DEEP LEARNING FOR VISUAL RECOGNITION

Lecture 3 – Neural Networks

Henrik Pedersen, PhD
Part-time lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

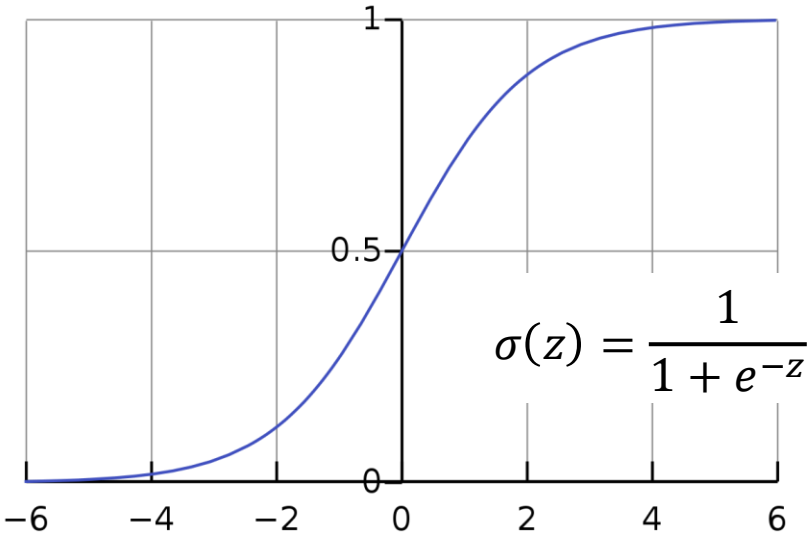
Today's agenda

- You will learn about basic neural networks and how to train them.
- Topics
  - Artificial neurons
  - Decision boundaries
  - Neural networks (NNs)
  - Multi-class classification with NNs
  - Loss functions
  - Optimization – backpropagation
  - Training NNs in practice
  - Backpropagation revisited – automatic differentiation

Recall: Logistic regression

- **Task:** Learn to predict y $\in \{0,1\}$ from $x \in \mathbb{R}^{m\times 1}$ given training data $\{y^{(i)}, x^{(i)}\}_{i=1}^{n}$
- **Model:** $P(y=1|x) = h_w(x) = \frac{1}{1+e^{-w^T x}} \equiv \sigma(w^T x)$
- **Optimization:** Minimize cross-entropy loss

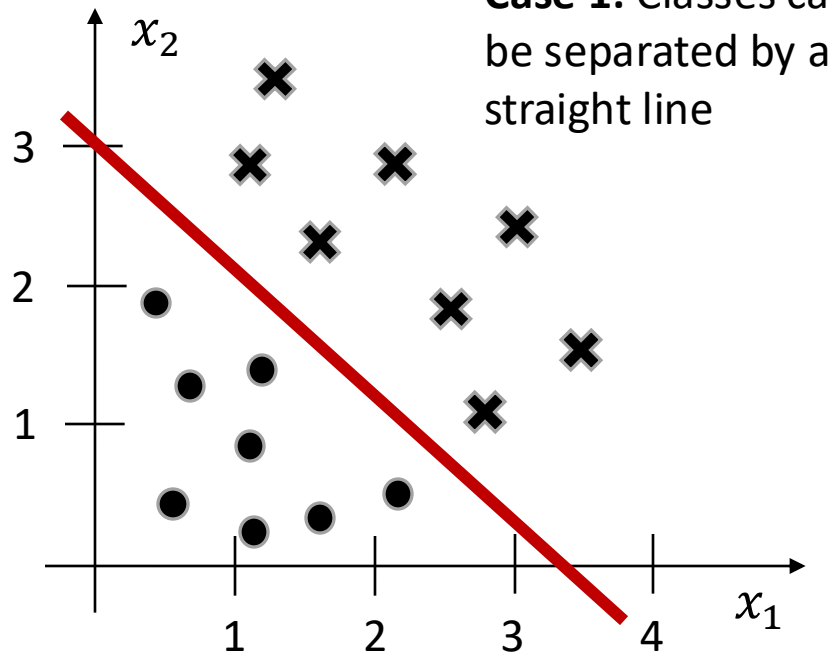
$$J(w) = -\sum_{i=1}^{n} y^{(i)} \log\left(h_w\left(x^{(i)}\right)\right) + (1-y^{(i)}) \log\left(1-h_w\left(x^{(i)}\right)\right)$$



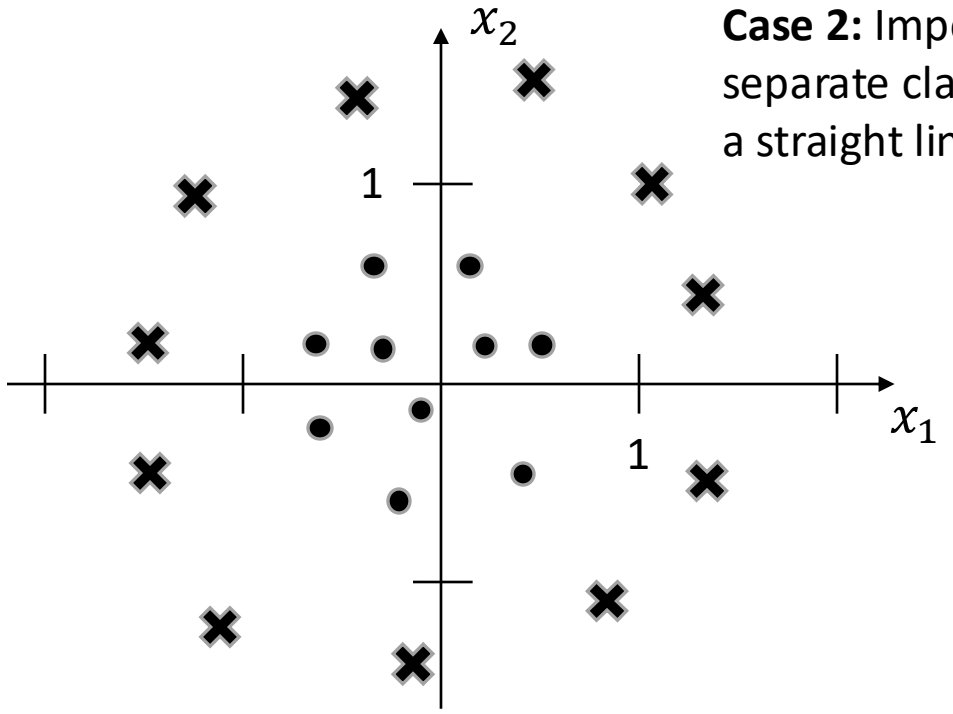
- **Softmax regression:** Natural extension of logistic regression that supports multiple classes.
- **Problem:** Unable to solve hard cases

Problem: Linear decision boundary

- Linear models have linear decision boundaries (explained later).



Case 1: Classes can be separated by a straight line

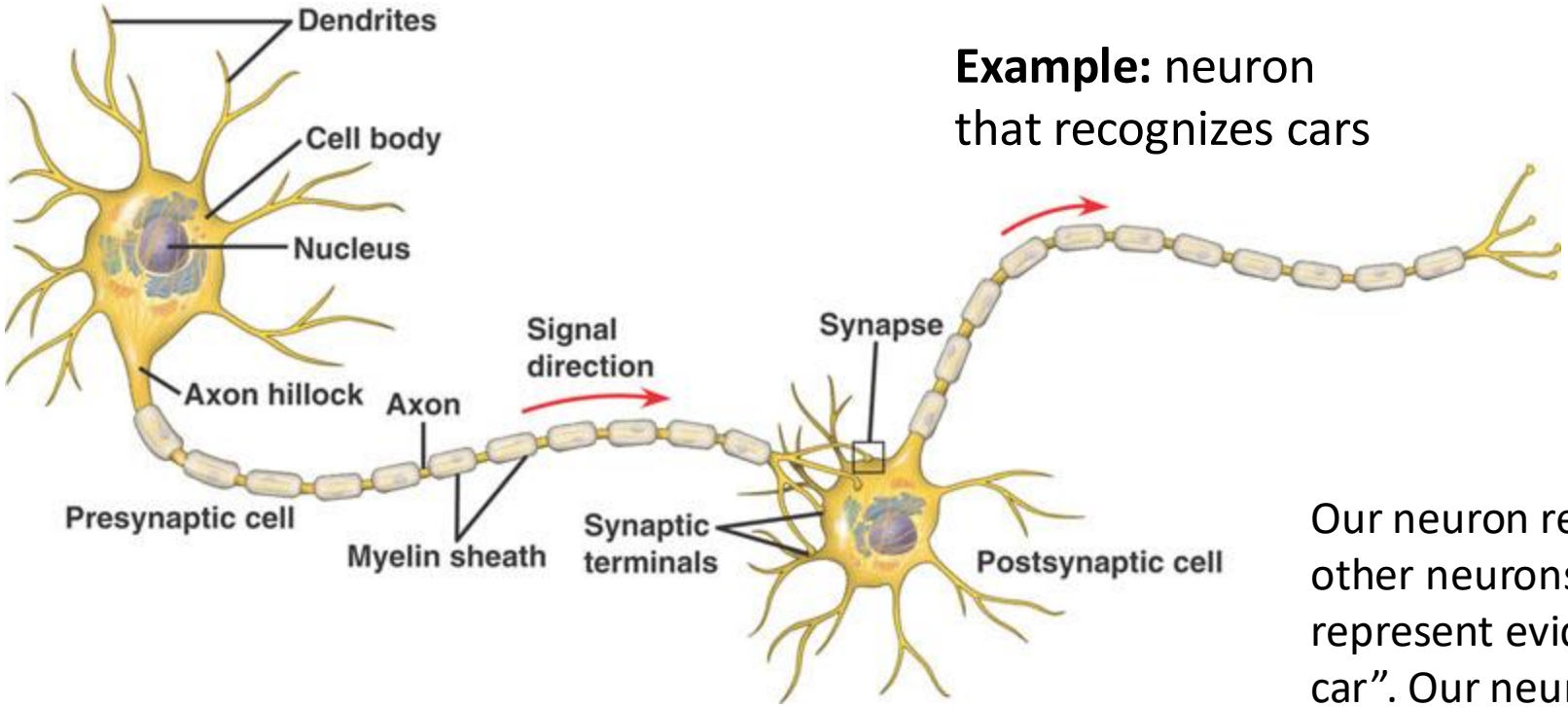


Case 2: Impossible to separate classes using a straight line

Solution: Artificial neural networks – universal function approximators!

Artificial neurons

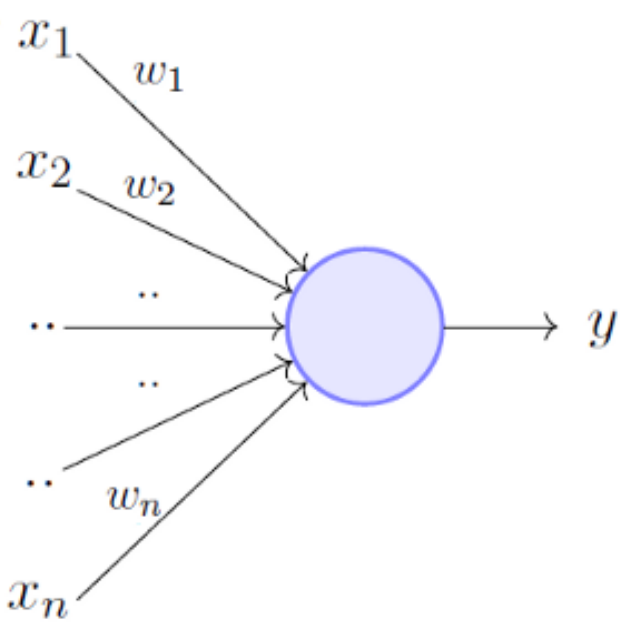
Biological neurons



Example: neuron that recognizes cars

Our neuron receives input from other neurons. These inputs may represent evidence of “car” or “not car”. Our neuron makes a decision by **weighing up evidence**.

Artificial neurons (perceptron)



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

This is called a **bias** or threshold.

Think of **w** as a kind of **template**. If x matches the template well, the inner product will be larger than the threshold, and the neuron "fires" (i.e., it outputs a 1). **Think similarity!**

Rewriting the above,

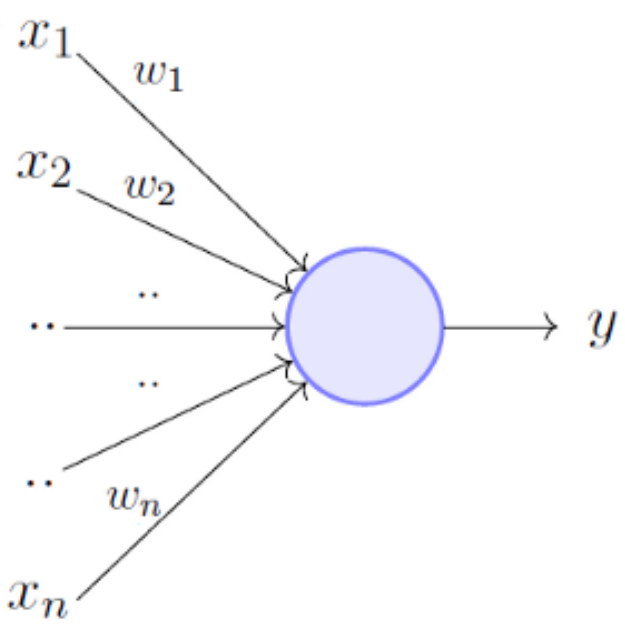
$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

A way you can think about the perceptron is that it's a device that makes decisions by **weighing up evidence**. For images, each x_i corresponds to a particular pixel in the input image. In MNIST a white pixel at a given position either increases or decreases the evidence of, say, the digit 0 being present in the image.

https://towardsdatascience.com/perceptron-the-artificial-neuron-4d8c70d5cc8d

Artificial neurons (perceptron)



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

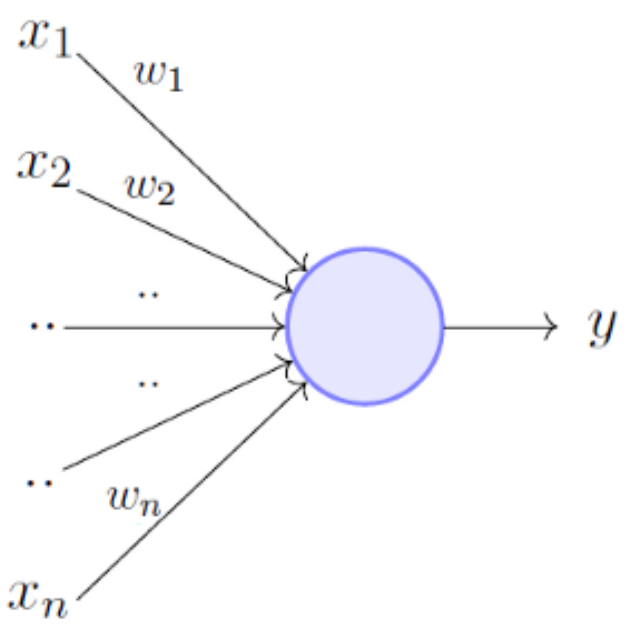
This is called a **bias** or threshold.

Think of **w** as a kind of **template**. If x matches the template well, the inner product will be larger than the threshold, and the neuron "fires" (i.e., it outputs a 1). **Think similarity!**

Note that we can simplify the notation by observing that $\sum_{i=1}^{n} w_i x_i$ is just the inner product $w^T x$

https://towardsdatascience.com/perceptron-the-artificial-neuron-4d8c70d5cc8d

Artificial neurons (perceptron)



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

This is called a **bias** or threshold.

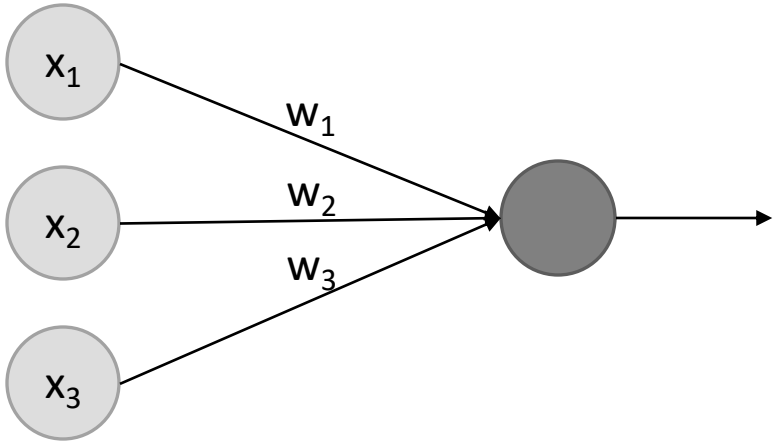
Think of **w** as a kind of **template**. If x matches the template well, the inner product will be larger than the threshold, and the neuron "fires" (i.e., it outputs a 1). **Think similarity!**

Perceptrons are not that well-suited for gradient descent optimization: Small change in w either has no effect on the predicted output (y), or it changes the output dramatically (e.g., from 1 to 0).

https://towardsdatascience.com/perceptron-the-artificial-neuron-4d8c70d5cc8d

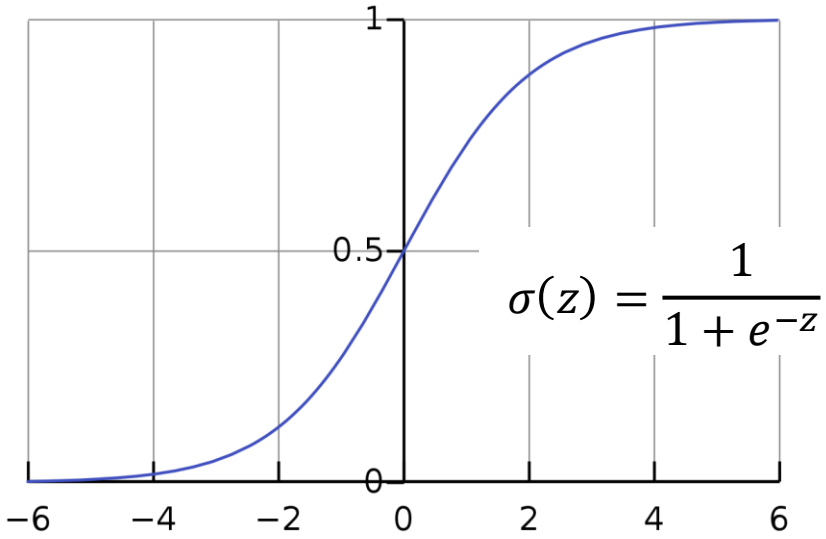
Logistic unit

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$



$$h_w(x) = \sigma(w^T x)$$

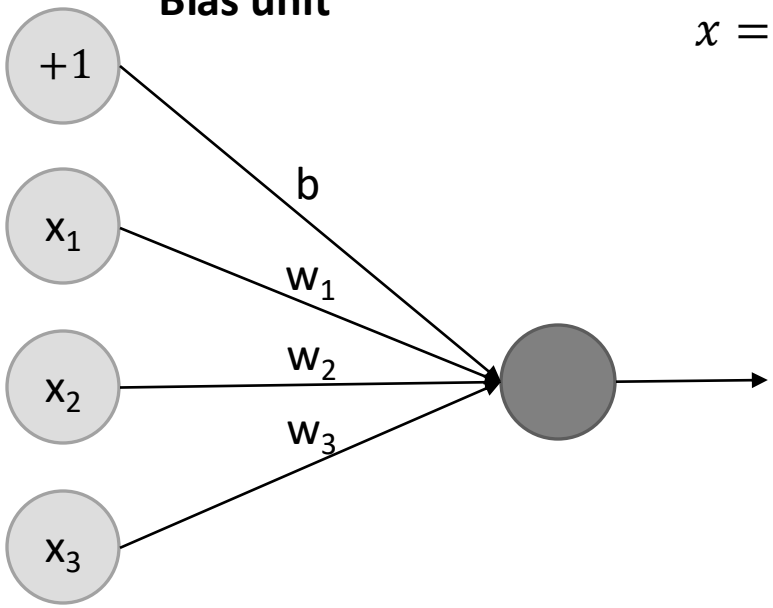
Natural extension of the perceptron that **outputs probabilities** instead of binary labels (0 or 1).



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

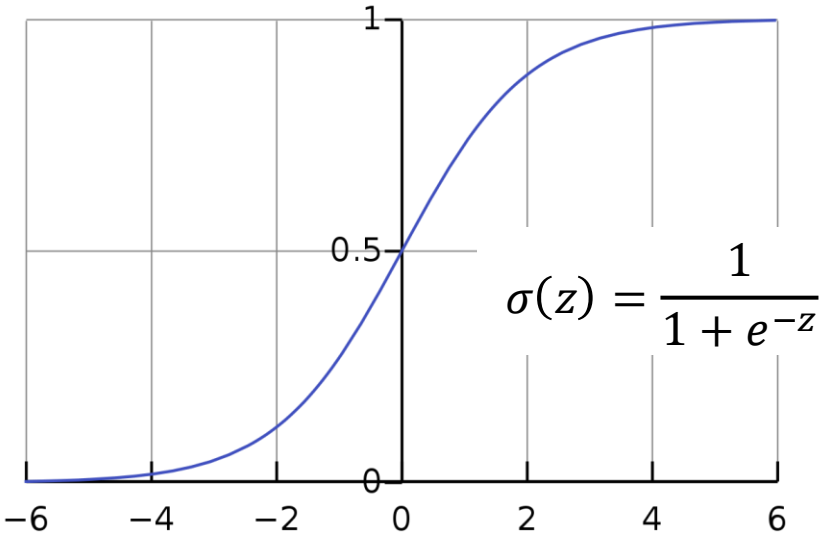
Activation function:
Sigmoid in this case (there are other options)

Logistic unit



$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

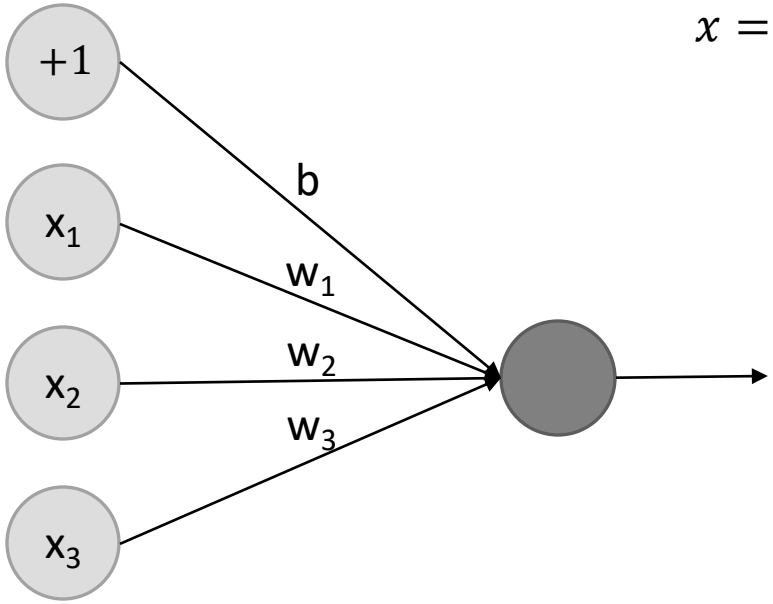
$$h_{w,b}(x) = \sigma(w^T x + b)$$



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

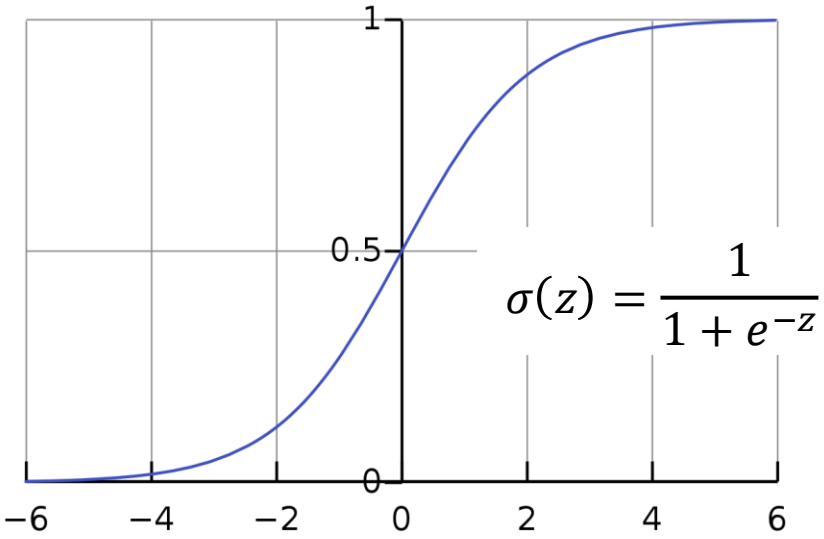
The **bias** unit is analogous to the threshold of the perceptron.
For a logistic unit with a large positive bias, it's easy to output a 1.
But if the bias is large and negative, then it's difficult to output a 1.

Logistic unit



$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

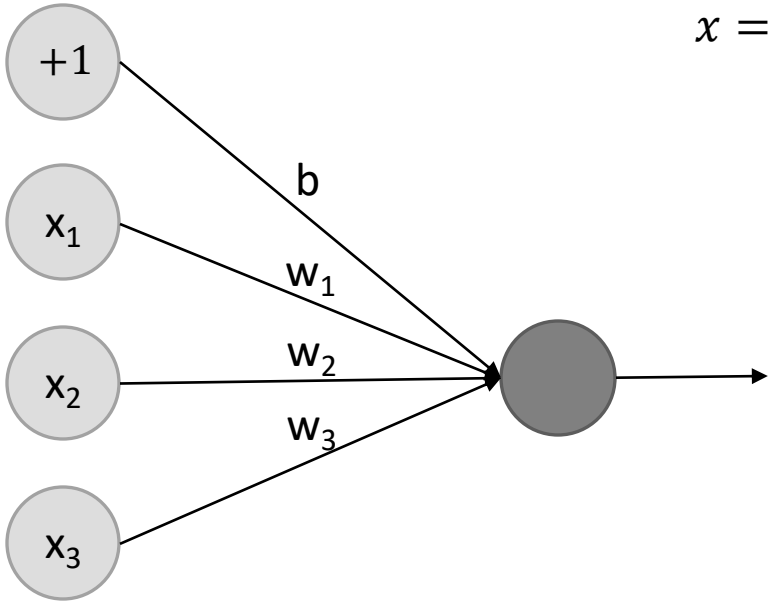
$$h_{w,b}(x) = \sigma(w^T x + b)$$



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

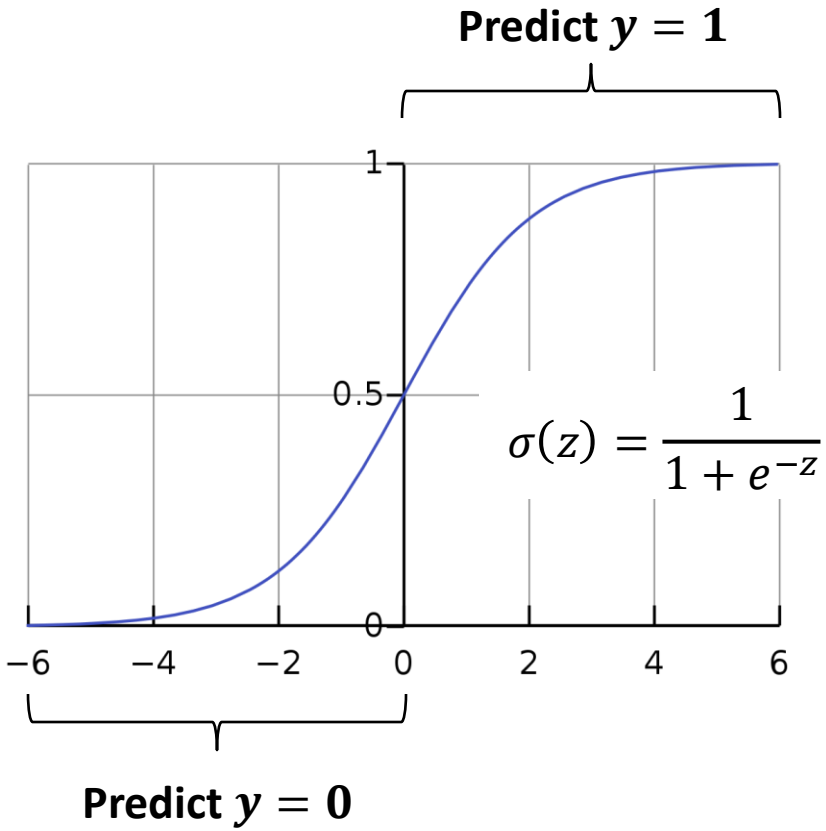
Saturation: The logistic unit is better suited for gradient descent than the perceptron: Small changes in w also result in small changes of the output, because the sigmoid is smooth. However, if $|w^T x + b|$ is numerically large, the gradient of $h_{w,b}(x)$ will be very small. This can lead to slow convergence of gradient descent (called saturation).

Logistic unit as a binary classifier



$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$h_{w,b}(x) = \sigma(w^T x + b)$$



- **Model:** $P(y = 1|x) = h_{w,b}(x)$
- **Prediction (inference):**
  - Predict $y = 1$ if $h_{w,b}(x) \geq 0.5$
  - Predict $y = 0$ if $h_{w,b}(x) < 0.5$

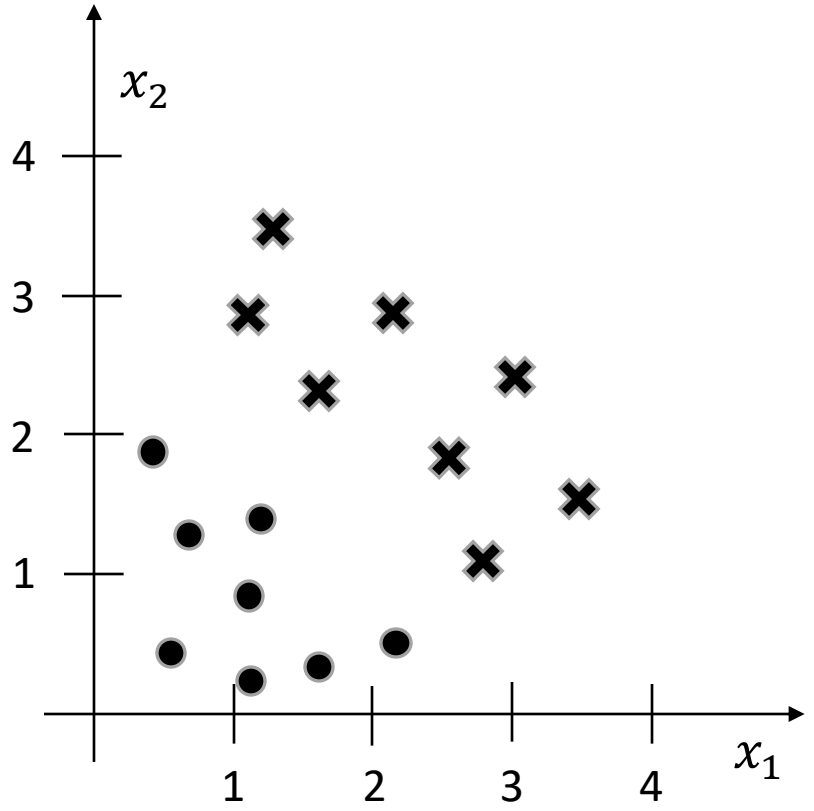
Logistic unit: decision boundary

- **Model:**

$$h_{w,b}(x) = \frac{1}{1 + e^{-w^T x + b}} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

- **Parameters (example):**

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad b = -3$$



Logistic unit: decision boundary

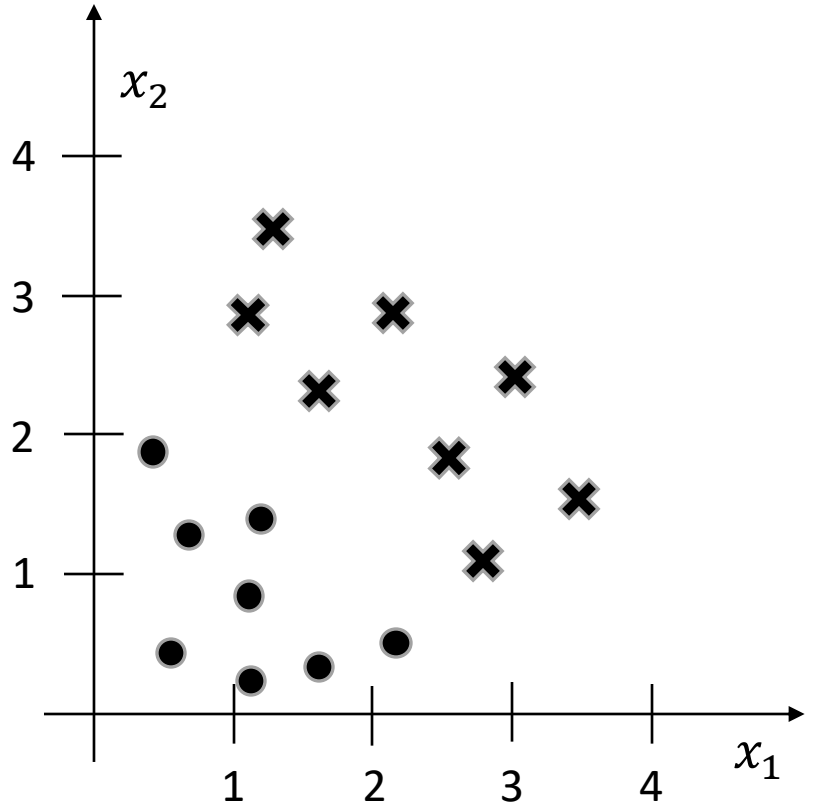
- **Model:**

$$h_{w,b}(x) = \frac{1}{1 + e^{-w^T x + b}} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

- **Parameters (example):**

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad b = -3$$

- Predict $y = 1$ if $x_1 + x_2 - 3 \geq 0$ because $h_{w,b}(x) \geq 0.5$
- Predict $y = 0$ if $x_1 + x_2 - 3 < 0$ because $h_{w,b}(x) < 0.5$



Logistic unit: decision boundary

- **Model:**

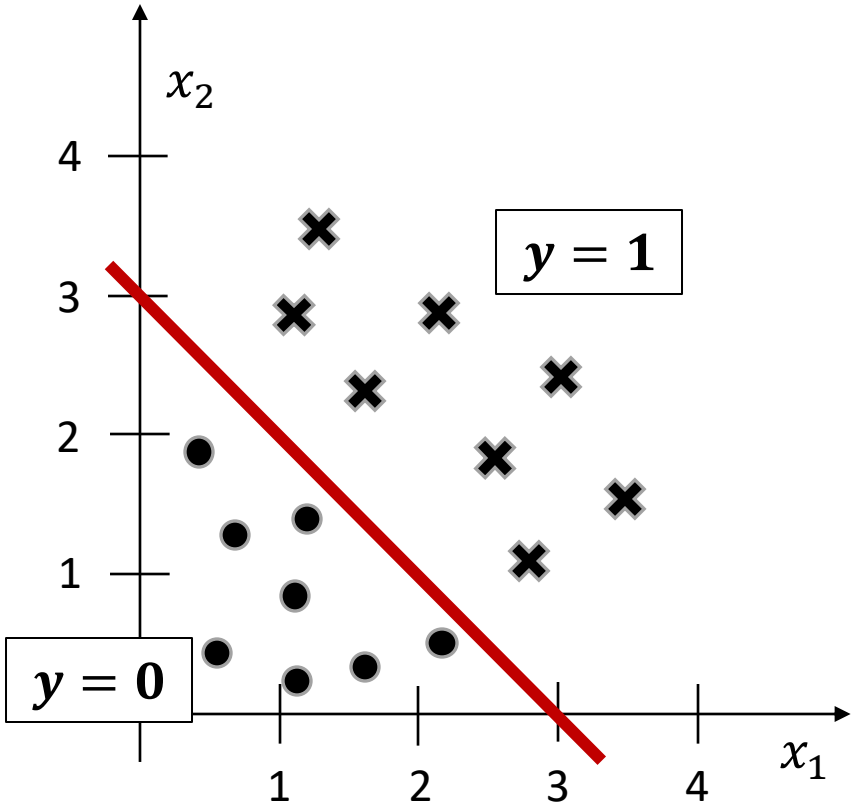
$$h_{w,b}(x) = \frac{1}{1 + e^{-w^T x + b}} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

- **Parameters (example):**

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \qquad b = -3$$

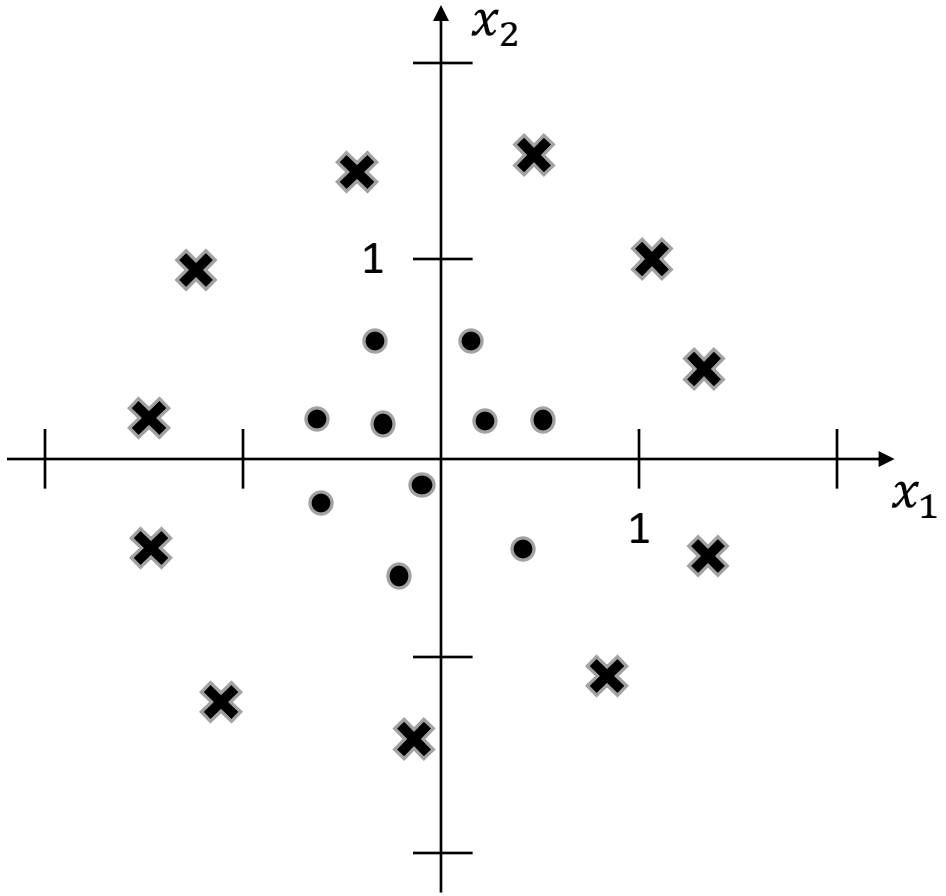
- Predict $y = 1$ if $x_1 + x_2 - 3 \geq 0$ because $h_{w,b}(x) \geq 0.5$
- Predict $y = 0$ if $x_1 + x_2 - 3 < 0$ because $h_{w,b}(x) < 0.5$
- **Decision boundary (line equation):**

$$x_1 + x_2 - 3 = 0 \rightarrow x_2 = -x_1 + 3$$



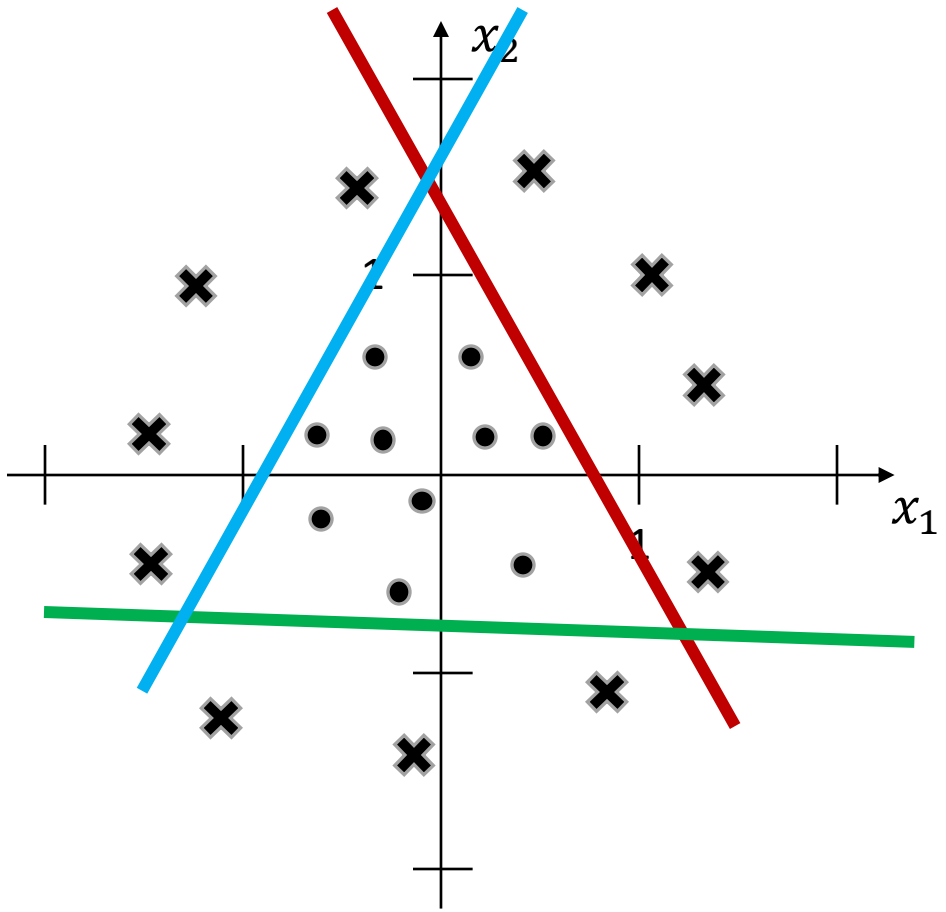
What about the hard cases?

- There is no way to separate the two classes using a straight line as decision boundary.



What about the hard case?

- There is no way to separate the two classes using a straight line as decision boundary.
- **Idea:** Could we somehow combine multiple logistic units?



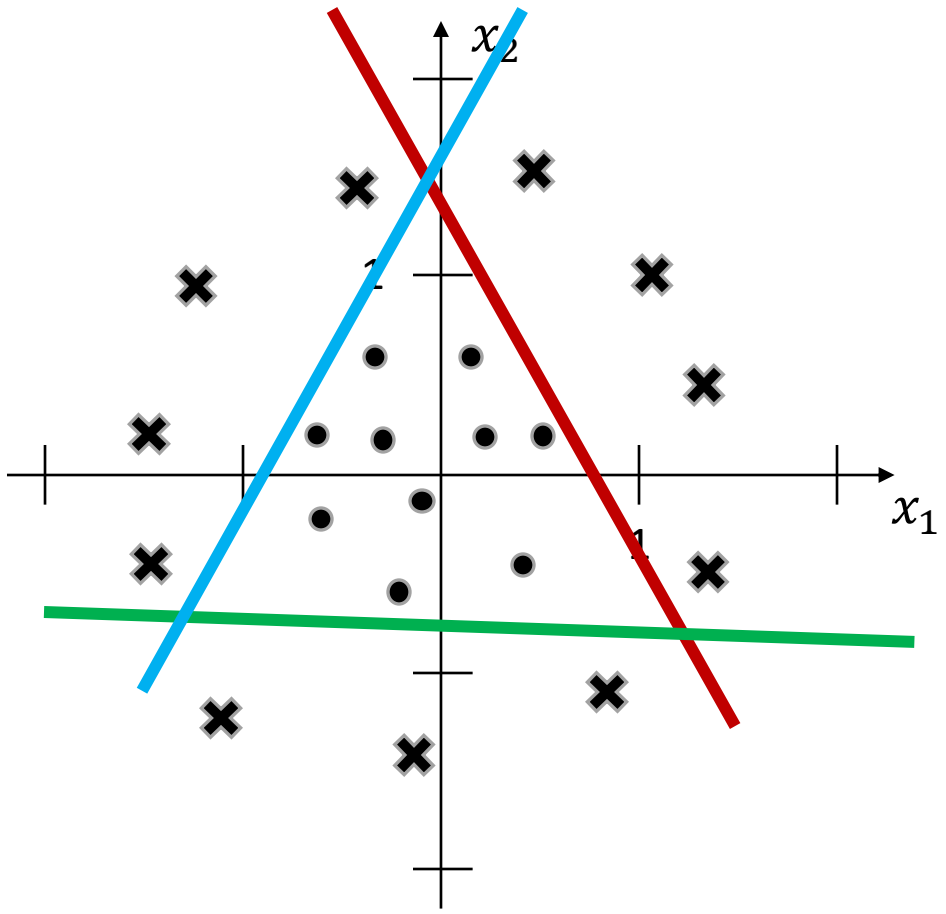
Combining multiple logistic units

- **Problem:** How do we combine them?
- **Idea:** Add them and apply a second sigmoid

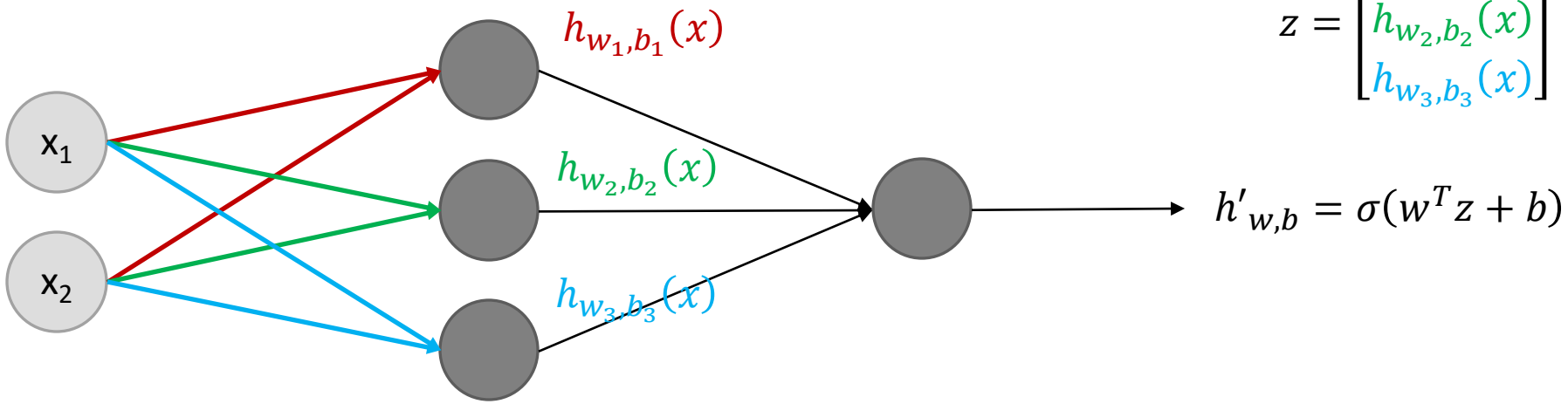
$$h'(x) = \sigma\left(h_{w_1,b_1}(x) + h_{w_2,b_2}(x) + h_{w_3,b_3}(x)\right)$$

- This is not quite going to work, because the input to the sigmoid is always going to be ≥ 0, meaning that our classifier will always predict the label y=1.
- **Solution:** Define h' to be another logistic unit and set the parameters w and b appropriately:

Define $z = \begin{bmatrix} h_{w_1,b_1}(x) \\ h_{w_2,b_2}(x) \\ h_{w_3,b_3}(x) \end{bmatrix}$ and set $h'_{w,b}(x) = \sigma(w^T z + b)$



We just built our first neural network!



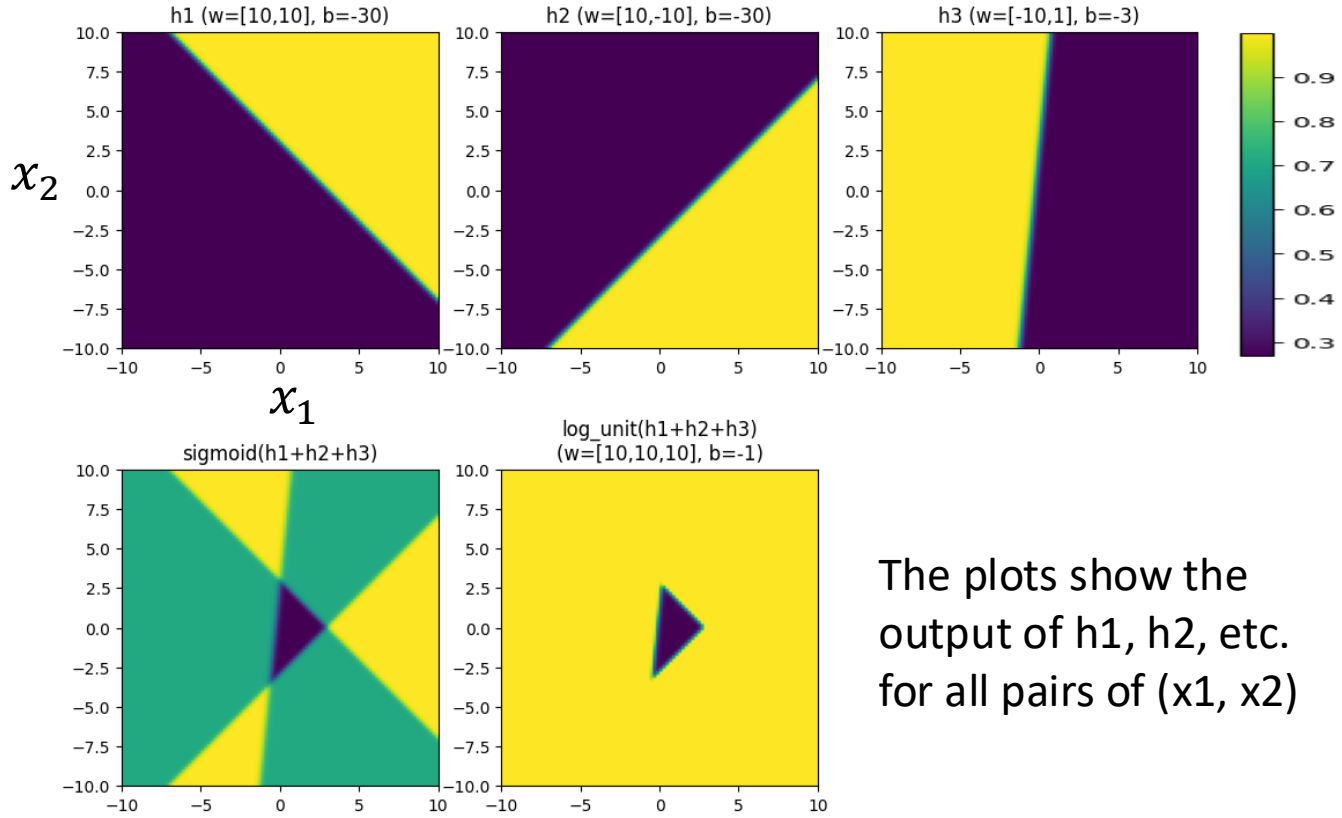
Layer 1:
Input layer
(x)

Layer 2:
Logistic units

Layer 3:
Logistic unit

$$h_{w,b}(x) = \sigma(w^T x + b)$$

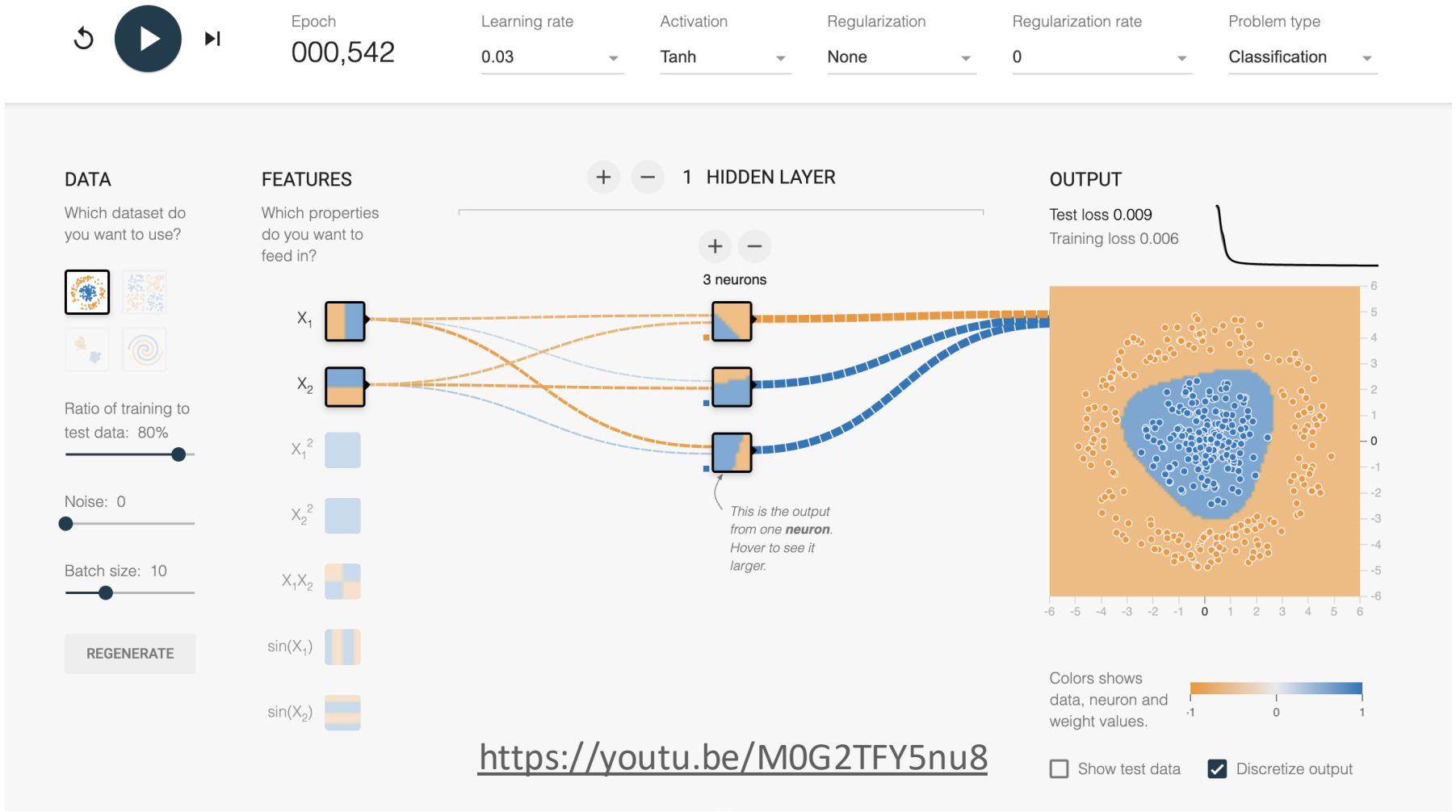
```
from matplotlib import pyplot as plt
import numpy as np

def sigmoid(z):
  return 1 / (1+np.exp(-z))

# Define 2D grid of coordinates (x1 and x2)
x_range = np.linspace(-10,10,100)
x1,x2 = np.meshgrid(x_range,x_range)

# Define three logistic units
h1 = sigmoid(10*x1+10*x2-30)
h2 = sigmoid(10*x1-10*x2-30)
h3 = sigmoid(-10*x1+x2-3)

# Take sigmoid of h1+h2+h3
h_123_sigmoid = sigmoid(h1+h2+h3)

# Use logistic unit instead
h_123_log_unit = sigmoid(10*(h1+h2+h3)-1)
```



The plots show the output of h1, h2, etc. for all pairs of (x1, x2)

```
plt.subplot(231); plt.imshow(h1, extent=[-10, 10, -10, 10],origin='lower');
plt.title('h1 (w=[10,10], b=-30)')
plt.subplot(232); plt.imshow(h2, extent=[-10, 10, -10, 10],origin='lower');
plt.title('h2 (w=[10,-10], b=-30)')
plt.subplot(233); plt.imshow(h3, extent=[-10, 10, -10, 10],origin='lower');
plt.title('h3 (w=[-10,1], b=-3)')
plt.subplot(234); plt.imshow(h_123_sigmoid, extent=[-10, 10, -10, 10],origin='lower');
plt.title('sigmoid(h1+h2+h3)')
plt.subplot(235); plt.imshow(h_123_log_unit, extent=[-10, 10, -10, 10],origin='lower');
plt.title('log_unit(h1+h2+h3)\n(w=[10,10,10], b=-1)')
```

You can try it out on your own here

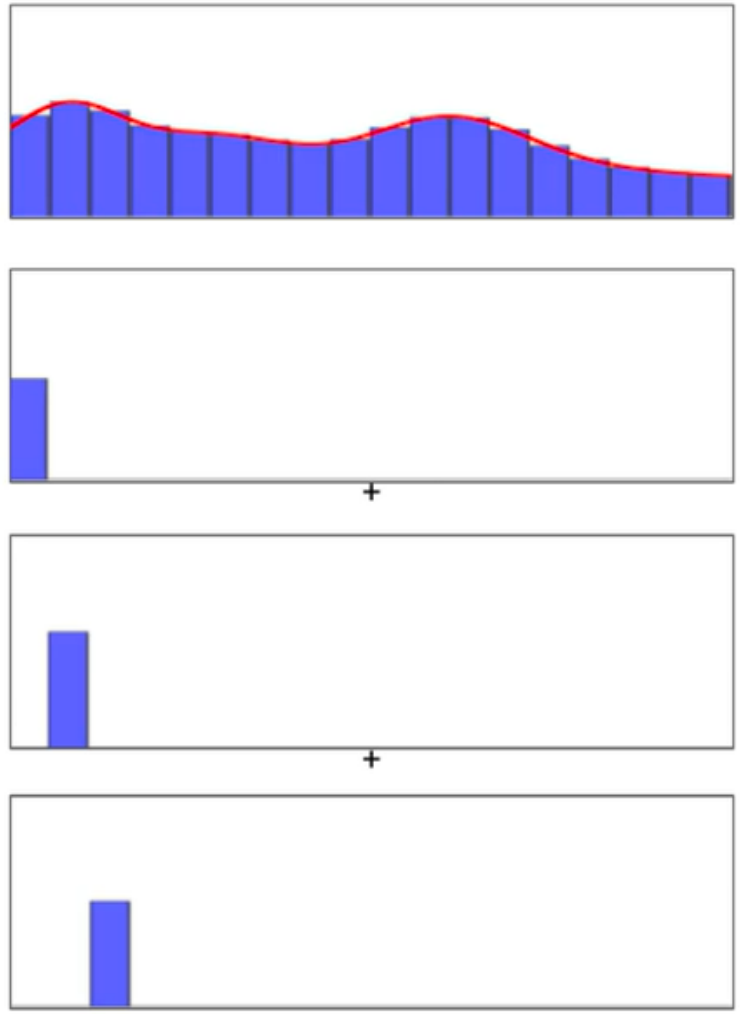


https://youtu.be/M0G2TFY5nu8

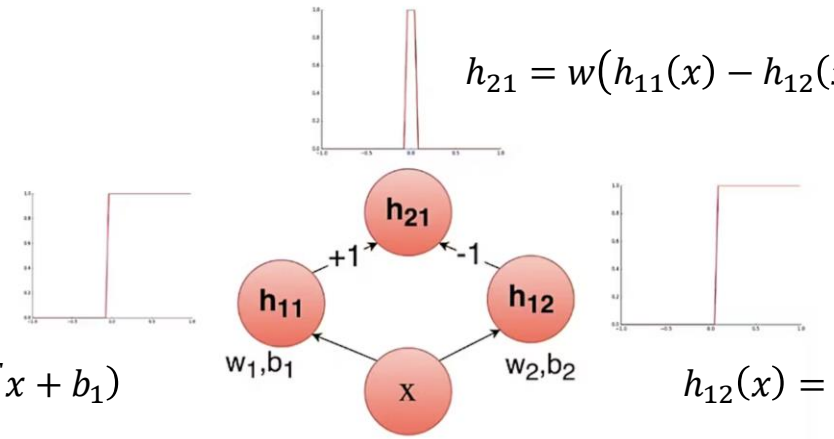
https://playground.tensorflow.org/

Universal approximation theorem

- An appropriately designed neural network can fit any function.
- Intuition explained in 1D
  - Combine triplets of neurons to generate ”tower-like” functions with **controllable** height, width and displacement.
  - Combine many tower functions to approximate the desired function (red curve).
- Generalizes to multiple dimensions.



$$h_{21} = w\big(h_{11}(x) - h_{12}(x)\big) + b$$



$$h_{11}(x) = \sigma(w_1^T x + b_1)$$

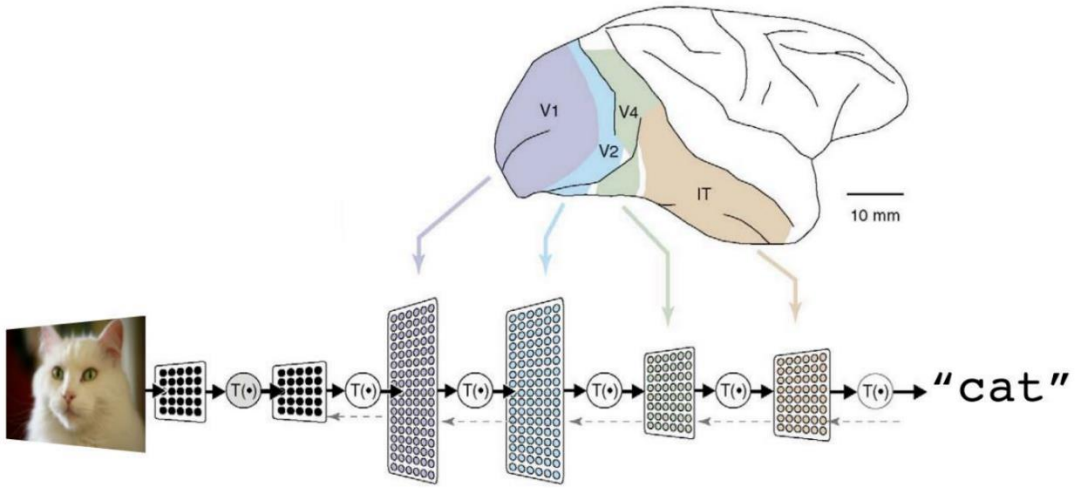
$$h_{12}(x) = \sigma(w_2^T x + b_2)$$

https://hackernoon.com/illustrative-proof-of-universal-approximation-theorem-5845c02822f6

Neural networks

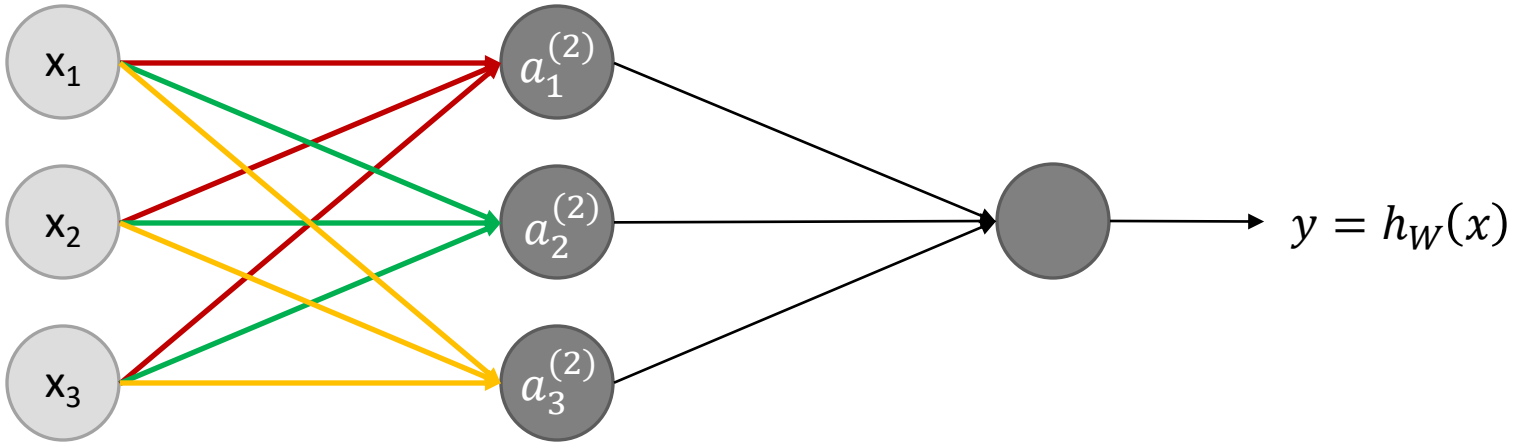
Artificial neural networks

- **Origins:** Algorithms that try to mimic the brain (most amazing learning machine we know of!)
- Were very popular in 80s and 90s, but popularity diminished in late 90s in favour of techniques like Support Vector Machines.
- **Recent resurgence:** State-of-the-art technique for many applications.
- Revolution started in mid 2000s but really kicked off in 2012 with AlexNet – mainly due to **Big Data and GPUs**, but also some algorithmic advances.
- Artificial neural networks are not exact models of how the brain works, but **they seem to perform really well on many tasks**.



A deep neural network consists of a hierarchy of layers, whereby each layer transforms the input data into more abstract representations (e.g. edge -> nose -> face). The output layer combines those features to make predictions.

Neural network

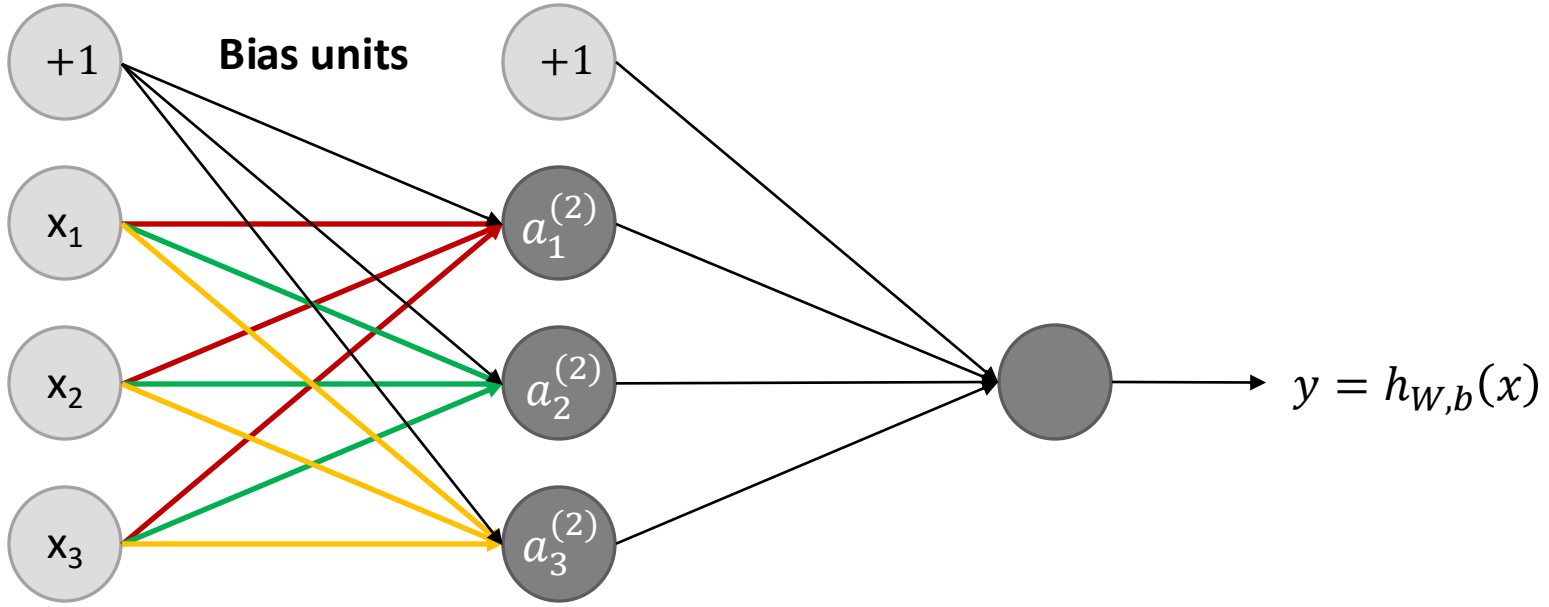


Layer 1:
Input layer
(x)

Layer 2:
Hidden layer
(a)

Layer 3:
Output layer
(y)

Neural network

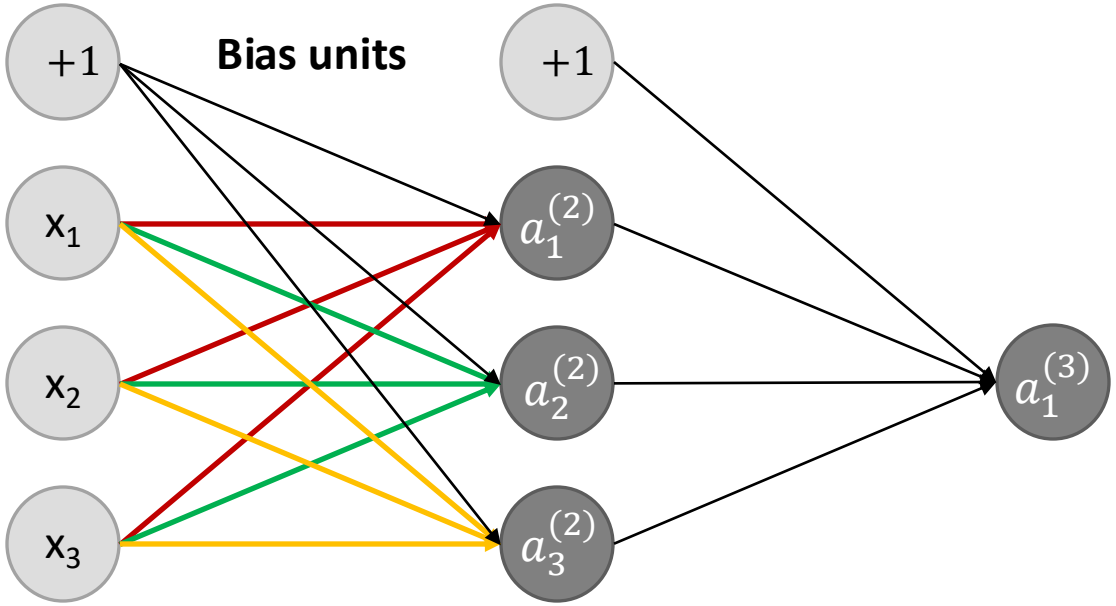


Layer 1:
Input layer
(x)

Layer 2:
Hidden layer
(a)

Layer 3:
Output layer
(y)

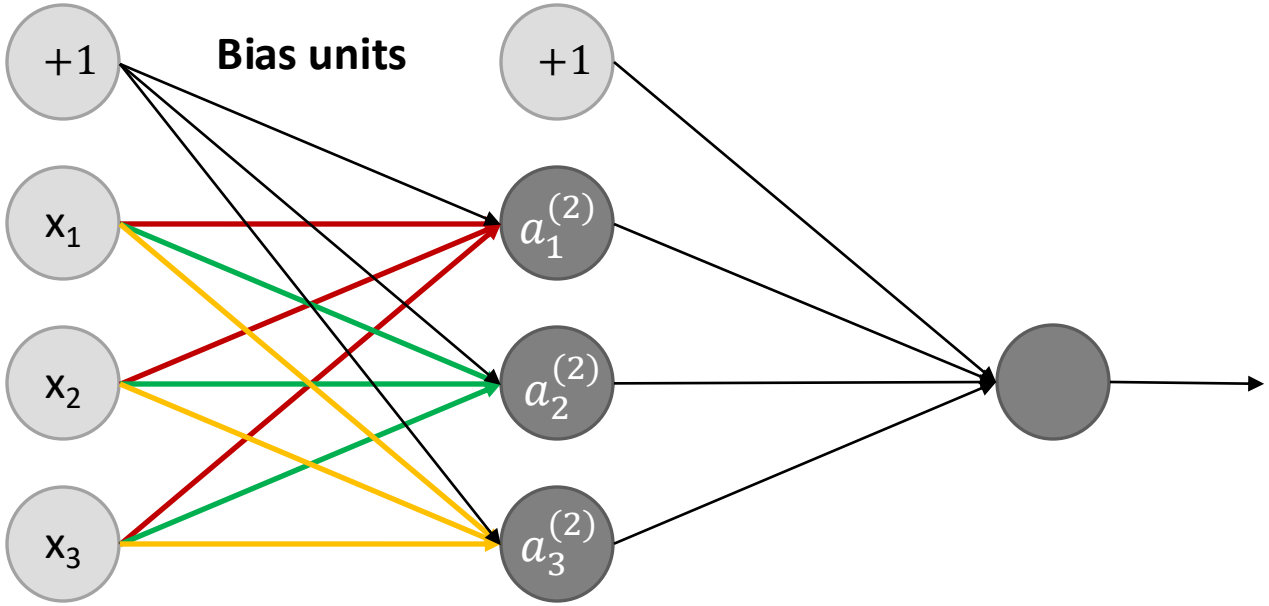
Forward propagation – summary



$$a^{(1)} = x$$
$$z^{(j+1)} = W^{(j)} a^{(j)} + b^{(j)}$$
$$a^{(j+1)} = \sigma\left(z^{(j+1)}\right)$$

$a_i^{(j)}$	= activation (i.e., output) of unit i in layer j
$W^{(j)}$	= matrix of weights mapping from layer j to layer j+1
$b^{(j)}$	= vector of biases used from layer j to layer j+1

Forward propagation



$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

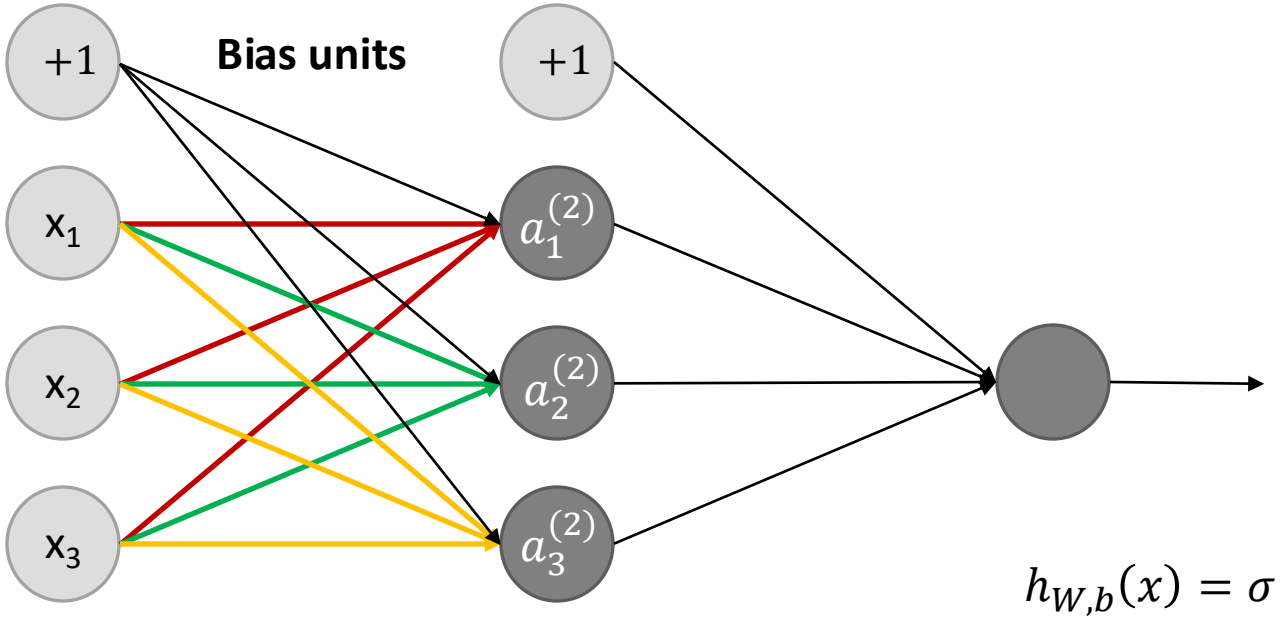
$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} & W_{33}^{(1)} \end{bmatrix}$$

$$b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix}$$

$$a_i^{(2)} = \sigma\left(W_{i1}^{(1)}x_1 + W_{i2}^{(1)}x_2 + W_{i3}^{(1)}x_3 + b_i^{(1)}\right) \text{ for } i = 1,2,3$$

Inner product between input x and i'th row of $W^{(1)}$ (think similarity…)

Forward propagation



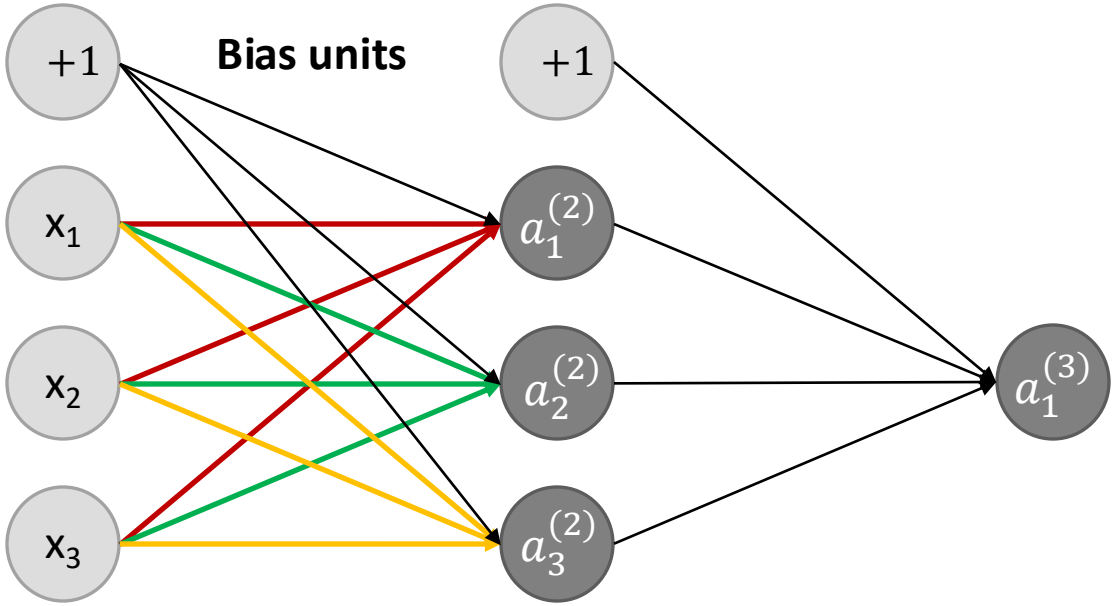
$$y = h_{W,b}(x)$$

$$W^{(2)} = \begin{bmatrix} W_{11}^{(2)} & W_{12}^{(2)} & W_{13}^{(2)} \end{bmatrix}$$

$$b^{(2)} = \text{scalar (in this case)}$$

$$h_{W,b}(x) = \sigma\left(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b^{(2)}\right)$$

Forward propagation

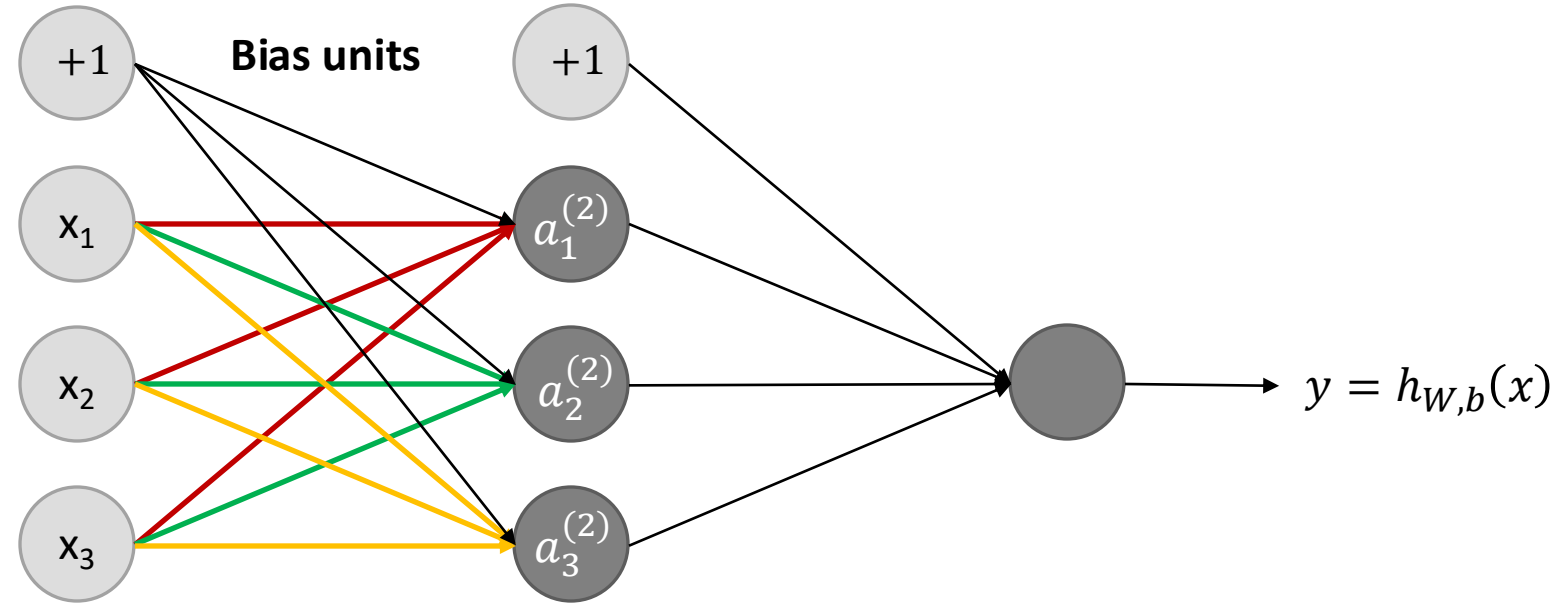


$$a^{(1)} = x$$
$$z^{(j+1)} = W^{(j)} a^{(j)} + b^{(j)}$$
$$a^{(j+1)} = \sigma\left(z^{(j+1)}\right)$$

$a_i^{(j)}$	= activation (i.e., output) of unit i in layer j
$W^{(j)}$	= matrix of weights mapping from layer j to layer j+1
$b^{(j)}$	= vector of biases used from layer j to layer j+1

NNs learn useful features

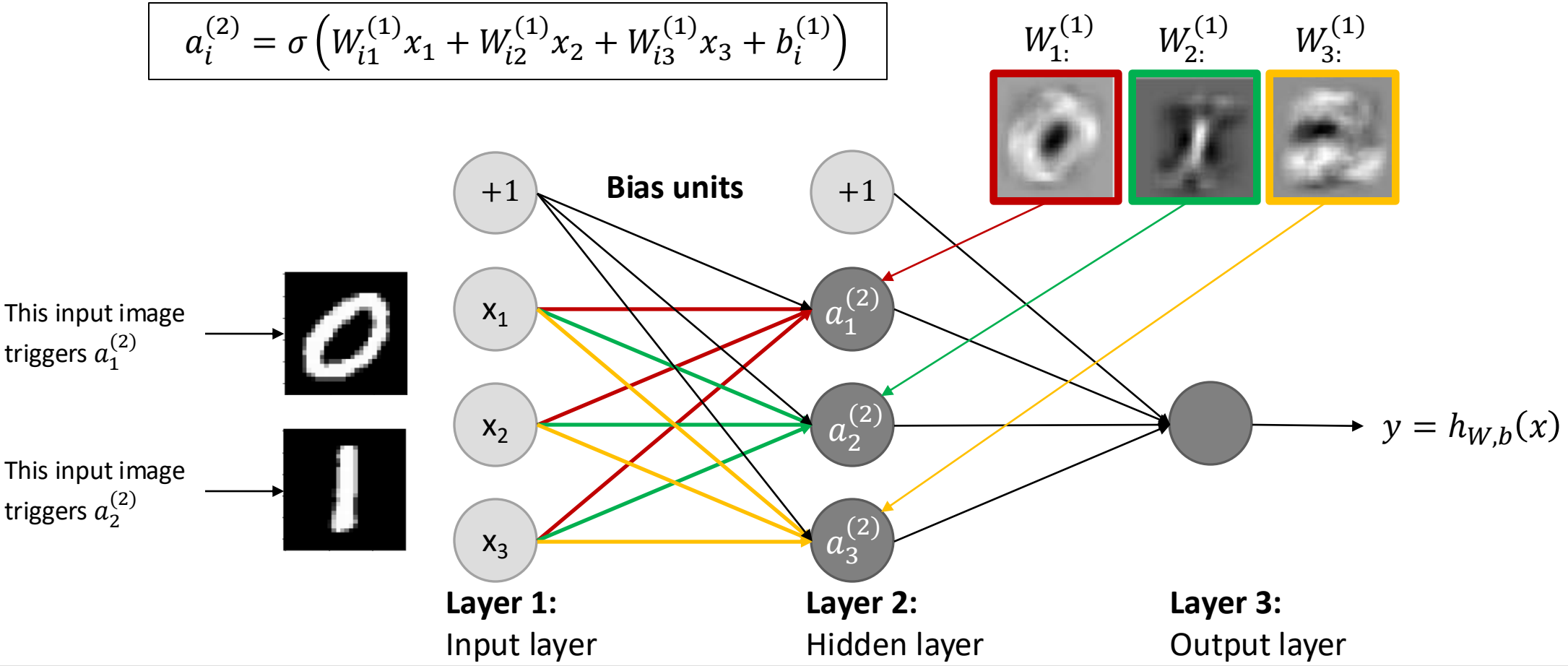
- What is so amazing about neural networks is that they **learn new feature representations** that are optimized for the task at hand.
- In terms of images, think of the hidden layer features as “templates”, and the hidden unit activations reflect how well the input matches the templates (see example on next slide).



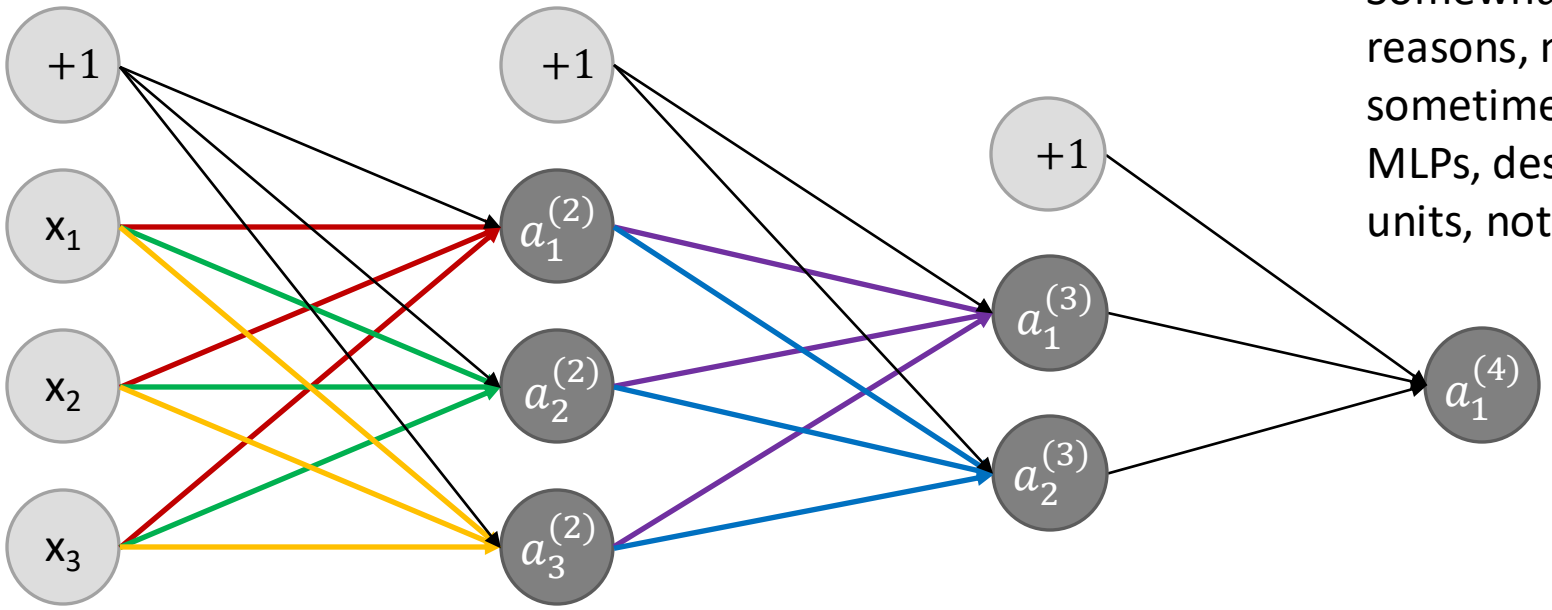
NNs learn useful features

- A hidden unit will fire or activate if the input matches the corresponding template.

$$a_i^{(2)} = \sigma\left(W_{i1}^{(1)}x_1 + W_{i2}^{(1)}x_2 + W_{i3}^{(1)}x_3 + b_i^{(1)}\right)$$

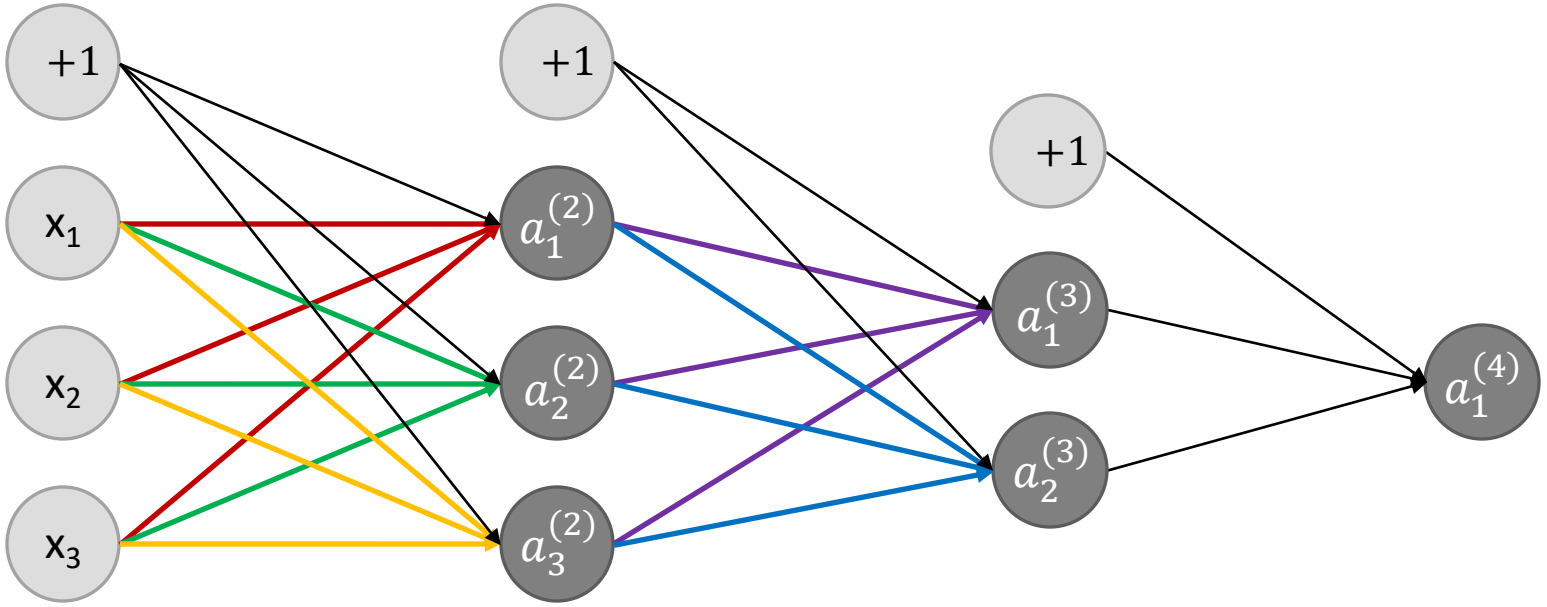


Multilayer perceptrons (MLPs)



Somewhat confusingly, and for historical reasons, multiple layer networks are sometimes called multilayer perceptrons or MLPs, despite being made up of logistic units, not perceptrons.

Multilayer perceptrons (MLPs)

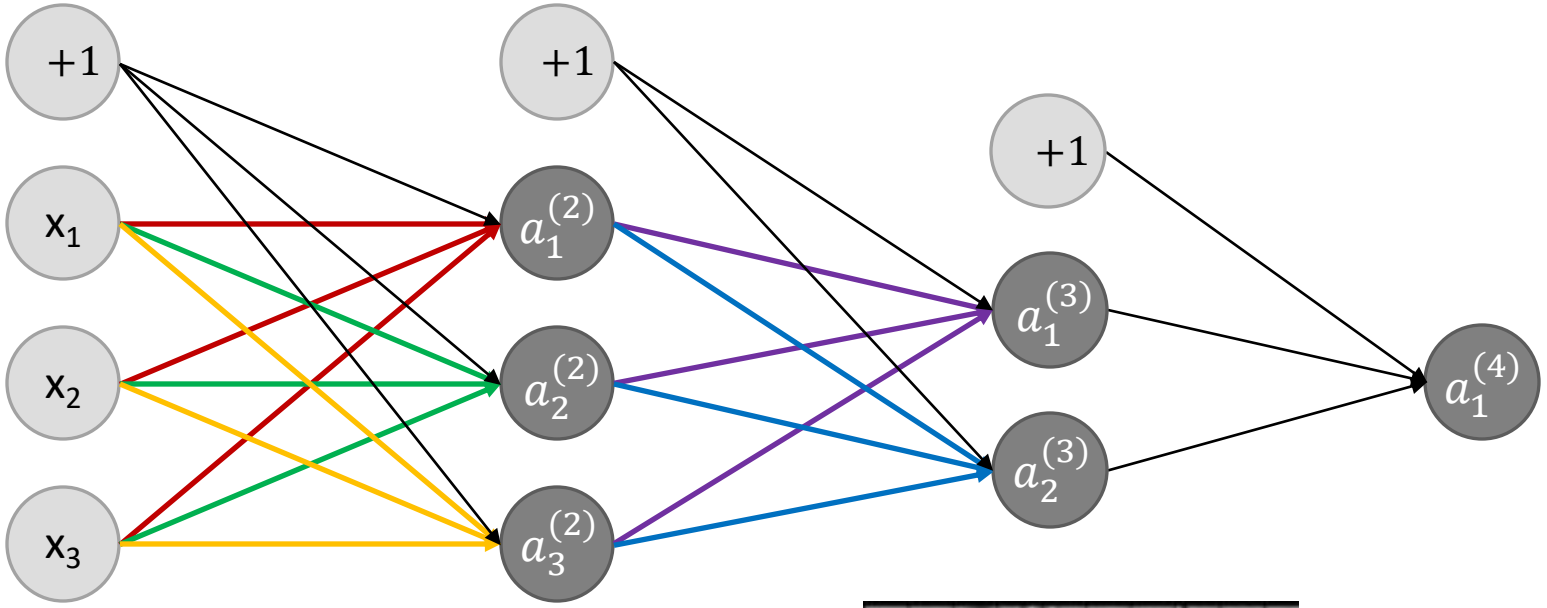


2nd layer features (based on pixels)

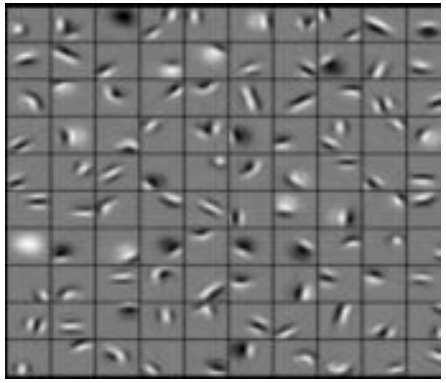
3rd layer features (based on 2nd layer features)

Why "Deep Learning"?
Neural networks learn features of features of …

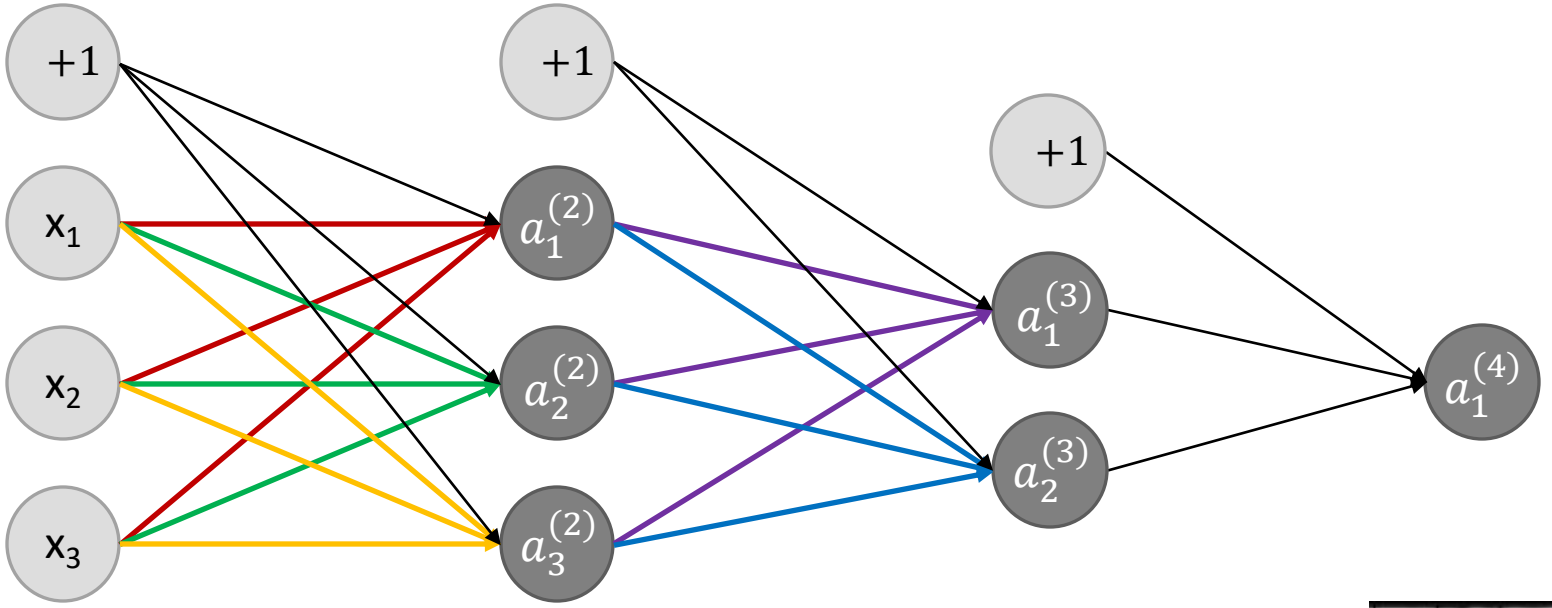
Multilayer perceptrons (MLPs)



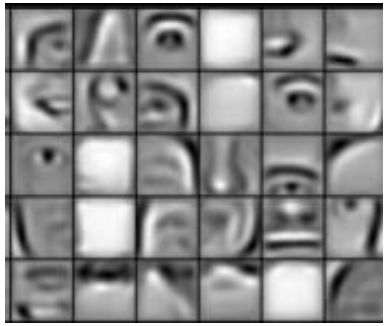
2nd layer features
(These neurons find simple features in the pixel space like edges and blobs.)



Multilayer perceptrons (MLPs)

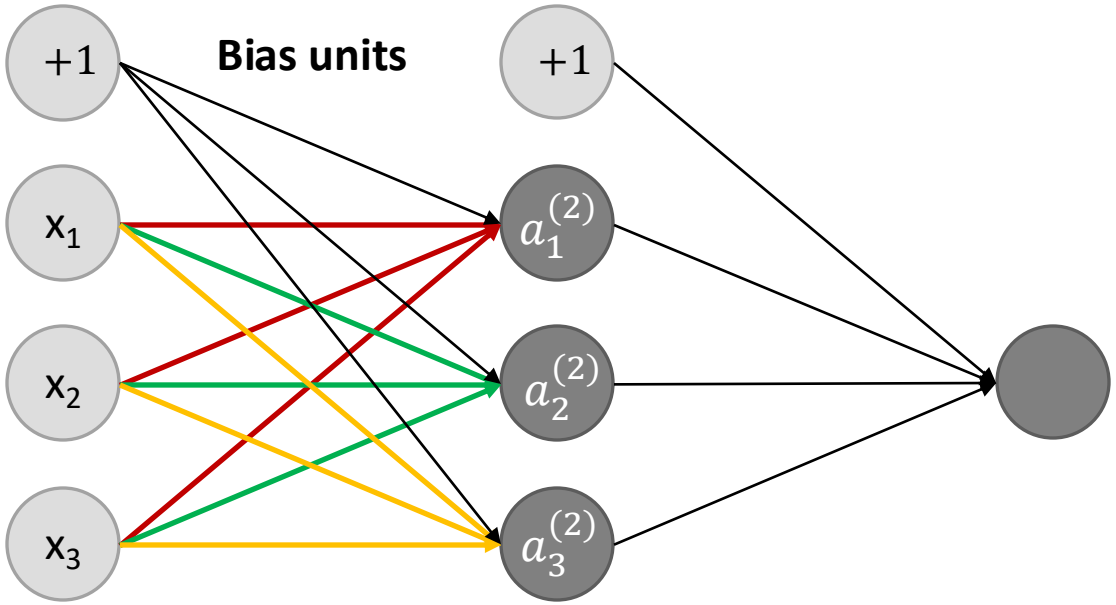


3rd layer features
(These neurons combine 2nd layer features to find more complex features like eyes, nose, etc.)



Multi-class classification

Binary classification ($K = 2$)



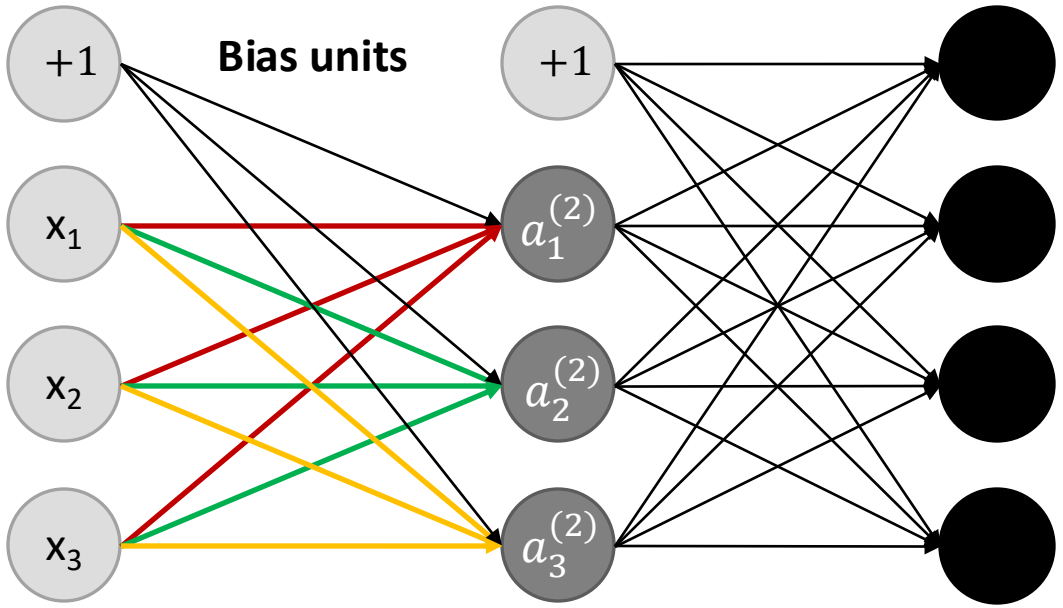
$$h_{w,b}(x) = \frac{1}{1 + e^{-w^T x + b}}$$

Task: Learn to predict y $\in \{0,1\}$ from $x \in \mathbb{R}^{m \times 1}$ given training data $\left\{y^{(i)}, x^{(i)}\right\}_{i=1}^{n}$

Model: $P(y = 1|x) = h_{w,b}(x) = \frac{1}{1+e^{-w^T x+b}} \equiv \sigma(w^T x + b)$

Multi-class classification $(K \geq 3)$

- To support multi-class classification, we can replace the last logistic unit with a **softmax** unit.



$$h_{W,b}(x) = \begin{bmatrix} P(y=1|x) \\ P(y=2|x) \\ \vdots \\ P(y=K|x) \end{bmatrix}$$

$$= \frac{1}{\sum_{j=1}^{K} \exp\left(w_j^T x + b_j\right)} \begin{bmatrix} \exp(w_1^T x + b_1) \\ \exp(w_2^T x + b_2) \\ \vdots \\ \exp(w_K^T x + b_K) \end{bmatrix}$$

Task: Learn to predict y $\in \{0,1,\ldots,K\}$ from $x \in \mathbb{R}^{m\times 1}$ given training data $\left\{y^{(i)}, x^{(i)}\right\}_{i=1}^{n}$

Model: $P(y=k|x) = \left(h_{W,b}(x)\right)_k$ = k′th element of $h_{W,b}(x)$

Multiple output units

Pedestrian

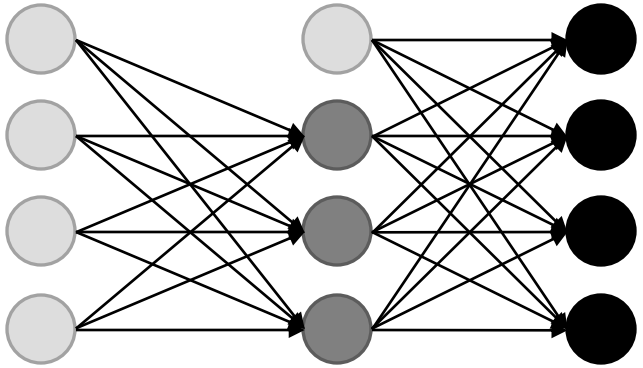
Car

Motorcycle

Truck

What we want →

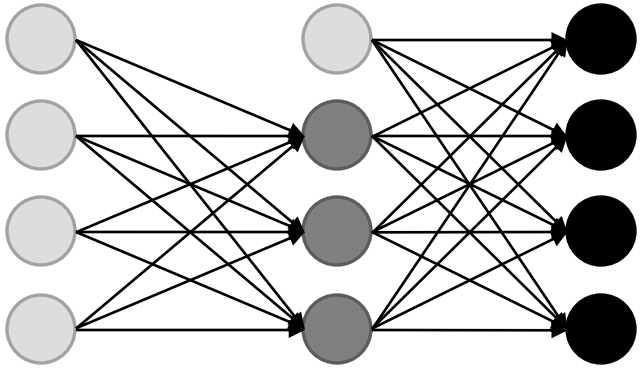
$$h_{W,b}(x) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad h_{W,b}(x) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad h_{W,b}(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad h_{W,b}(x) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$



$$h_{W,b}(x) = \begin{bmatrix} P(y = Pedestrain|x) \\ P(y = Car|x) \\ P(y = Motorcycle|x) \\ P(y = Truck|x) \end{bmatrix}$$

One-hot vectors

- The training set is $\{y^{(i)}, x^{(i)}\}_{i=1}^{n}$
- where $y^{(i)}$ is one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$
- **Goal of training:** Learn parameters such that $h_{W,b}(x^{(i)}) = y^{(i)}$



$$h_{W,b}(x) = \begin{bmatrix} P(y = Pedestrain|x) \\ P(y = Car|x) \\ P(y = Motorcycle|x) \\ P(y = Truck|x) \end{bmatrix}$$

Loss functions

- Recall the cross-entropy loss (for softmax regression):

$$J(W) = -\sum_{i=1}^{n}\sum_{k=1}^{K} \mathbf{1}\{y^{(i)} = k\} \log\left(P\left(y^{(i)} = k \middle| x^{(i)}\right)\right)$$

- With vector notation we can write the loss of our neural network in the following way:

$$J(W, b) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_{W,b}\left(x^{(i)}\right)_k\right)$$

- Where $y^{(i)}$ and $h_{W,b}\left(x^{(i)}\right)$ are K-dimensional vectors, and the notation v_k is used to refer to "the k'th element" of vector v.
- **Observation:** Because $y^{(i)}$ is a **one-hot vector**, there is only one term in the inner sum that is non-zero. Therefore, the model is rewarded for predicting the correct class label ($y_k^{(i)} = 1$), but it is not punished for predicting wrong labels (terms where $y_k^{(i)} = 0$).

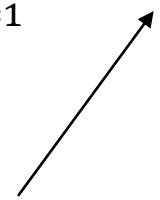
Loss functions

- How can we make all terms in the inner sum contribute to the loss?

$$J(W,b) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_{W,b}\left(x^{(i)}\right)_k\right)$$

- Simple extension (inspired by logistic loss):

$$J(W,b) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_{W,b}\left(x^{(i)}\right)_k\right) + \left(1 - y_k^{(i)}\right) \log\left(1 - h_{W,b}\left(x^{(i)}\right)_k\right)$$

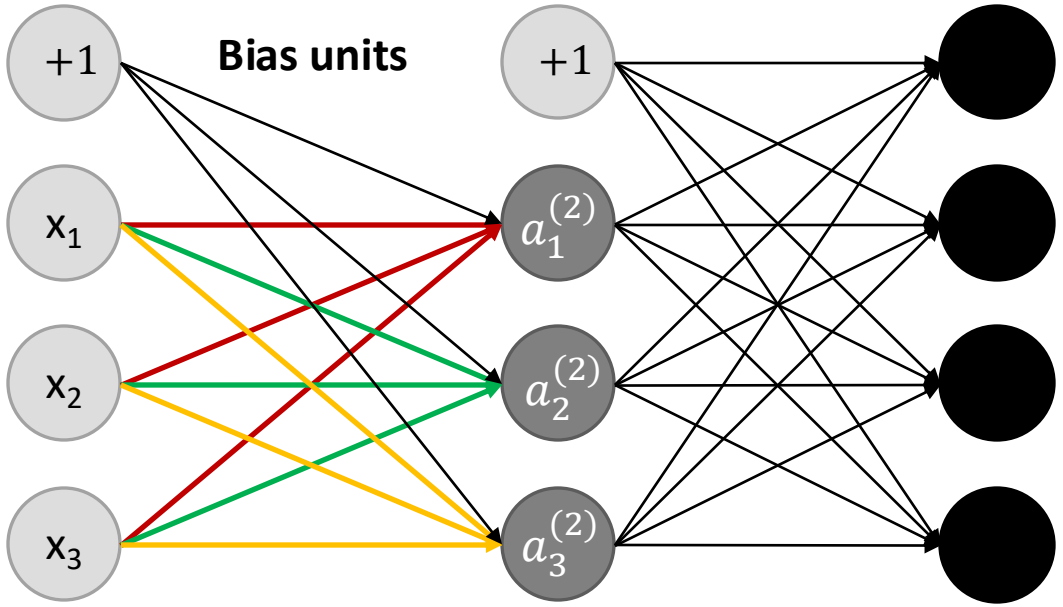


Close to zero if the model outputs high probability of positive class.

Close to zero if the model outputs low probability of negative class.

Side-note: Multi-label classification

- When using softmax the class labels are mutually exclusive. If there are multiple mutually non-exclusive classes or “labels”, we can use a separate sigmoid for each output element.



$$h_{W,b}(x) = \begin{bmatrix} \sigma(w_1^T x + b_1) \\ \sigma(w_2^T x + b_2) \\ \vdots \\ \sigma(w_K^T x + b_K) \end{bmatrix}$$

Example:
- ← Dog or cat?
- ← Black or white?
- …

Task: Learn to predict y $\in \{0,1\}^K$ from $x \in \mathbb{R}^{m\times 1}$ given training data $\{y^{(i)}, x^{(i)}\}_{i=1}^n$

Model: $P(y = 1|x) = \left(h_{W,b}(x)\right)_k$ = k′th element of $h_{W,b}(x)$

Optimization

BACKPROPAGATION

Gradient computation

- For illustration purposes, let's just pick the simple quadratic loss:

$$J(W,b) = \frac{1}{2}\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\left(h_{W,b}\left(x^{(i)}\right)_k - y_k^{(i)}\right)^2$$

- To minimize the loss using gradient descent we need to be able to compute

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} \text{ and } \frac{\partial J(W,b)}{\partial b_i^{(l)}}$$

Why do we need backpropagation?

- Suppose we want to compute the partial derivative of the loss function with respect to some weight w_j. We could approximate the gradient like this:

$$\frac{\partial J(w)}{\partial w_j} \approx \frac{J\left(w + \varepsilon e_j\right) - J(w)}{\varepsilon}$$

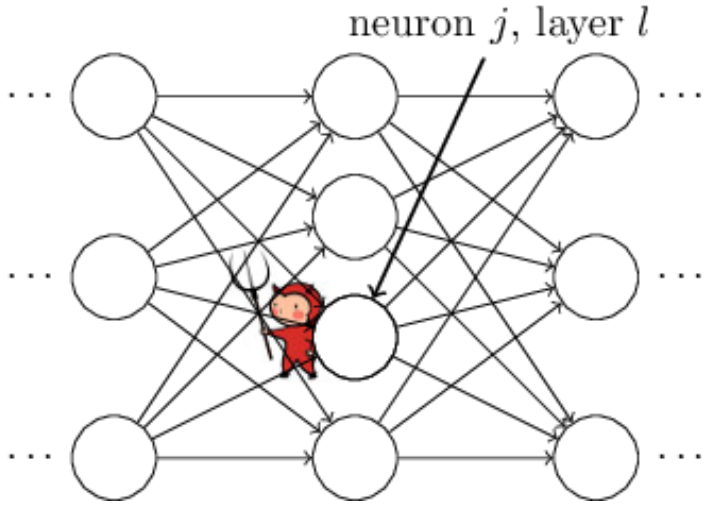
- where e_j is a unit vector pointing in the direction of w_j, and ε is a very small number.
- Unfortunately, when you implement the code, it turns out to be **extremely slow**. To understand why, imagine we have a million weights in our network. Then for each distinct weight w_j we need to compute $J\left(w + \varepsilon e_j\right)$ in order to compute $\partial J(w)/\partial w_j$. That means that **to compute the gradient we need to compute the cost function a million different times**, requiring a million forward passes through the network (per training example).
- What's clever about backpropagation is that it enables us to simultaneously compute all the partial derivatives $\partial J(w)/\partial w_j$ using just one forward pass through the network, followed by one backward pass through the network.

Backpropagation intuition

- Imagine there is a demon in our neural net: The demon sits at the j'th neuron in layer l. As the input to the neuron comes in, the demon messes with the neuron's operation. It adds a little change $\Delta z_j^{(l)}$ to the neuron's weighted input, so that instead of outputting $\sigma\left(z_j^{(l)}\right)$, the neuron instead outputs $\sigma\left(z_j^{(l)} + \Delta z_j^{(l)}\right)$.
- This change propagates through later layers in the network, finally causing the overall loss to change by an amount

$$\frac{\partial J(W,b)}{\partial z_j^{(l)}} \Delta z_j^{(l)}$$

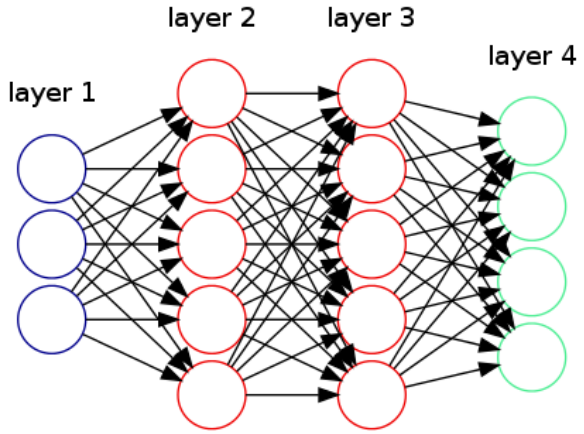
- Defining $\delta_j^{(l)} = \frac{\partial J(W,b)}{\partial z_j^{(l)}}$, we can use it to figure out how to set $\Delta z_j^{(l)}$, such that the loss is minimized.



http://neuralnetworksanddeeplearning.com/chap2.html

Notation

- L: Total number of layers
- s_l: No. of units (not counting bias unit) in layer l.
- $(W, b) = \left(W_{ij}^{(l)}, b_i^{(l)}\right)_{l=1}^{n_L}$: Model parameters, where we write $W_{ij}^{(l)}$ to denote the weight associated with the connection between unit j in layer l, and unit i in layer $l+1$ (note the order of the indices). Also, $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$. $W^{(l)}$ has size $s_{l+1} \times s_l$ and $b^{(l)}$ has length s_{l+1}.



Example:

$L = 4, s_1 = 3, s_2 = 5, s_3 = 5, s_4 = 4$

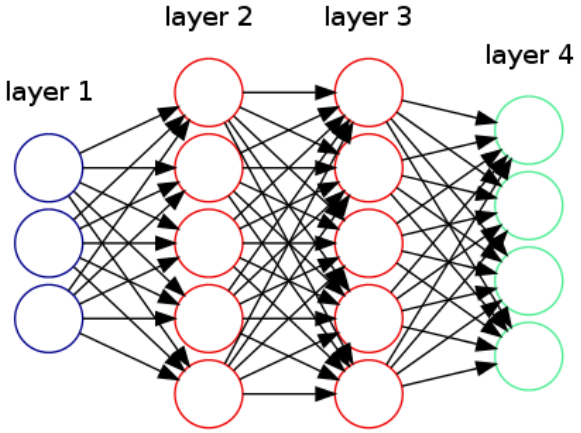
$W^{(1)} \in \mathbb{R}^{5\times3}, W^{(2)} \in \mathbb{R}^{5\times5}, W^{(3)} \in \mathbb{R}^{4\times5}$

$b^{(1)} \in \mathbb{R}^5, b^{(2)} \in \mathbb{R}^5, b^{(3)} \in \mathbb{R}^4$

First step: Forward propagation

$x^{(k)}$ is the k'th training example

$$a^{(1)} = x^{(k)}$$
$$z^{(2)} = W^{(1)}a^{(1)} + b^{(1)}$$
$$a^{(2)} = \sigma\left(z^{(2)}\right)$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$
$$a^{(3)} = \sigma\left(z^{(3)}\right)$$
$$z^{(4)} = W^{(3)}a^{(3)} + b^{(3)}$$
$$a^{(4)} = \sigma\left(z^{(4)}\right)$$
$$= h_{W,b}\left(x^{(k)}\right)$$



$$a^{(1)} = x$$
$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = \sigma\left(z^{(l+1)}\right)$$

$y^{(k)}$ is the target/label of $x^{(k)}$

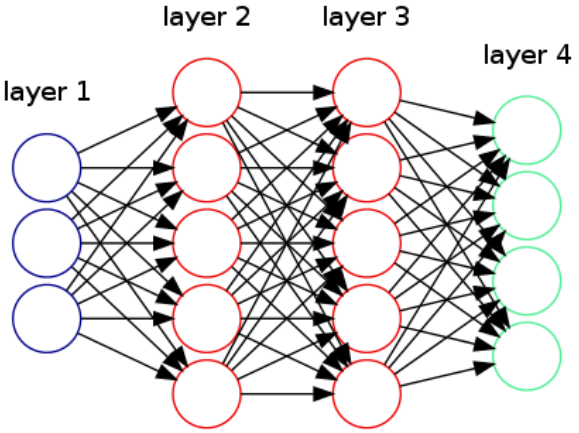
Goal of training:

Learn parameters such that $h_{W,b}\left(x^{(k)}\right) = y^{(k)}$

Second step: Backpropagation

- **Intuition:** $\delta_j^{(l)}$ = "error" of unit j in layer l.
- $\delta^{(l)}$ is a vector

$$\delta^{(4)} = \left(a^{(4)} - y^{(k)}\right) \odot \sigma'\left(z^{(4)}\right)$$
$$\delta^{(3)} = \left(W^{(3)}\right)^T \delta^{(4)} \odot \sigma'\left(z^{(3)}\right)$$
$$\delta^{(2)} = \left(W^{(2)}\right)^T \delta^{(3)} \odot \sigma'\left(z^{(2)}\right)$$



- There is no $\delta^{(1)}$ term, because layer 1 corresponds to the input (x).
- $\odot$ denotes element-wise matrix or vector multiplication
- And $\sigma'\left(z^{(j)}\right) = \sigma\left(z^{(j)}\right) \odot \left(1 - \sigma\left(z^{(j)}\right)\right) = a^{(j)} \odot \left(1 - a^{(j)}\right)$ is the derivate of the sigmoid function (σ).

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial J(W,b)}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

Backpropagation algorithm

- The training set is $\{y^{(k)}, x^{(k)}\}_{k=1}^{n}$
- Set $\Delta W_{ij}^{(l)} = 0$ (for all i, j, l) and $\Delta b_i^{(l)} = 0$ (for all i, l)
- For $k = 1$ to n:
  - Set $a^{(1)} = x^{(k)}$
  - Perform forward propagation to compute $a^{(l)}$ for all $l = 2, \ldots, L$
  - Using $y^{(k)}$, compute $\delta^{(L)} = \left(a^{(L)} - y^{(k)}\right) \odot \sigma'\left(z^{(L)}\right)$
  - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  - $\Delta W_{ij}^{(l)} = \Delta W_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ for all i, j in layers $l = 1, \ldots, L-1$
  - $\Delta b_i^{(l)} = \Delta b_i^{(l)} + \delta_i^{(l+1)}$ for all i in layers $l = 1, \ldots, L-1$
- Update parameters (gradient descent)

Note that we use *k* to index over training examples. We cannot use *i* here, because *i* refers to a unit in the network.

Backpropagation algorithm

- The training set is $\{y^{(k)}, x^{(k)}\}_{k=1}^{n}$
- Set $\Delta W_{ij}^{(l)} = 0$ (for all i, j, l) and $\Delta b_i^{(l)} = 0$ (for all i, l)
- For $k = 1$ to n:
  - Set $a^{(1)} = x^{(k)}$
  - Perform forward propagation to compute $a^{(l)}$ for all $l = 2, \ldots, L$
  - Using $y^{(k)}$, compute $\delta^{(L)} = \left(a^{(L)} - y^{(k)}\right) \odot \sigma'\left(z^{(L)}\right)$
  - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  - $\Delta W_{ij}^{(l)} = \Delta W_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ for all i, j in layers $l = 1, \ldots, L-1$
  - $\Delta b_i^{(l)} = \Delta b_i^{(l)} + \delta_i^{(l+1)}$ for all i in layers $l = 1, \ldots, L-1$
- Update parameters (gradient descent)

Calculating the gradients

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = \frac{1}{n} \Delta W_{ij}^{(l)}$$

$$\frac{\partial J(W,b)}{\partial b_i^{(l)}} = \frac{1}{n} \Delta b_i^{(l)}$$

Backpropagation algorithm

- The training set is $\{y^{(k)}, x^{(k)}\}_{k=1}^{n}$
- Set $\Delta W_{ij}^{(l)} = 0$ (for all i, j, l) and $\Delta b_i^{(l)} = 0$ (for all i, l)
- For $k = 1$ to n:
  - Set $a^{(1)} = x^{(k)}$
  - Perform forward propagation to compute $a^{(l)}$ for all $l = 2, \ldots, L$
  - Using $y^{(k)}$, compute $\delta^{(L)} = \left(a^{(L)} - y^{(k)}\right) \odot \sigma'\left(z^{(L)}\right)$
  - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  - $\Delta W_{ij}^{(l)} = \Delta W_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ for all i, j in layers $l = 1, \ldots, L-1$
  - $\Delta b_i^{(l)} = \Delta b_i^{(l)} + \delta_i^{(l+1)}$ for all i in layers $l = 1, \ldots, L-1$
- Update parameters (gradient descent)

Calculating the gradients

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = \frac{1}{n} \Delta W_{ij}^{(l)}$$

$$\frac{\partial J(W,b)}{\partial b_i^{(l)}} = \frac{1}{n} \Delta b_i^{(l)}$$

Parameter update (gradient descent)

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial J(W,b)}{\partial W_{ij}^{(l)}}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial J(W,b)}{\partial b_i^{(l)}}$$

Backpropagation algorithm

- The training set is $\{y^{(k)}, x^{(k)}\}_{k=1}^{n}$
- Set $\Delta W_{ij}^{(l)} = 0$ (for all i, j, l) and $\Delta b_i^{(l)} = 0$ (for all i, l)
- For $k = 1$ to n:
  - Set $a^{(1)} = x^{(k)}$
  - Perform forward propagation to compute $a^{(l)}$ for all $l = 2, \ldots, L$
  - Using $y^{(k)}$, compute $\delta^{(L)} = \left(a^{(L)} - y^{(k)}\right) \odot \sigma'\left(z^{(L)}\right)$
  - Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  - $\Delta W_{ij}^{(l)} = \Delta W_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ for all i, j in layers $l = 1, \ldots, L-1$
  - $\Delta b_i^{(l)} = \Delta b_i^{(l)} + \delta_i^{(l+1)}$ for all i in layers $l = 1, \ldots, L-1$
- Update parameters (gradient descent)

Vectorized computations

$$\Delta W^{(l)} = \Delta W^{(l)} + \delta^{(l+1)} \left(a^{(l)}\right)^T$$

$$\Delta b^{(l)} = \Delta b^{(l)} + \delta^{(l+1)}$$

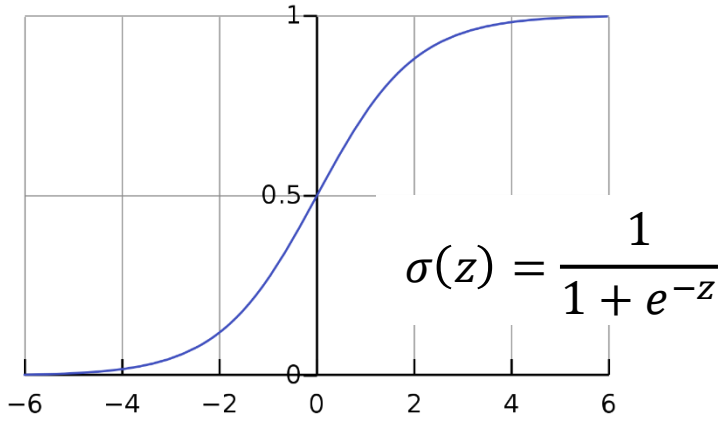
Training NNs in practise

Saturation

- The sigmoid function becomes very flat when $|z|$ is large.
- When this occurs, we will have $\sigma'(z) \approx 0$.
- Now, since the derivatives of the loss w.r.t. to the parameters

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} \text{ and } \frac{\partial J(W,b)}{\partial b_i^{(l)}}$$

- depend on $\delta_i^{(l+1)}$, which again depends on $\sigma'\left(z^{(l+1)}\right)$, a parameter will **learn slowly** if the neuron is either low activation ($\sigma \approx 0$) or high activation ($\sigma \approx 1$).
- In this case it's common to say the neuron has saturated and, as a result, the parameter has stopped learning (or is learning slowly).



Other activation functions

- The four fundamental equations of backprop turn out to hold for any activation function, not just the standard sigmoid function (because the proofs don't use any special properties of σ).

$$\delta^{(L)} = \left(a^{(L)} - y\right) \odot f'\left(z^{(L)}\right) = \left(h_{W,b}(x) - y\right) \odot f'\left(z^{(L)}\right)$$

$$\delta^{(l)} = \left(W^{(l)}\right)^T \delta^{(l+1)} \odot f'\left(z^{(l)}\right) \text{ for } l = L-1, L-2, \ldots, 2$$

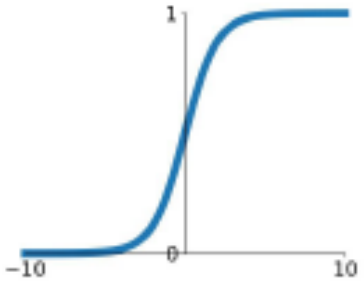
$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \text{ and } \frac{\partial J(W,b)}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

- where f denotes any activation function
- Could we use the four basic equations to design activation functions which have particular desired learning properties?
- As an example, to give you the idea, suppose we were to choose a non-sigmoid activation function f so that f' is always positive, and never gets close to zero. That would prevent the slow-down of learning that occurs when ordinary sigmoid neurons saturate.

Examples of activation functions

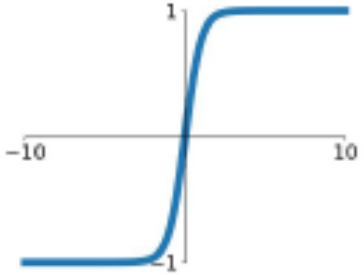
Sigmoid

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



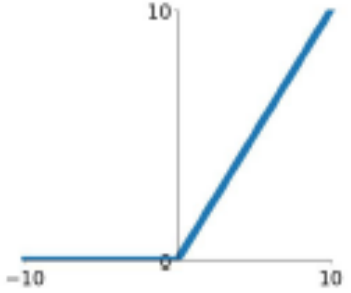
tanh

$$\tanh(x)$$



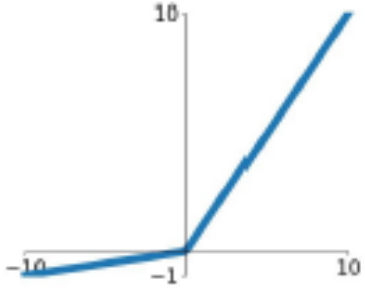
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

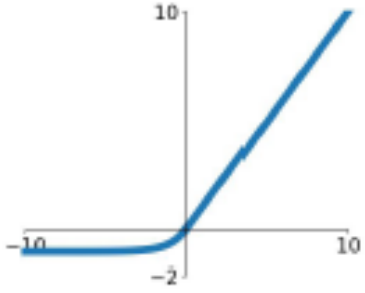


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

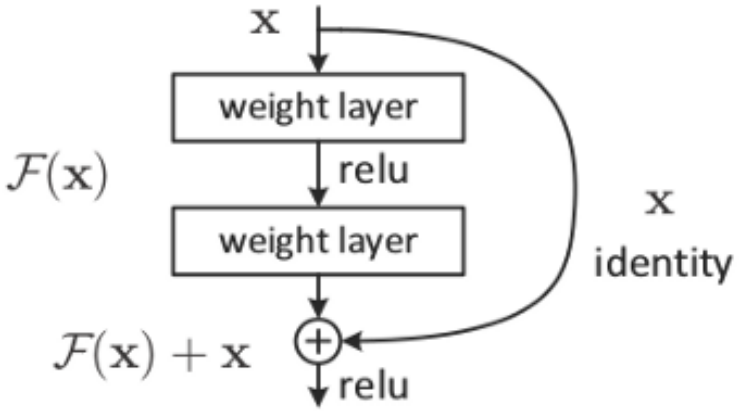


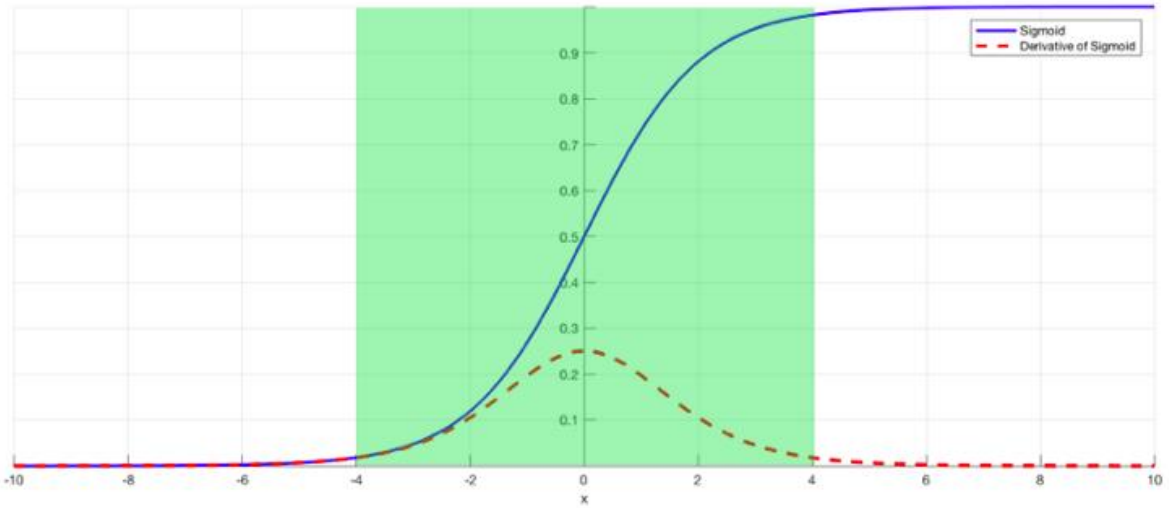
Vanishing gradient problem

- Saturation is related to the vanishing gradient problem.
- As more layers are added to a neural network, the **gradients of the loss function can approach zero**, making the network hard to train.
- The gradient gets smaller and smaller the further you move back from the output layer of the network towards the input layer.
- Intuition
  - The sigmoid function squishes a large input range into a small output range between 0 and 1.
  - In backpropagation this has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer network, meaning that the gradient decreases exponentially with n (i.e. the first layers train very slowly).
- Two possible solutions: Batch normalization and residual networks (later lecture).

Solutions to vanishing gradient problem

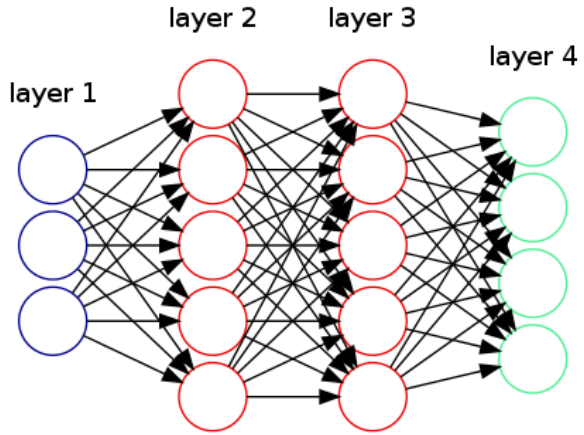
- The simplest solution is to use another activation functions, such as ReLU, which doesn't cause a small derivative.
- Residual networks are another solution, as they provide residual connections straight to earlier layers.
- Finally, batch normalization layers can also help resolve the issue. Batch normalization reduces the problem by simply normalizing the input so |x| doesn't reach the outer edges of the sigmoid function.
- We will cover these techniques in more detail later in the course.





Model size

- How many parameters are there?
- Example: MNIST
- Input images are 784 dimensional vectors (recall 28 x 28 pixels).
- Outputs are 10 dimensional vectors (one for each of the 10 digits)
- Let's use two hidden layers with 30 units each.
- Then $L = 4, s_1 = 784, s_2 = 30, s_3 = 30, s_4 = 10$
- $W^{(1)} \in \mathbb{R}^{30\times 784}, W^{(2)} \in \mathbb{R}^{30\times 30}, W^{(3)} \in \mathbb{R}^{10\times 30}$
- $b^{(1)} \in \mathbb{R}^{30}, b^{(2)} \in \mathbb{R}^{30}, b^{(3)} \in \mathbb{R}^{10}$
- $\text{N. o. parameters} = 30 \times 784 + 30 \times 30 + 10 \times 30 + 30 + 30 + 10 = 24{,}790$
- **With only 60,000 images (observations) in the training set, what do you think will happen?**



Neural networks are prone to overfitting

- Our 2 x 30 hidden neuron network for classifying MNIST digits has nearly 25,000 parameters.
- That's a lot of parameters, but state-of-the-art deep neural nets sometimes contain millions or even billions of parameters.
- Models with a large number of free parameters can describe an amazingly wide range of phenomena, but even if such a model agrees well with the available training data, that doesn’t necessarily make it a good model.
- **It may just mean there's enough capacity in the model that it can describe the training dataset, without capturing any genuine insights into the underlying phenomenon.**
- When that happens, the model will work well for the training data but will fail to generalize to new (unseen) data.

L2 regularization (weight decay)

- Of course, this demands for some regularization.
- The simplest solution is weight decay, where we simply add an L2 regularization term to the chosen cost function.
- For the quadratic loss that would be

$$J(W,b) = \frac{1}{2}\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\left(h_{W,b}\left(x^{(i)}\right)_k - y_k^{(i)}\right)^2 + \frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{j=1}^{s_l}\sum_{i=1}^{s_{l+1}}\left(W_{ij}^{(l)}\right)^2$$

- The partial derivative then simply gets an extra term to it

Without L2 regularization:

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = \frac{1}{n}\Delta W_{ij}^{(l)}$$

As derived earlier (see backprop)

With L2 regularization:

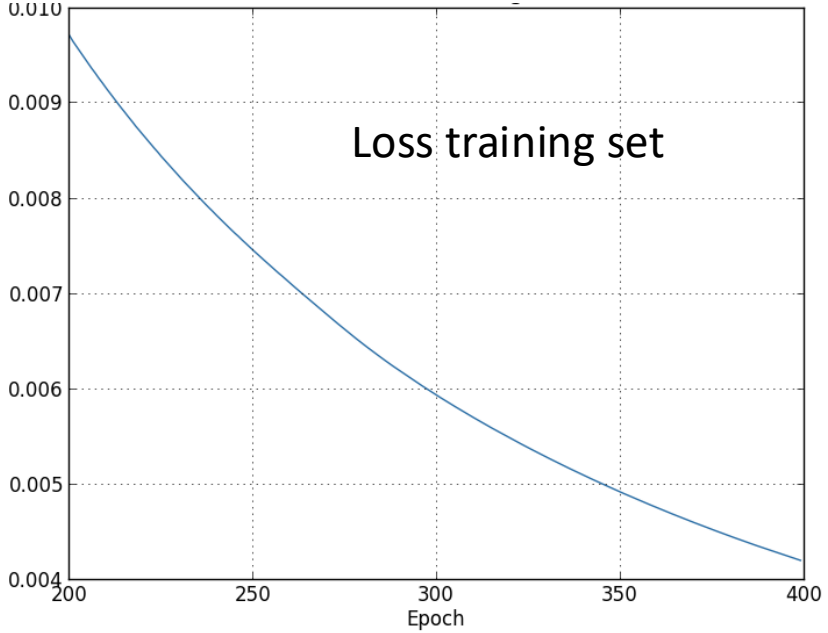
$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = \frac{1}{n}\Delta W_{ij}^{(l)} + \lambda W_{ij}^{(l)}$$

Other types of regularization

- We will be looking more into regularization techniques later in the course.
- Other techniques include
  - Dropout
  - Early stopping
  - Batch normalization
  - Data augmentation
  - Stochastic gradient descent
  - Using other activation functions (like ReLU)
  - …

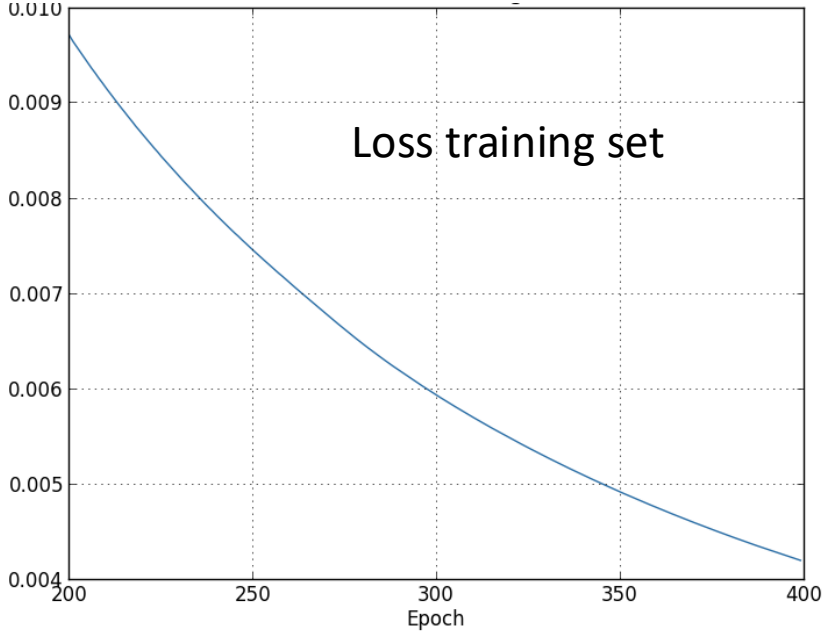
Detecting overfitting

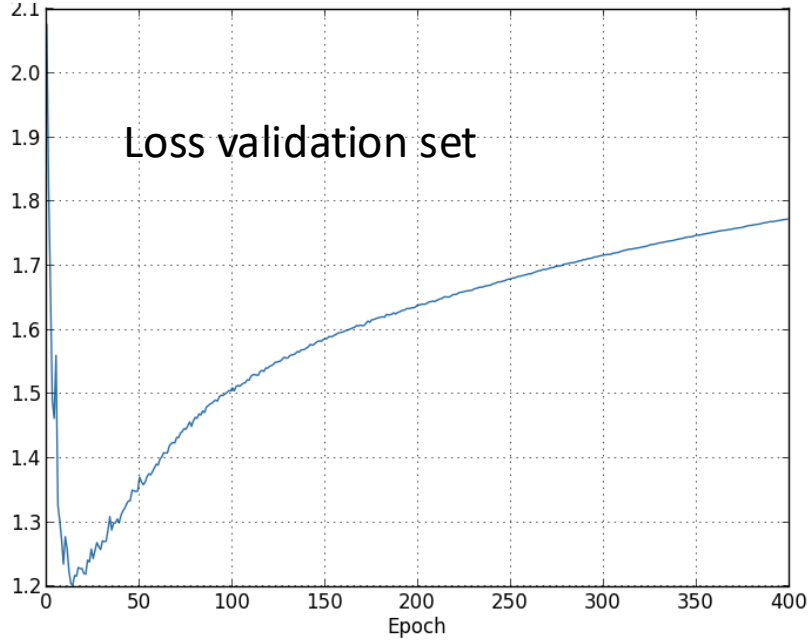
- If you look at the loss function evaluated on the training set, it will always decrease over time (if you got all the hyperparameters right, like learning rate).
- Remember that we have also talked about validation sets and test sets?



Detecting overfitting

- If you look at the loss function evaluated on the training set, it will always decrease over time (if you got all the hyperparameters right, like learning rate).
- Remember that we have also talked about validation sets and test sets?
- If the loss evaluated on the validation set starts to increase, your model is overfitting.

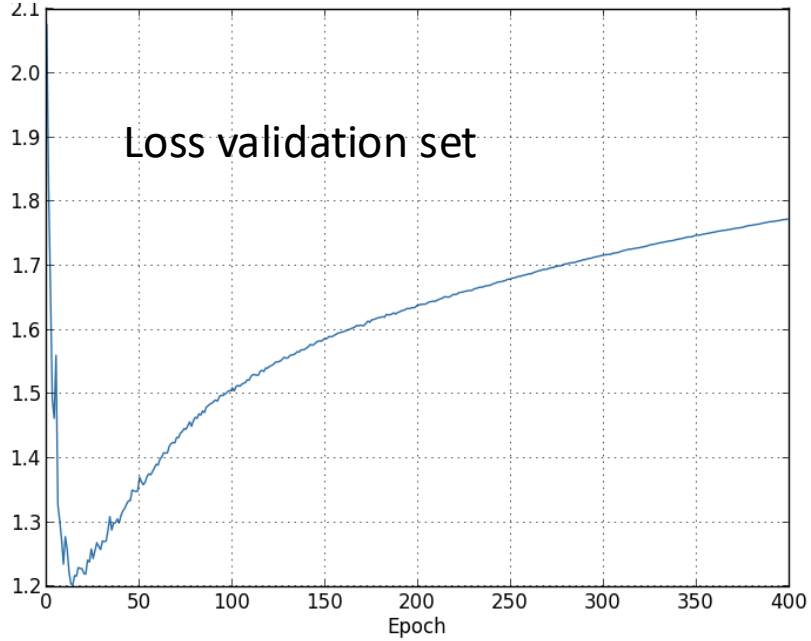




Detecting overfitting

- If you look at the loss function evaluated on the training set, it will always decrease over time (if you got all the hyperparameters right, like learning rate).
- Remember that we have also talked about validation sets and test sets?
- If the loss evaluated on the validation set starts to increase, your model is overfitting.

Explanation:
Overfitting happens when the model fits the training data well but fails to generalize to unseen data. The validation set is the unseen data in this case. When the loss evaluated on the validation set increases, it means that the model no longer generalizes to unseen data.



Early stopping

- Early stopping is a form of regularization used to avoid overfitting when training for instance a neural network model.
- Up to a certain point, gradient descent improves the model's performance on data outside of the training set (i.e., the validation set).
- Past that point, however, improving the model's fit to the training data comes at the expense of increased generalization error.
- Early stopping rules provide guidance as to how many iterations can be run before the model begins to overfit.
- Early stopping rules have been employed in many different machine learning methods, with varying amounts of theoretical foundation.

Stochastic Gradient Descent (SGD)

- In deep learning, training sets can be really large (millions of images).
- In practice, computing the loss and gradient for the entire training set can be very slow and sometimes intractable on a single machine if the dataset is too big to fit in main memory.
- Stochastic Gradient Descent (SGD) addresses this issue by following the negative gradient of the loss function after seeing **only a single or a few training examples**.
- The use of SGD in the neural network setting is motivated by the high cost of running back propagation over the full training set.
- SGD can overcome this cost and still lead to fast convergence.
- Generally, each parameter update in SGD is computed w.r.t. a few training examples or a **minibatch** as opposed to a single example. The reason for this is twofold: first this reduces the variance in the parameter update and can lead to more stable convergence, second this allows the computation to take advantage of highly optimized matrix operations that should be used in a well vectorized computation of the loss and gradient.

Backpropagation revisited

AUTOMATIC DIFFERENTIATION

Backpropagation revisited

- For traditional neural networks, we manually derived that if we define

$$\delta^{(L)} = \left(a^{(L)} - y\right) \odot \sigma'\left(z^{(L)}\right) = \left(h_{W,b}(x) - y\right) \odot \sigma'\left(z^{(L)}\right)$$
$$\delta^{(l)} = \left(W^{(l)}\right)^T \delta^{(l+1)} \odot \sigma'\left(z^{(l)}\right) \text{ for } l = L-1, L-2, \ldots, 2$$

- then

$$\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \text{ and } \frac{\partial J(W,b)}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

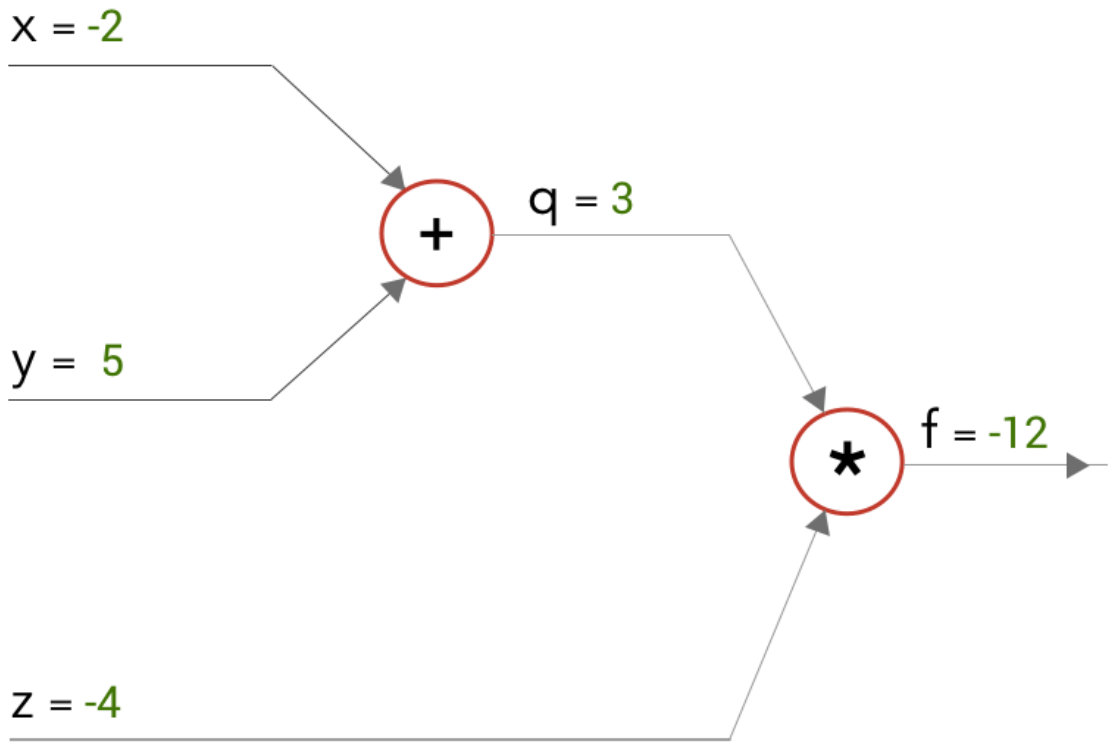
- **Problem:** If we change the network architecture (e.g., by adding convolutional layers) or the loss function, we have to derive everything again.
- **Solution:** Computational graphs and auto-differentiation

Computational graphs

- All modern deep learning frameworks use computational graphs.
- Basically, they enable automatic calculation of all partial derivatives.
- All you have to do is specify the underlying **computational graph** of your network.
- Really nice intro here: https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c
- Other resources
  - https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/
  - https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytorchs-magic-2686cd94ec95
  - https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
  - https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

Example

- Consider the equation $f(x, y, z) = (x + y)z$
- Our goal is to **calculate the gradients**: ∂f/∂x, ∂f/∂y and ∂f/∂z.
- To make it simpler, let us split it into two equations.
  - $q = x + y$
  - $f = q * z$
- Now, let us draw a computational graph for it with values x = -2, y = 5, z = 4.
- **Forward pass** results in f = - 12.
- Now, let's do the **backward pass** to calculate the gradients.



Example

- Working from right to left, at the multiply gate we can differentiate f to get the gradients at q and z — ∂f/∂q and ∂f/∂z.
- And at the add gate, we can differentiate q to get the gradients at x and y — ∂q/∂x and ∂q/∂y.
- How do we compute ∂f/∂x and ∂f/∂y?

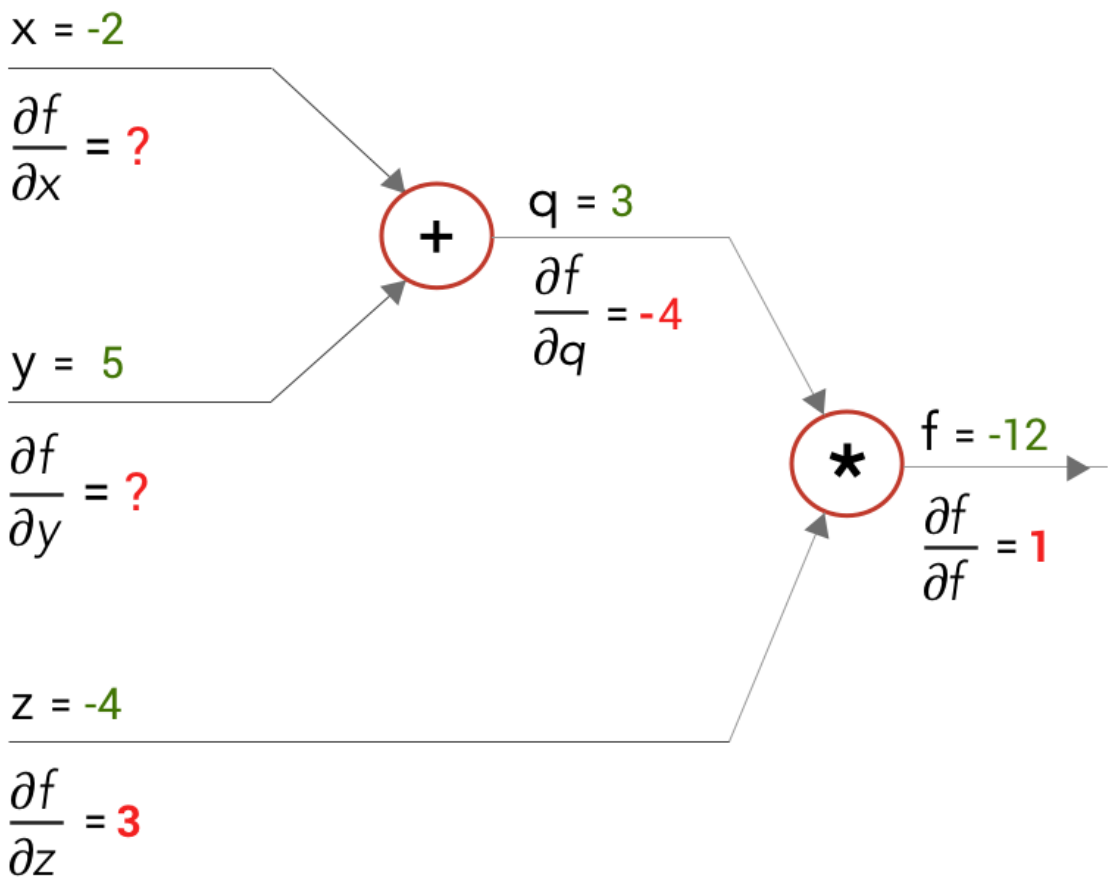
$f = q * z$

$\frac{\partial f}{\partial q} = z \mid z = -4$

$\frac{\partial f}{\partial z} = q \mid q = 3$

$q = x + y$

$\frac{\partial q}{\partial x} = 1$ $\frac{\partial q}{\partial y} = 1$



Example

- Use the chain rule to compute ∂f/∂x and ∂f/∂y:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial x} = -4 * 1 = -4$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial y} = -4 * 1 = -4$$

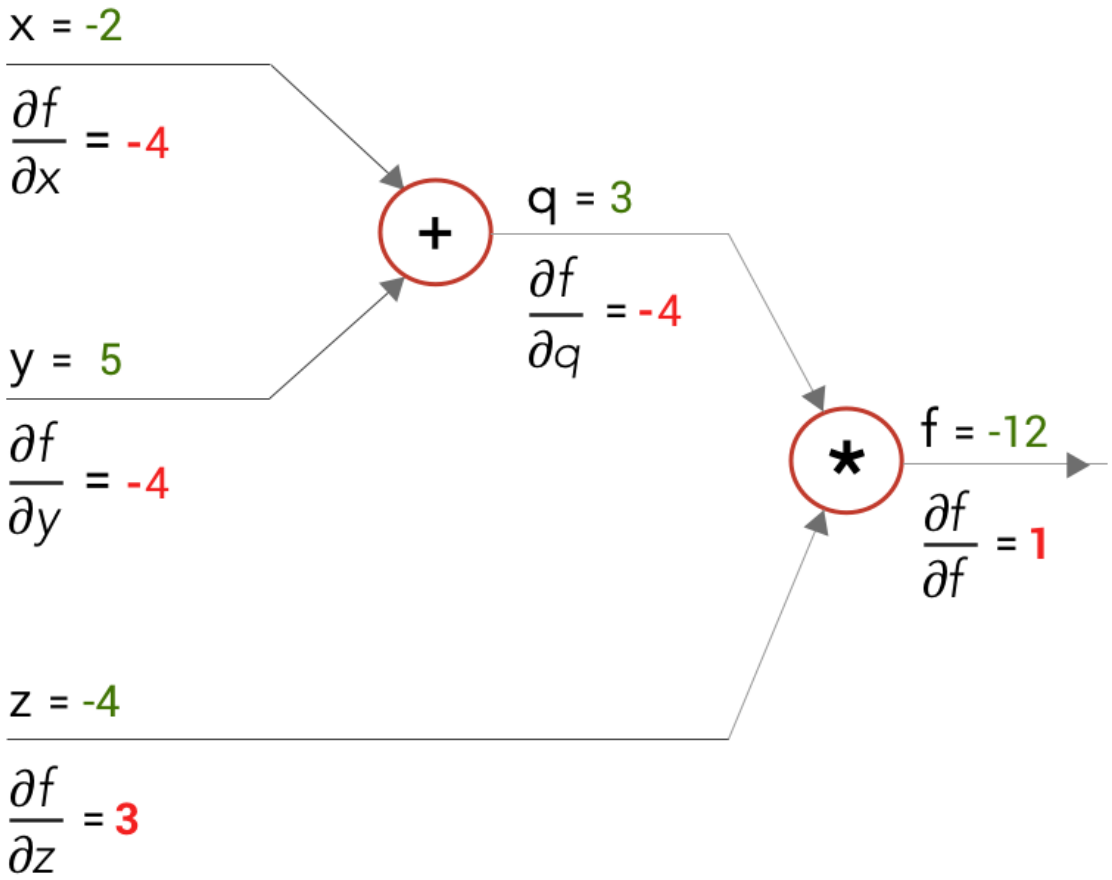
$f = q * z$

$\frac{\partial f}{\partial q} = z \mid z = -4$

$\frac{\partial f}{\partial z} = q \mid q = 3$

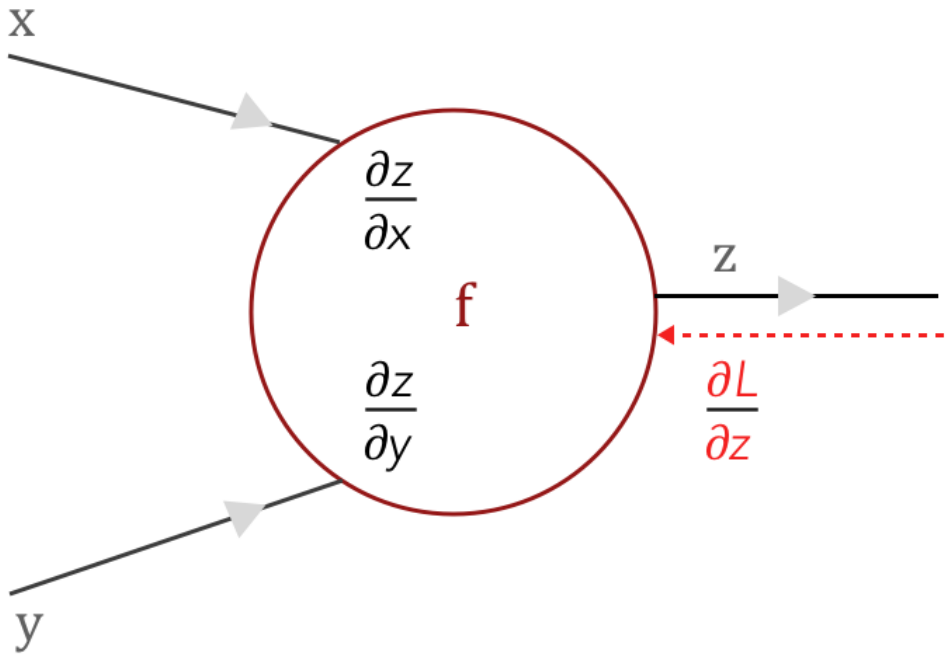
$q = x + y$

$\frac{\partial q}{\partial x} = 1 \quad \frac{\partial q}{\partial y} = 1$



Chain rule in a neuron

- Now that we have worked through a simple computational graph, we can imagine a neural network as a massive computational graph.
- Let us say we have a “neuron” f in that computational graph with inputs x and y which outputs z.
- We can easily compute the local gradients — differentiating z with respect to x and y as ∂z/∂x and ∂z/∂y.
- From the forward pass, we obtain the loss (L).
- When we start to work the loss backwards, we get the partial derivative of the loss from the previous layer (∂L/∂z).
- In order for the loss to be propagated to the other gates, we need to **find ∂L/∂x and ∂L/∂y**.

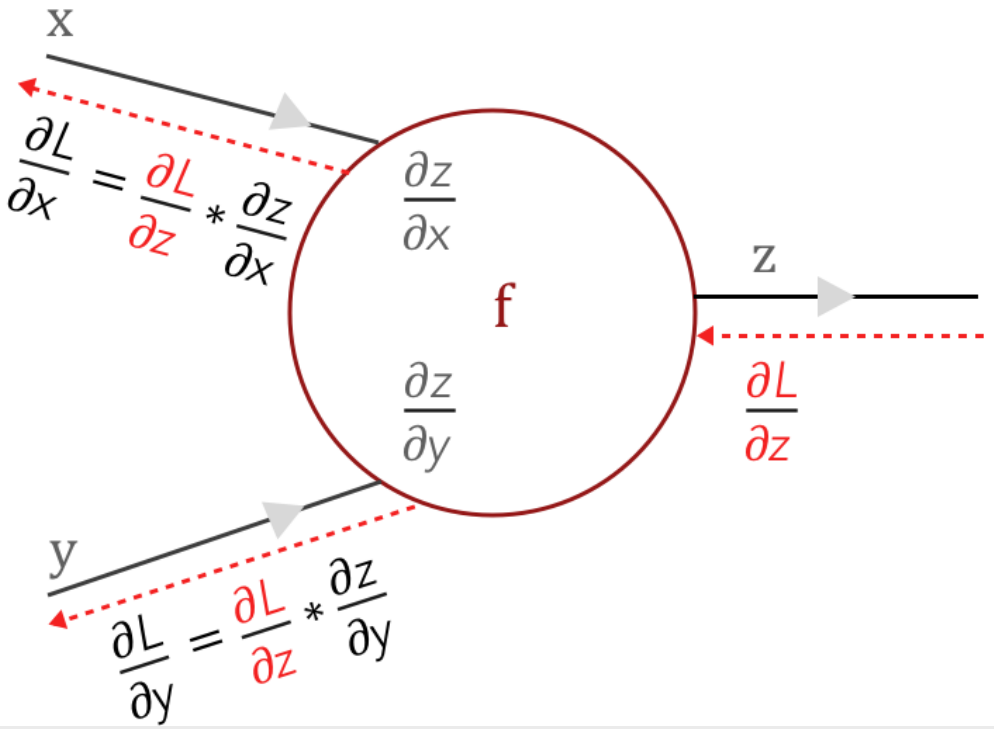


$\frac{\partial z}{\partial x}$ & $\frac{\partial z}{\partial y}$ *are local gradients*

$\frac{\partial L}{\partial z}$ *is the loss from the previous layer which has to be backpropagated to other layers*

Chain rule in a neuron

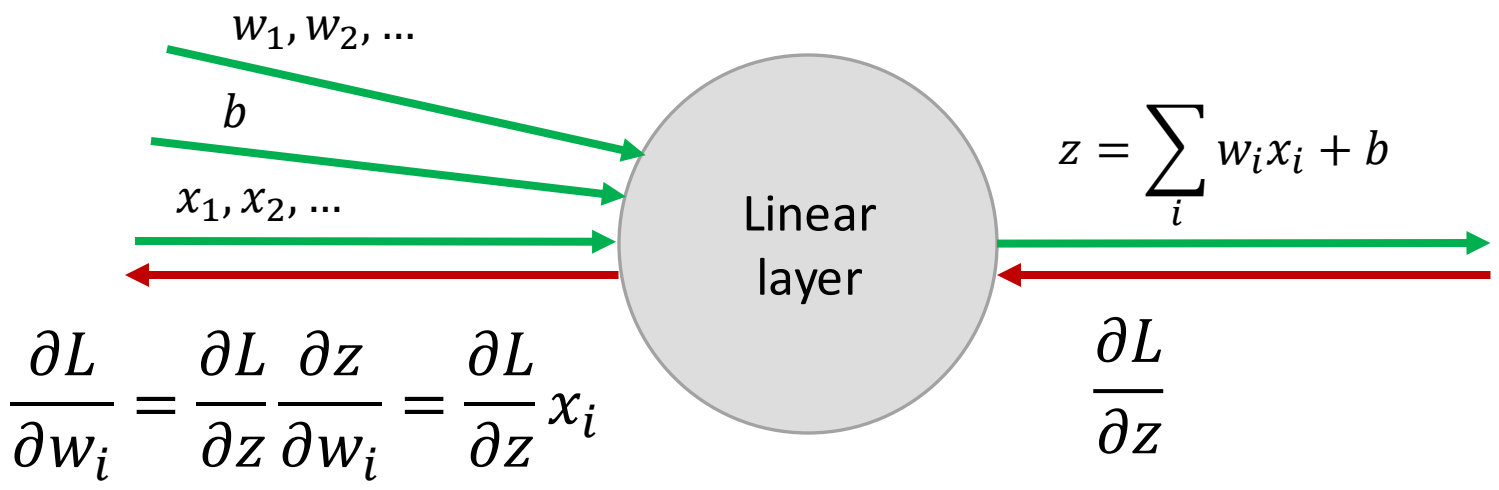
- The chain rule provides the solution.
- Using the chain rule, we can calculate ∂L/∂x and ∂L/∂y, which would feed the other gates in the extended computational graph.
- Defining new autograd functions in PyTorch: https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html



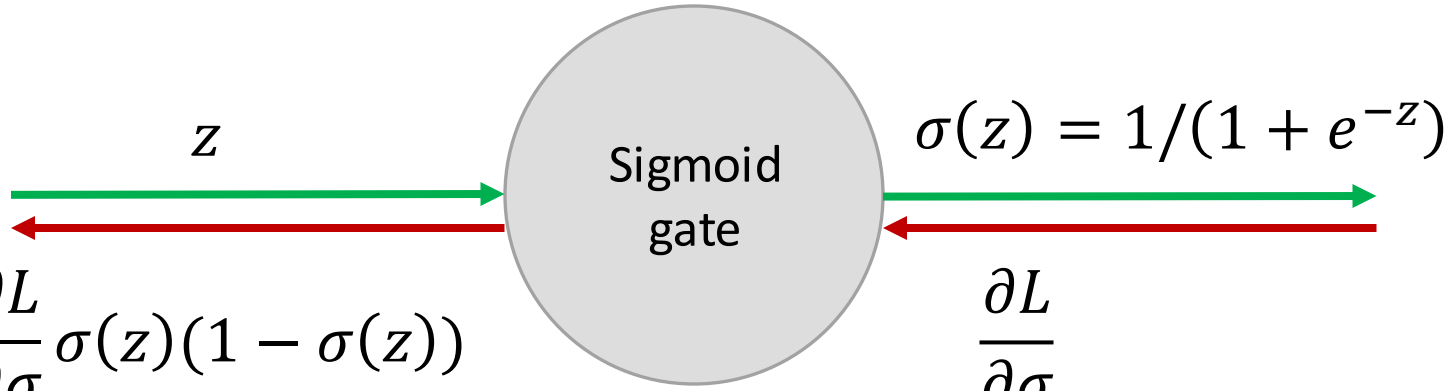
$\frac{\partial z}{\partial x}$ & $\frac{\partial z}{\partial y}$ *are local gradients*

$\frac{\partial L}{\partial z}$ *is the loss from the previous layer which has to be backpropagated to other layers*

Example: Linear layer

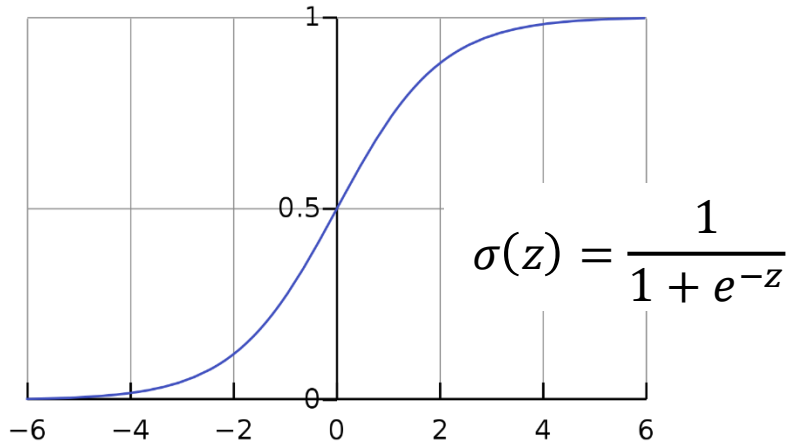


Example: Sigmoid gate

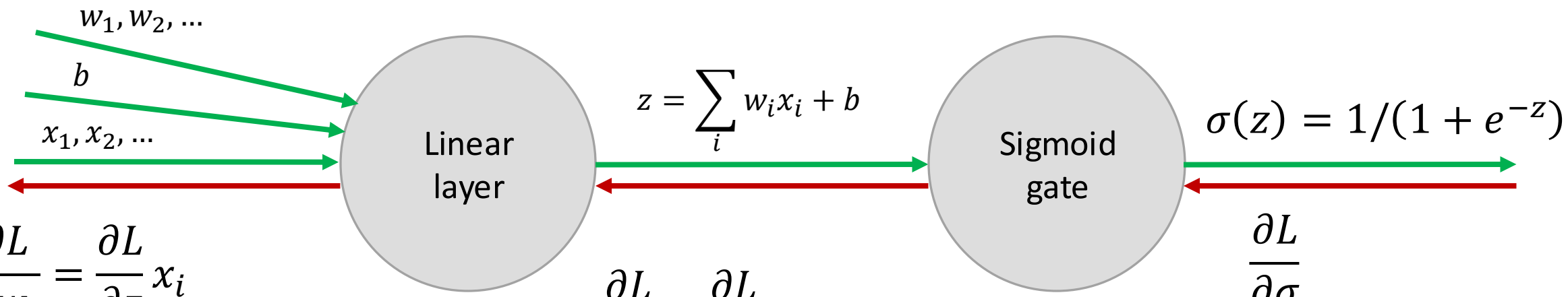


$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z} = \frac{\partial L}{\partial \sigma}\sigma(z)(1 - \sigma(z))$$

$$\frac{\partial L}{\partial \sigma}$$



Example: Logistic unit



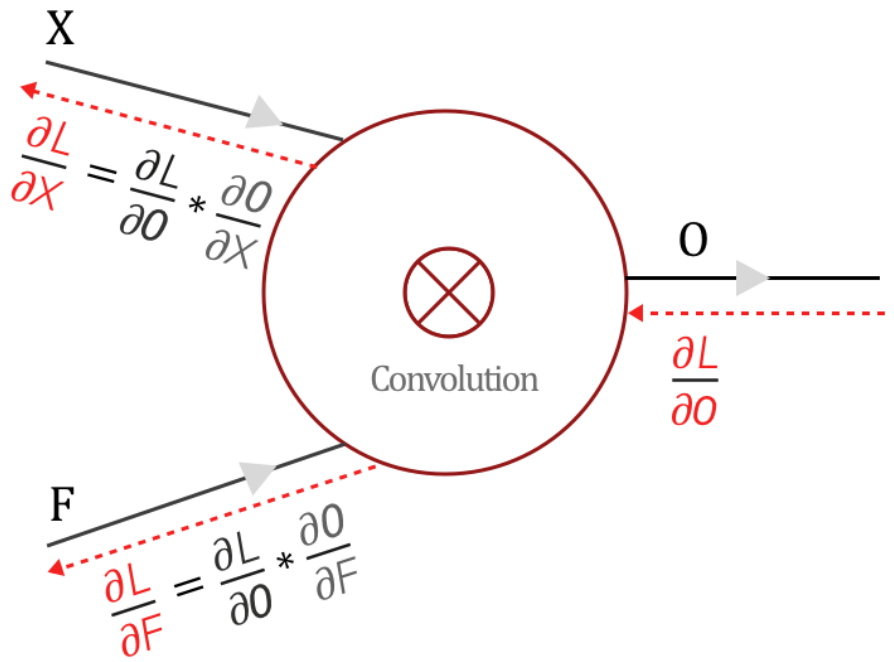
Example: Backpropagation for convolution

Backpropagation in a Convolutional Layer of a CNN

Finding the gradients:

$$\frac{\partial L}{\partial F} = \text{Convolution}\left(\text{Input } X\,,\ \text{Loss gradient } \frac{\partial L}{\partial O}\right)$$

$$\frac{\partial L}{\partial X} = \text{Full Convolution}\left(180^{\circ}\text{rotated Filter } F\,,\ \text{Loss Gradient } \frac{\partial L}{\partial O}\right)$$



$\frac{\partial O}{\partial X}$ & $\frac{\partial O}{\partial F}$ *are local gradients*

$\frac{\partial L}{\partial O}$ *is the loss from the previous layer which has to be backpropagated to other layers*

Further reading + online videos/demos

- Derivation of the backpropagation algorithm
  - http://neuralnetworksanddeeplearning.com/chap2.html
- Cross-entropy, regularization and other practical considerations:
  - http://neuralnetworksanddeeplearning.com/chap3.html
- Machine Learning, Andrew Ng (Stanford), full course:
  - https://www.youtube.com/playlist?list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN
- Deep learning frameworks like PyTorch and Keras perform automatic differentiation/backprop. We will talk about it later in the course. Here are a few useful links if you are interested:
  - https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/
  - https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytorchs-magic-2686cd94ec95
  - https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
  - https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

Further reading + online videos/demos

- Computational graphs and backpropagation in CNNs:
  - https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199
  - https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c
  - http://cs231n.github.io/optimization-2/
  - http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture04.pdf