# Artificial Intelligence and Machine Learning

# Neural Networks

# Lecture Outline

- Logistic Regression Review
- Neural Networks
  - Forward pass
  - Backward pass

# Neural Networks

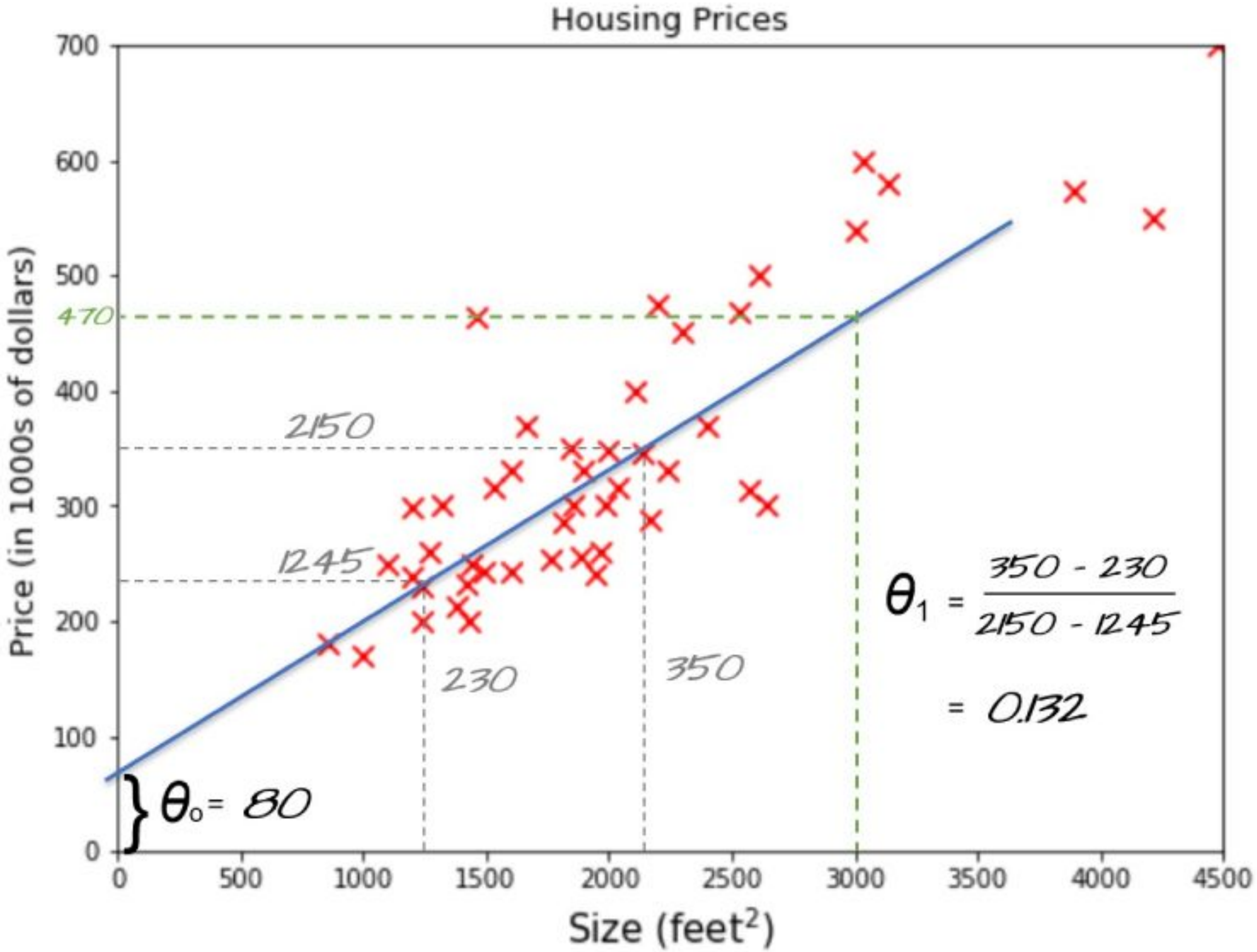
Origins can be traced back to algorithms that try to mimic the brain

40s and 50s: Hebbian learning and Perceptron

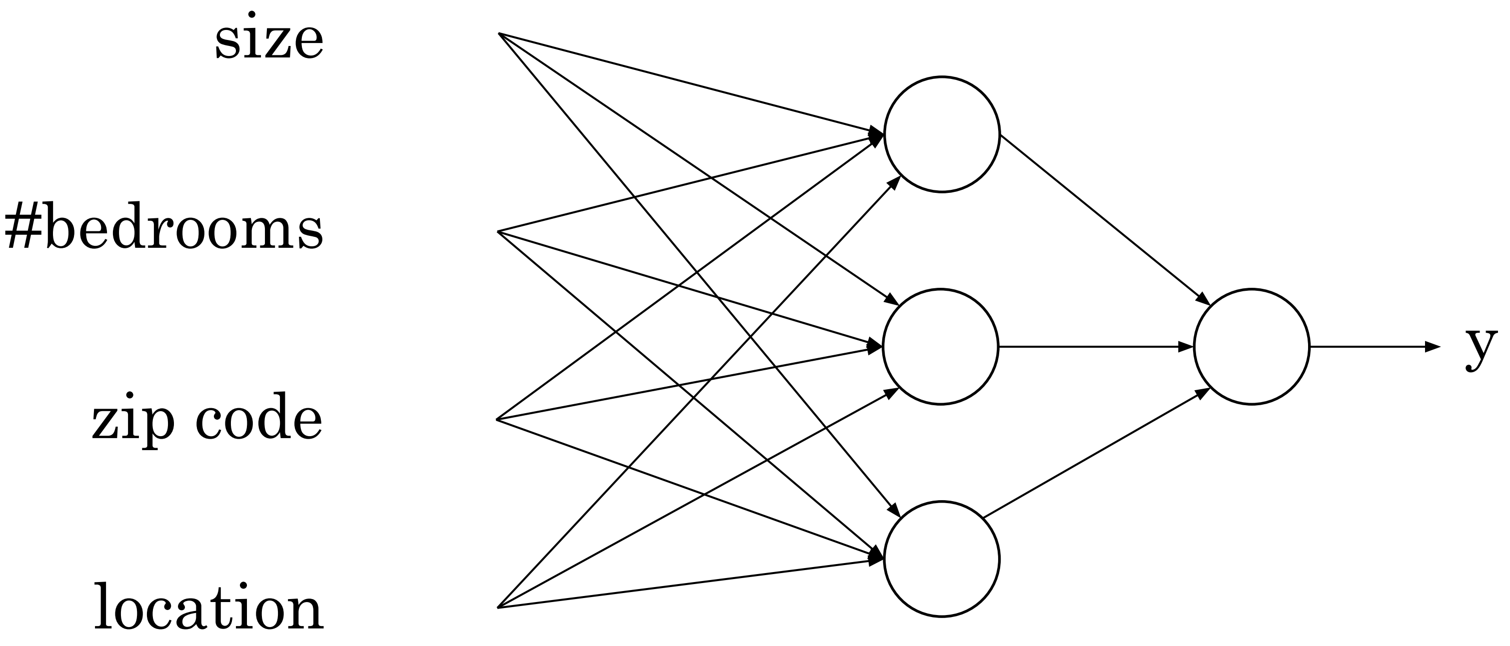
Widely used from the 80s

Recent resurgence of state-of-the-art techniques for different problems

# Housing Price Prediction



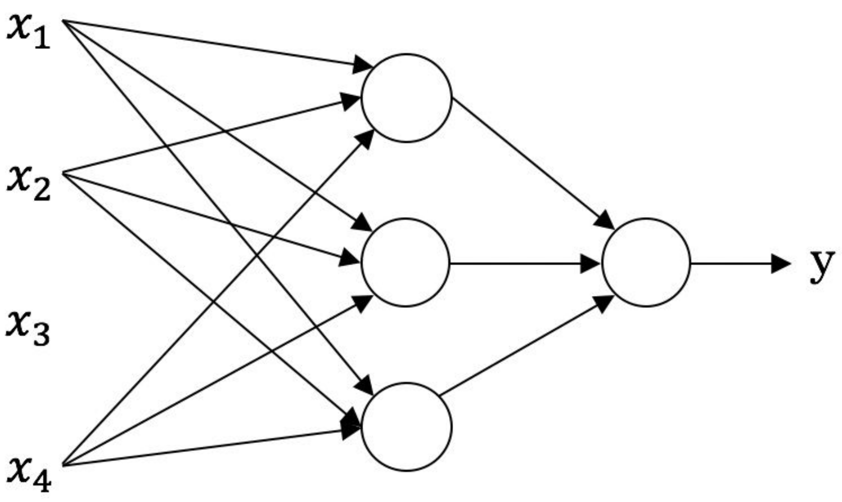
# Housing Price Prediction



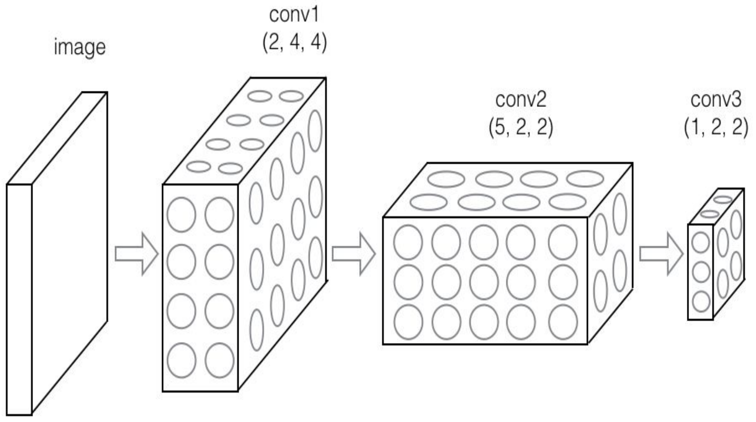
# Supervised Learning

| Input(x) | Output (y) | Application |
|---|---|---|
| Home features | Price | Real Estate |
| Ad, user info | Click on ad? (0/1) | Online Advertising |
| Image | Object (1,…,1000) | Photo tagging |
| Audio | Text transcript | Speech recognition |
| English | Chinese | Machine translation |
| Image, Radar info | Position of other cars | Autonomous driving |

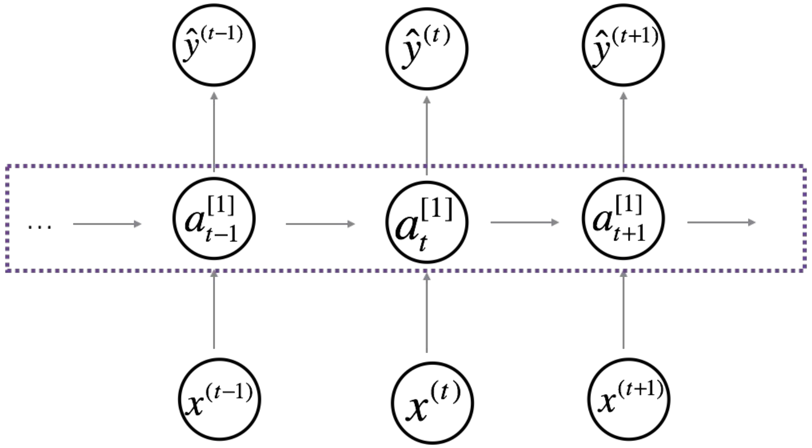
# Neural Network examples




Standard NN



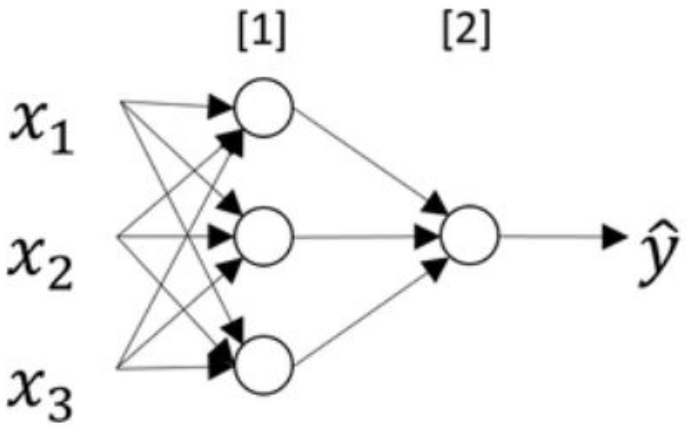

Convolutional NN



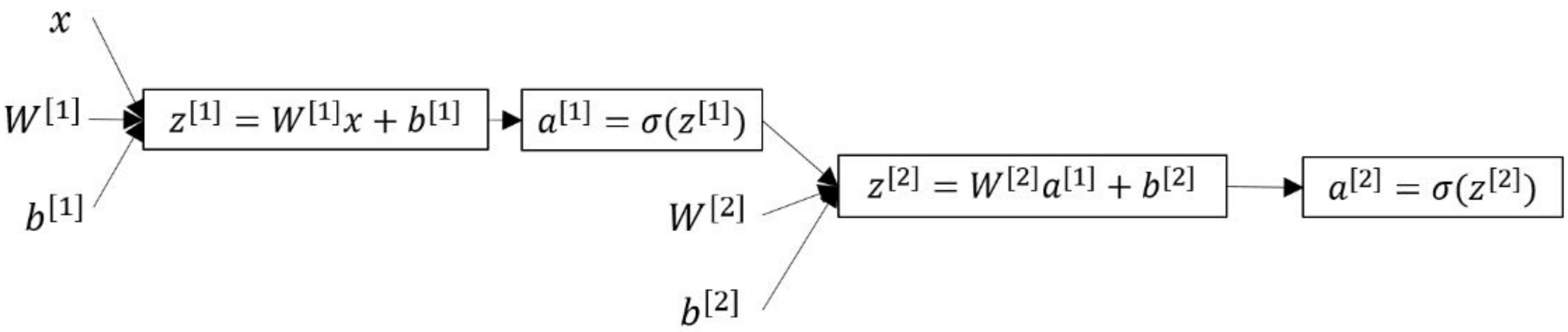

Recurrent NN

# What is a Neural Network?

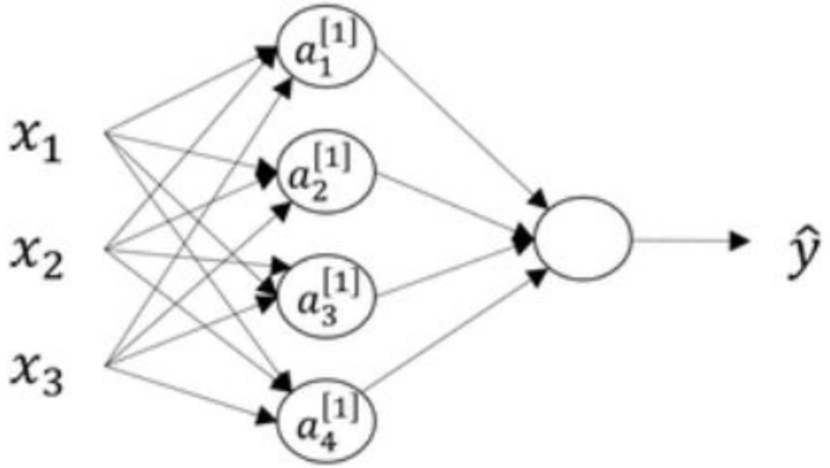
This is a simple 2-layer neural network.




Using computation graph, the forward computation process is like this.



# Neural Network Representation




$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \; a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \; a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \; a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \; a_4^{[1]} = \sigma(z_4^{[1]})$$

In the above example, `z[1]` is the result of linear computation of the input values and the parameters of the hidden layer and `a[1]` is the activation as a sigmoid function of `z[1]`.

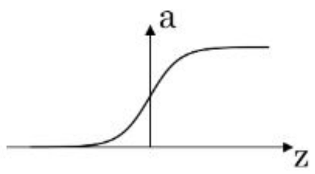
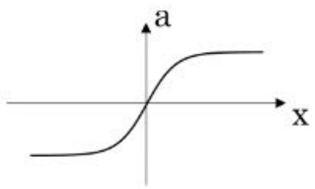
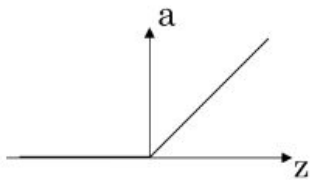
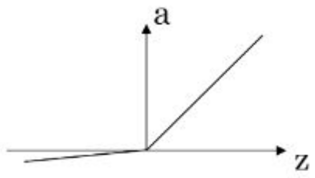
Generally, in a two-layer neural network, if we have `nx` features of input `x` and `n1` neurons of hidden layer and one output value, we have the following dimensions of each variable. Specifically, we have `nx=3, n1=4` in the above network.

# Neural Network Representation

Generally, in a two-layer neural network, if we have `nx` features of input `x` and `n1` neurons of hidden layer and one output value, we have the following dimensions of each variable. Specifically, we have `nx=3, n1=4` in the above network.

| variable | shape | description |
|---|---|---|
| `x` | `(nx,1)` | input value with `nx` features |
| `W[1]` | `(n1,nx)` | weight matrix of first layer, i.e., hidden layer |
| `b[1]` | `(n1,1)` | bias terms of hidden layer |
| `z[1]` | `(n1,1)` | result of linear computation of hidden layer |
| `a[1]` | `(n1,1)` | activation of hidden layer |
| `W[2]` | `(1,n1)` | weight matrix of second layer, i.e., output layer here |
| `b[2]` | `(1,1)` | bias terms of output layer |
| `z[2]` | `(1,1)` | result of linear computation of output layer |
| `a[2]` | `(1,1)` | activation of output layer, i.e., output value |

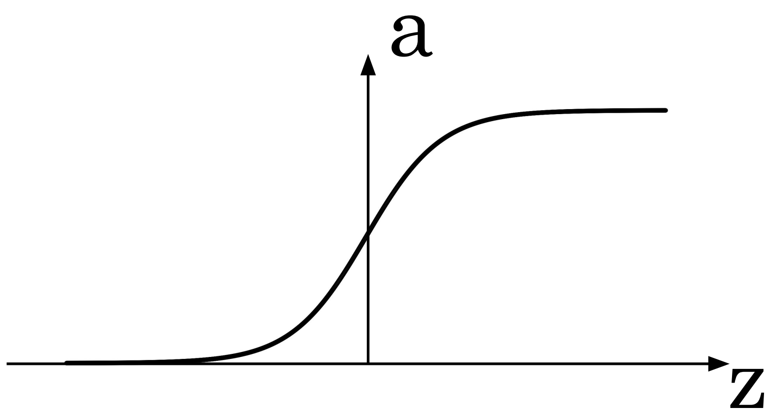
# Activation functions

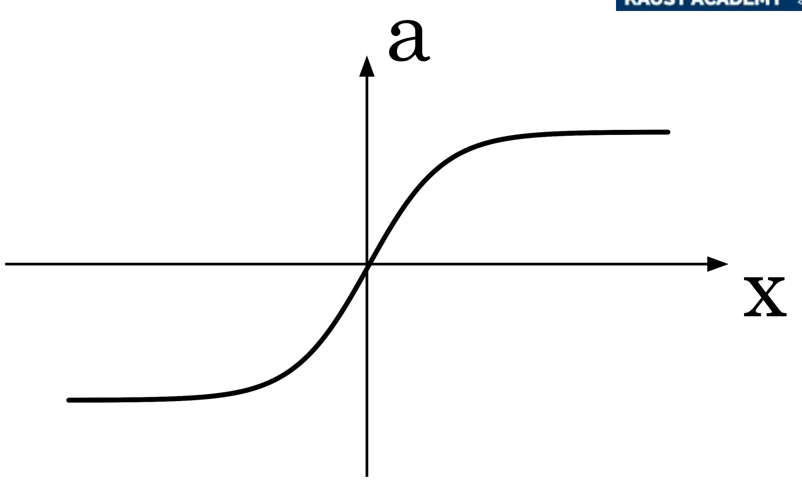
| activation | formula | graph | description |
|---|---|---|---|
| sigmoid | $a = \dfrac{1}{1+e^{-z}}$ |  | also called logistic activation function, looks like an S-shape, if your output value between 0 and 1 choose sigmoid |
| tanh | $a = \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ |  | tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer |
| ReLU | `a=max(0,z)` |  | rectified linear unit, the most widely used activation function |
| Leaky ReLU | `a=max(0.01z,z)` |  | an improved version of ReLU, 0.01 can be a parameter |

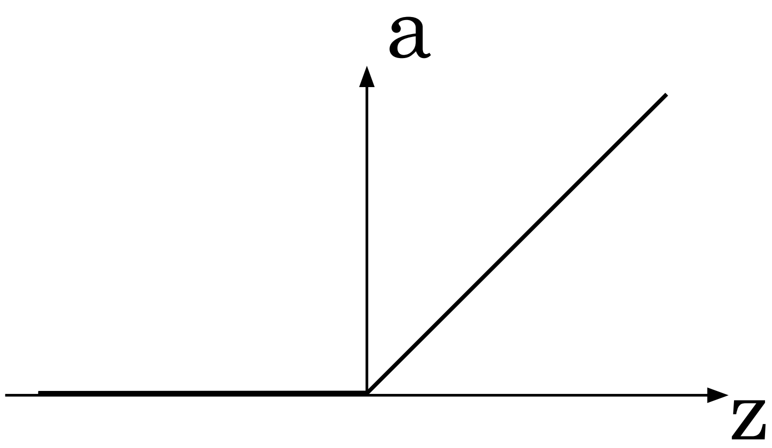
# Activation functions derivatives

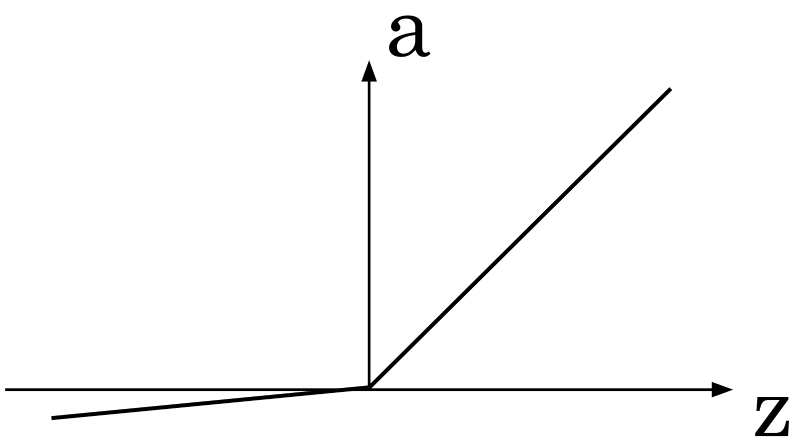
| activation | formula | derivative |
|---|---|---|
| sigmoid | $a = \frac{1}{1+e^{-z}}$ | a(1-a) |
| tanh | $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ | 1-a^2 |
| ReLU | `a=max(0,z)` | 0 if z<0; 1 if z>=0 |
| Leaky ReLU | `a=max(0.01z,z)` | 0.01 if z<0; 1 if z>=0 |

# Pros and cons of activation functions



# Why nonlinear activation functions?

The activation function introduces non-linearity to the network, enabling the modeling of a response variable that varies non-linearly with explanatory variables.

Without a non-linear activation function, a neural network, regardless of its layers, behaves like a single-layer perceptron, producing only linear functions.

# Gradient descent for neural networks

- Initialize $\boldsymbol{\theta}$
- Repeat until convergence

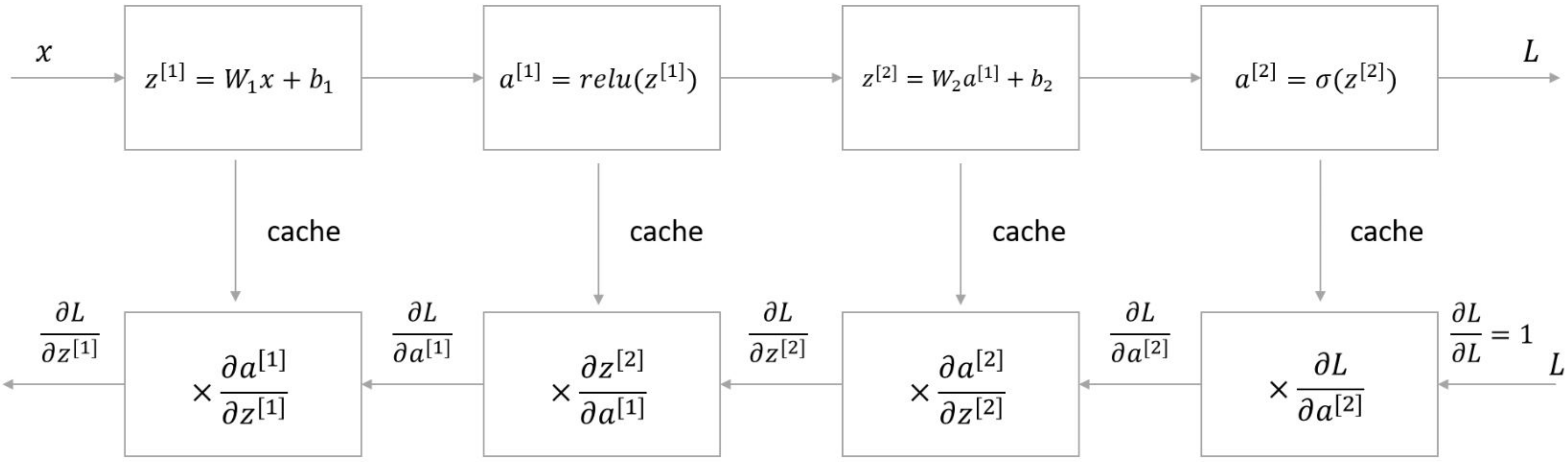
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta})$$

simultaneous update
for j = 0 ... d

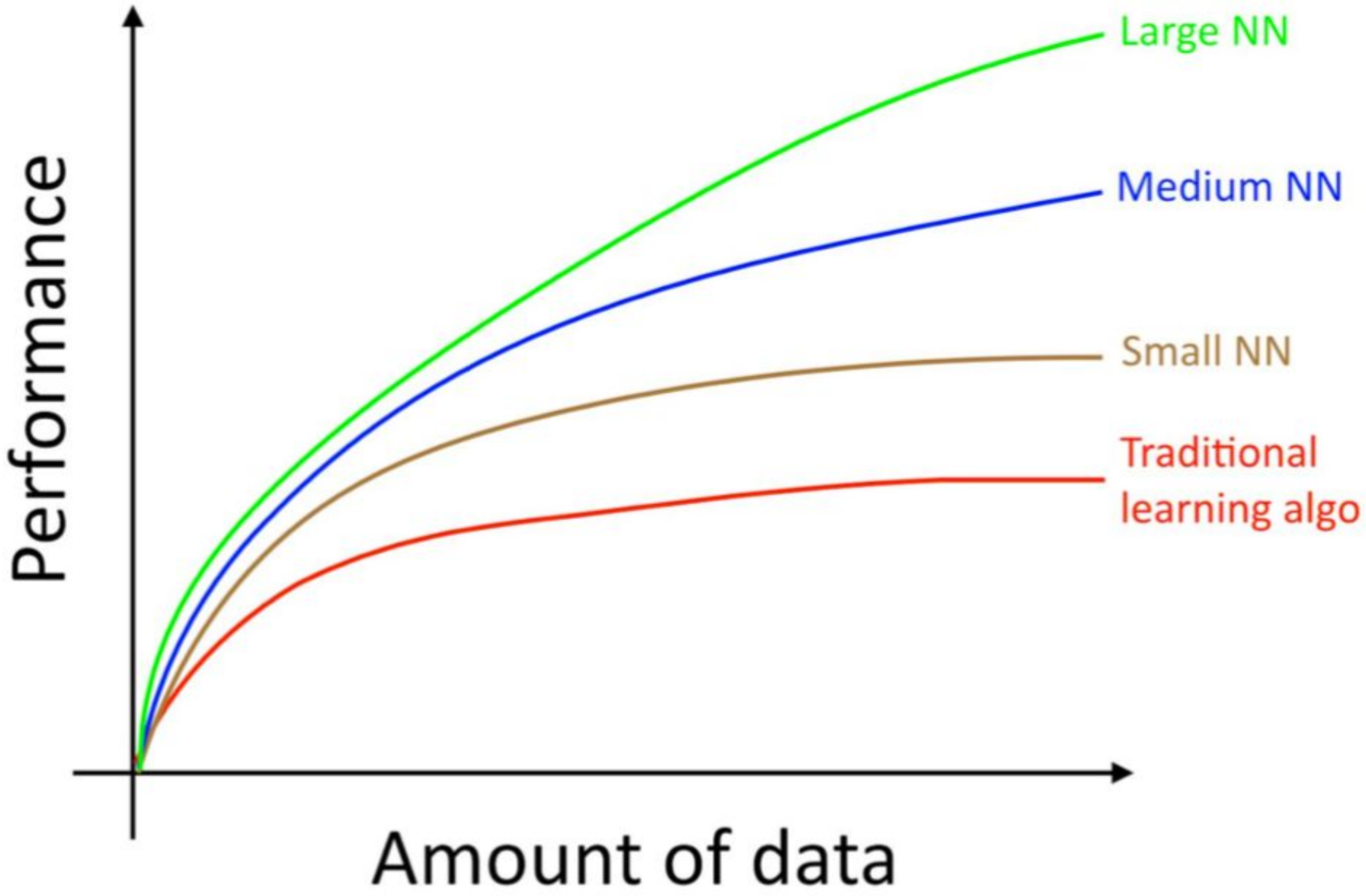
# Forward and backward

## Forward and Backward Propagation

In the algorithm implementation, outputting intermediate values as caches (basically `Z` and `A`) of each forward step is crucial for backward computation.

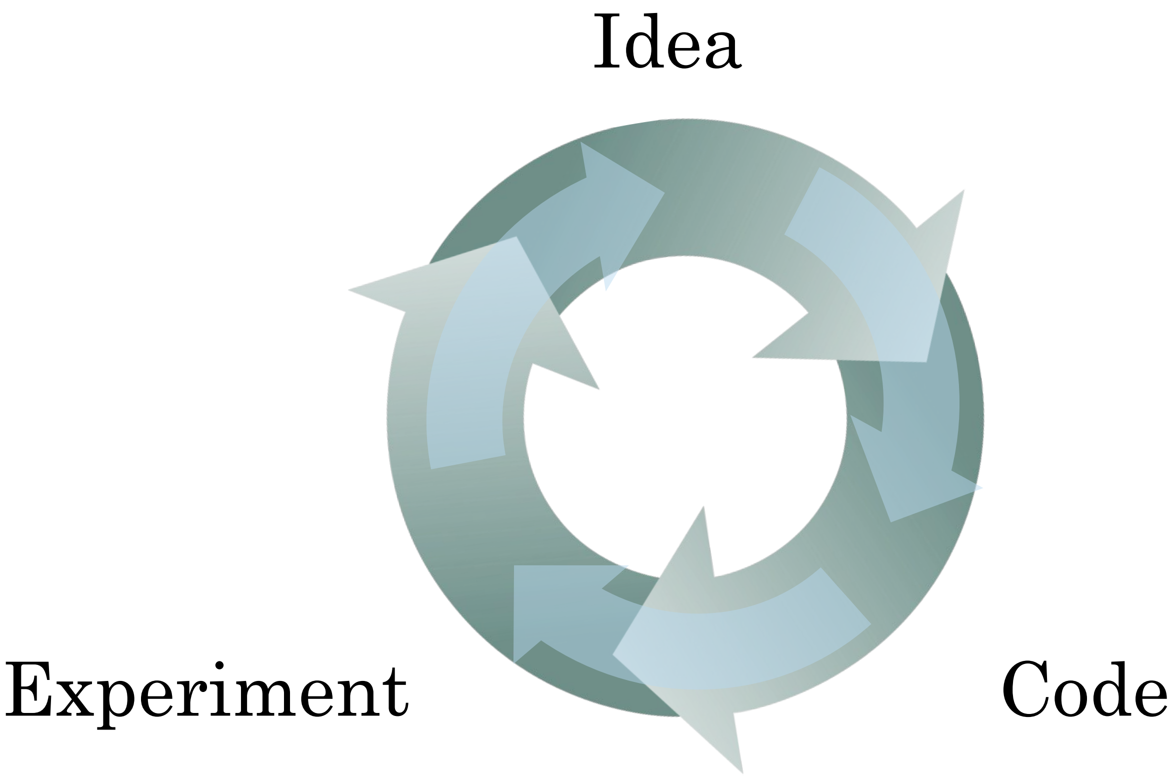


# Scale drives deep learning progress



# Scale drives deep learning progress

- Data
- Computation
- Algorithms



# What happens if you initialize weights to zero?

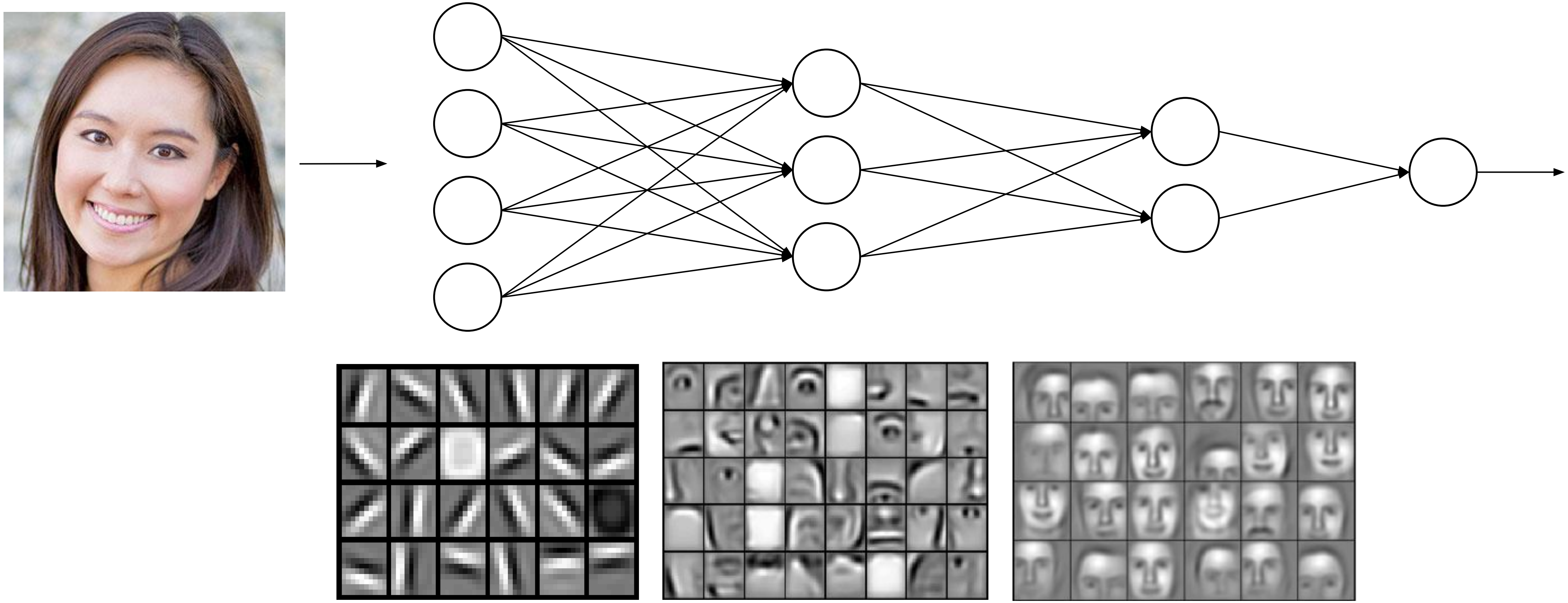
**Zero Initialization** In some cases, learning does not occur. For example, when using activation functions such as tanh and ReLU, where a(0)=0, all weights will be initialized to zero, leading to zero outputs for any input value. Consequently, the gradients will be zeros, resulting in no learning at all.

# Random initialization

**Constant Initialization** Constant initialization does not break the symmetry within the neural network. When all weights are set to a constant value, all neurons in all layers perform the same calculation, yielding identical outputs and rendering the deep net meaningless.

**Random Initialization** Random initialization in neural networks aids in the symmetry-breaking process and enhances accuracy.

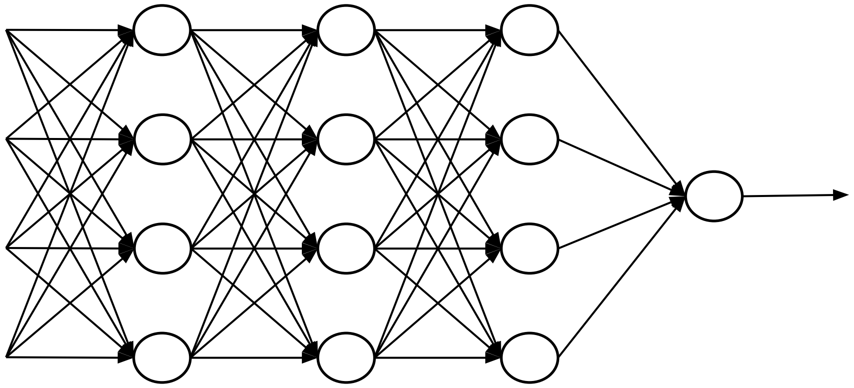
# Intuition about deep representation



# Circuit theory and deep learning

Informally: There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

# Forward and backward functions



# Questions?