Research and Innovation Center of Excellence

MSc on Intelligent Critical Infrastructure Systems

Machine Learning
Lecture 5

Christos Kyrkou
Research Lecturer
KIOS Research and Innovation Center of Excellence
University of Cyprus

Imperial College London

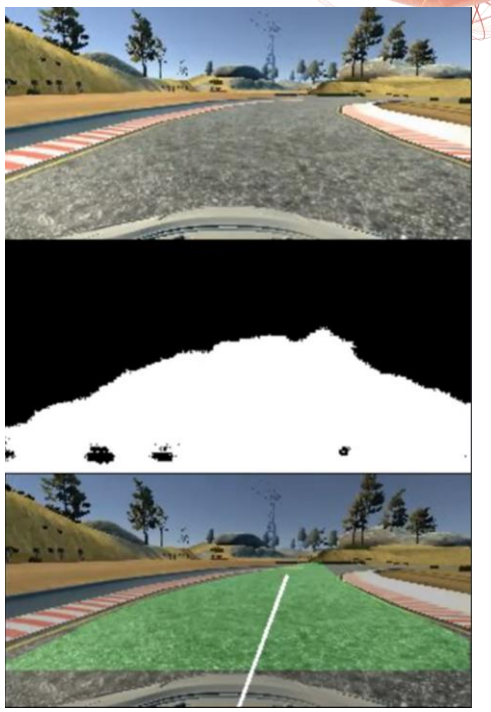
University of Cyprus

funded by:

Artificial Neural Networks (ANNs)

- ANNs have been used on a variety of tasks,
  - computer vision,
  - Speech recognition,
  - machine translation,
  - social network filtering,
  - playing board and video games,
  - medical diagnosis,
  - and even in activities considered as reserved to humans, like painting

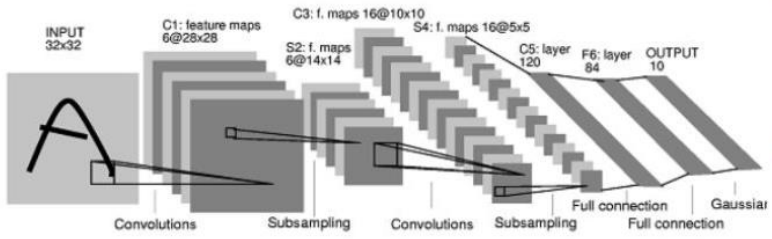
Generative Models



Regression Models

- MNIST digit recognition dataset
- 10^7 pixels used in training

1998 LeCun et al.

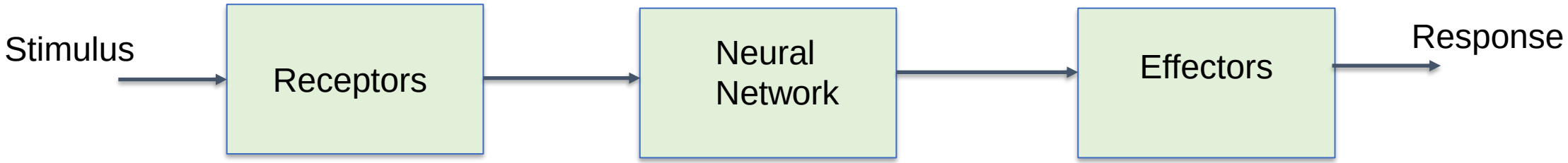


Discriminative Classifiers/Detectors

Goodfellowet al. “Deep Learning” (2016), –Chapter 6: Deep Feedforward Networks

http://neuralnetworksanddeeplearning.com/

Neural Networks – Motivation



- The brain is a complex, nonlinear and distributed computer having neurons as its basic information processing units (different than traditional computers).
- The brain has the ability to perform several tasks such as pattern recognition, perception and motor control very well, despite being slow in information processing.
- Therefore, the motivation is to mimic the functioning neurons and neural networks *in-silico* so as to build machines that have very high capabilities.

Neural Networks have been around for many years

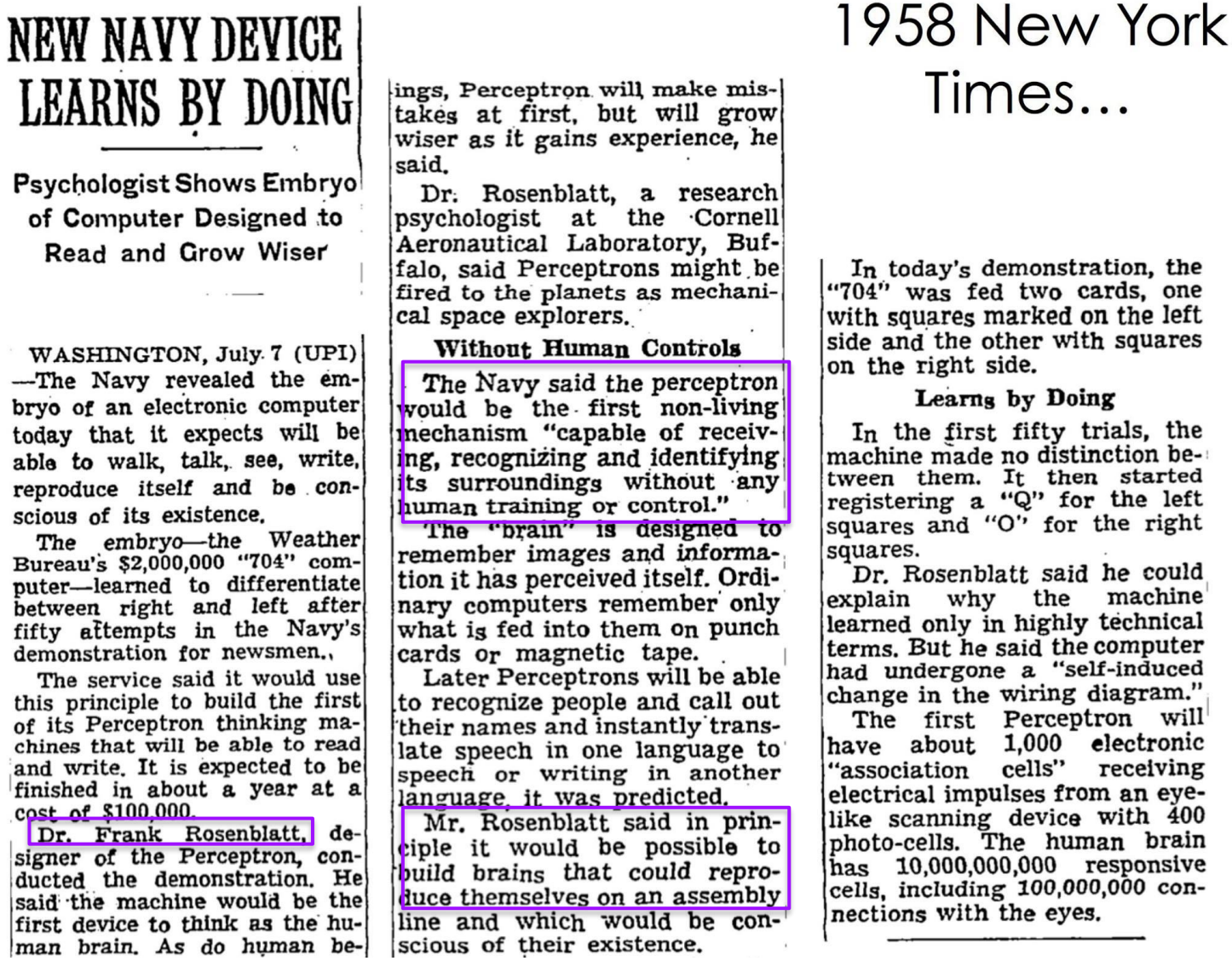
1950		Turing's Learning Machine	Alan Turing proposes a 'learning machine' that could learn and become artificially intelligent. Turing's specific proposal foreshadows genetic algorithms.
1951		First Neural Network Machine	Marvin Minsky and Dean Edmonds build the first neural network machine, able to learn, the SNARC.
1969		Limitations of Neural Networks	Marvin Minsky and Seymour Papert publish their book Perceptrons, describing some of the limitations of perceptrons and neural networks. The interpretation that the book shows that neural networks are fundamentally limited
1982	Discovery	Recurrent Neural Network	John Hopfield popula as content-addressa
1995	Discovery	Random Forest Algorithm	Tin Kam Ho publishes a paper describing random decision forests.
1995	Discovery	Support Vector Machines	Corinna Cortes and
1997	Discovery	LSTM	Sepp Hochreiter and networks, greatly imp
		Torch Machine Learning Library	Torch, a software library for machine learning, is first released.
2009	Achievement	ImageNet	ImageNet is created University, who reali reflect the real world
2012	Achievement	Recognizing Cats on YouTube	The Google Brain team, led by Andrew Ng and Jeff Dean, create a neural network that learns to recognize cats by watching unlabeled images taken from frames of YouTube videos.
2014		Leap in Face Recognition	Facebook researchers publish their work on DeepFace, a system that uses neural networks that identifies faces with 97.35% accuracy. The results are an improvement of more than 27% over previous systems and rivals human performance.
2016	Achievement	Beating Humans in Go	Google's AlphaGo program becomes the first Computer Go program to beat an unhandicapped professional human player using a combination of machine learning and tree search techniques.

'AI Winter' caused by pessimism about machine learning effectiveness.

(1980s) Popularization of backpropagation causes a resurgence in machine learning research (Hinton et al).

Deep learning becomes feasible, which leads to machine learning becoming integral to many widely used software services and applications.

Headlines from the past…



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

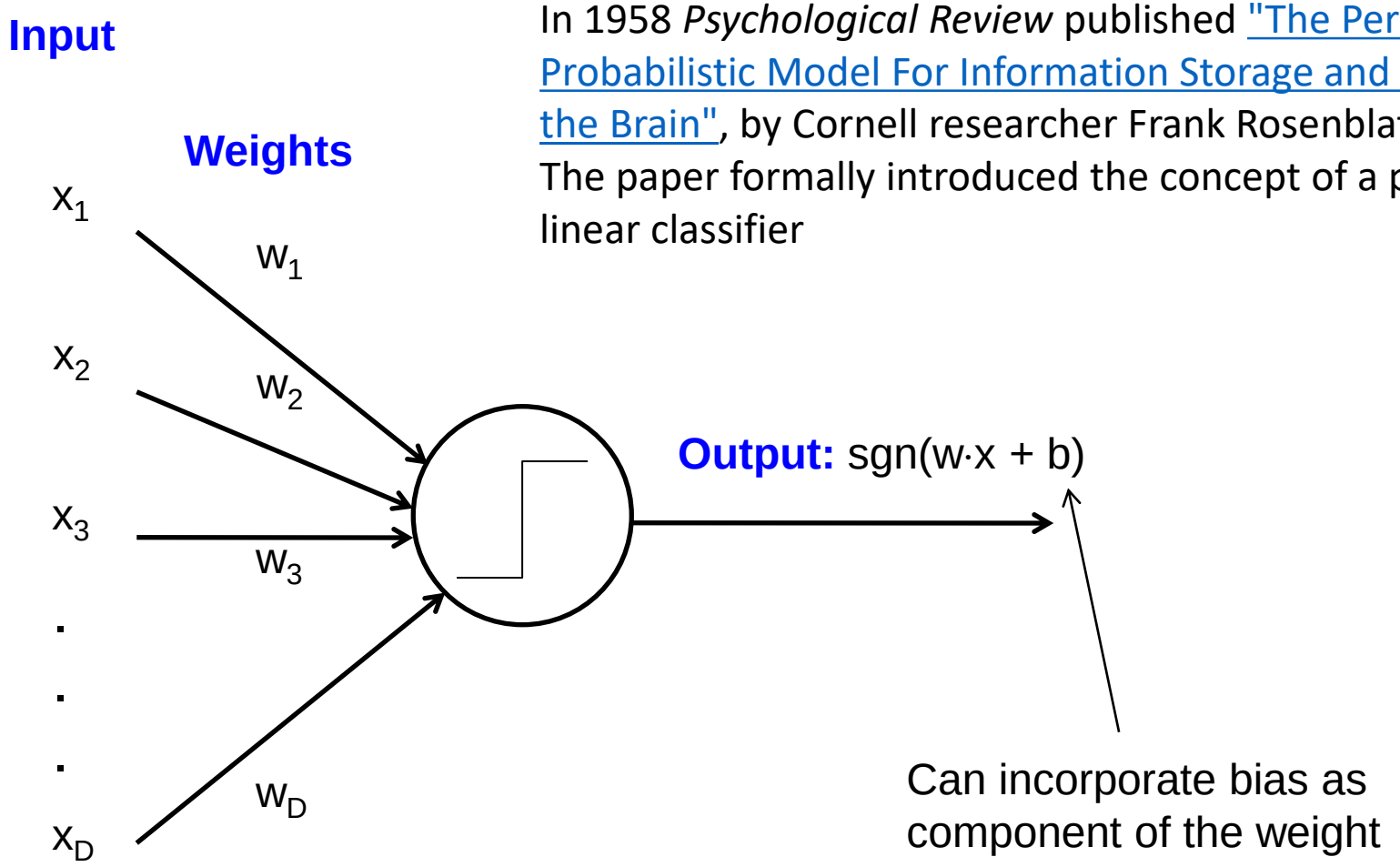
The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

1958 New York Times…

"Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted."

NY Times, "New Navy Device Learns By Doing" 7/7/1958

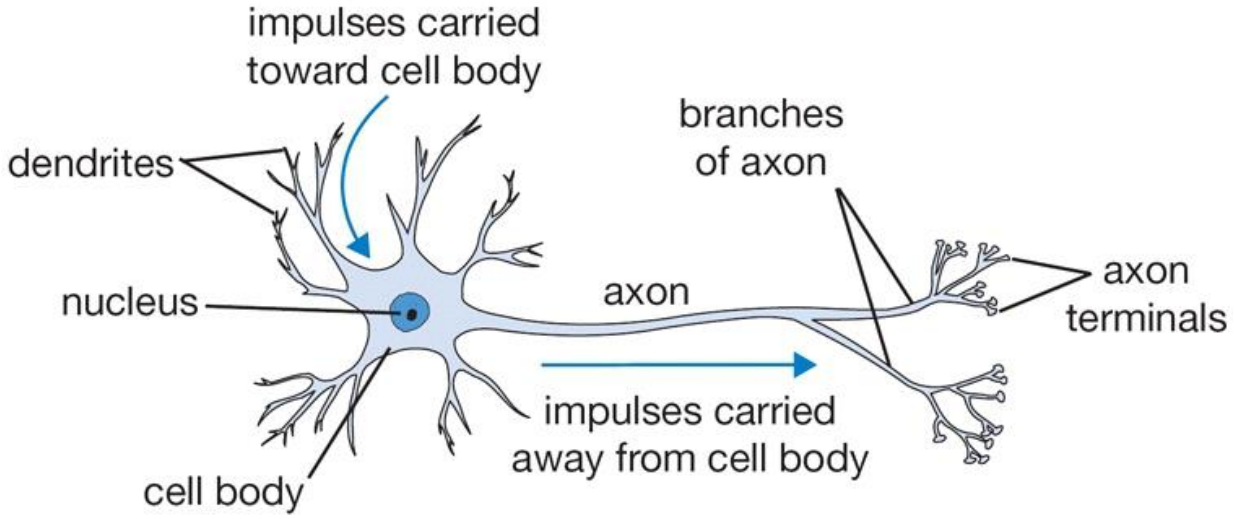
Linear classifiers revisited: Perceptron

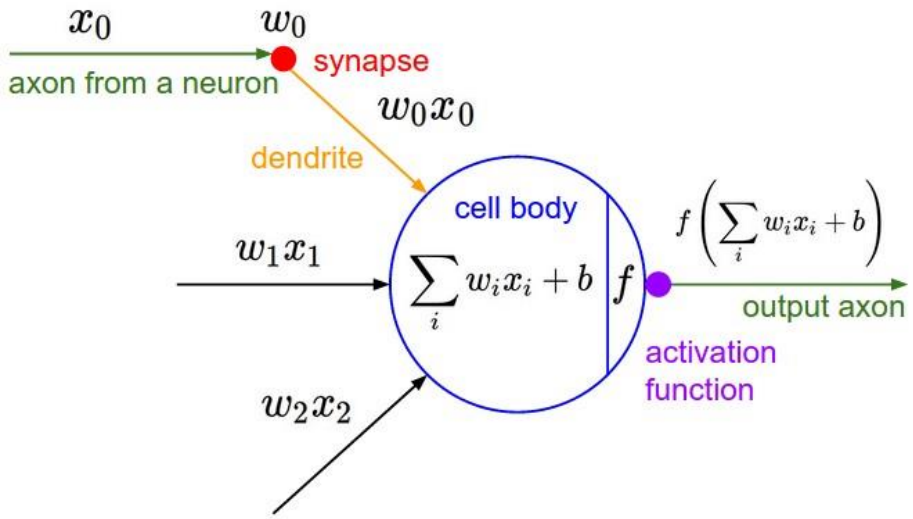


In 1958 *Psychological Review* published "The Perceptron: A Probabilistic Model For Information Storage and Organization in the Brain", by Cornell researcher Frank Rosenblatt.
The paper formally introduced the concept of a perceptron — a linear classifier

Neurons

- Neuron
  - Computational building block for the brain
- (Artificial) Neuron
  - Computational building block for the (artificial) “neural network”
- Human brains have ~10000 computational power than computer





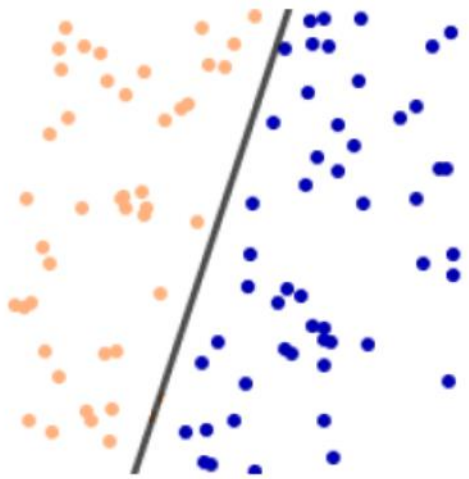
Perceptron Algorithm

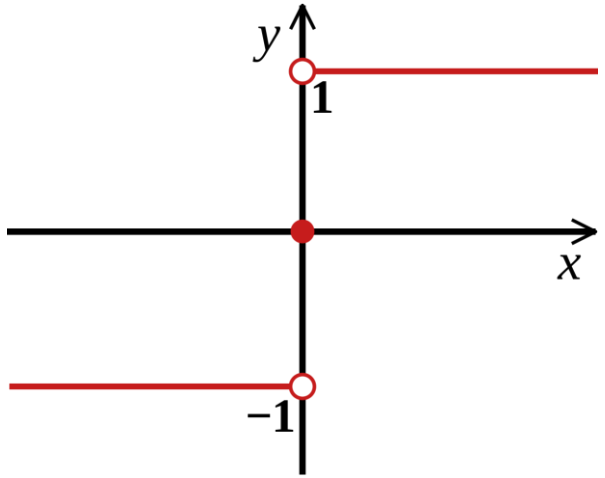
- In the modern sense, the perceptron is an algorithm for learning a **binary classifier** i.e., a function that maps its input x (a real-valued vector) to an **output** value $f(x)$.
- The value of $f(x)$ is used to classify x as either a positive or a negative instance.

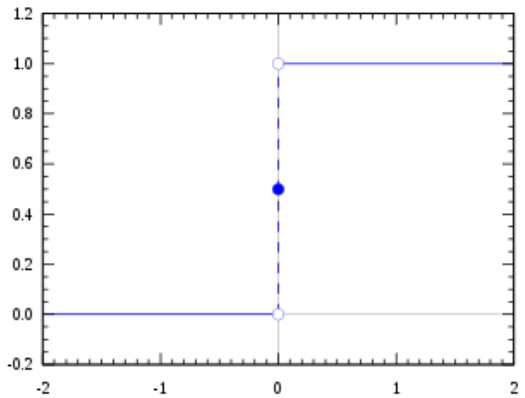
$\boldsymbol{w} \cdot \boldsymbol{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$

$$f(x) = \boldsymbol{sign}(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

$$f(x) \begin{cases} 1 \; if \; \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \\ 0 \; otherwise \end{cases}$$







Heaviside step function

Perceptron Training Algorithm

- Provide training set of (input, output) pairs and run:
  - Initialize perceptron with random weights
  - For the inputs of an example in the training set, compute the Perceptron's output
  - If the output of the Perceptron does not match the output that is known to be correct for the example:
    - If the output should have been -1 but was 1, decrease the weights
    - If the output should have been 1 but was -1 increase the weights
    - Go to the next example in the training set and repeat…

Algorithm 5 PERCEPTRONTRAIN($\mathbf{D}$, *MaxIter*)

```
w_d ← 0, for all d = 1 ... D                          // initialize weights
b ← 0                                                 // initialize bias
for iter = 1 ... MaxIter do
    for all (x,y) ∈ D do
        a ← Σ_{d=1}^{D} w_d x_d + b                   // compute activation for this example
        if ya ≤ 0 then
            w_d ← w_d + y x_d, for all d = 1 ... D    // update weights
            b ← b + y                                 // update bias
        end if
    end for
end for
return w_0, w_1, ..., w_D, b
```

$y \in \{-1,1\}$

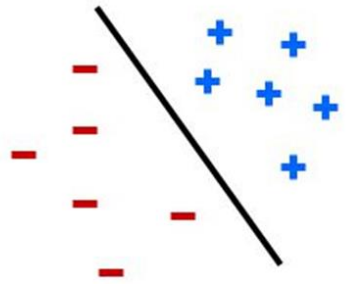
Algorithm 6 PERCEPTRONTEST($w_0, w_1, \ldots, w_D, b, \hat{\mathbf{x}}$)

```
a ← Σ_{d=1}^{D} w_d x̂_d + b                           // compute activation for the test example
return SIGN(a)
```

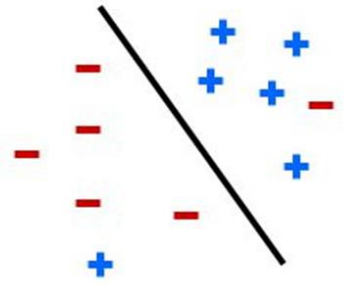
Convergence of perceptron update rule

- Linearly separable data
  - Guaranteed to converge to *some* solution
  - it may still pick any solution
- Non-separable data
  - converges to a minimum-error solution assuming examples are presented in random sequence in a **finite number of steps**
    - Provided the and learning rate r decays appropriately $(1/t)$ where t is the number of iterations
- In 1969 a famous book entitled **Perceptron's** by Marvin Minsky and Seymour Papert showed that it was impossible for a single perceptron to learn an XOR function.

Separable



Non-Separable

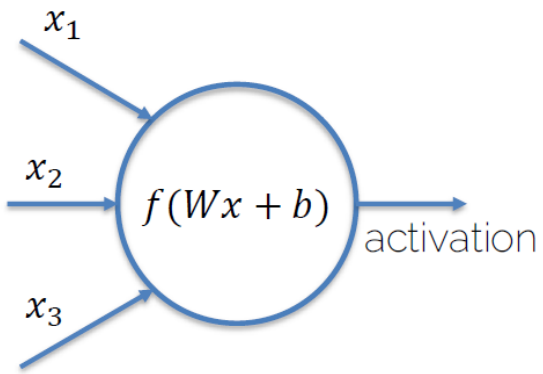


1	1	0
0	0	1
XOR	0	1

From perceptron to Neuron

- To allow neural networks to compute nontrivial problems pass the linear combination through a non-linear function (e.g., sigmoid)
  - There are some desirable properties.

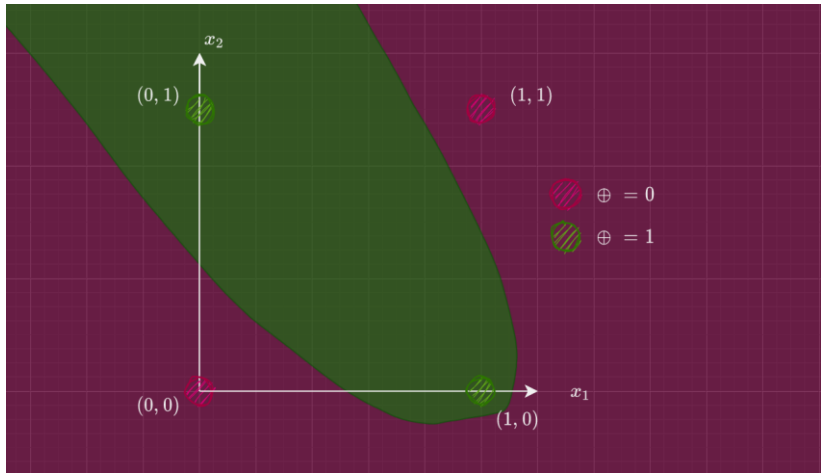
- A Generic Neuron now becomes:



Linear function: $Wx + b$
Non-linearity: activation: $f(x)$

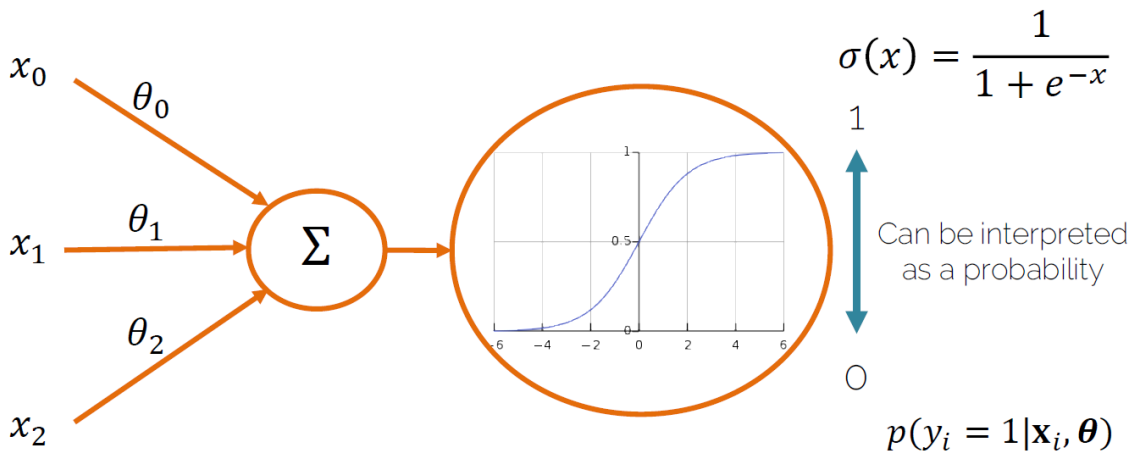
Every neuron computes: $f(Wx + b)$

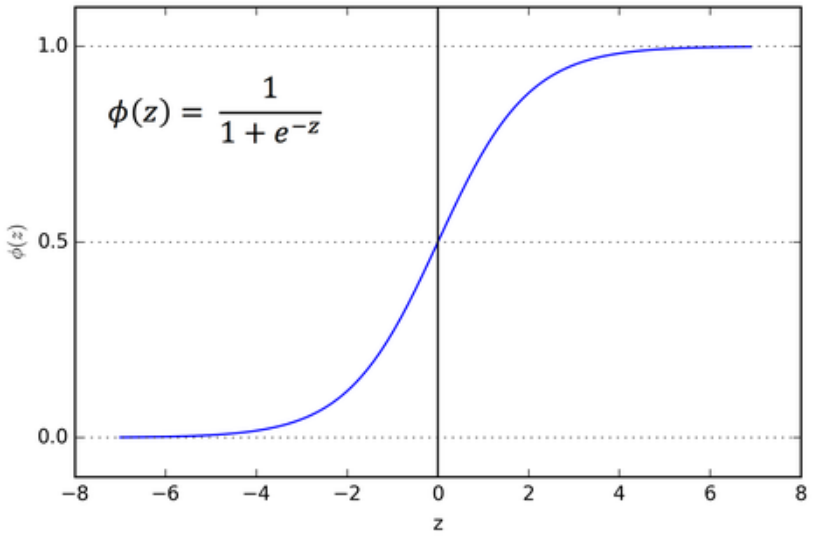
Non-linearity allows for more complex decision boundaries:



Sigmoid Neuron

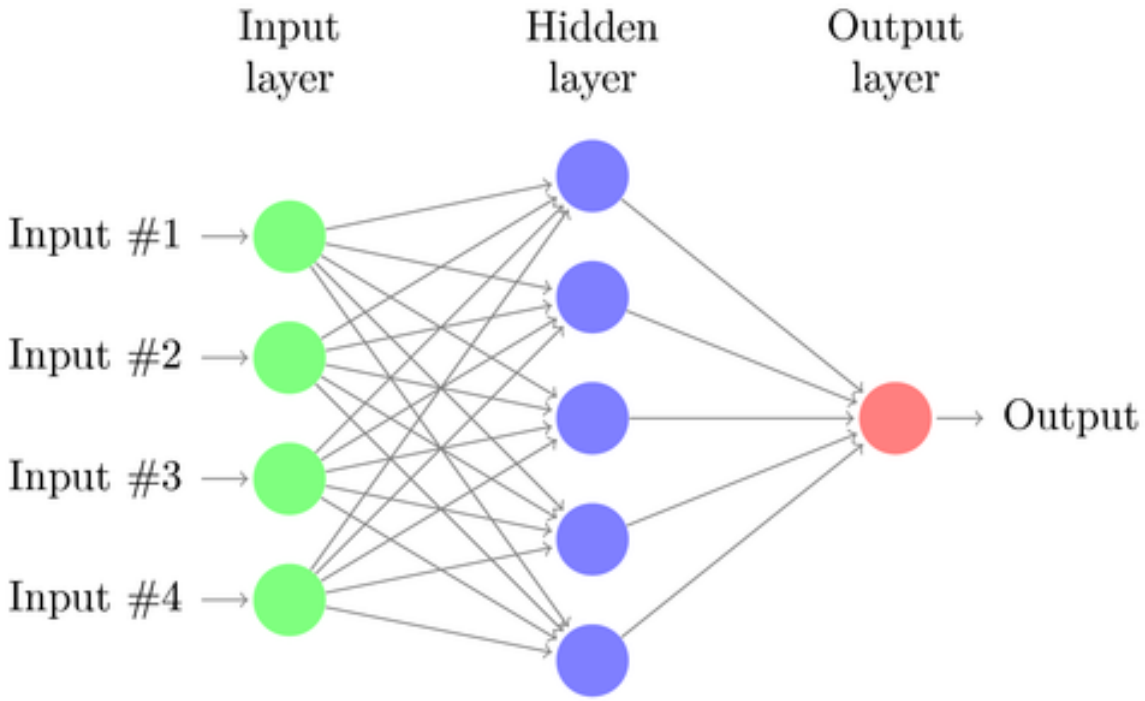
- Sigmoid function is a mathematical function with a characteristic “S”-shaped curve, also called the **sigmoid** curve.
  - Probabilistic output





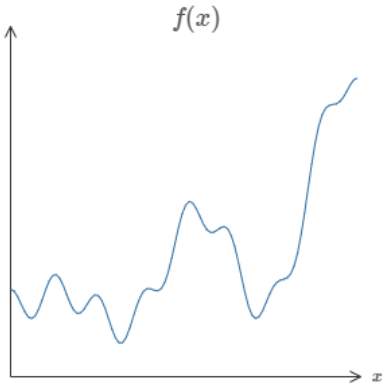
Network with a single hidden layer

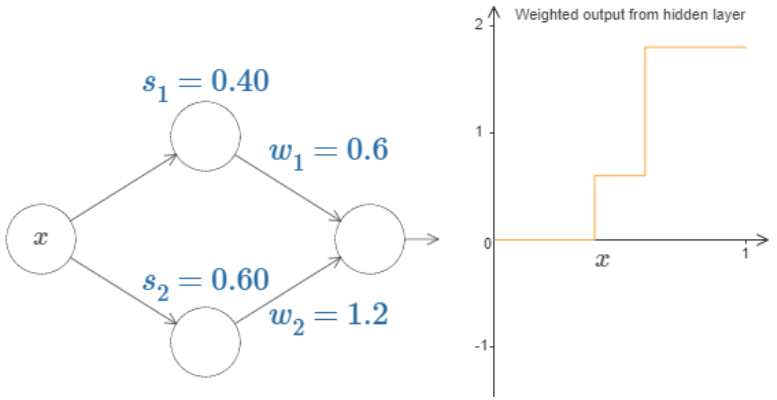
- Neural networks with at least one hidden layer are **universal** function **approximator theorem** (UAT)
- Universality: For any arbitrary function $f(x)$, there exists a neural network that can closely approximate it for any input x.

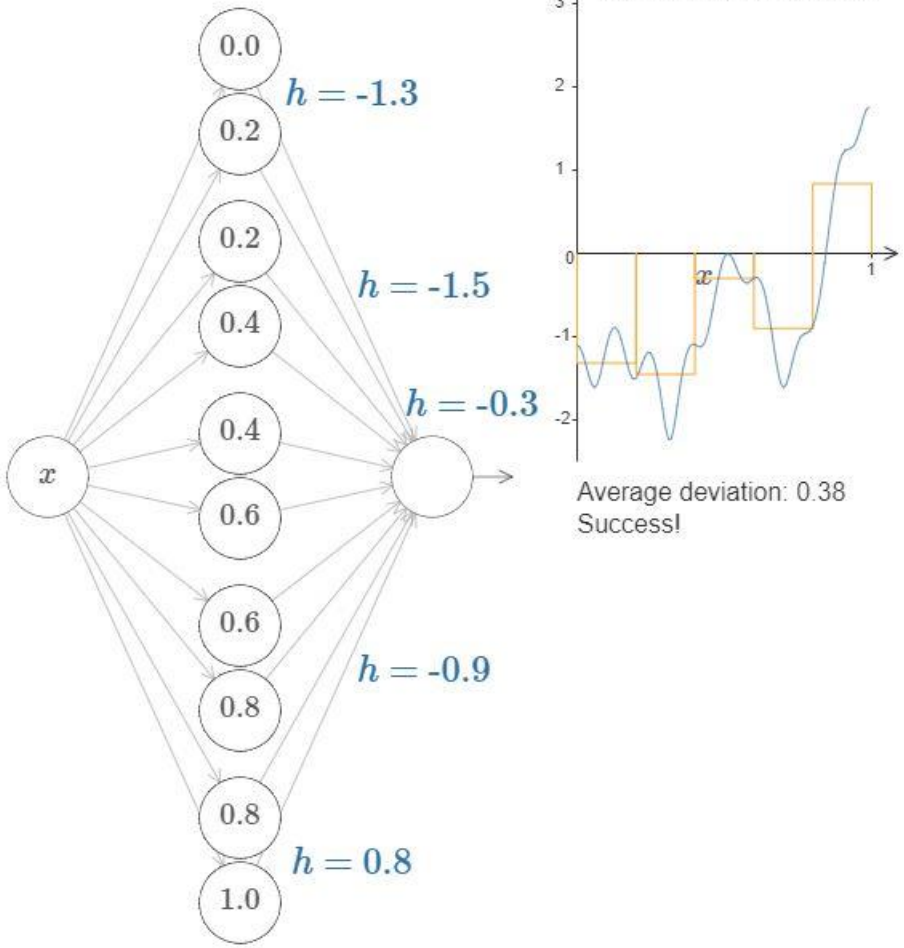


Universal approximation theorem (UAT)

- By increasing the number of hidden neurons we can improve the approximation.







http://neuralnetworksanddeeplearning.com/chap4.html

Universal approximation theorem (UAT)

- Neural Networks with at least one hidden layer are universal approximators.
  - Doesn't mean that a network can be used to exactly compute any function.
  - Rather, we can get an approximation that is as good as we want.

- That is, it can be shown (e.g. see Approximation by Superpositions of Sigmoidal Function from 1989) that given any continuous function $f(x)$ and some $\epsilon > 0$, there exists a Neural Network $g(x)$ with one hidden layer (with a non-linearity) such that $\forall x, | f(x) - g(x) | < \epsilon$.

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

Universal approximation theorem (UAT)

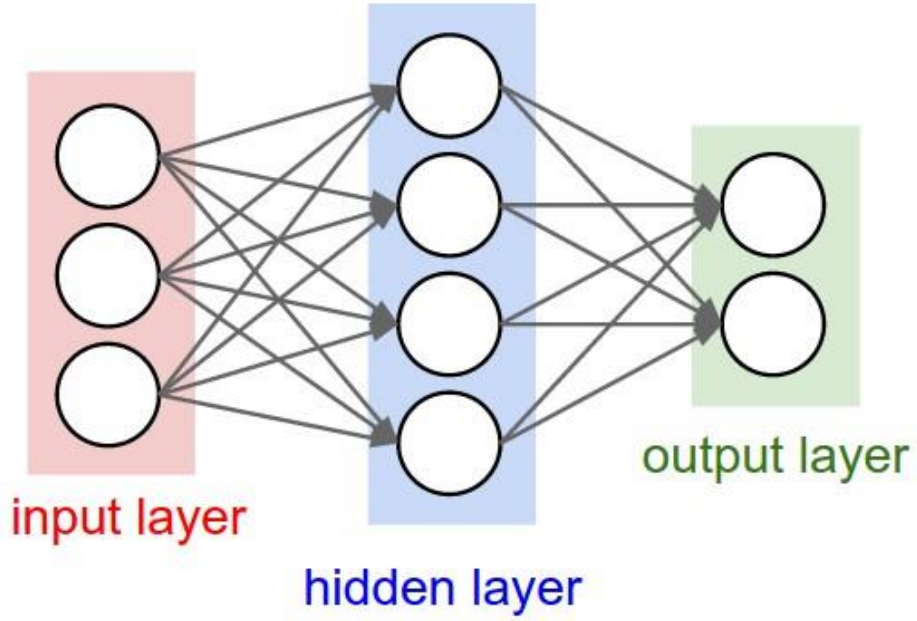
- If a 1-hidden-layer NN is a universal approximator, then why do we need deep nets?

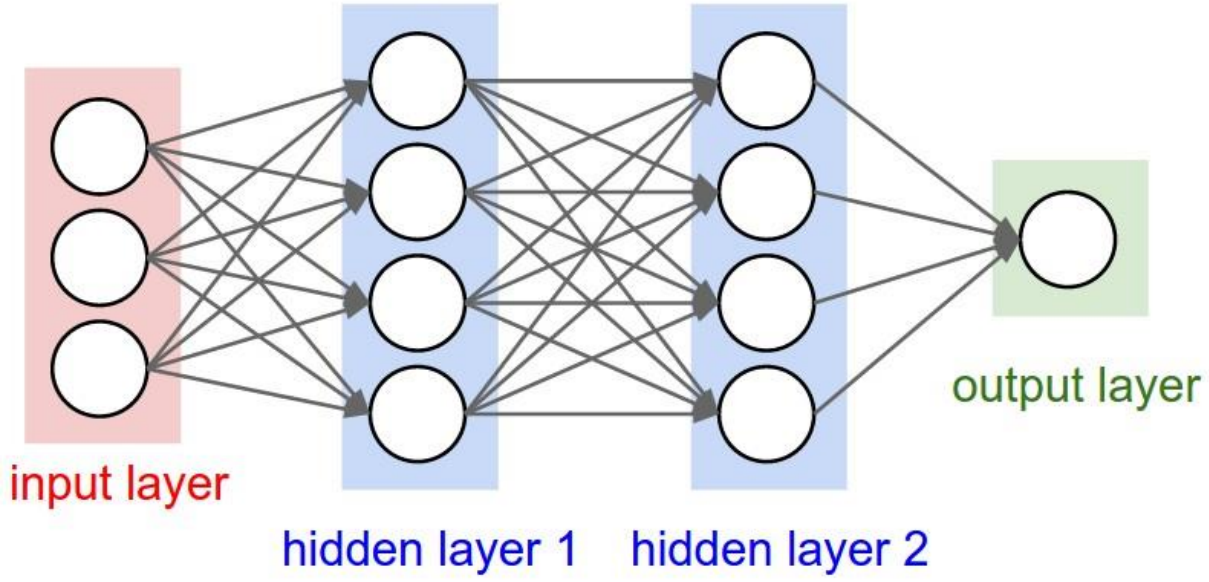
- The fact that a 1-layer Neural Network is a universal approximate does not guarantee anything ☹
  - Difficult to learn with such networks, says nothing about how difficult is to fit such approximators.
  - In some cases, infinite number of neurons maybe necessary

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

Multi-layer Neural Networks

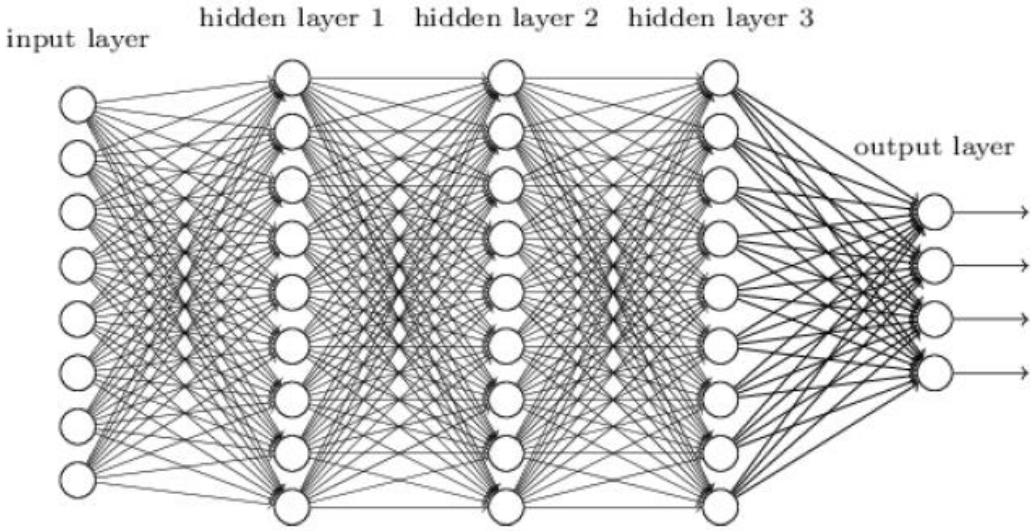
- Generally, the networks are defined to be feed-forward: a unit feeds its output to all the units on the next layer, but there is no feedback to the previous layer.
- Neural Networks are modeled as collections of neurons that are connected in an acyclic graph.





Deep Neural Networks

- In practice deeper networks usually represent mode complex functions with less total neurons (and therefore less parameters)
  - Compactly express nice, smooth functions that fit well with the statistical properties of data we encounter in practice
  - Easy to learn using the optimization algorithms we have available



Why organize a neural network into layers?

Non-Linear Neural Networks

- $f = W_3 \cdot (W_2 \cdot (W_1 \cdot x))$

- $f = W_4 \cdot x$
  - Where $W_4 = W_3 \cdot W_2 \cdot \mathrm{W}_1$
  - Still Linear!
  - Equivalent to 1-hidden layer

Why activation functions?

Why not just concatenate?

Would be much cheaper to compute....

What about our activation functions?

- We need to apply an **Activation function** $\varphi(x)$ so as to make the network more powerful and add ability to it to learn something complex

- Non-linear functions are those which have degree more than one and they have a curvature when we plot a Non-Linear function.

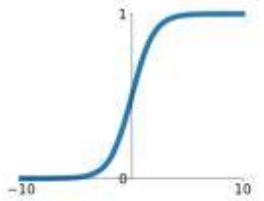
- Many options, want them to be easy to take derivative (More Later)

Non-linear activation functions

- Choosing the right activation function is another new hyper-parameter!
  - ***Derivative or Differential:*** *Change in y-axis w.r.t. change in x-axis.It is also known as slope.*
  - ***Monotonic function:*** *A function which is either entirely non-increasing or non-decreasing.*

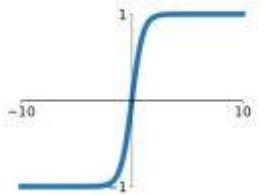
Sigmoid

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



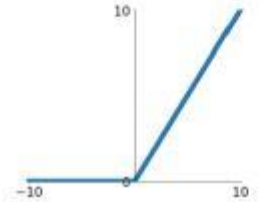
tanh

$$\tanh(x)$$



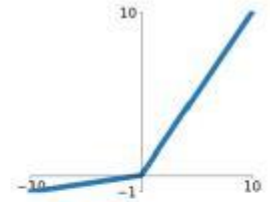
ReLU

$$\max(0, x)$$



Leaky ReLU

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

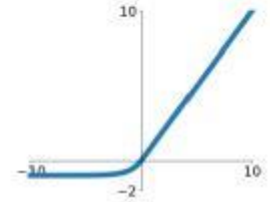


Maxout

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

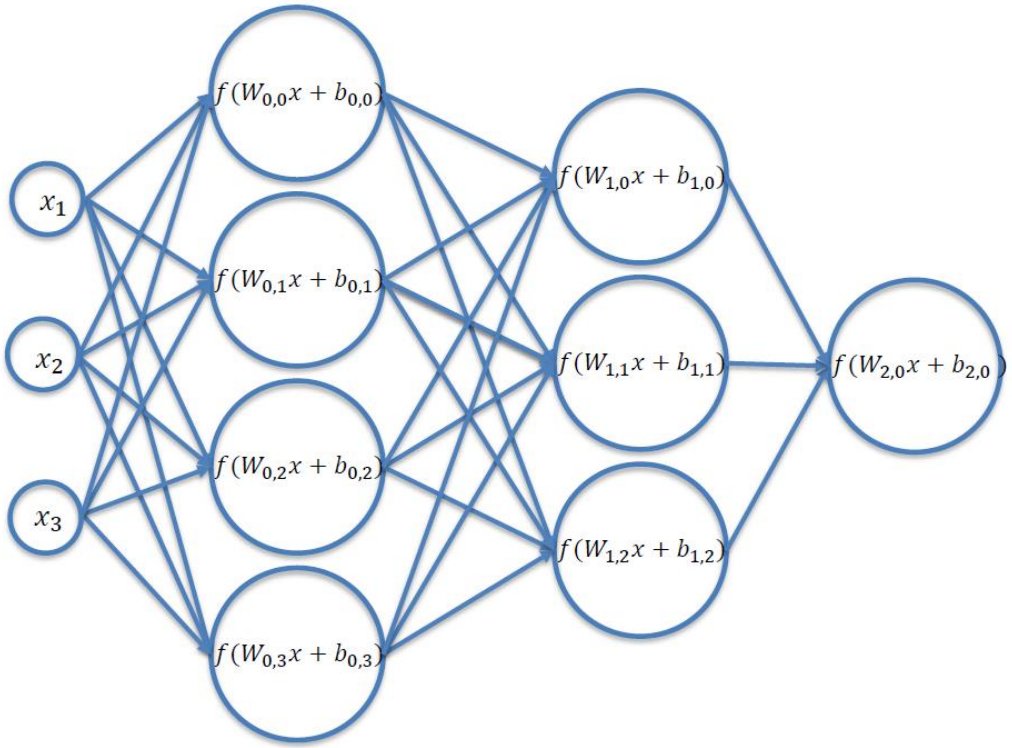
ELU

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



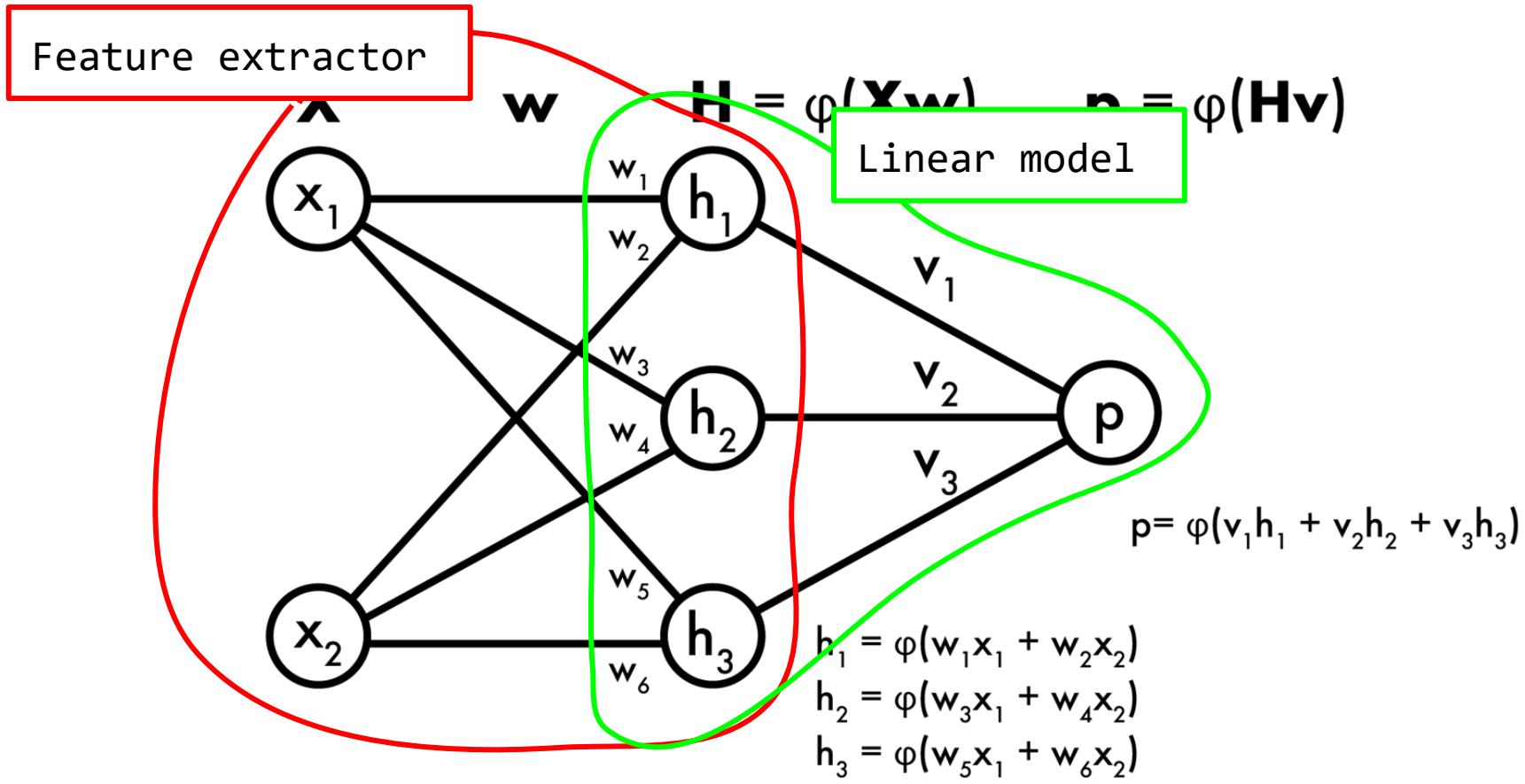
Neural Network

- Linear score function: $f = Wx$
- Neurons compute the same functions
- Neural network is a nesting of ‘functions’
  - 2-layers: $f = W_2 \max(0, W_1 x)$
  - 3-layers: $f = W_3 \max(0, W_2 \max(0, W_1 x))$
  - 4-layers: $f = W_4 \tanh(W_3 \max(0, W_2 \max(0, W_1 x)))$
  - 5-layers: $f = W_5 \sigma(W_4 \tanh(W_3 \max(0, W_2 \max(0, W_1 x))))$
  - … up to hundreds of layers

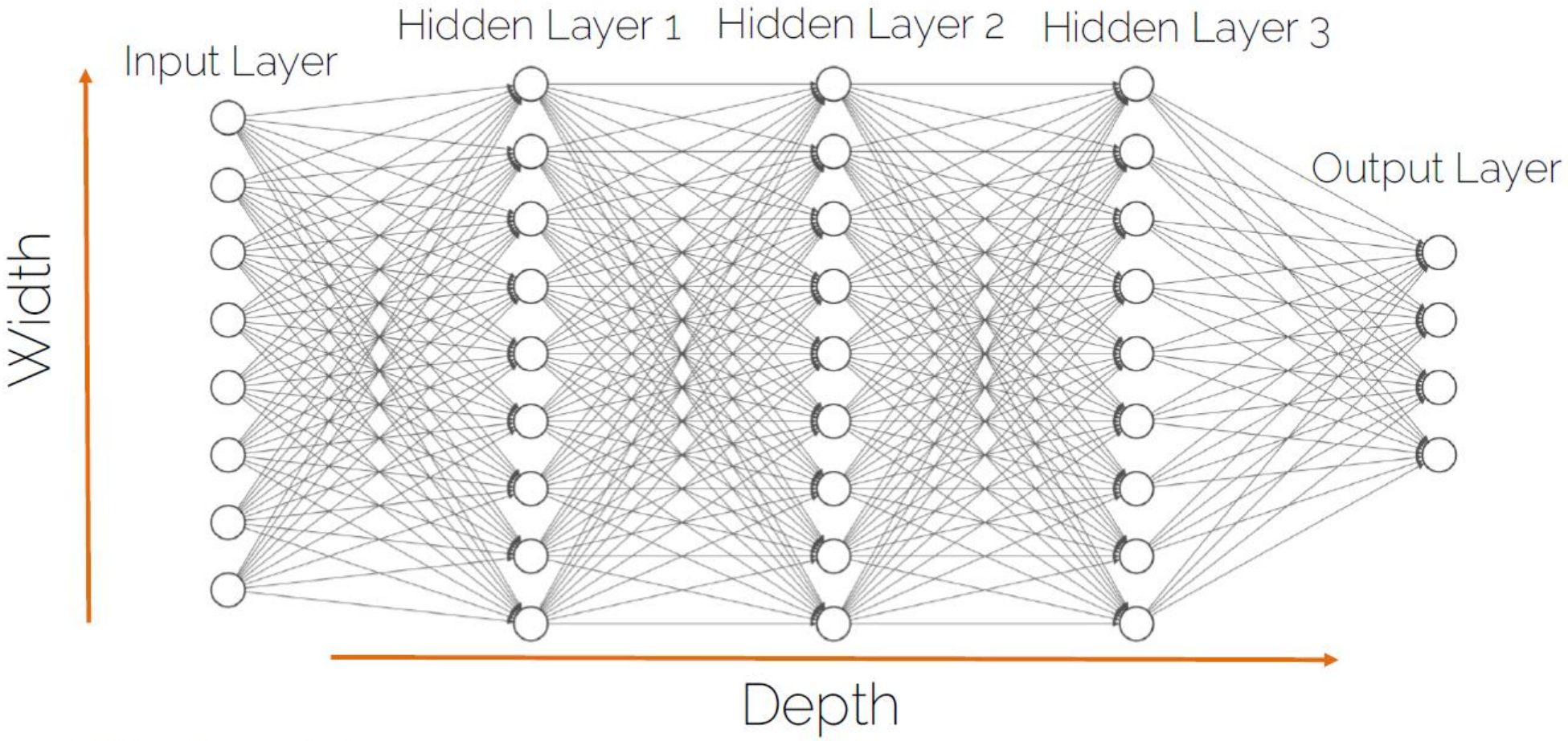


This is a neural network!

- Neurons, arranged in layers convert an input vector into some output.
- Each unit takes an input, applies a (often non-linear) function to it and then passes the output on to the next layer.

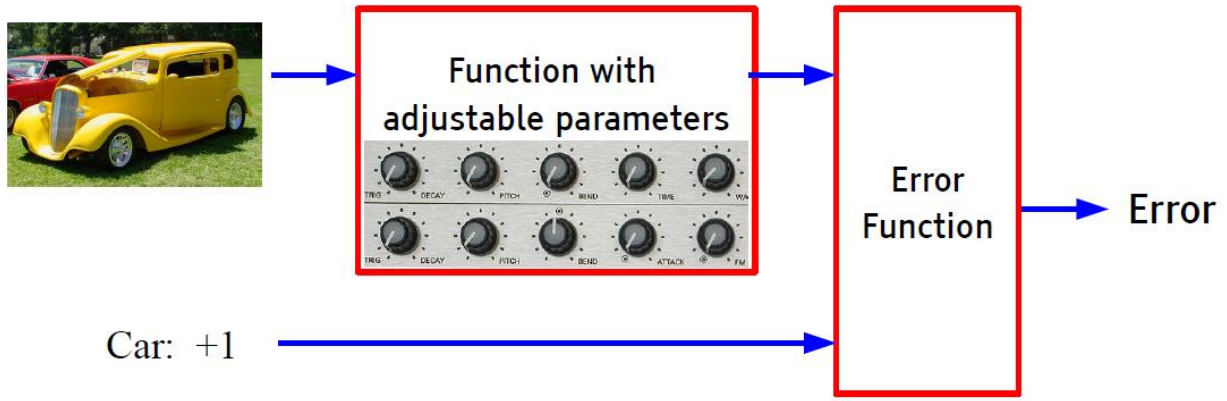


Neural Network



Training a Neural Network

- Given a dataset with ground truth training pairs $[x_i; y_i]$
- Find optimal **weights** W using **stochastic gradient decent**, such that the **loss function** is minimized
- Compute **gradients** with **backpropagation** (use **batch mode**; more later)
- Iterate many times over training set (SGD; later)



Training Neural Networks

- **Loss/Error/Cost function** measures the **quality** of a particular set of parameters (W, b) based on how well the output of the network **agrees** with the truth labels in the training data

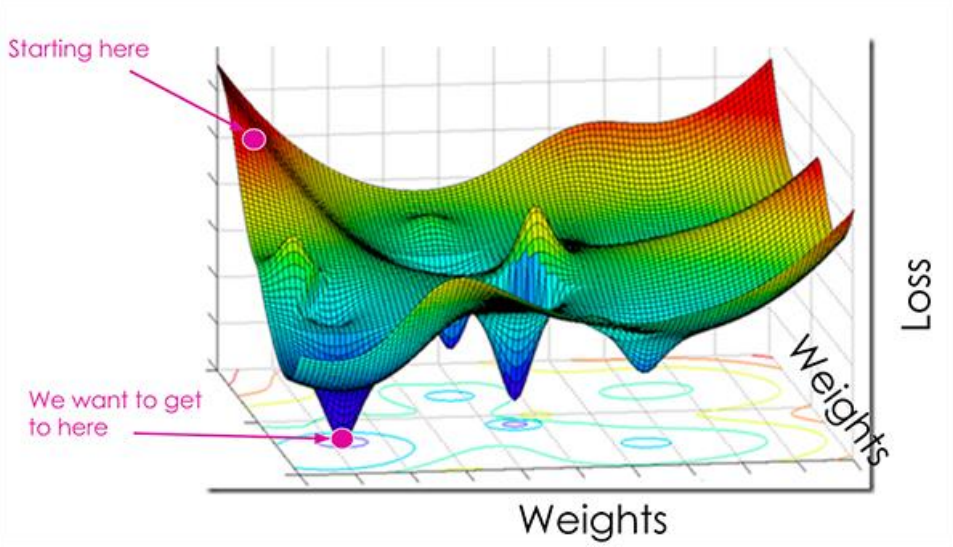
- Find network weights to minimize the prediction loss between true and estimated labels of training examples.

- **Optimization**: Changes the model in order to improve the loss function (i.e., to improve the estimation)

- It's a balancing act and our choice of loss function and model optimizer can dramatically impact the quality, accuracy, and generalizability of our final model.

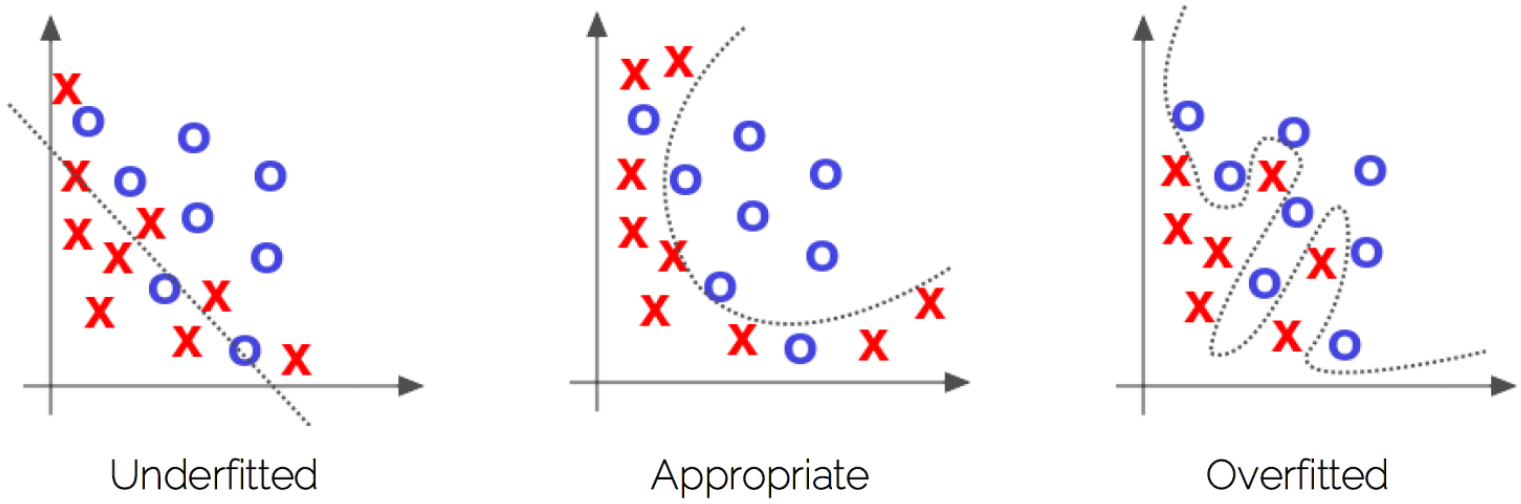
What is “loss” when training a neural network?

- At the most basic level, a **loss** function quantifies how “good” or “bad” a given predictor is at classifying the input data points in a dataset.
- The smaller the loss, the better a job the classifier is at modeling the relationship between the input data and the output targets.



Training of multi-layer networks

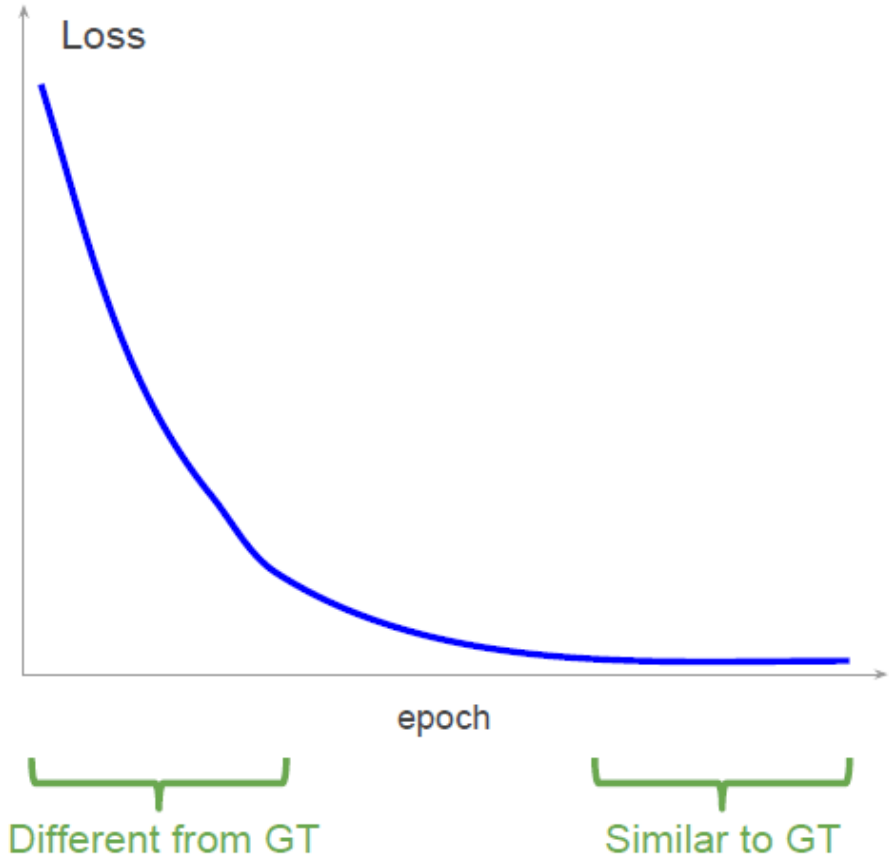
- We, therefore, seek to:
  - Drive our loss down, thereby improving our model accuracy.
  - Ideally, do so as fast as possible and with as little hyperparameter updates/experiments.
  - All without overfitting our network and modeling the training data too closely.



Loss functions properties

- Minimum (0 value) when the output of the network is equal to the ground truth data.

- Increase value when output differs from ground truth.

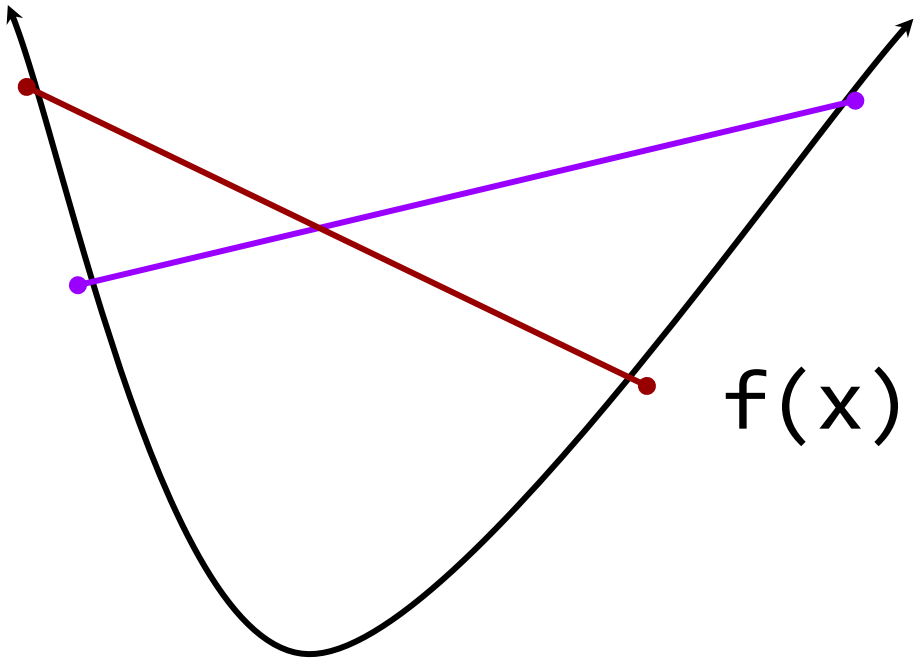


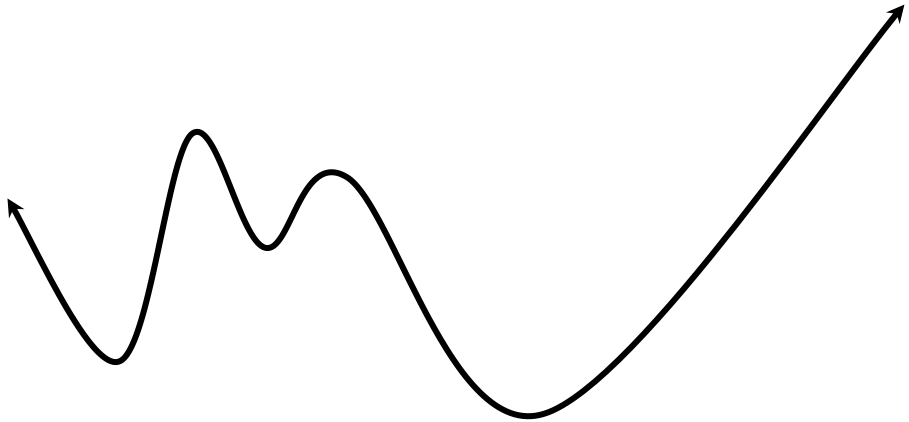
Loss functions properties

- Ideally
  - convex function (no more than one minimum.)
- In reality
  - so many parameters (in the order of millions) than it is not convex
- Varies smoothly with changes on the output
  - Better gradients for gradient-based optimization (More Later)
  - Easy to compute small changes in the parameters to get an improvement in the loss

Convex vs. Non-convex

- Connect any two points on graph with a line, that line lies above function everywhere.
- Any local extrema is global extrema!





https://en.wikipedia.org/wiki/Convex_function#/media/File:ConvexFunction.svg

Typical loss functions

- Typical loss functions (also called “objective functions” or “scoring functions”) include:
  - Binary cross-entropy
  - Categorical cross-entropy
  - Sparse categorical cross-entropy
  - Standard Hinge
  - Squared Hinge

  } Classification

  - Mean Squared Error (MSE)
  - Mean Absolute Error (MAE)

  } Regression

Loss function for classification

- Binary cross-entropy: $\mathrm{L}(\mathbf{x}_i, y_i; \mathbf{w}) = -\log P_{\mathbf{w}}(y_i \mid \mathbf{x}_i)$
- Neural network estimates a probability $P_{\mathbf{w}}(y_i \mid \mathbf{x}_i)$ for class label y_i with parameters w, models a Bernoulli distribution: $P(y|x) = \hat{p}^y(1-\hat{p})^{1-y}$, where $\hat{p}$ is the network output.
  - We could maximize the probabilities of y_i given the training set
  - To frame it as a minimization we **minimize the negative log -likelihood**
  - The $\log$ is used to simplify numerical stability and math simplicity. (turn products into sums)
- Categorical cross-entropy: $\mathrm{L}(\mathbf{x}_i, y_i; \mathbf{w}) = -\sum_c \log P_{\mathbf{w}}(y_i = \mathrm{c} \mid \mathbf{x}_i)$

Loss function for regression

- Mean Squared Error (MSE): $\mathrm{L}(\mathbf{x}_i, y_i; \mathbf{w}) = (f_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2$
  - L_2 loss
  - Sum of Squared Differences (SSD)
  - Prone to outliers
  - Compute-efficient optimization
  - Optimum is the mean
- Mean Absolute Error (MAE): $\mathrm{L}(\mathbf{x}_i, y_i; \mathbf{w}) = |f_{\mathbf{w}}(\mathbf{x}_i) - y_i|$
  - L_1 Loss
  - Sum of absolute differences
  - More robust to outliers
  - Costly to optimize

Training of multi-layer networks

- Our model has parameters/weights and we want to find the best values for them.
- To start we pick random values and we need a way to measure how well the algorithm performs using those random weights.
- Loss functions can be written as an average over loss functions for individual training examples:
  - $E(\mathbf{w}) = \frac{1}{N}\sum_i \mathrm{L}(\mathbf{x}_i, y_i; \mathbf{w})$ or $E(\mathbf{w}) = \frac{1}{N}\sum_i \mathrm{L}(\mathrm{f}(\mathbf{x}_i, \mathbf{w}), y_i)$
    - $where\ f$ is the neural network with parameters $\boldsymbol{w}$
    - (x_i, y) is a training sample-label pair
    - L is the loss function
    - N – number of training samples

Neural Network Training with Gradient Descent

- Our goal is to minimize the loss function and the way we have to achieve it is by increasing/decreasing the weights, i.e. fitting them.

- The question is, how do we know what parameters should be bigger and what parameters should be smaller?

- The answer is given by the **derivative** of the loss function with **respect to each weight**: $\nabla_w L(w)$
  - It tells us how loss would change if we modified the parameters.

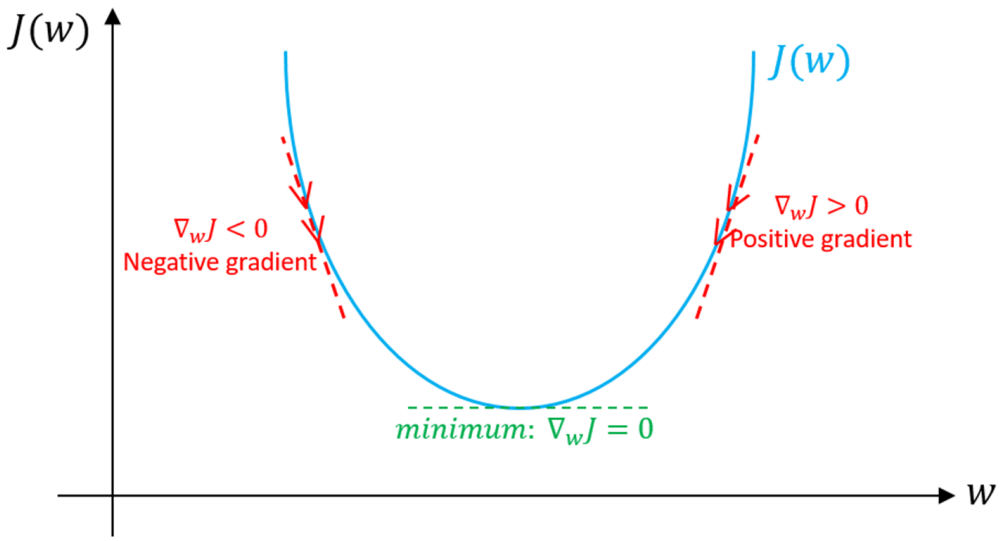
Neural Network Training with Gradient Descent

- Find network weights to minimize the prediction loss between true and estimated labels of training examples:

- $E(\mathbf{w}) = \frac{1}{N}\sum_i \mathrm{L}(\mathbf{x}_i, y_i; \mathbf{w})$
- Update weights by gradient descent
  - α – Learning Rate

Direction of greatest increase of the function

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E}{\partial \mathbf{w}}$$



- **Epoch:** An entire pass through the entire training set.
- **Iteration**: An update of the weights using the computed gradient

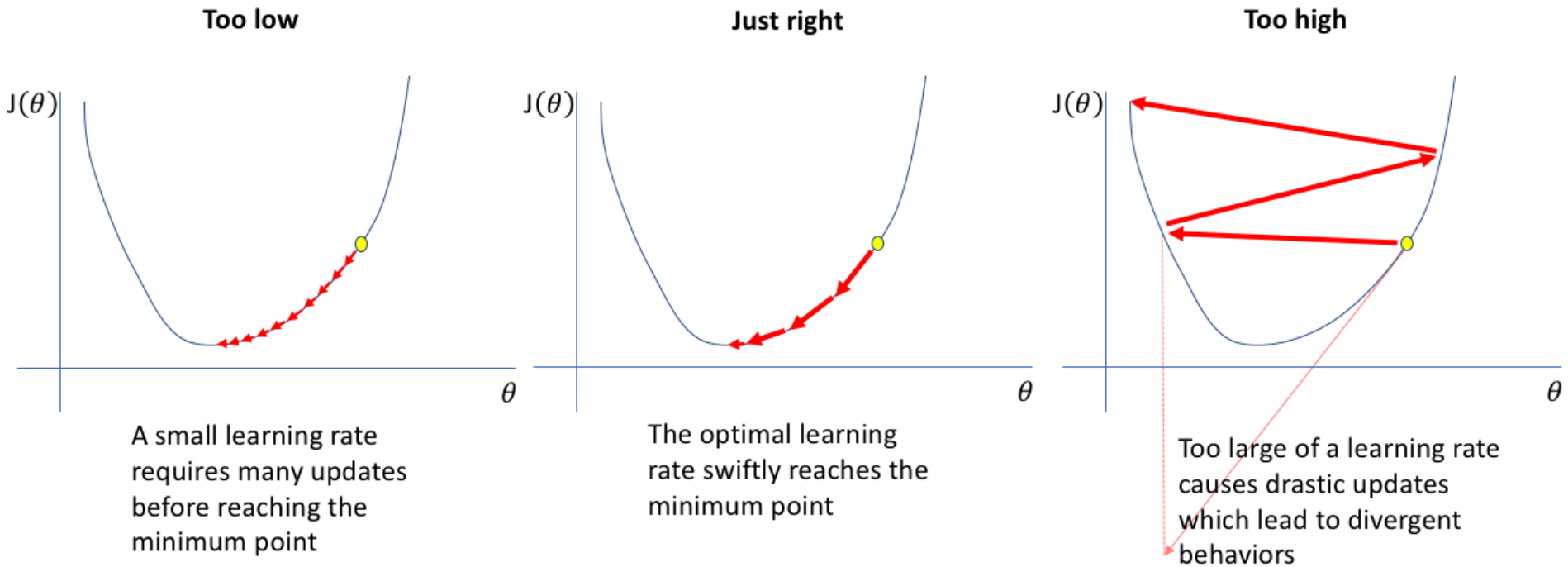
Stochastic Gradient Descent (SGD)

- **IDEA:** use a random subset (mini-batch) of the data (of a given size) to compute an approximation of the gradient of the loss function
- **Pros**
  - Simple but sufficiently effective
  - Fast (depending on mini-batch size)
  - Scale the problem with data and model size
- **Cons**
  - Needs more iterations to converge → but tricks to are present

Effect of learning rate

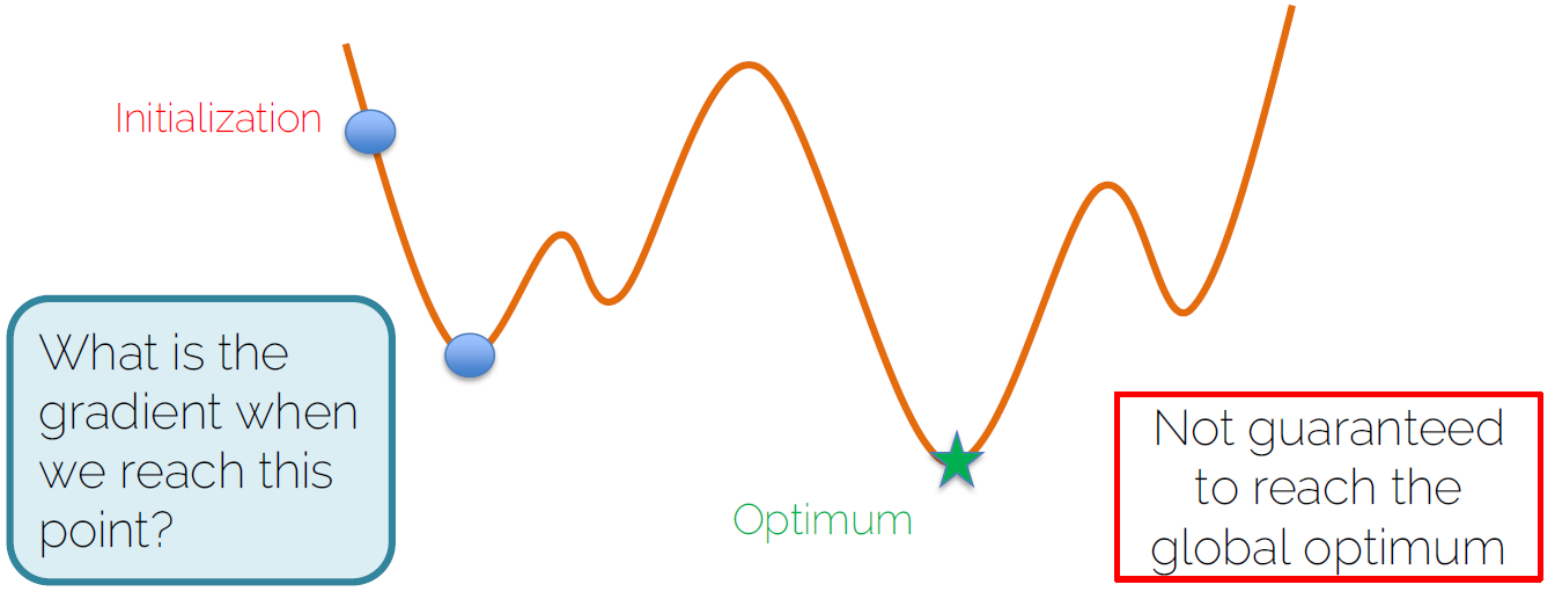
- Constant learning rate $0 < \alpha \leq 1$
  - Small: slow convergence.
  - Large: Oscillatory behavior near a local minimum

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E}{\partial \mathbf{w}}$$



Convergence of Gradient Descent

- Neural networks are non-convex
  - many (different) local minima
  - no (practical) way to say which is globally optimal
  - Learning rate schedulers change the learning rate dynamically



Gradient Descent Pseudocode

- Given a loss function L and a single training sample $\{x_i, y_i\}$
- Find best model parameters $\theta = \{W, b\}$
- Cost $L_i(\theta, \chi_i, y_i)$
- Gradient Descent
  - Initialize with random values (more to that later)
  - $\theta^{k+1} = \theta^k - \alpha \nabla_\theta L_i(\theta^k, x_i, y_i)$
  - Iterate until convergence: $\left|\theta^{k+1} - \theta^k\right| < \varepsilon$

```
for _ in {0,…,num_epochs}:
    L = 0
    for x_i, y_i in data:
        ŷ_i = f(x_i, W)
        L += L_i(y_i, ŷ_i)
    dL/dW =???
    W := W − α dL/dW
```

How to find the gradients?

- We first need to compute the loss function L which involves computing the network output $f(x)$ given input x.
  - This is called the **forward** pass

- One can compute gradients analytically but what if our function is too complex?
  - $f = W_5\sigma(W_4 \tanh(W_3 \max(0, W_2 \max(0, W_1 x))))$
  - How are weights not directly connected to the output layer associated with the objective function?

- Break down gradient computation
  - Backpropagation

Gradient descent with back propagation

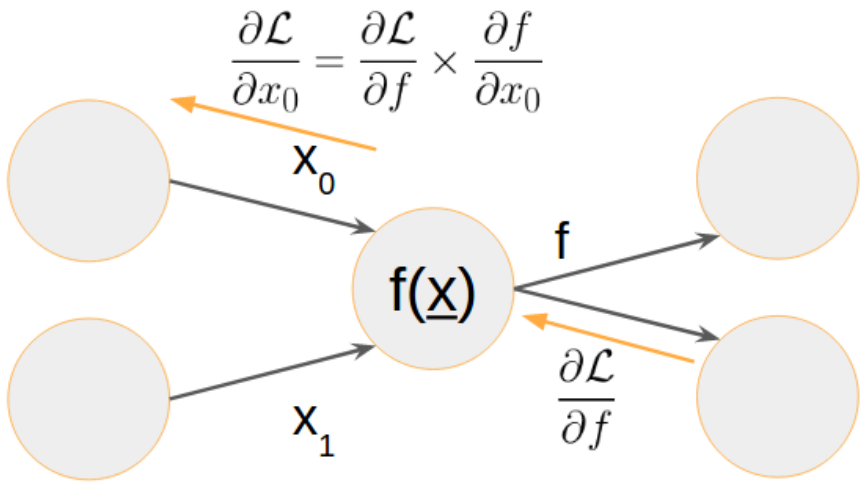
- To compute all the derivatives, we use a **backward** sweep called the **back-propagation algorithm**
  - Gradients are computed in the direction from output to input layers and combined using chain rule
  - Local derivatives are easy to compute because they care only about their own input and output.
  - Backward phase

- **Chain Rule**: If $z = f(g(\underline{x}))$ and $f(.)$, $g(.)$ are continuous differentiable functions then:

$$\frac{dz}{dx_i} = \frac{\partial f}{\partial g}\frac{\partial g(\underline{x})}{\partial x_i}$$

- More chain rule, remember:

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$



The Backpropagation Algorithm

- Create a fully connected network. **Initialize weights.**
- Until all examples produce the correct output within ϵ (or other criteria)

For each example (x_i, t_i) in the training set do:

1. Compute the network output y_i for this example
2. Compute the error between the output and target value

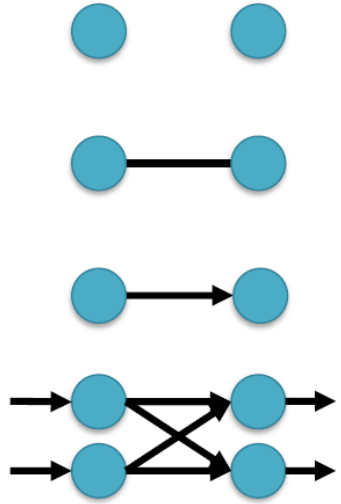
$$E = \sum \left(y_i^k - \hat{y}_i^k\right)^2$$

3. Compute the gradient for all weight values, Δw_{ij}
4. Update network weights with $w_{ij} = w_{ij} - \lambda * \Delta w_{ij}$

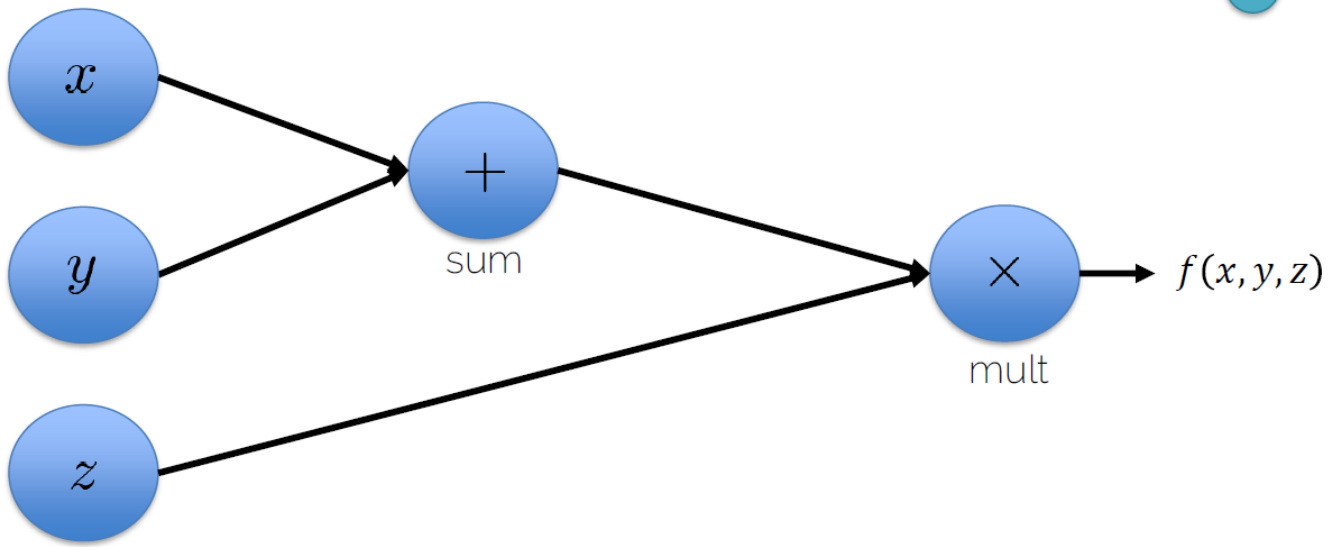
End epoch

Neural Networks as Computational Graphs

- Neural network is a computational graph
  - It has compute nodes
  - It has edges that connect nodes
  - It is directional
  - It is organized in ‘layers’



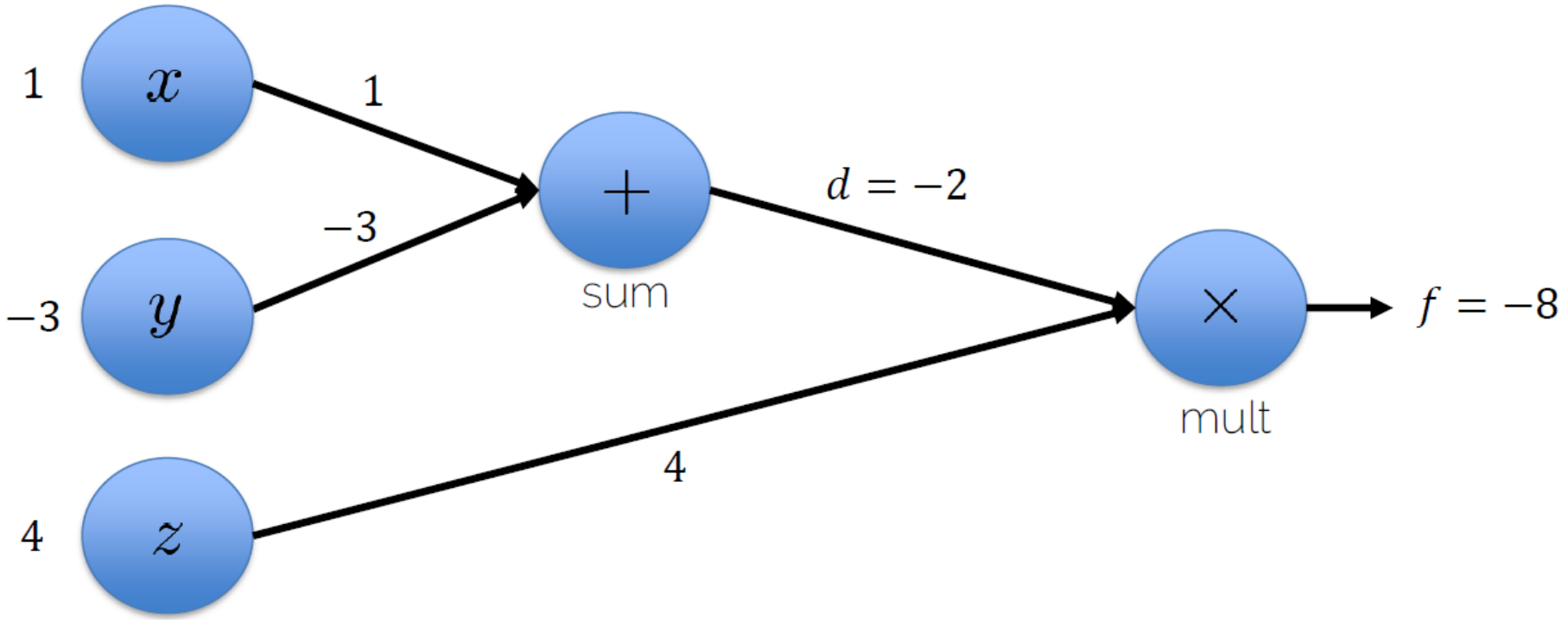
- $f(x, y, z) = (x + y) \cdot z$



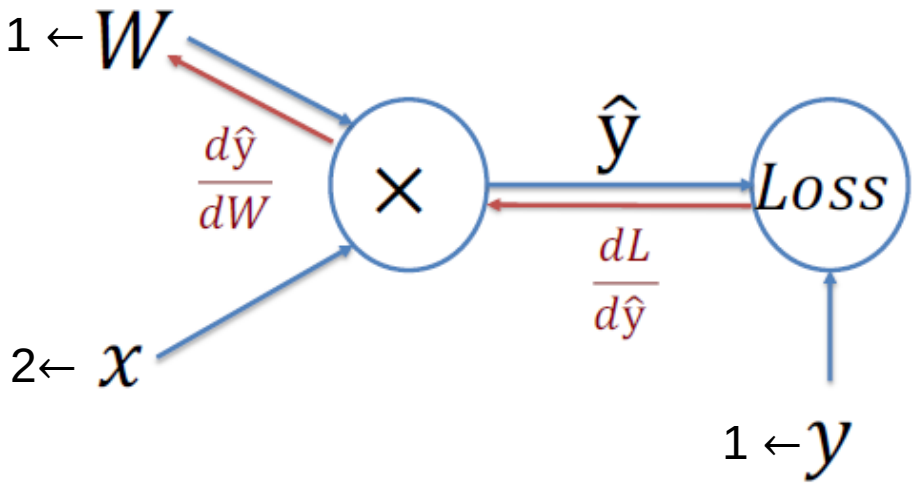
Evaluation: Forward Pass

- $f(x, y, z) = (x + y) \cdot z$

Initialization $x = 1, y = -3, z = 4$



Backprop Example in 1D



$$w \leftarrow w - a\frac{dL}{dW}$$

$$\frac{dL}{dW} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dW}$$

$$= (\hat{y} - y)x$$

$$= 1 \times 2 = 2$$

$$\alpha = 0.1$$

$$\rightarrow w = 0.8$$

$$\xrightarrow{Next\ Epoch} \hat{y}=1.8$$

$$L = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(\hat{y}^2 - 2y\hat{y} + y^2)$$

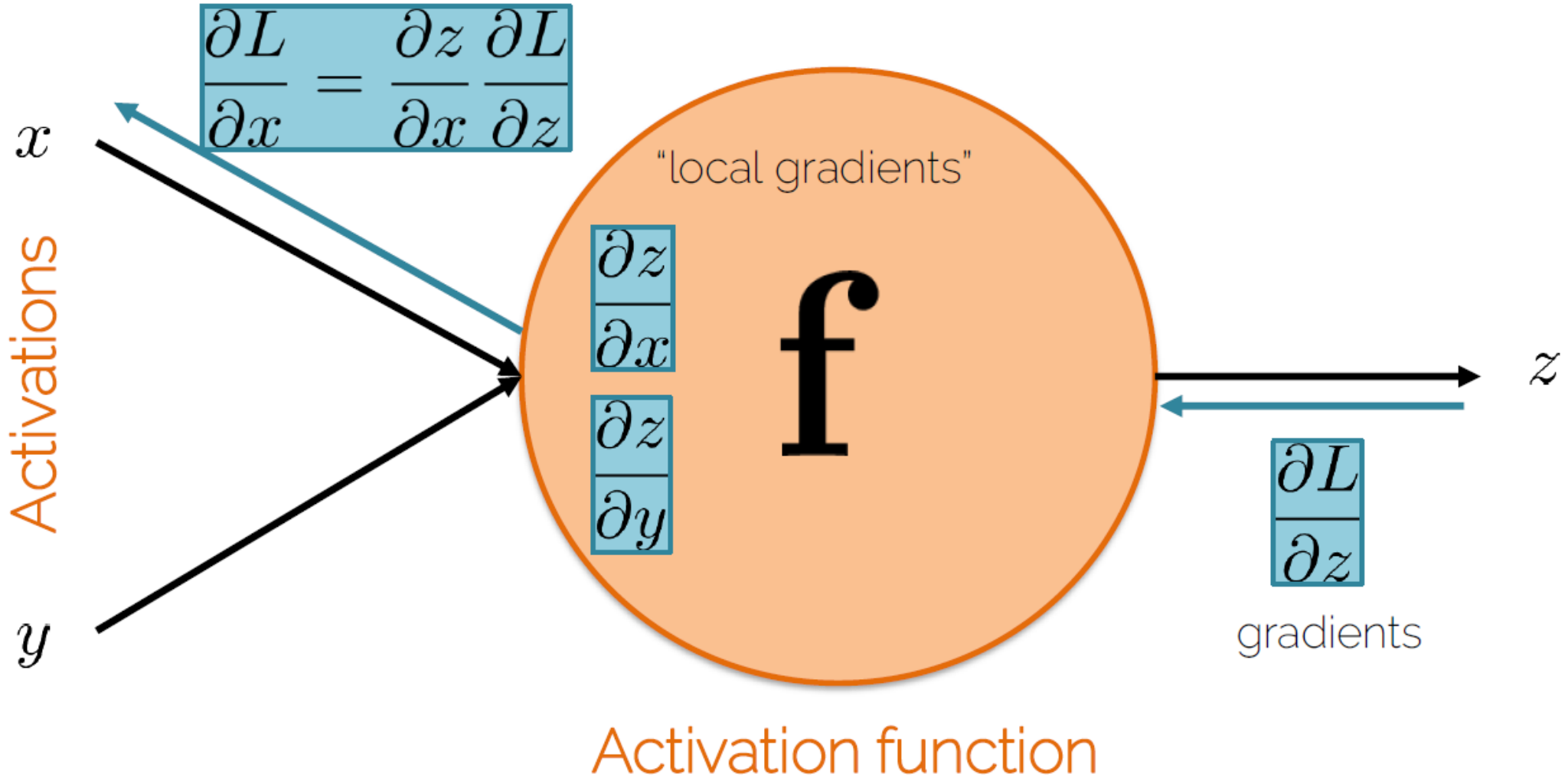
$$\hat{y} = x \cdot w = 2$$

$$\frac{d\hat{y}}{dW} = x = 2 \qquad \frac{dL}{d\hat{y}} = \hat{y} - y = 1$$

https://sebastianraschka.com/faq/docs/mse-derivative.html

Main Steps:
- Visualize the computation as a graph
- Compute the forward pass to calculate the loss.
- Compute all gradients for each computation backwards

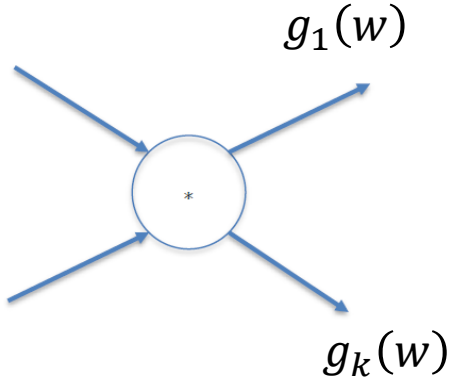
The Flow of Gradients



Backpropagation Considerations

- Neural networks contain multiple nodes in each layer
- Consider that a node that computes function $g(w)$ and propagates its input to k other units each computing $f(\cdot)$
- It will receive k derivatives during backpropagation

What happens if there are multiple outputs in a compute node?

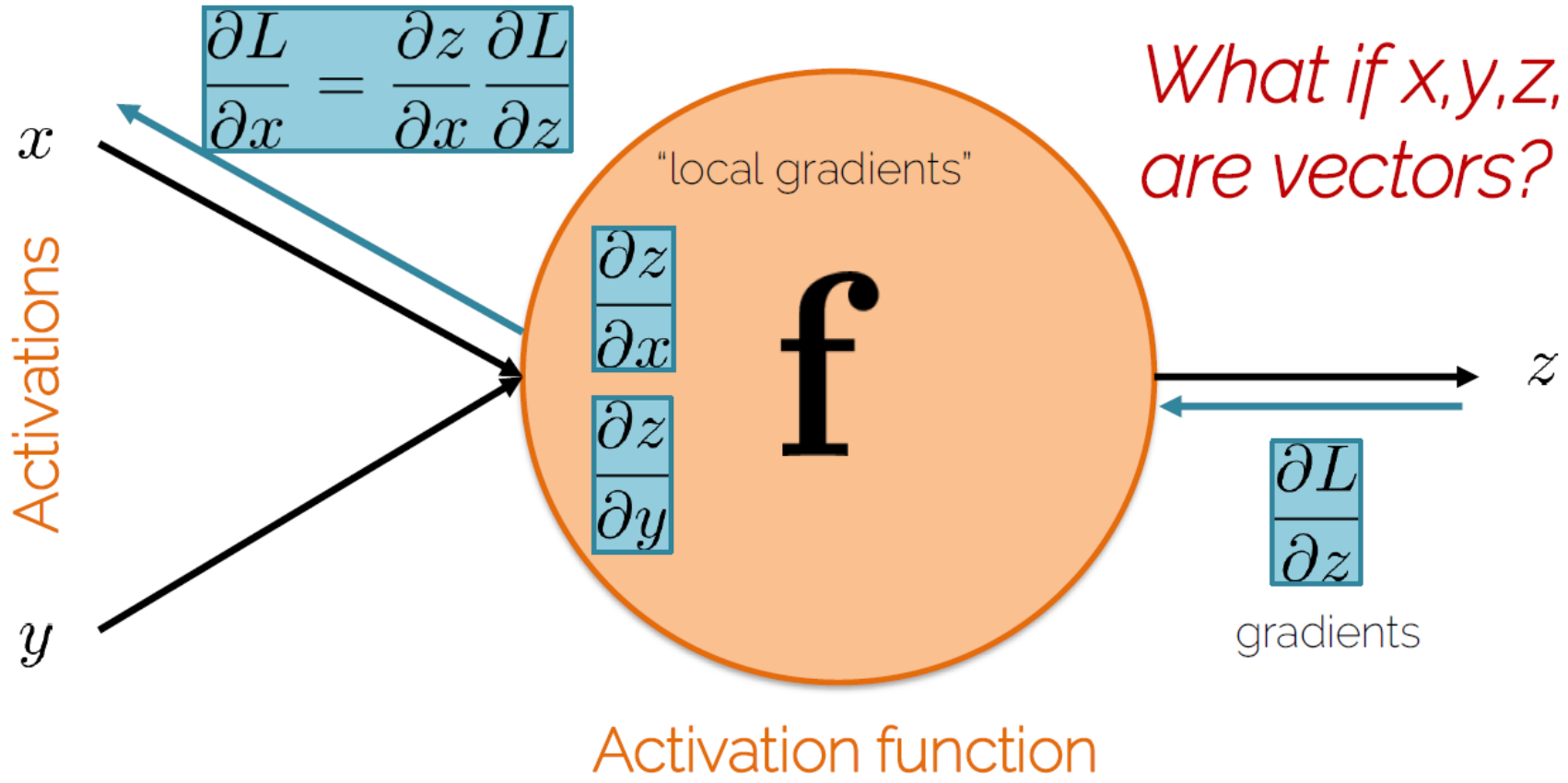


$$\frac{dg}{dw} \quad \frac{df(g_1(w),\ldots,g_k(w))}{dg}$$

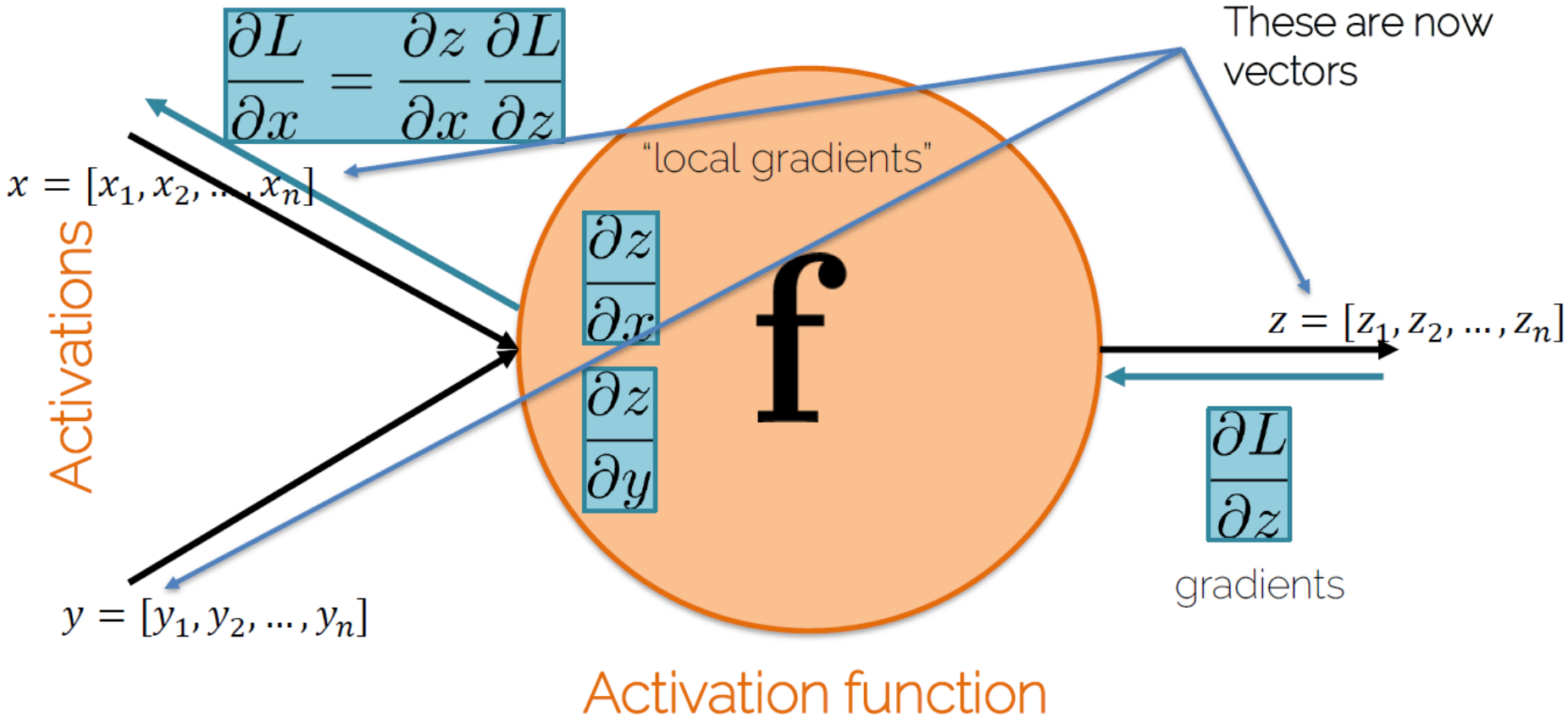
$$\frac{\partial f(g_1(w),\ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w),\ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w}$$

https://www.offconvex.org/2016/12/20/backprop/

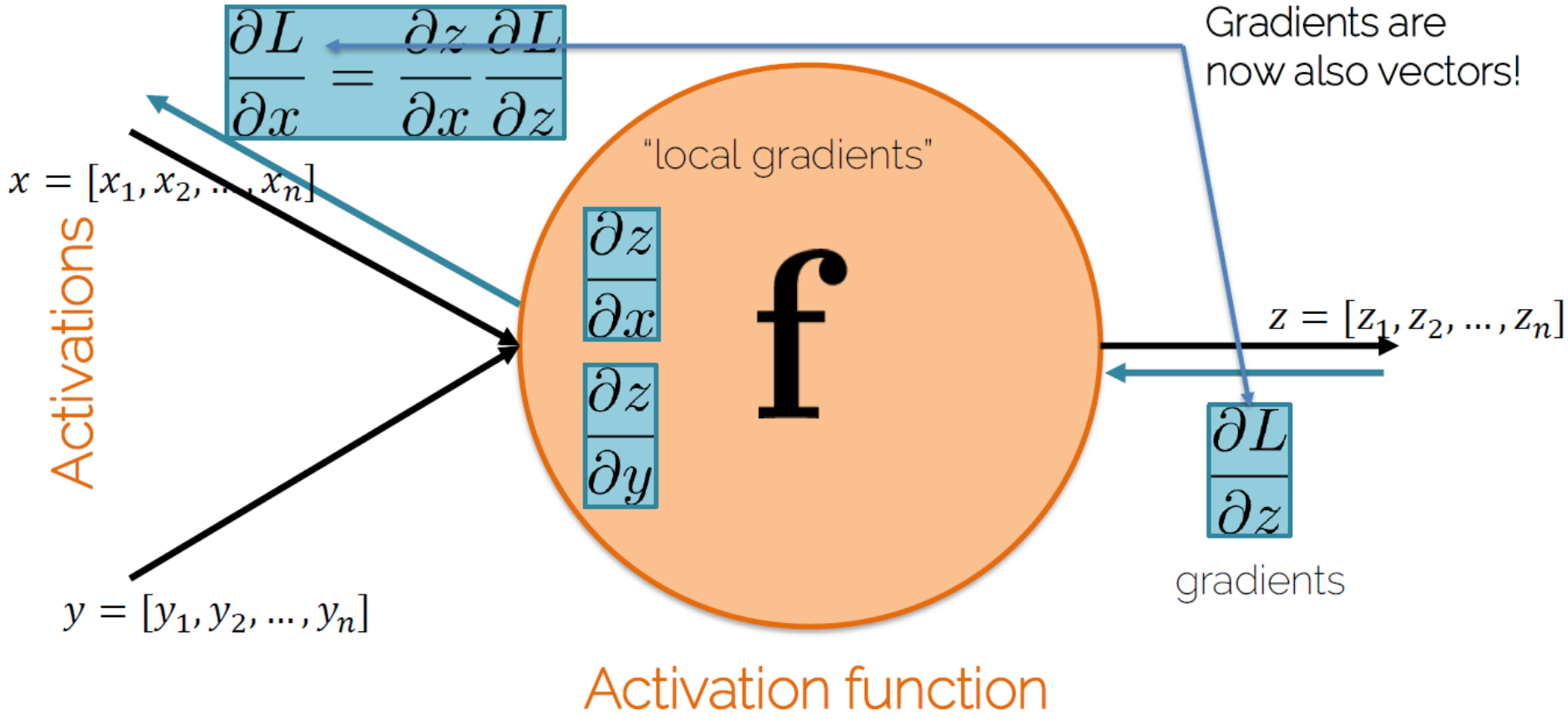
Vectorized Operations



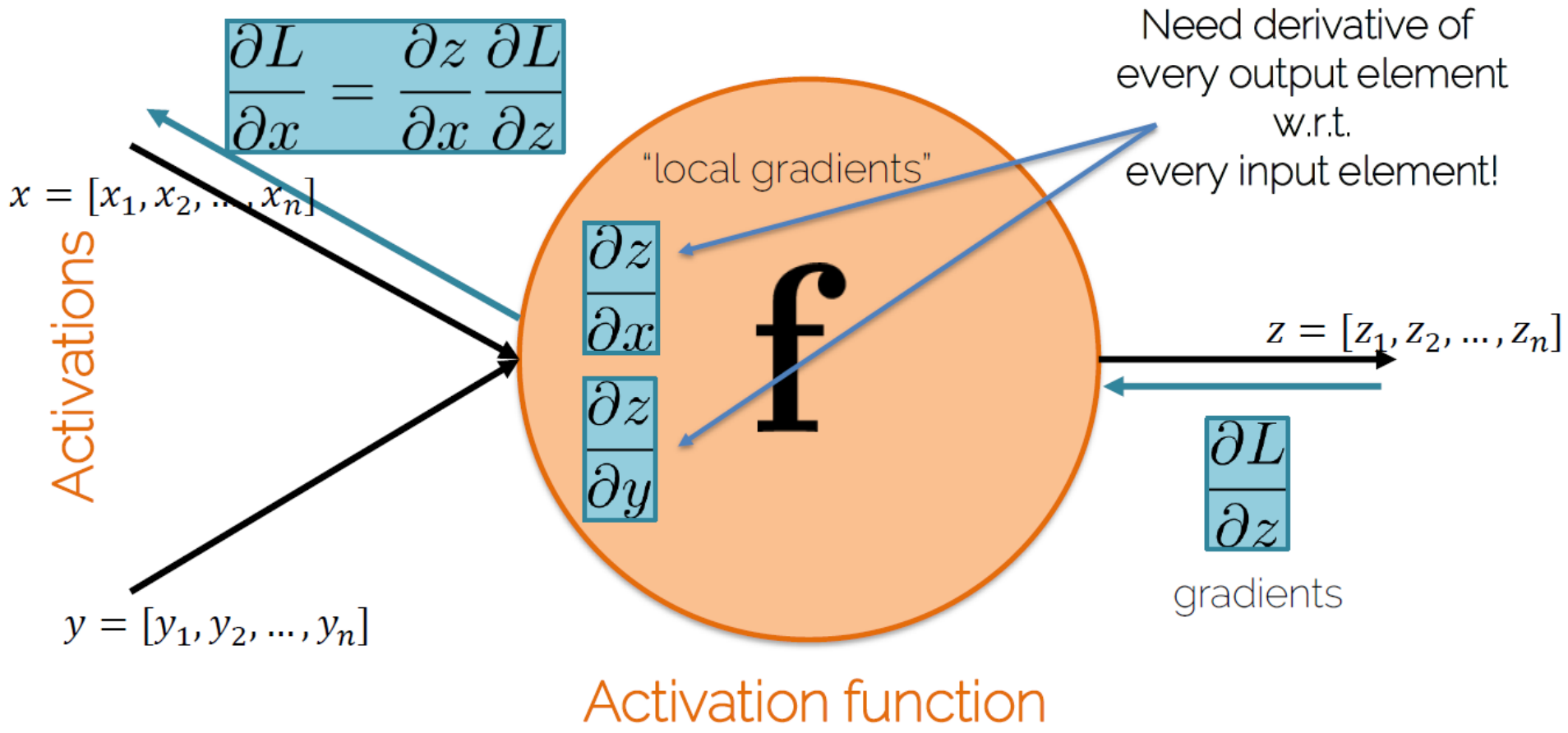
Vectorized Operations



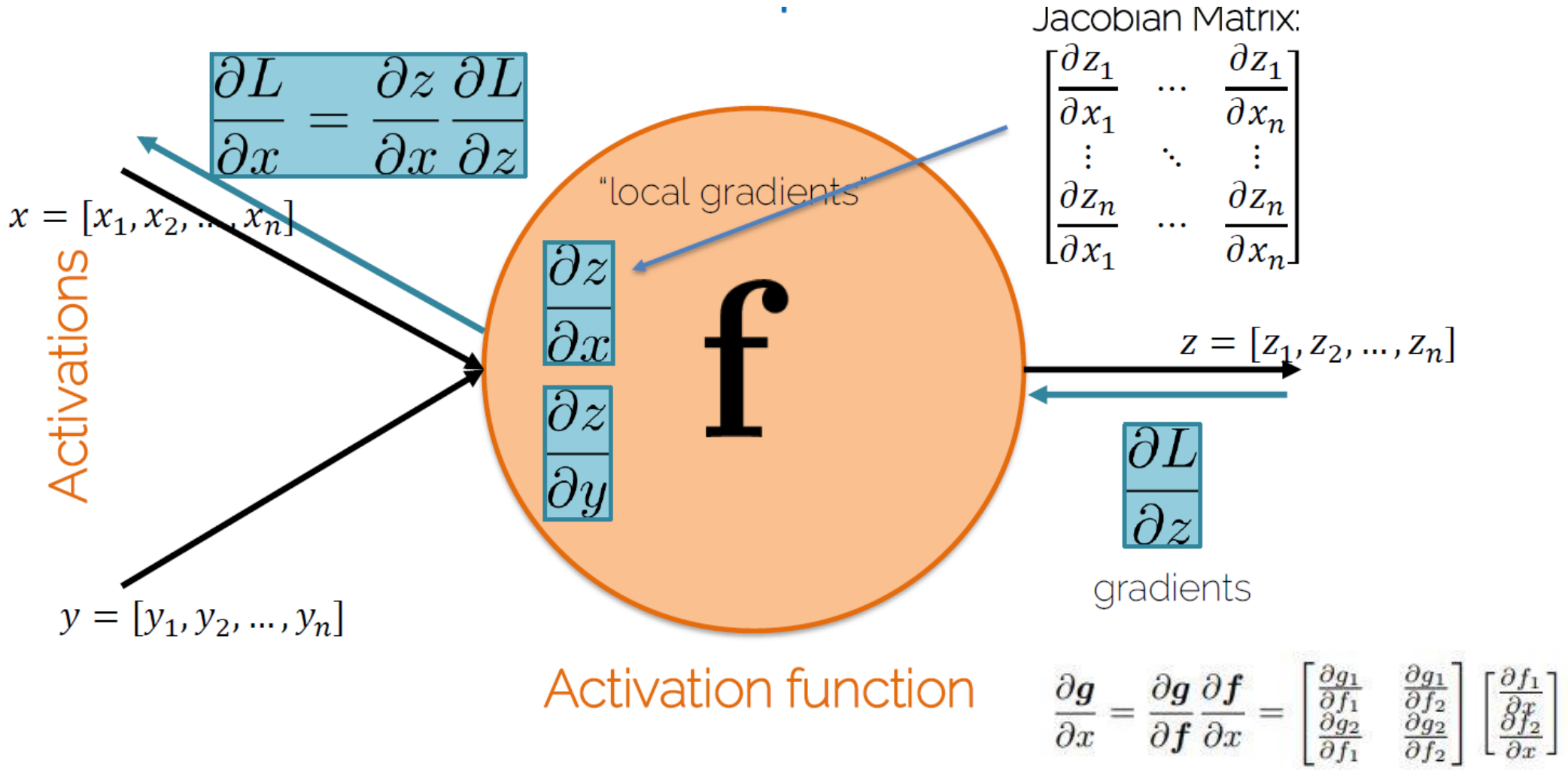
Vectorized Operations



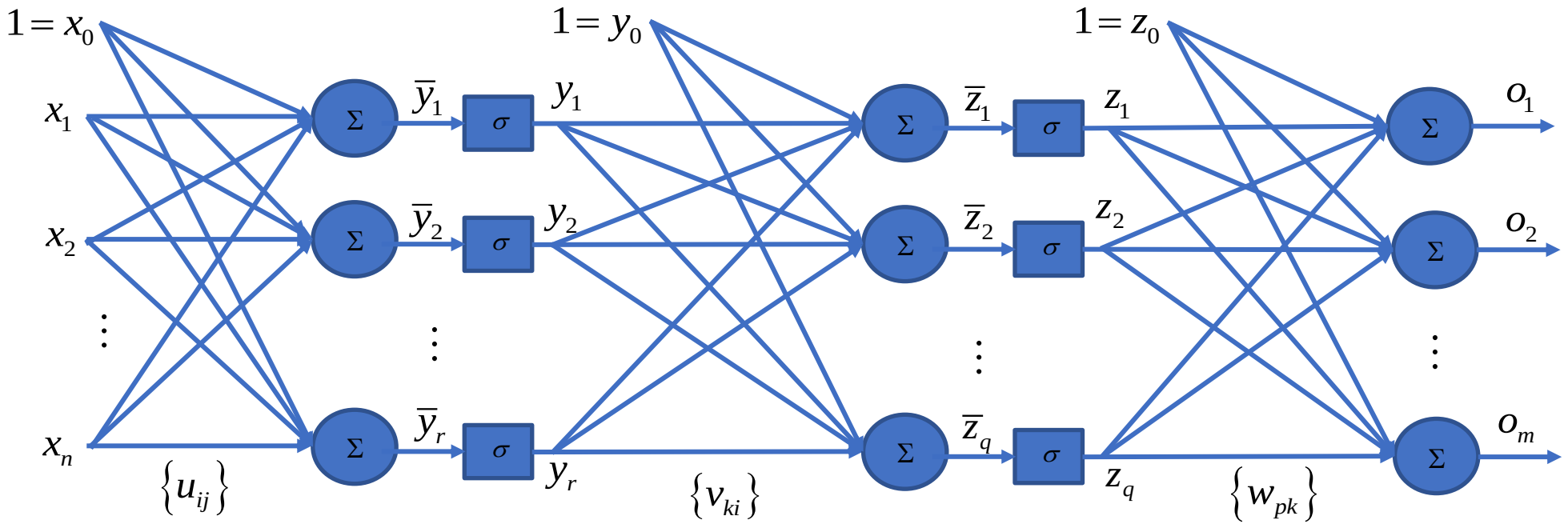
Vectorized Operations



Vectorized Operations



Multilayer Neural Networks



$$O = WZ$$
$$[m \times 1] = [m \times (q+1)] \times [(q+1) \times 1]$$

$$z_i = \sigma(\bar{z}_i),\ i = 1, \ldots, q \quad , \quad \bar{Z} = VY$$
$$[(q+1) \times 1] = [q \times (r+1)] \times [(r+1) \times 1]$$

$$y_j = \sigma(\bar{y}_j),\ j = 1, \ldots, r \quad , \quad \bar{Y} = UX$$
$$[(r+1) \times 1] = [r \times (n+1)] \times [(n+1) \times 1]$$

Weights: W, V, U

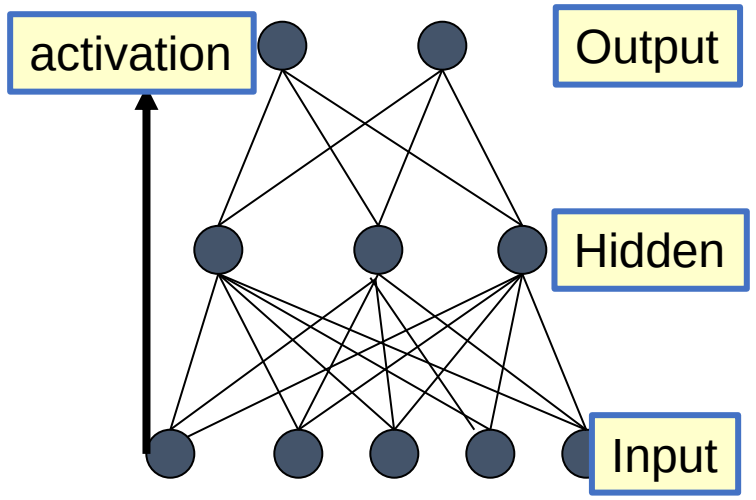
Input: X

Output: O

Activation or squashing function: $\sigma(.) : \mathbb{R} \mapsto \mathbb{R}$

Recap: Multi-Layer Perceptrons

- Multi-layer network
  - A global approximator
  - Different rules for training it
- The Back-propagation
  - Forward step
  - Back propagation of errors



- Congrats! Now you know the one of the important algorithms in machine learning!
- Next:
  - Techniques for initialization/training/optimization
  - Convolutional Neural Networks