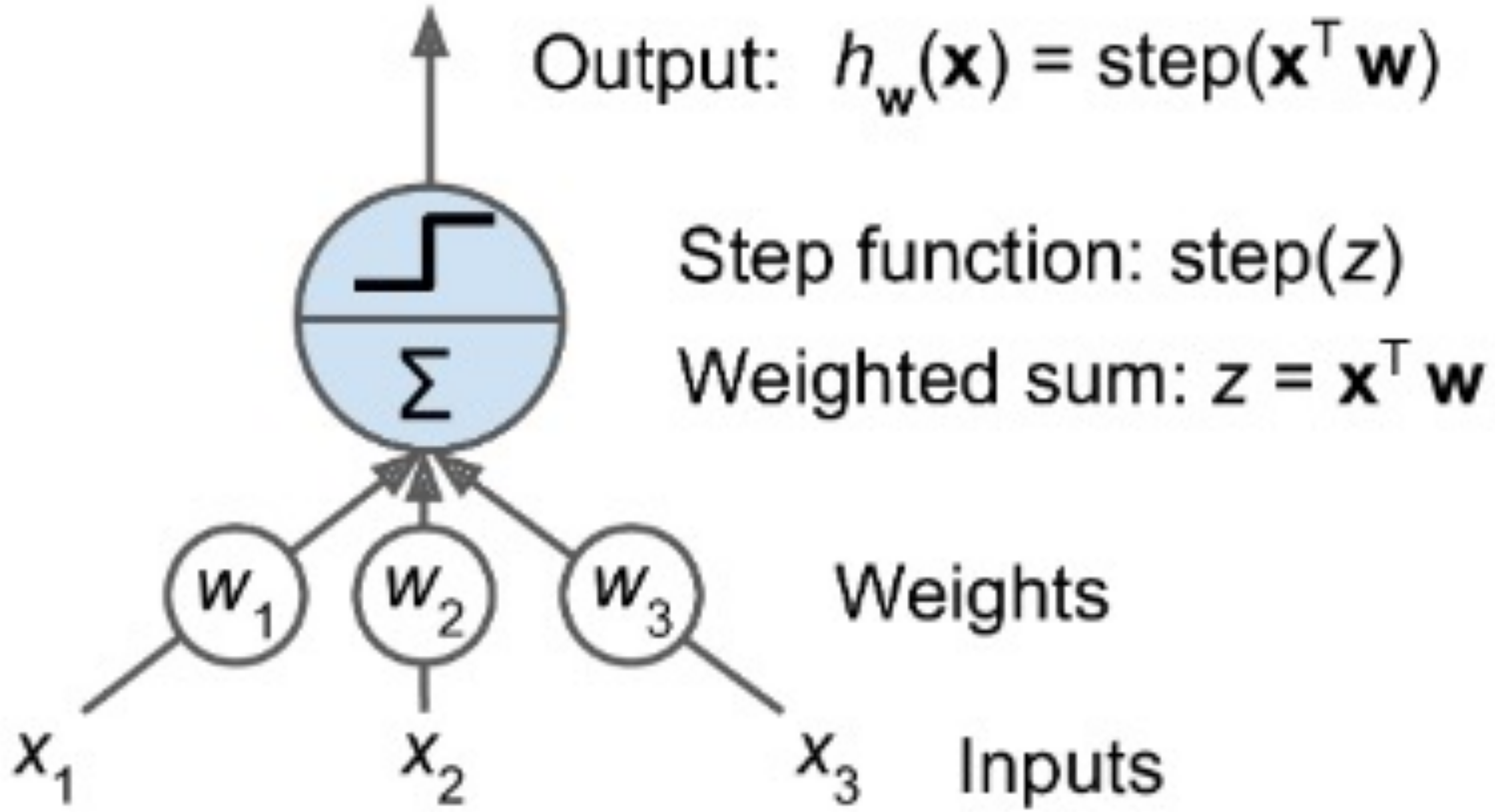


Drawing of a cortical lamination by S. Ramon y Cajal (public domain)

The Perceptron

- One of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt

- These artificial neurons are called *Threshold Logic Units.* We will see why.


Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{x}^T \mathbf{w})$
Step function: step(z)
Weighted sum: $z = \mathbf{x}^T \mathbf{w}$
Σ
w_1 w_2 w_3
Weights
x_1 x_2 x_3
Inputs

The Step Function

- Most commonly used is the *Heaviside Step Function :*

$$\text{heaviside}\,(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ if } z \geq 0 \end{cases}$$

- Does this unit ring any bells…?

Binary Classification

- A single TLU can be used for simple linear binary classification.

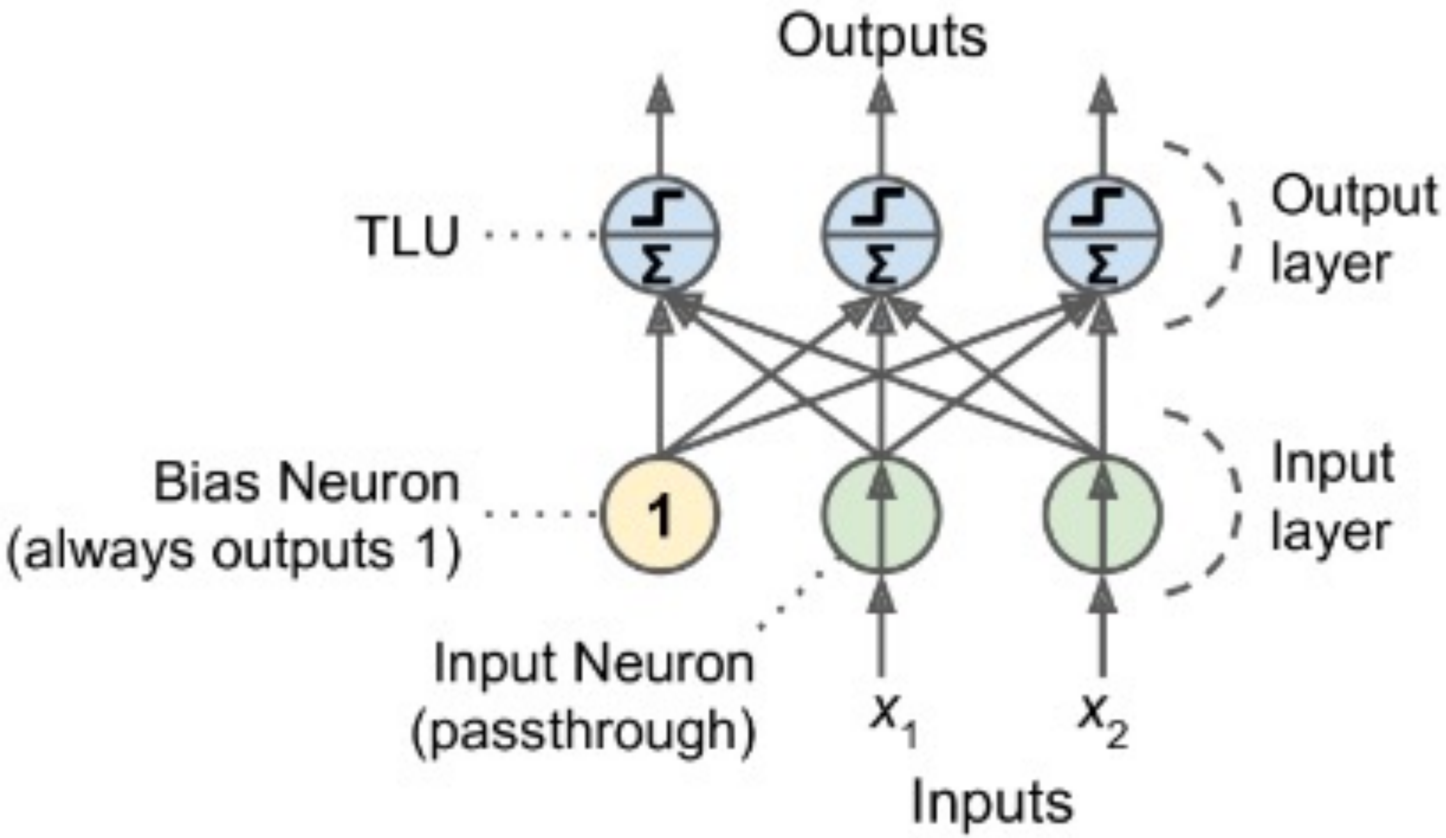
- It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class.

Perceptron

- The Perceptron can refer to this single TLU, but can also refer to a single *layer* of TLUs.

- When all the neurons in a layer are connected to every neuron in the previous layer, it is called a *fully connected layer* or a *dense layer*.

Perceptron Diagram



Perceptron

- Common to draw special passthrough neurons called *input neurons*

- Also, an extra bias feature is can be added (our dummy feature from linear/logistic regression), but this can be handled differently

- This particular Perceptron setup can classify instances simultaneously into three different binary classes, which makes it a multi-output classifier

Computing the outputs of a fully connected layer

$$h_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{X}) = activation(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b})$$

(no dummy features need to be added to **X** in this case)

- **X** represents the matrix of input features. It has one row per instance, one column per feature.
- The weight matrix **W** contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector **b** contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The activation function: when the artificial neurons are TLUs, it is a step function. We will discuss other activation functions.

How is the Perceptron trained ?

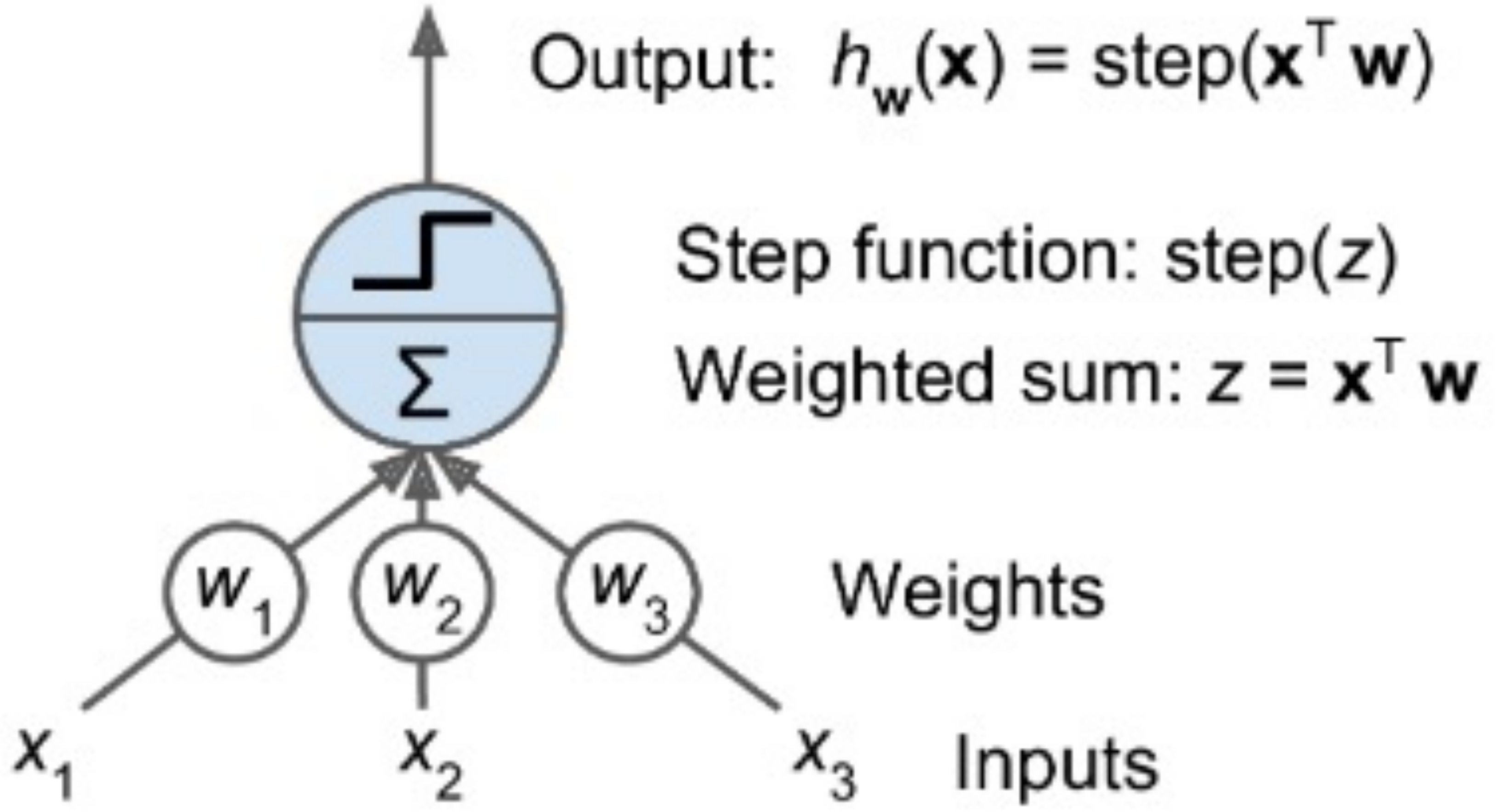
- In his book *The Organization of Behavior*, published in 1949, Donald Hebb suggested that *when a biological neuron often triggers another neuron*, the *connection* between these two neurons *grows stronger*.

- "**Cells that fire together, wire together**." (Siegrid Löwel)

- Perceptrons are trained using a variant of this rule that takes into account the error made by the network

- For every output neuron that produced a wrong prediction, it *reinforces the connection weights from the inputs that would have contributed to the correct prediction*.
- Ie. Increases the weight in the connection

Example


Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{x}^T \mathbf{w})$
Step function: step(z)
Weighted sum: $z = \mathbf{x}^T \mathbf{w}$
Σ
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

What can the Perceptron Learn ?

- Cannot learn complex patterns in the data

- However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.

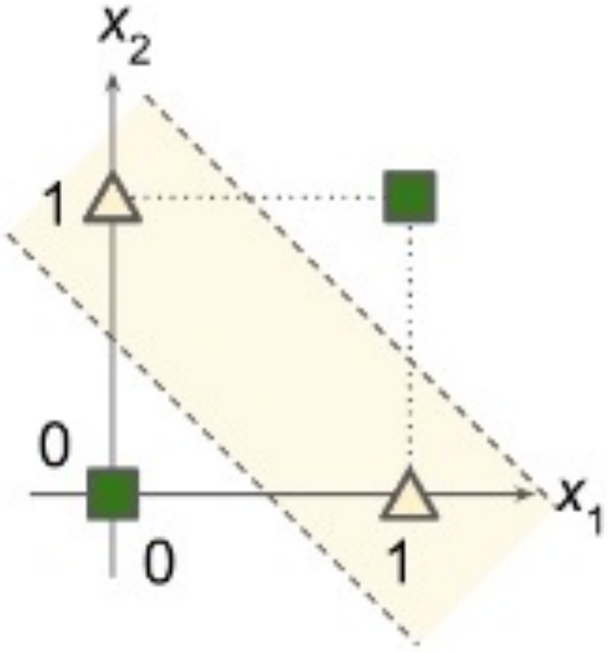
Sklearn Perceptron

- Scikit-Learn provides a Perceptron class that implements a **single TLU** network (on the iris dataset example here):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris Setosa?

per_clf = Perceptron()
per_clf.fit(X, y)


y_pred = per_clf.predict([[2, 0.5]])
```

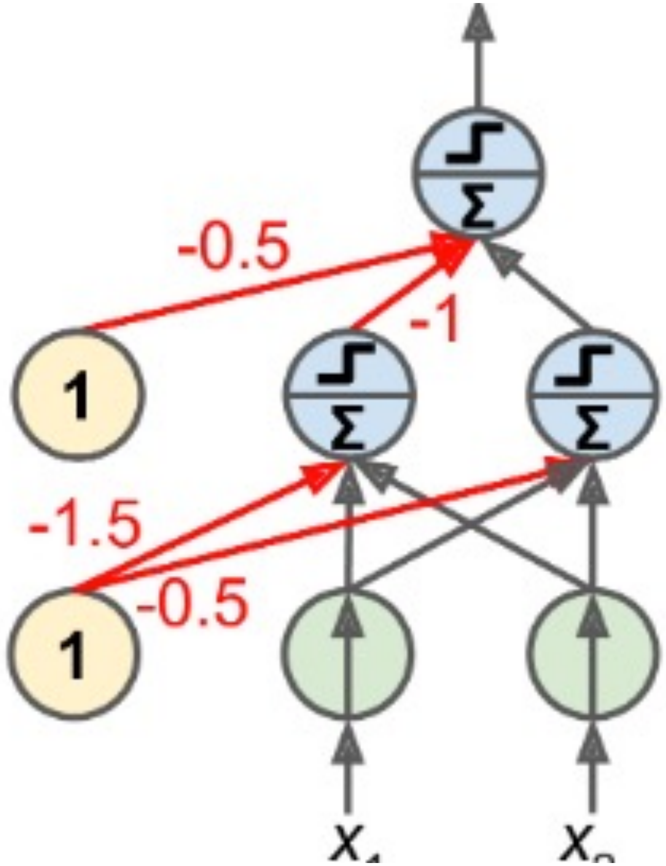
The Multi-Layer Perceptron: Why Multiple Layers ?

- Certain trivial problems prove unsolvable for the simple perceptron.
- The XOR (*exclusive OR*) problem in particular is one of the most famous examples.

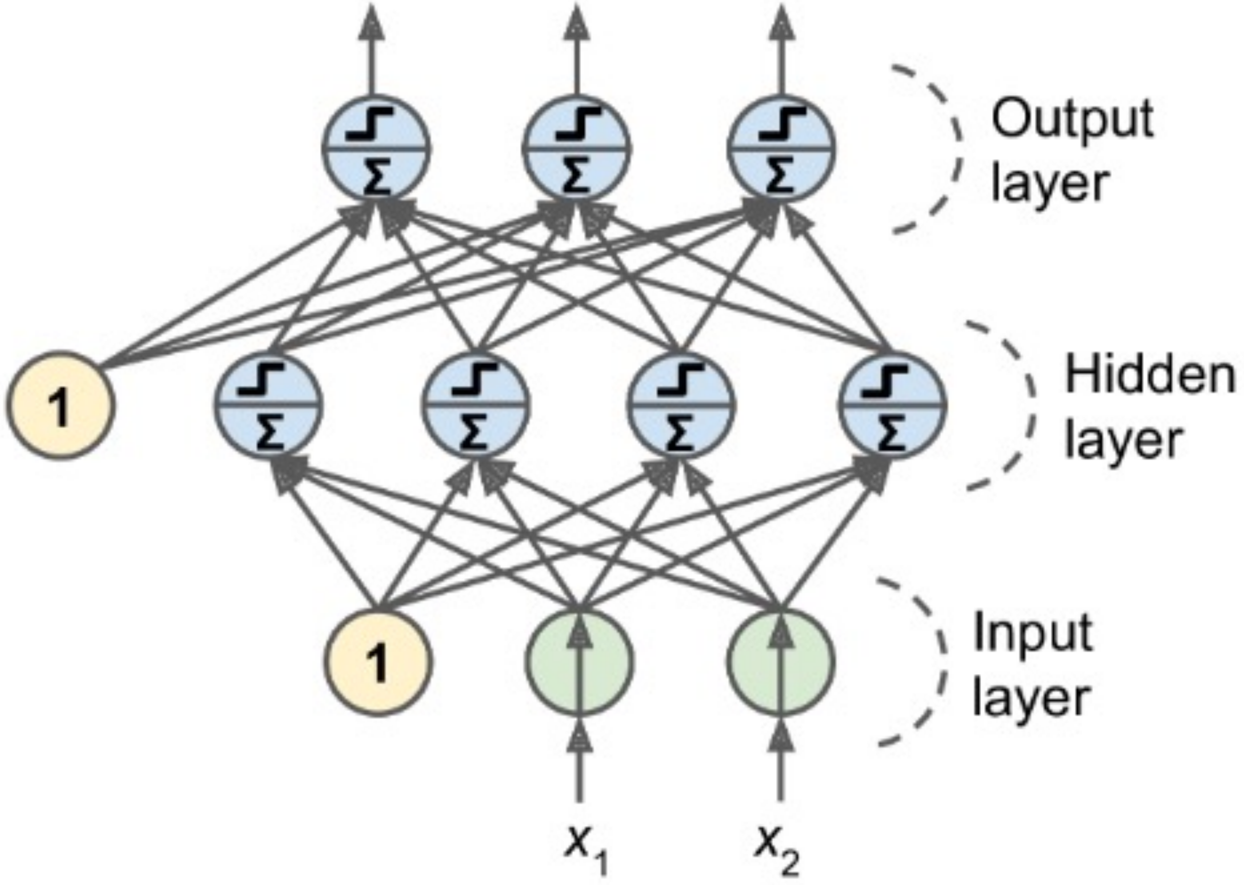


Why Multiple Layers ?

- Turns out some of these limitations can be elminiated by **adding layers** :
- This network solves the XOR problem for example
- With inputs (0,0) or (1, 1) the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1.
- All connections have a weight equal to 1, except the four connections where the weight is shown.



Multi-Layer Perceptron



Multi-Layer Perceptron

- An MLP is composed of one (passthrough) input layer, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer*

- The layers close to the input layer are usually called the *lower* layers, and the ones close to the outputs are usually called the *upper* layers.

- Every layer except the output layer includes a *bias neuron* and is *fully connected* to the next layer.

- The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a **feedforward neural network** (FNN).

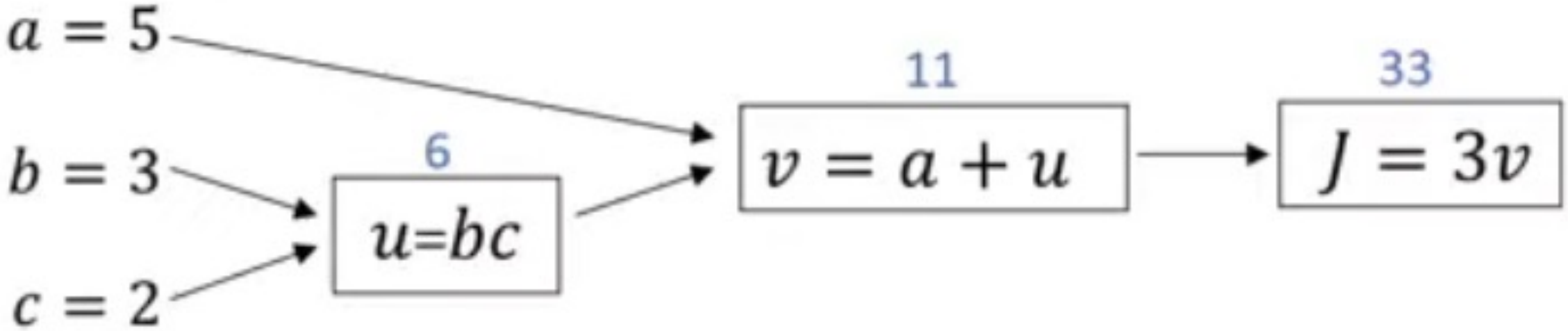
How do we train this neural network ?

- 1986 : Rumelhart & al. Introduced the ***Backpropagation*** algorithm, still used today.
- Basically a version of Gradient Descent :
  - It's able to compute the gradient of the network's error with regards to *every single model parameter.* This is done using the chain rule.
  - In other words it finds out how each connection weight and each bias term should be tweaked in order to reduce the error.
  - Then a gradient descent step is performed (e.g. for the 1st neuron of the 1st layer) :

$$weights_{N=1,\,L=1} = weights_{N=1,L=1} - learningRate * \frac{\partial Cost}{\partial weights_{N=1,L=1}}$$

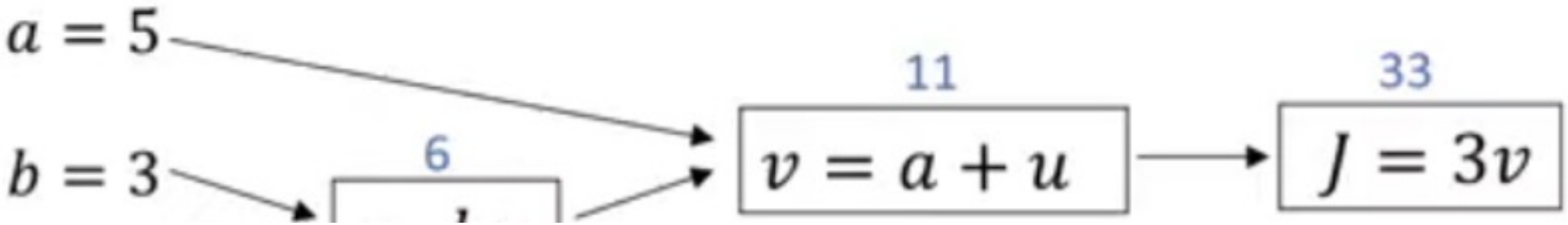
Chain Rule (slides from previous class on linear regression)

- Let’s apply derivatives to a computation graph that has multiple nodes, each representing a function :



- $\frac{dJ}{da} = ?$
- To figure this out, let's look at how a affects v, and then how this change in v affects J :

$$a = 5 \ \rightarrow 5.001$$
$$v = 11 \ \rightarrow 11.001$$
$$\mathrm{J} = 33 \ \rightarrow 33.003$$

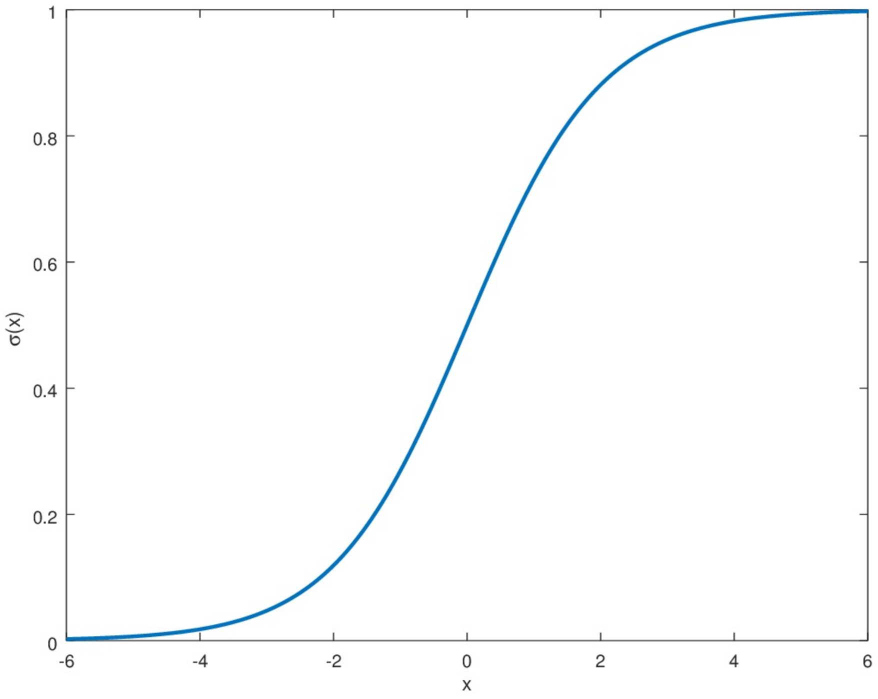


- Reasoning by looking at the waterfall effect of nudging a is the basis of the chain rule : $a \rightarrow v \rightarrow J$
- We can know how a nudge in a changes J by multiplying how much this nudge **first** changes v **and then** J.

$$\frac{dJ}{da} = \frac{dJ}{dv} \times \frac{dv}{da}$$

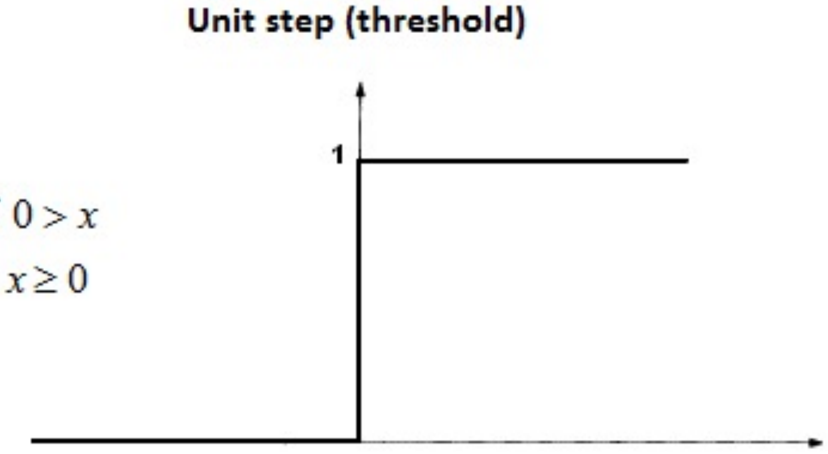
Activation functions

- One key change was made to the neurons for backprop to work:
- The *step* function was replaced with the logistic function.
- Step function contains only flat segments (derivative is 0 everywhere) vs. Sigmoid/logistic function which has a non-zero derivative everywhere
- This is important for the chain rule to work.



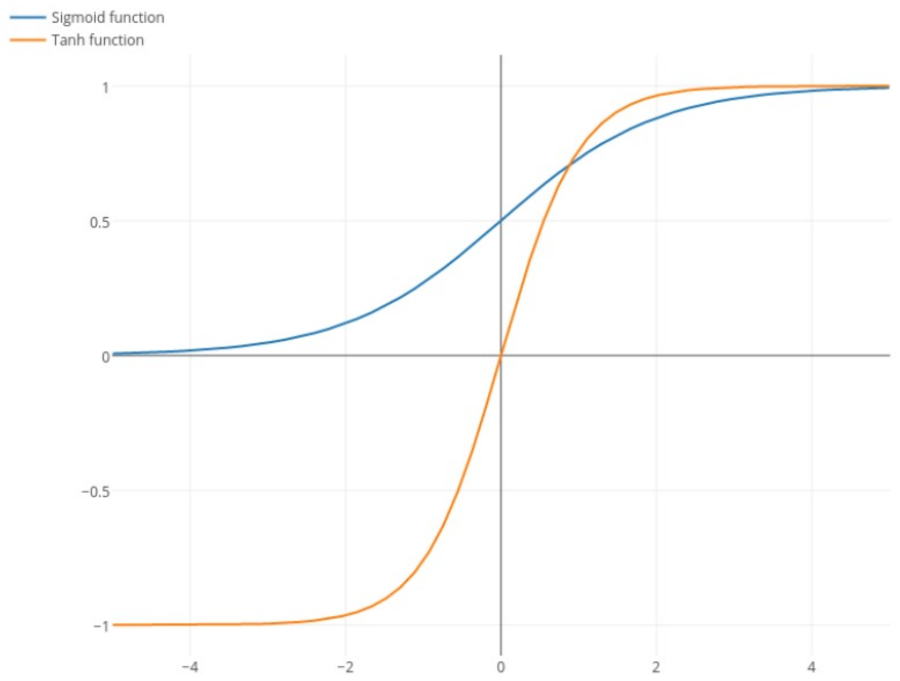
Sigmoid

$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$



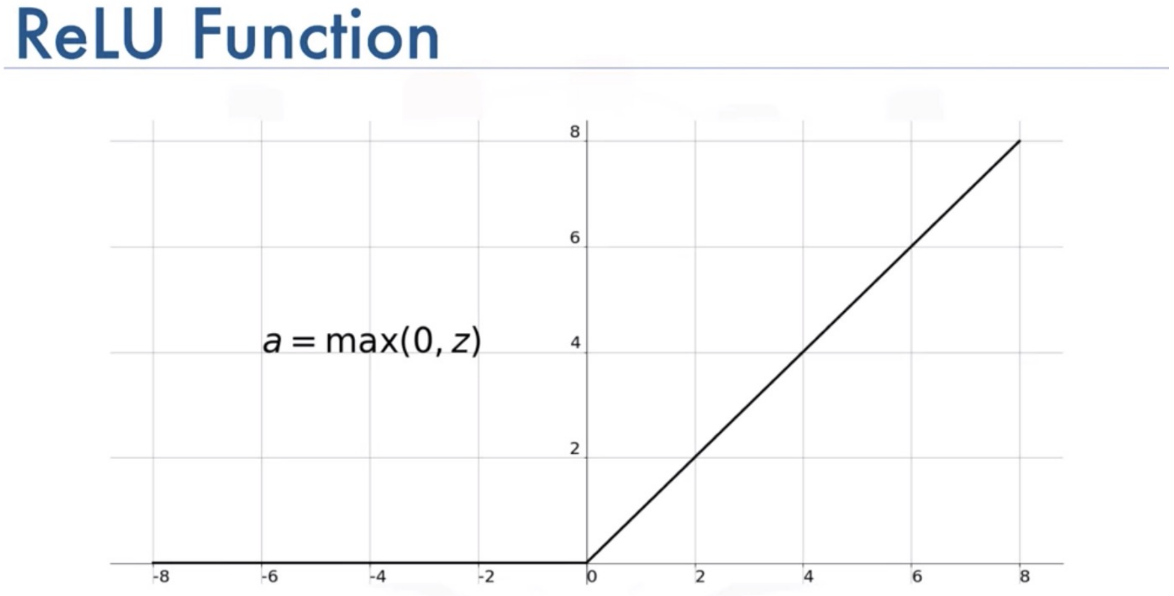
Other Activation functions

- *The hyperbolic tangent function* $\tanh(z) = 2\sigma(2z) - 1$
  Just like the logistic function it is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function)



Other Activation functions

- The Rectified Linear Unit function: ReLU(z) = max(0, z)
  It is continuous but unfortunately not differentiable at z = 0 (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for z < 0.
- However, in practice it works very well and has the advantage of being fast to compute.



Why do we need activation functions ?

- Why does a neuron not just output the weighted some of its inputs and pass it on to the next neurons ?

- Why is a **non-linear** activation function needed ?

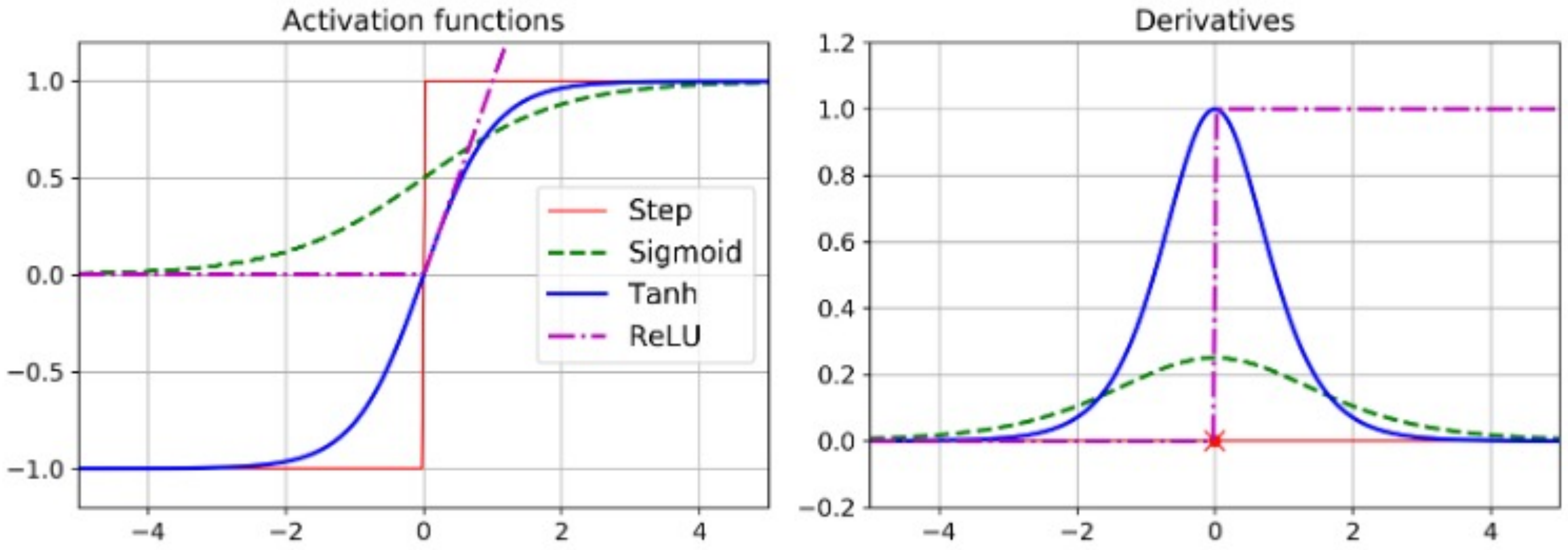
Why do we need activation functions ?

- => If you compose several linear transformations, you actually get 1 linear transformation, which doesn’t allow us to solve complex problems….
- If $f(x) = 2x + 3$ and $g(x) = 5x - 1$, then composing these two linear functions gives us another linear function:

$$f(g(x)) = 2(5x - 1) + 3 = \mathbf{10x + 1}$$

- So if there isn’t a non-linear function betwen the layers, then even a deep stack of layers is equivalent to a single layer…
- We will see this with a practical example in pytorch later

Activation Functions and their derivatives



Extra (illustrated) resources

- Jay Alammar's blogposts on neural nets
- 3Bue1Brown's videos on neural nets
- To play around with a neural net
- The Absolutely Simplest Backpropagation Example
- Why neural nets can learn almost anything

Classification MLPs

- **Binary Classifiaction** :
- 1 single output neuron using the logistic activation function.
- It outputs a number between 0 and 1, which you can interpret as the estimated probability of the positive class.
- The estimated probability of the negative class is equal to 1 - that number.

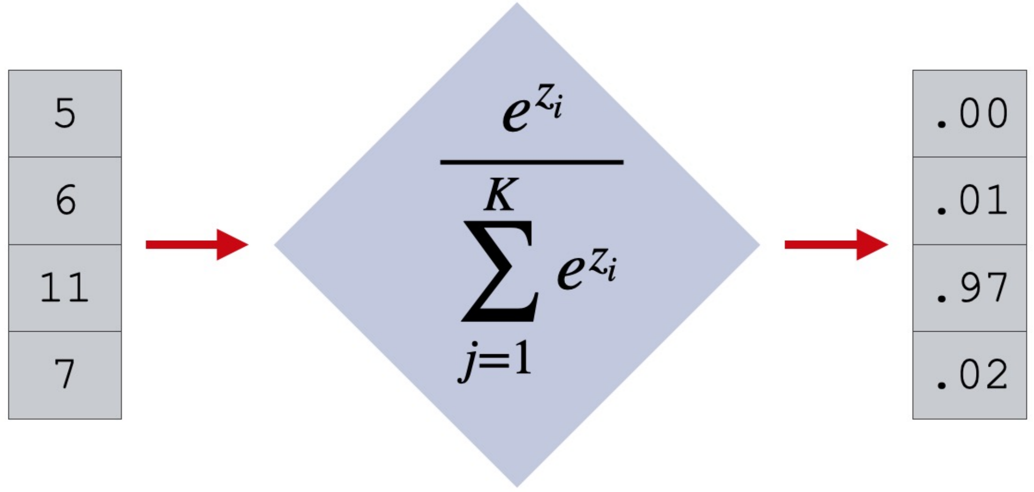
- What happens in this output neuron is equivalent to what we saw for logistic regression.
- The only difference is that the input values are now the computed outputs from the neurons in the previous layer vs. the features.

Classification MLPs

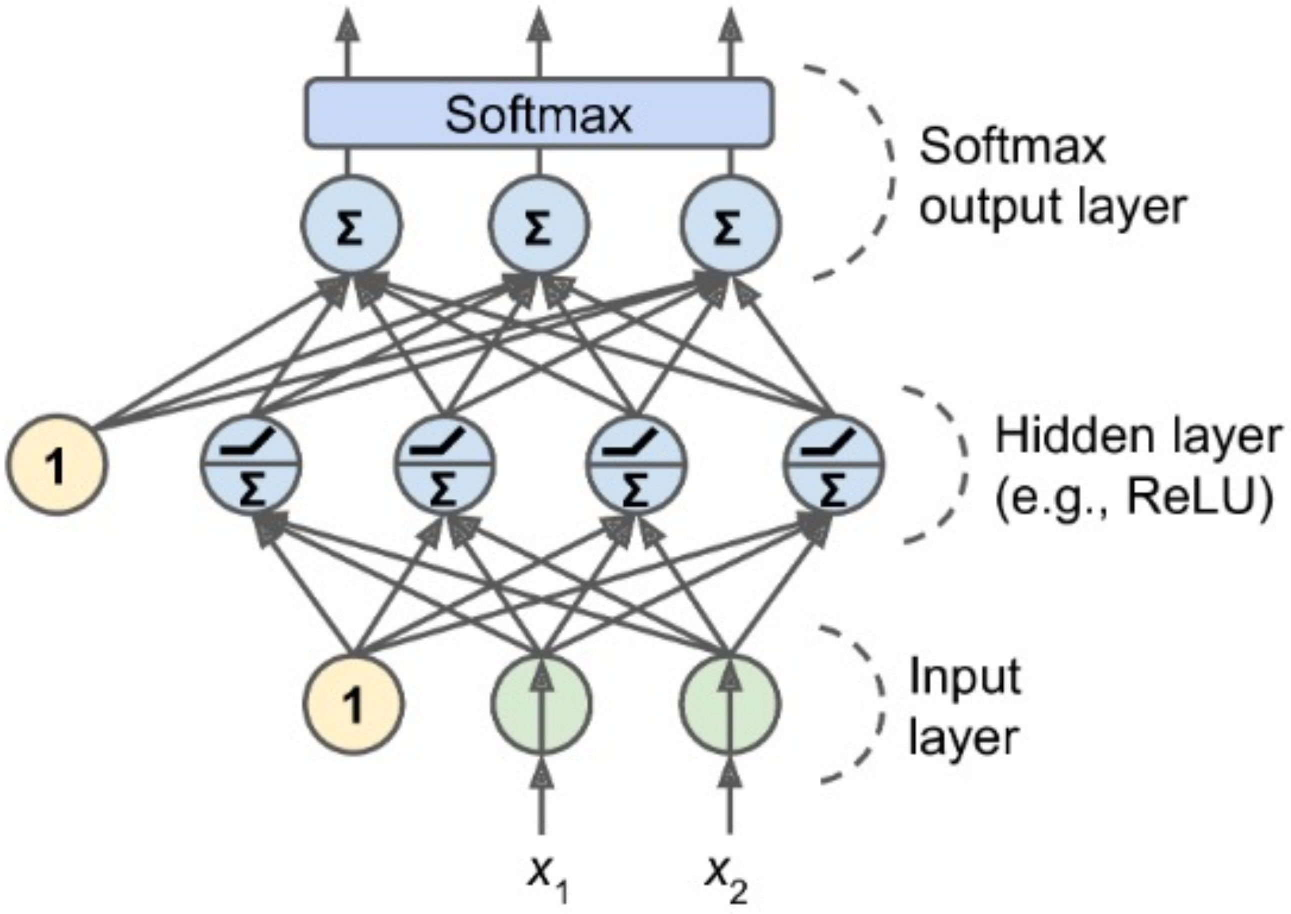
- **Multiclass Classification:**
  - If each example fed into the network can only belong to a single class (the classes are mutually exclusive — there can't be a combination of classes, only 1)
  - Then you need 1 output neuron *per class* and you need to use the ***softmax*** activation function for the whole output layer.
  - This function ensures the scores produced by each output neuron are between 0 and 1, and add up to 1.

  - Softmax wikipedia
  - Softmax vs argmax

SOFTMAX TRANSFORMS A VECTOR OF NUMBERS INTO A VECTOR OF RELATIVE "PROBABILITIES"



Multiclass Classification MLP



Loss function

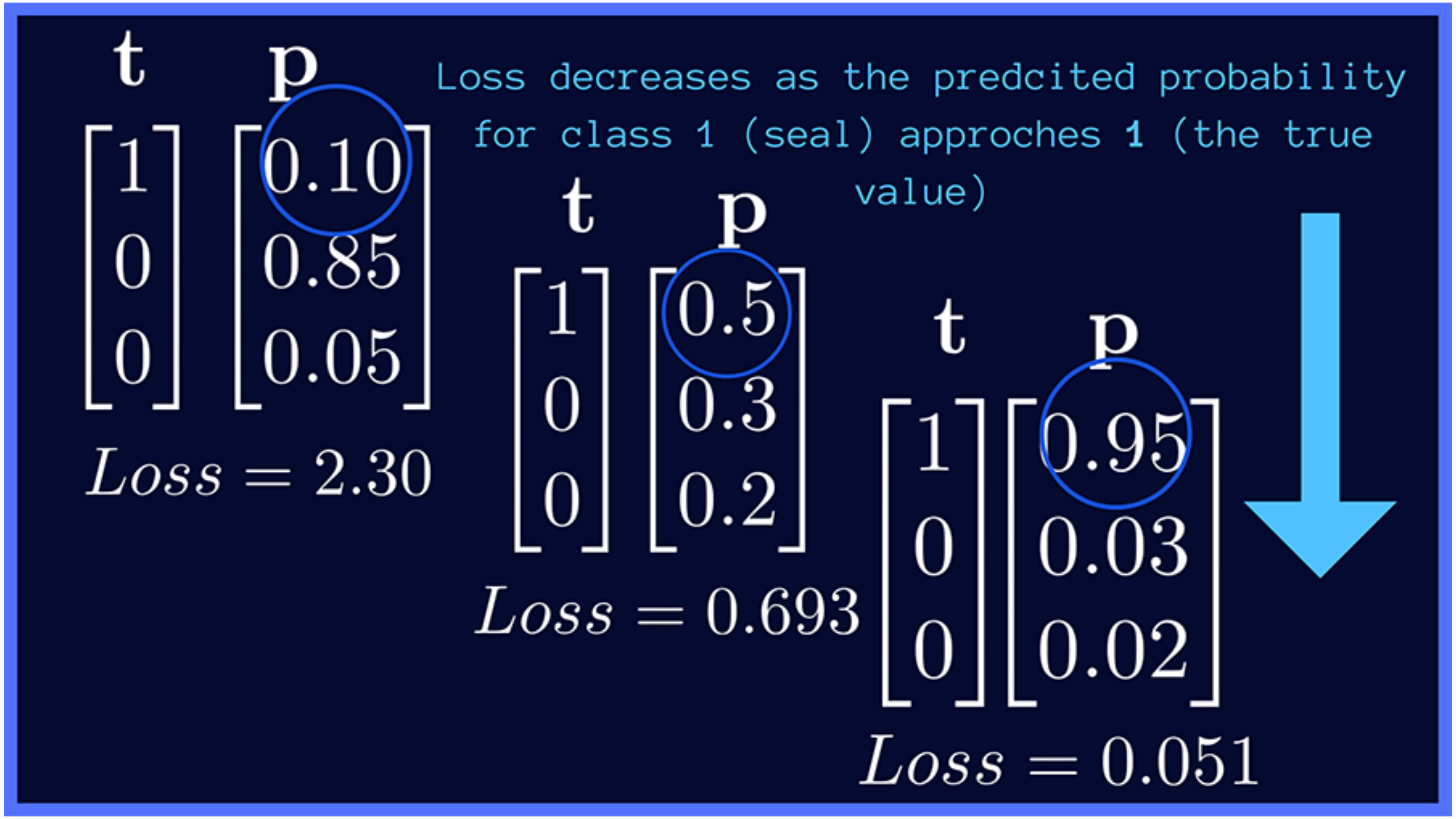
- We are predicting a *probability distribution*
- Hence we need to use a function which can output
  - a small loss when the probability distributions are very similar
  - a large loss otherwise.

- This is called the ***cross-entropy* loss** and originates from a field called *information theory*.

- We've actually used a version of this loss for logistic regression. It's generally referred to as BinaryCrossEntropy.

- Some Explanations/Resources with the formula:
  - Entropy, Cross-Entropy and KL Divergence (video resource)
  - Cross-entropy Loss

Cross Entropy Loss



https://www.pinecone.io/learn/cross-entropy-loss/

Typical Network Architecture for Classification

Hyperparameter	Binary Classification	Multiclass Classification (also works for 2 classses)
# input neurons	1 per feature	Same
# hidden layers	Depends on the problem. Typically 1-5, but as many as you want is possible.	Same
# neurons per hidden layer	Depends on the problem. Typically 10-100.	Same
# output neurons	1	1 per class
Output Layer Activation	Sigmoid	Softmax
Loss Function	Binary Cross-Entropy	Cross-Entropy