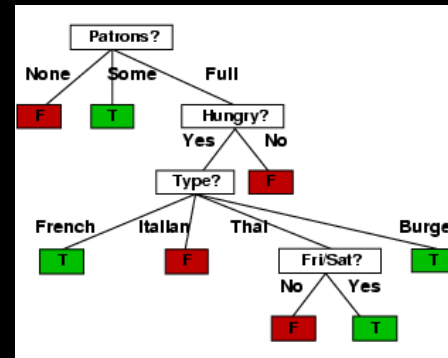
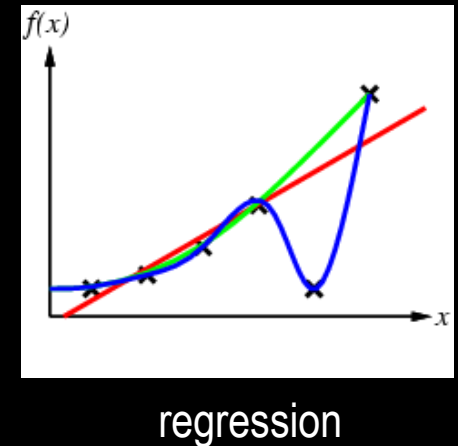
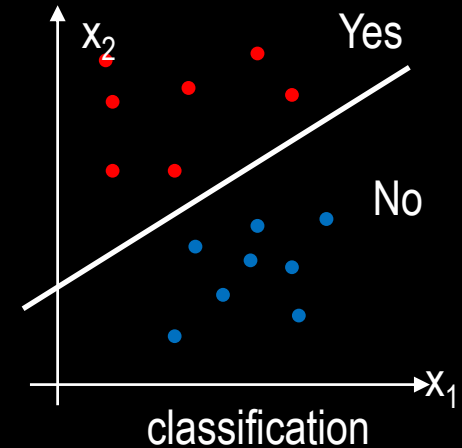


# KNN and ANN

---

## Last time...

- In supervised learning, given a set of training data with labels, need to learn a function that maps from data to label
- **Classification problem:** Label is a discrete variable
- **Regression problem:** Label is continuous
- Decision trees: an easy way to do classification
  - Difficult for continuous data



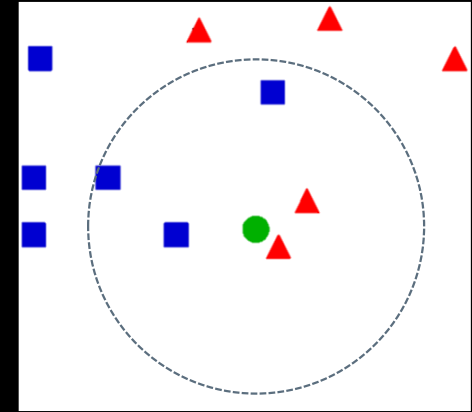
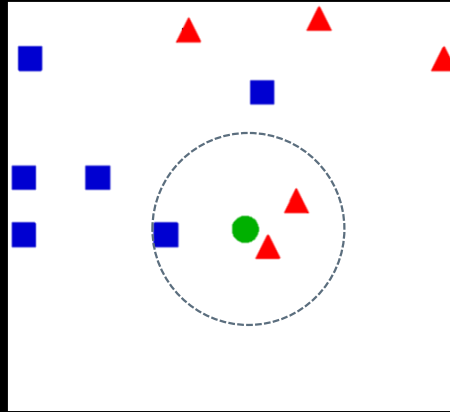
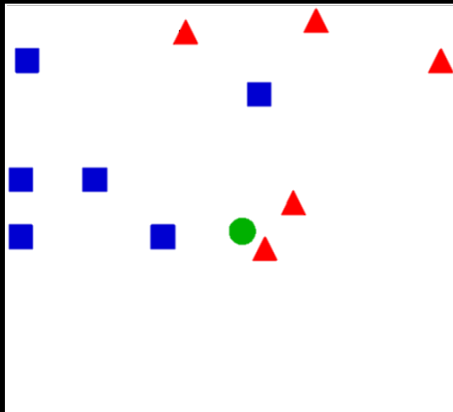
What other methods can we use?

# Outline

- K-Nearest Neighbors (KNN)
- Artificial Neural Networks (ANN)

# K-Nearest Neighbors

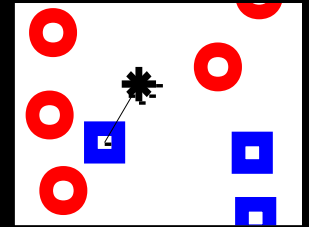
- What class do we assign to the green sample?



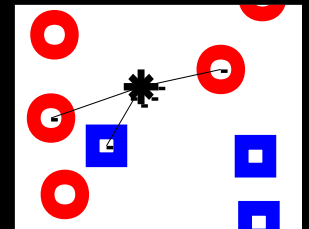
# K-Nearest Neighbors for Classification

- 1-NN:
  - For a given query point  $q$ , assign the class of the nearest neighbour.
- K-NN
  - Compute the  $k$  nearest neighbours and assign the class by majority vote.

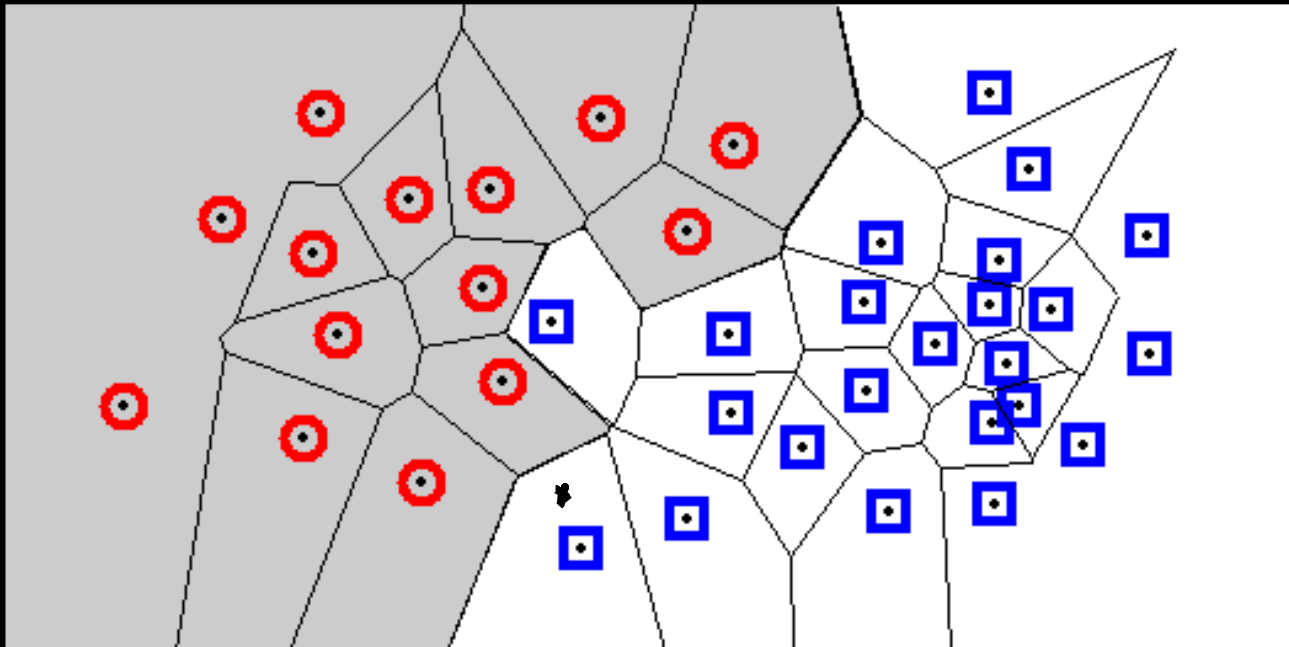
$k = 1$



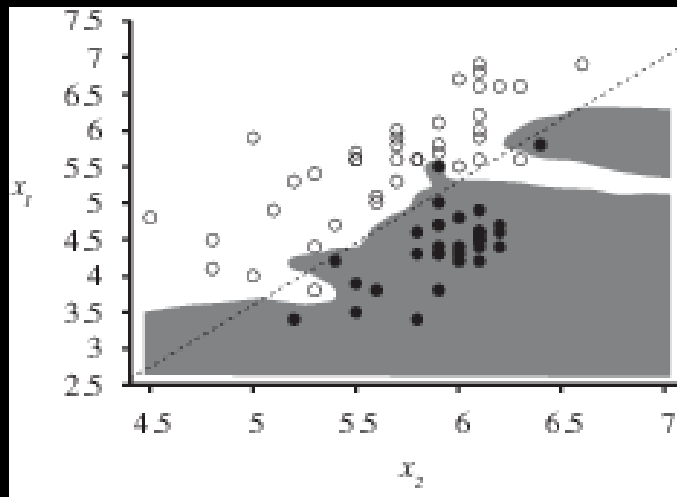
$k = 3$



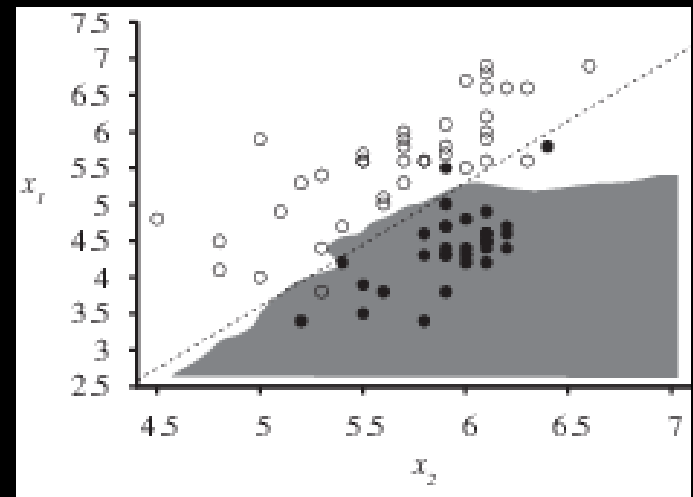
# Decision Regions for 1-NN Classification



## Effect of $k$



$k = 1$



$k = 5$

# K-Nearest Neighbors: Distance metric

- Euclidean Distance:

$$D(x_1, x_2) = \sqrt{\sum_i (x_{1,i} - x_{2,i})^2}$$

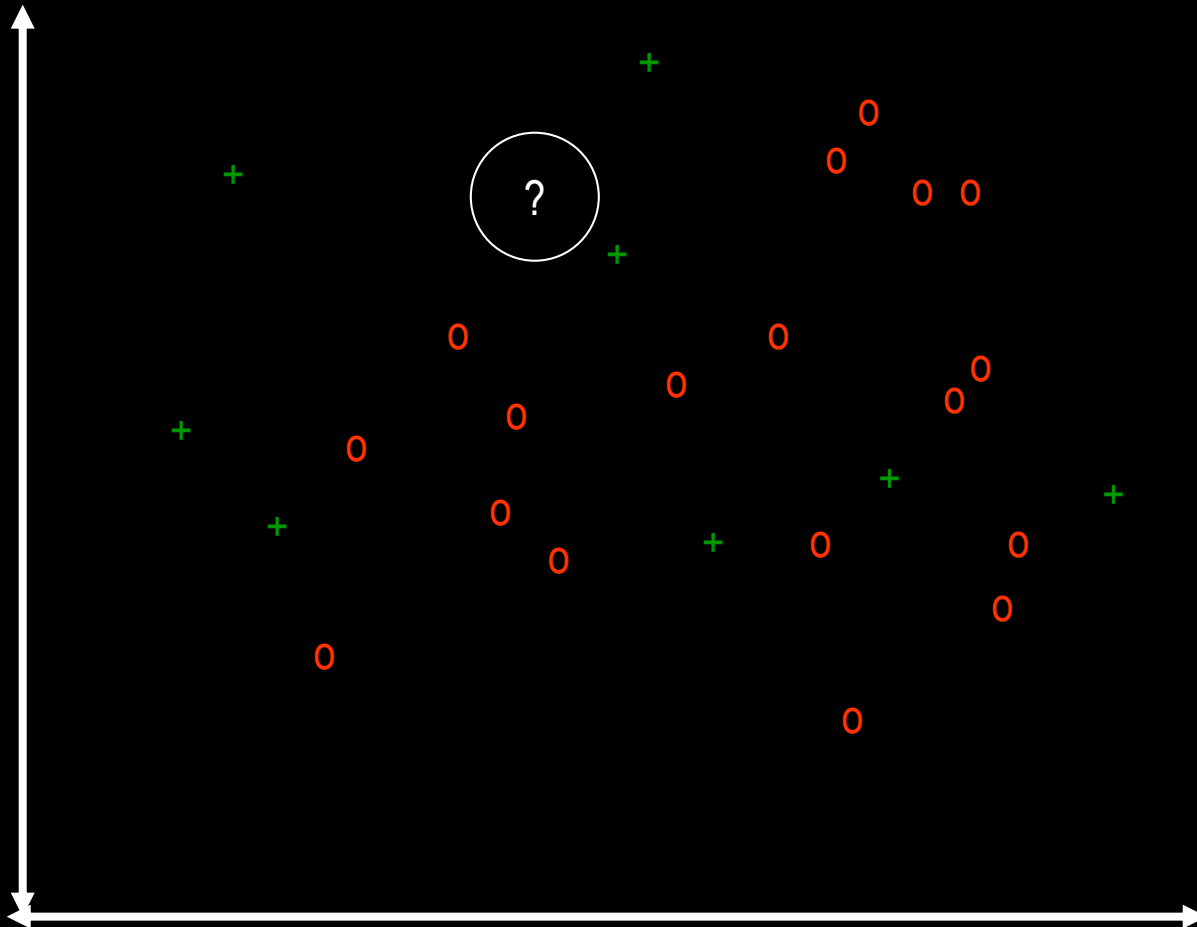
- Weighted Euclidian Distance:

$$D(x_1, x_2) = \sqrt{\sum_i w_i (x_{1,i} - x_{2,i})^2}$$

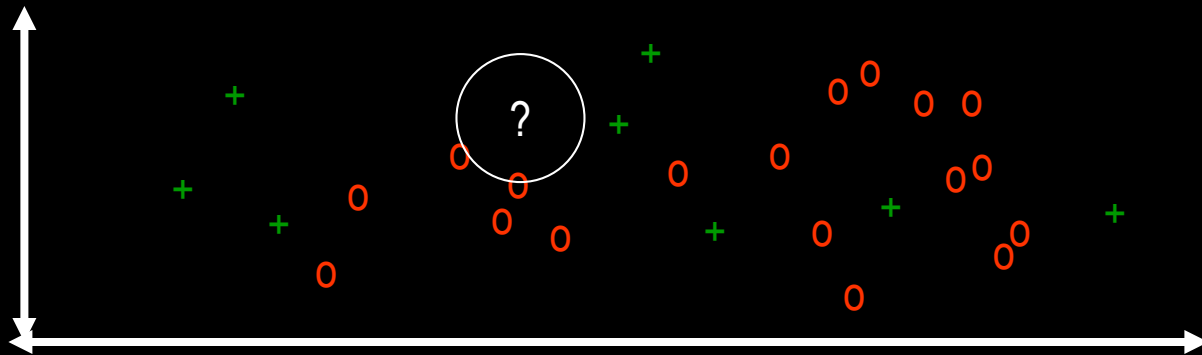
Where  $i$  indexes over each dimension of the data.



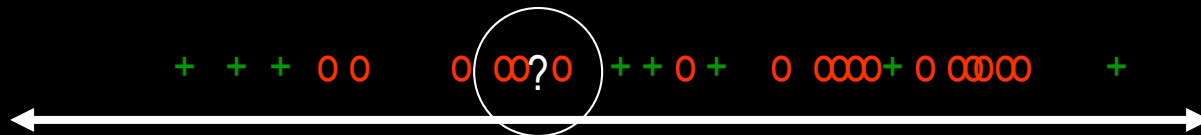
# Weighting the Distance to Remove Irrelevant Features



# Weighting the Distance to Remove Irrelevant Features

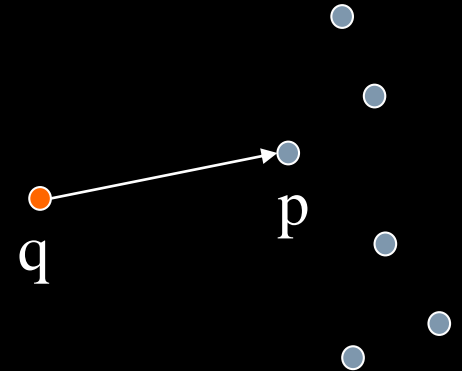


# Weighting the Distance to Remove Irrelevant Features



# Nearest Neighbors Search

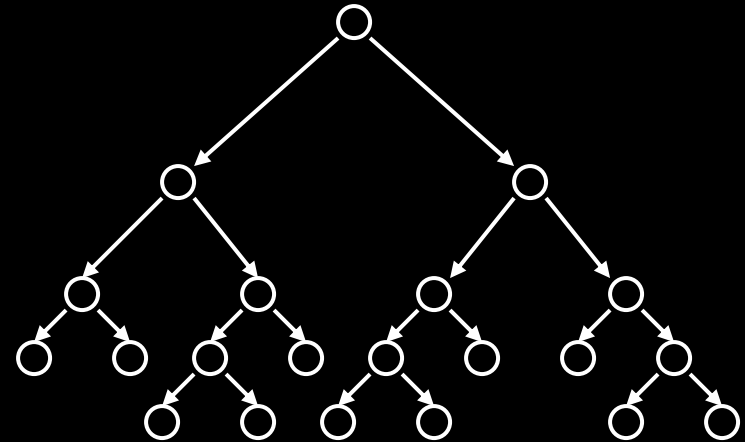
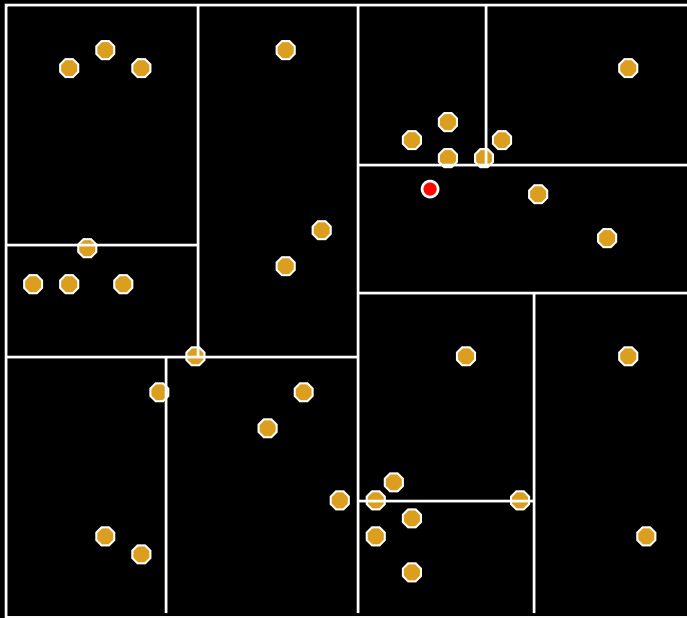
- Let  $P$  be a set of  $n$  training points
- Given a query point  $q$ , find the nearest neighbor  $p$  of  $q$  in  $P$ .
- Naïve approach
  - Compute the distance from the query point to every other point in the database, keeping track of the "best so far".
  - Running time is  $O(n)$ .
- Data Structure approach
  - Construct a data structure which makes this search more efficient



# kd-trees

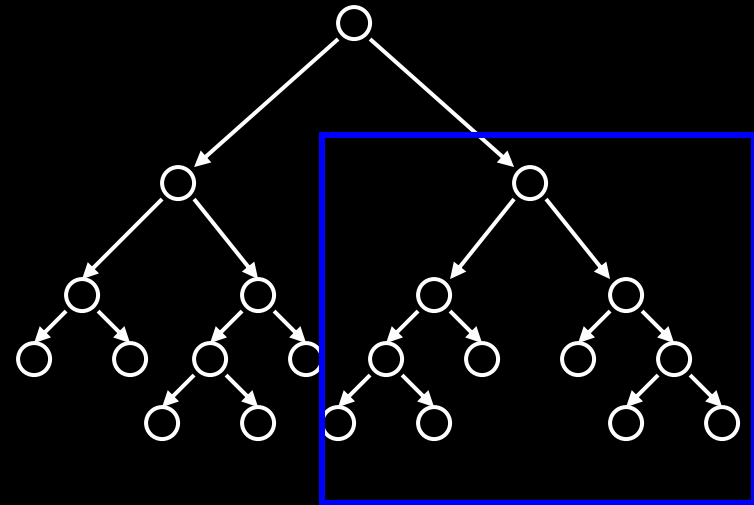
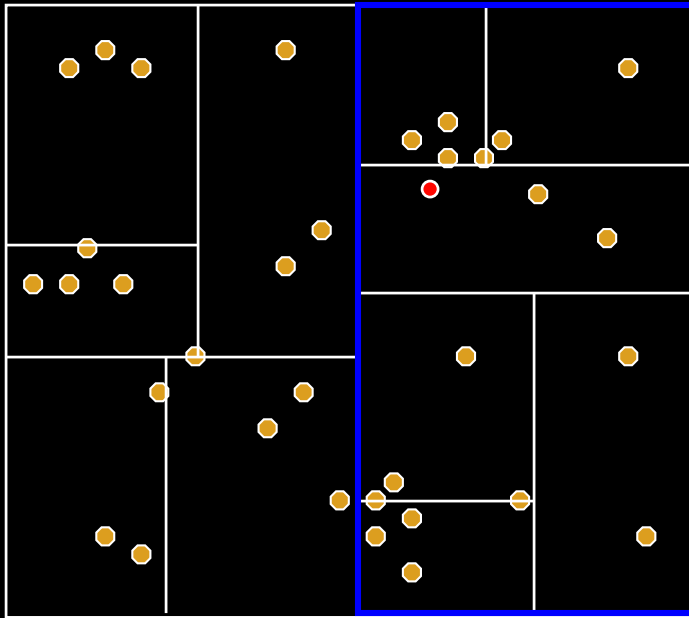
- Data structure with one-dimensional splits in the data
  - remember we used this for nearest-neighbor queries in sampling-based planning
- Algorithm
  - Choose x or y coordinate (alternate between them)
  - Choose the median of the coordinate for the points in this cell
    - this defines a horizontal or vertical line
    - can use other variants instead of median to select split line
  - Recurse on both sides until there is only one point left, which is stored as a leaf
- We get a binary tree
  - Size  $O(n)$
  - Construction time  $O(n \log n)$
  - Depth  $O(\log n)$

# Nearest Neighbor with KD Trees



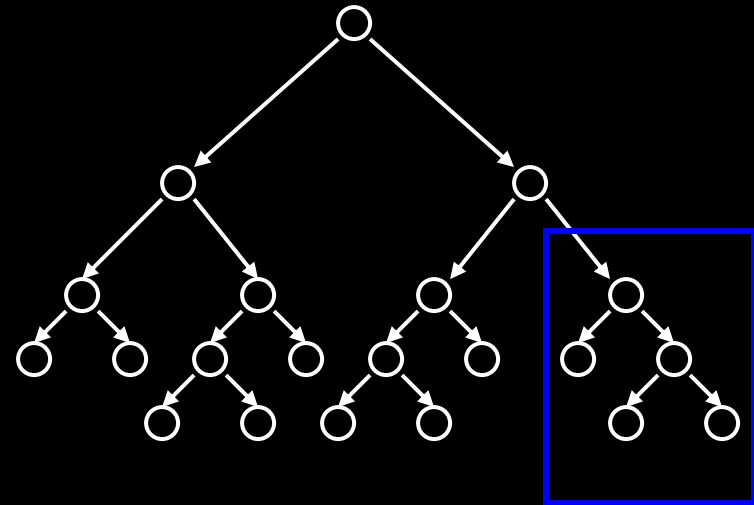
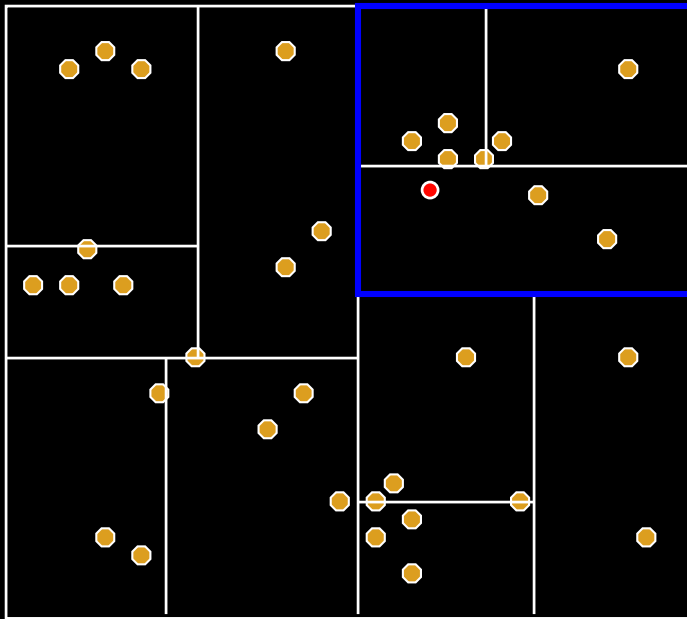
We traverse the tree looking for the nearest neighbor of the query point

# Nearest Neighbor with KD Trees



Examine nearby points first: Explore the branch of the tree that is closest to the query point first.

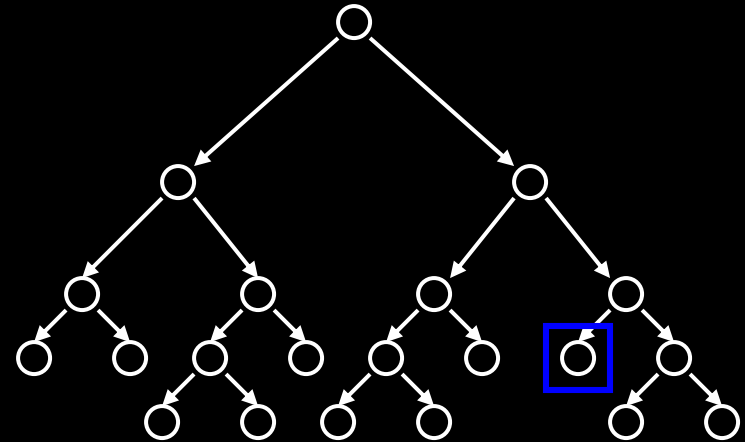
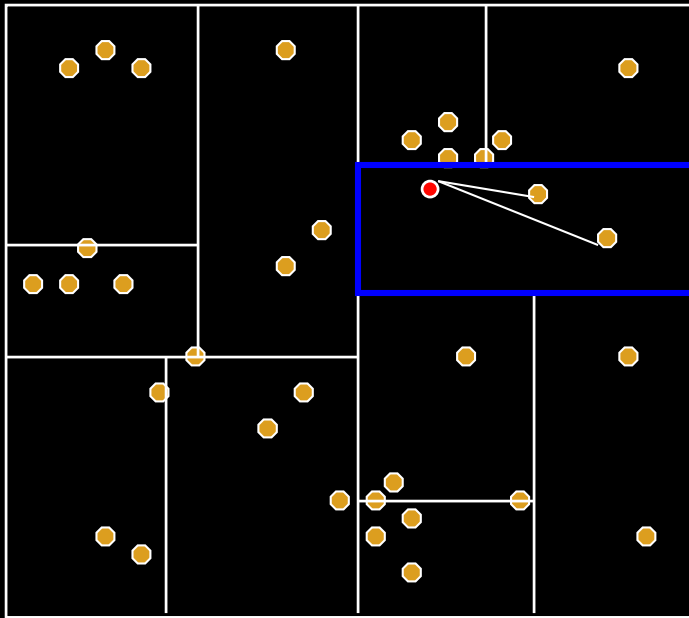
# Nearest Neighbor with KD Trees



Examine nearby points first: Explore the branch of the tree that is closest to the query point first.

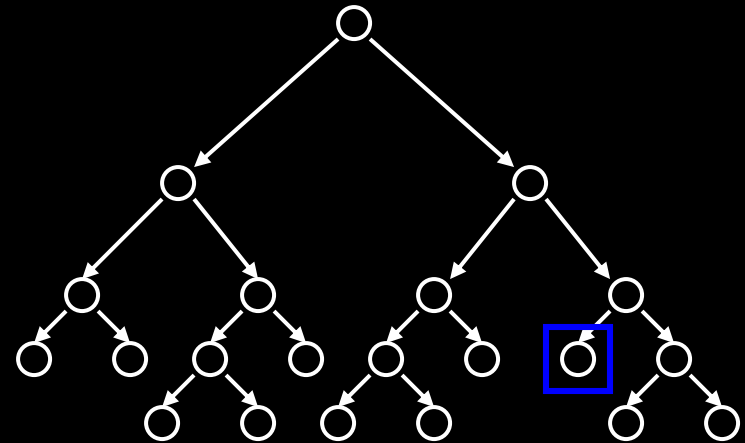
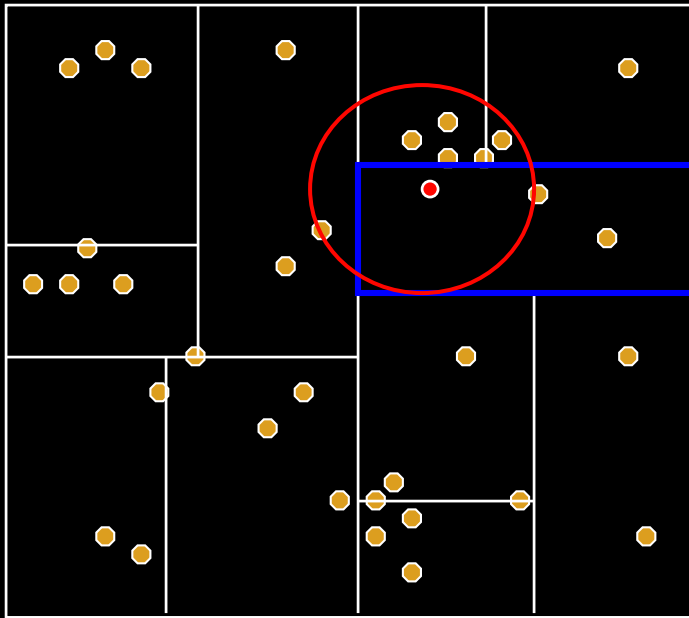


# Nearest Neighbor with KD Trees



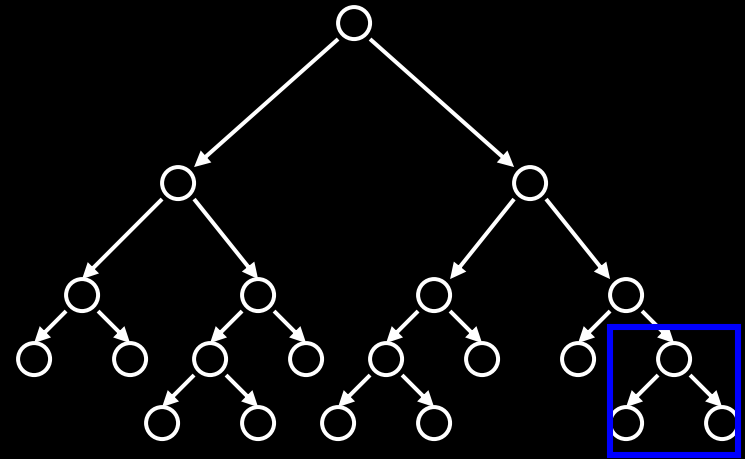
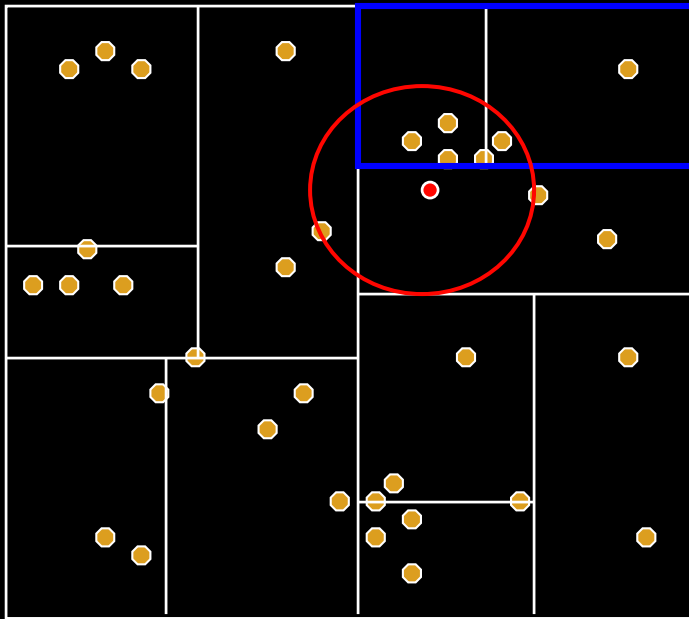
When we reach a leaf node: compute the distance to each point in the node.

# Nearest Neighbor with KD Trees



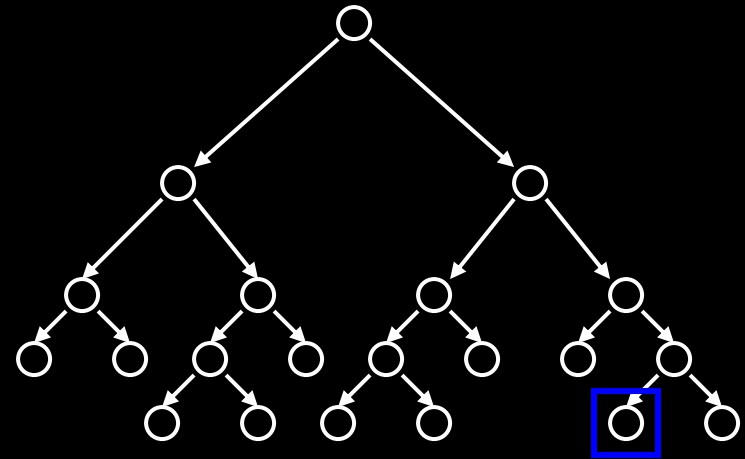
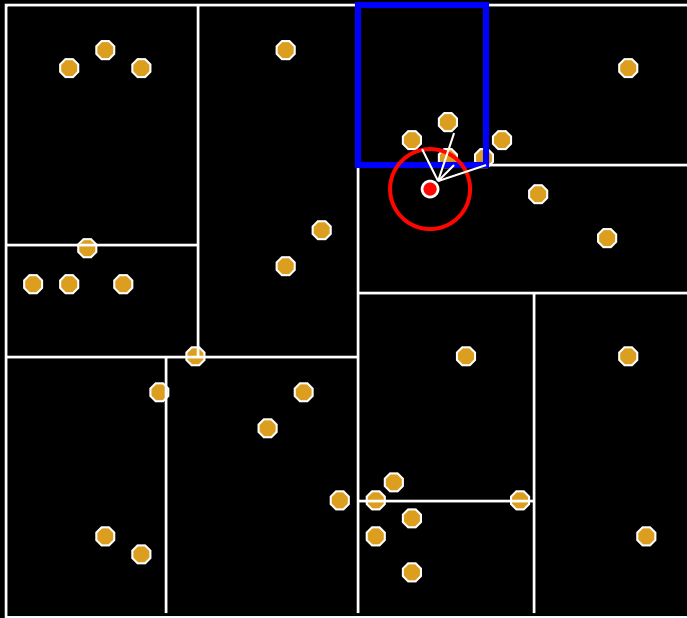
When we reach a leaf node: compute the distance to each point in the node.

# Nearest Neighbor with KD Trees



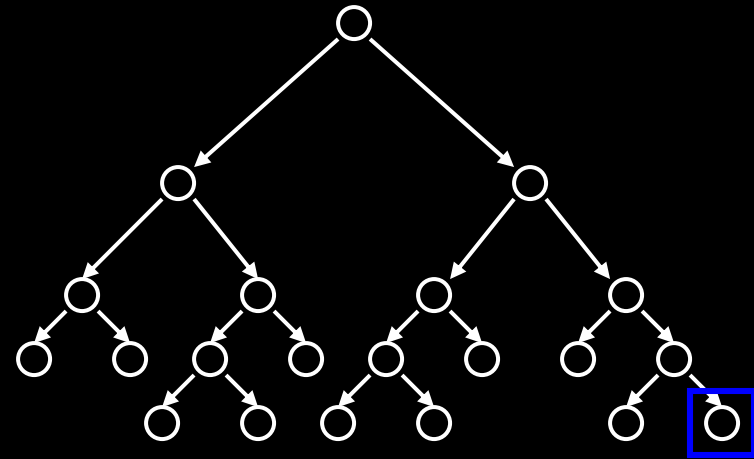
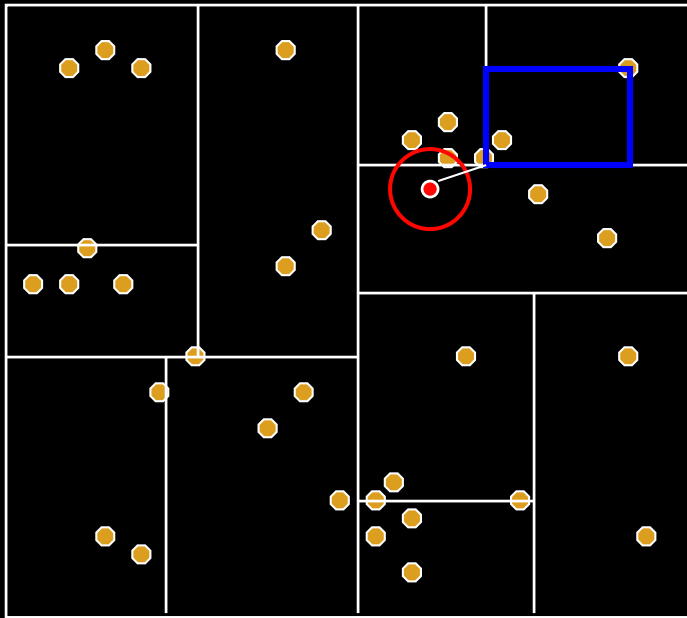
Then we can backtrack and try the other branch at each node visited.

# Nearest Neighbor with KD Trees



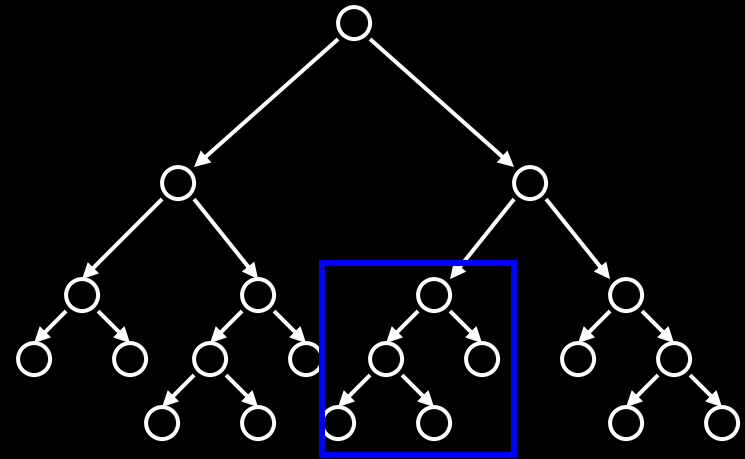
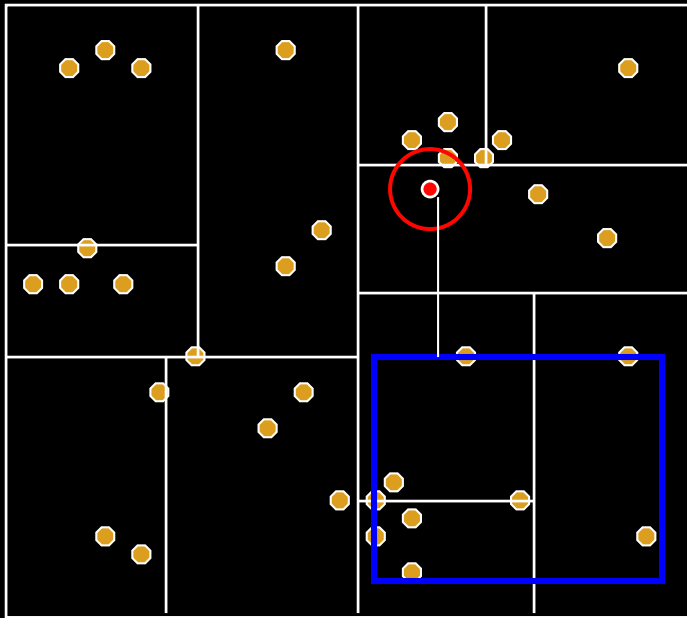
Each time a new closest node is found, we can update the distance bounds.

# Nearest Neighbor with KD Trees



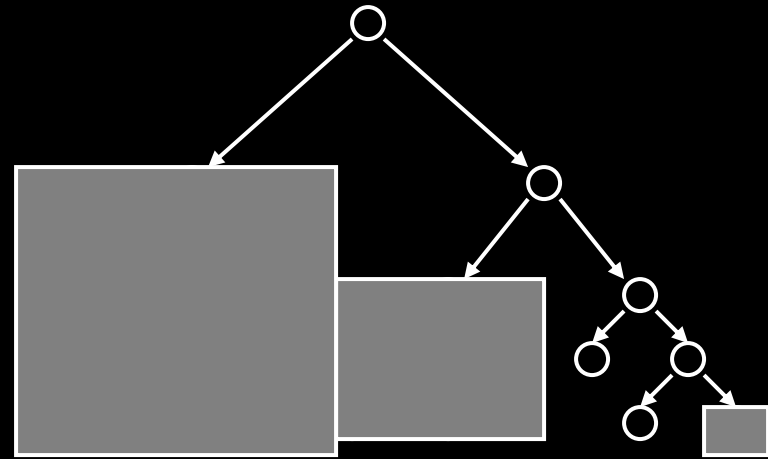
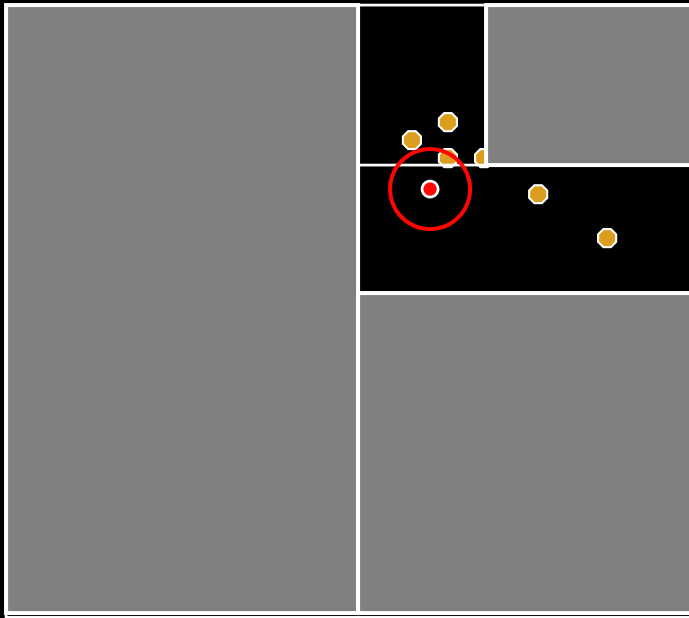
Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.

# Nearest Neighbor with KD Trees



Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.

# Nearest Neighbor with KD Trees



Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.

# Summary of K-Nearest Neighbor

- Stores all training data in memory – large space requirement
- Can improve query time by representing the data within a kd-tree
- Kd-trees are only efficient when there are many more examples than dimensions, preferably at least  $2^n$  examples for  $n$  dimensions



# Artificial Neural Networks

---

# Biological inspirations

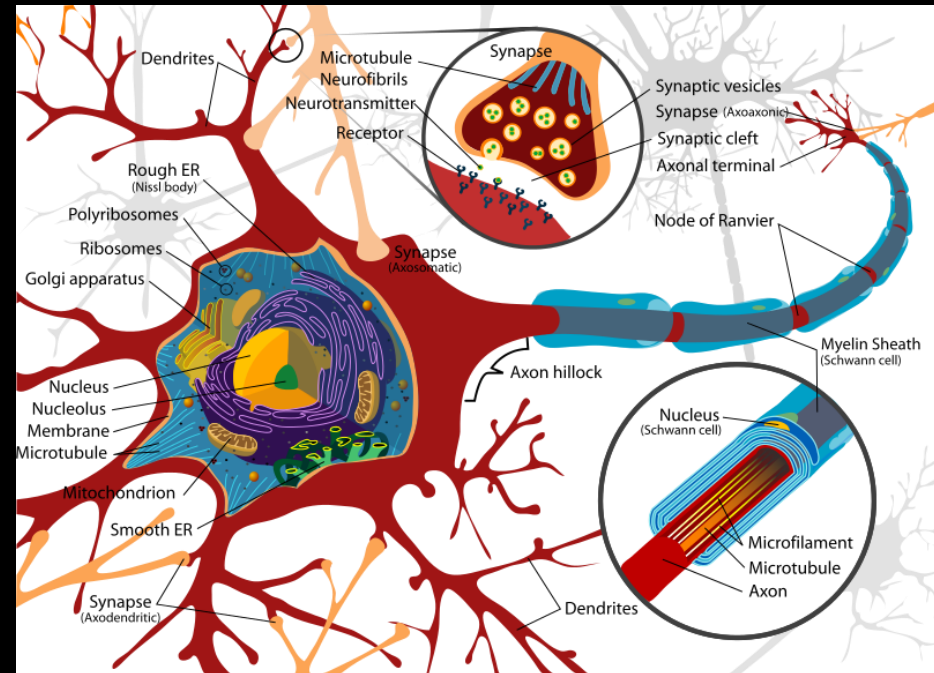
- The human brain contains about 10 billion nerve cells (neurons)
- Each neuron is connected to the others through 10000 synapses
- Properties of the brain
  - It can learn, reorganize itself from experience
  - It adapts to the environment
  - It is robust and fault tolerant
- Sounds like something we'd want for a robot!

# ANN Outline

- Artificial Neural Network (ANN) definition
- Learning ANN parameters
- Deep Learning

# The neuron

- Neurons often exhibit all-or-nothing property: Either they fire or they don't
- Greater intensity of stimulation does not produce a stronger signal magnitude
  - But can produce a higher-frequency firing
- Some neurons can get “bored”; i.e. firing decreases or stops with steady stimulus
  - Example: skin with steady contact
- Much is still unknown about how neurons communicate



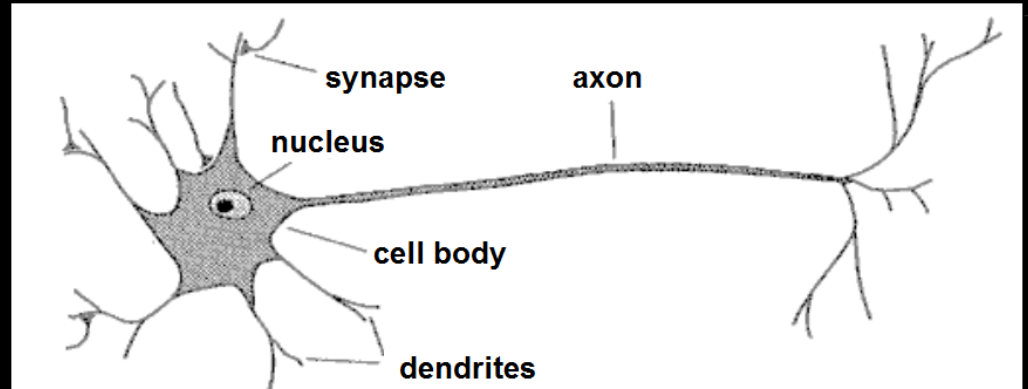
Anatomy of the Neuron



Fitting neuron models to spike trains

[Rossant et al., Frontiers in Neuroscience, 2011]

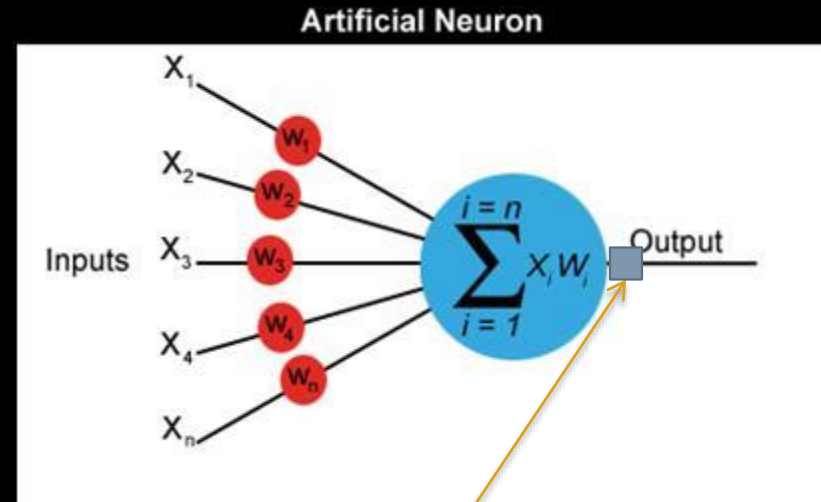
# The neuron (simplified model)



- A neuron has
  - Inputs (dendrites)
  - A branching output (the axon)
- The information moves from the dendrites to the axon via the cell body
- The cell body aggregates the input signals and *fires* – generates a signal through the axon – if the result is greater than some threshold

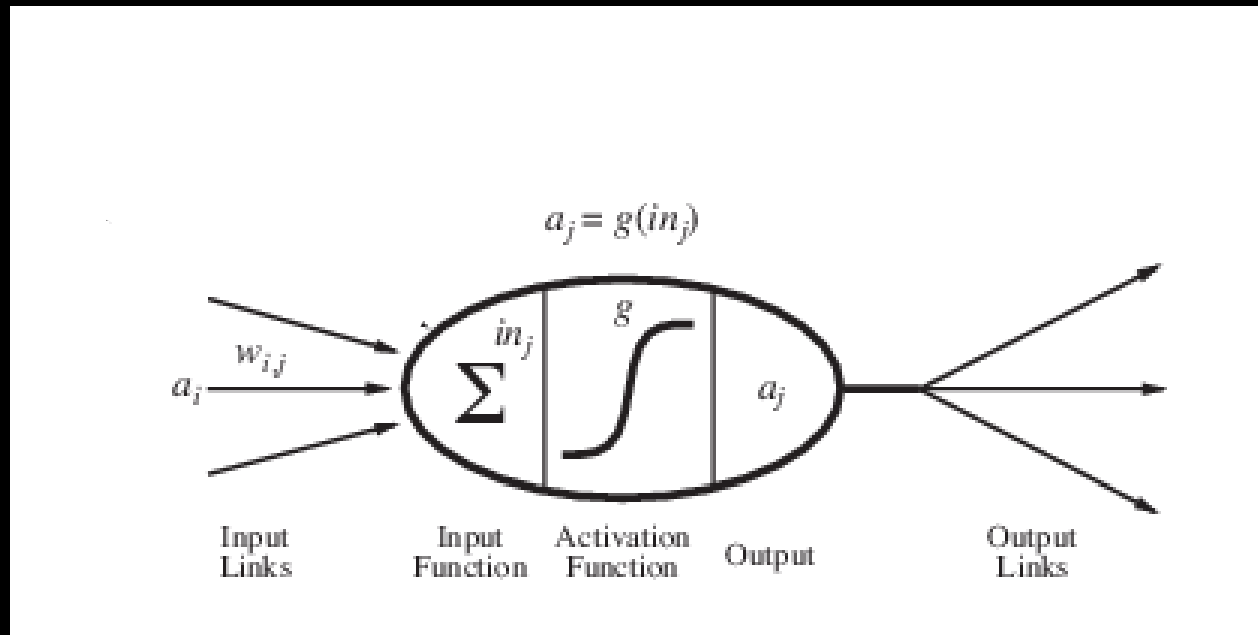
# An Artificial Neuron

- $w_i$  - weights
- $x_i$  - inputs
- Activation function: Non-linear, parameterized function with restricted output range



Activation Function

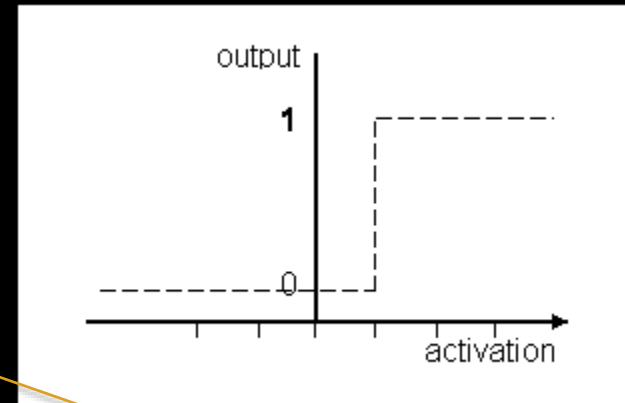
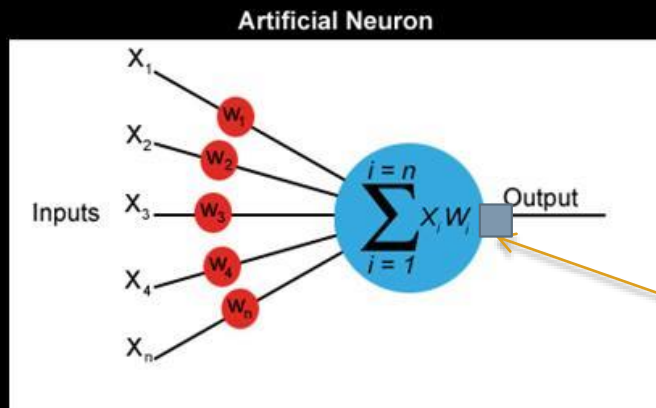
# Same Idea using the Notation in the Book



$$in_j = \sum_{i=0}^n w_{i,j} a_i$$
$$a_j = g(in_j) = g\left(\sum_{i=0}^n w_{i,j} a_i\right)$$

# The Output of a Neuron

- A neuron where the activation is a step function is called a **perceptron**

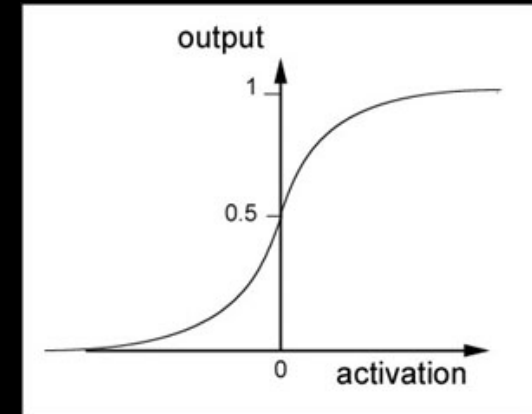
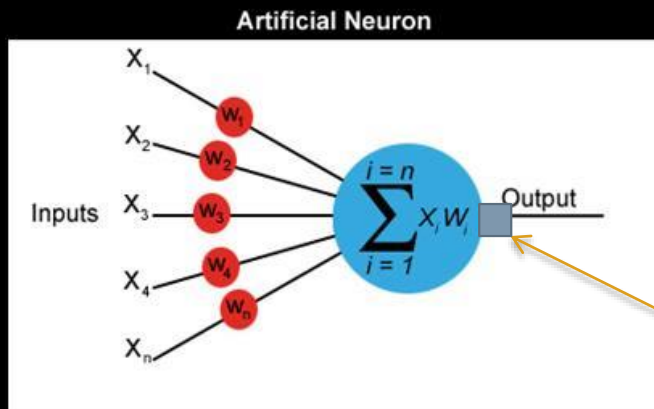


Step Function



# The Output of a Neuron

- Other possibilities, such as the **sigmoid function** for continuous output.

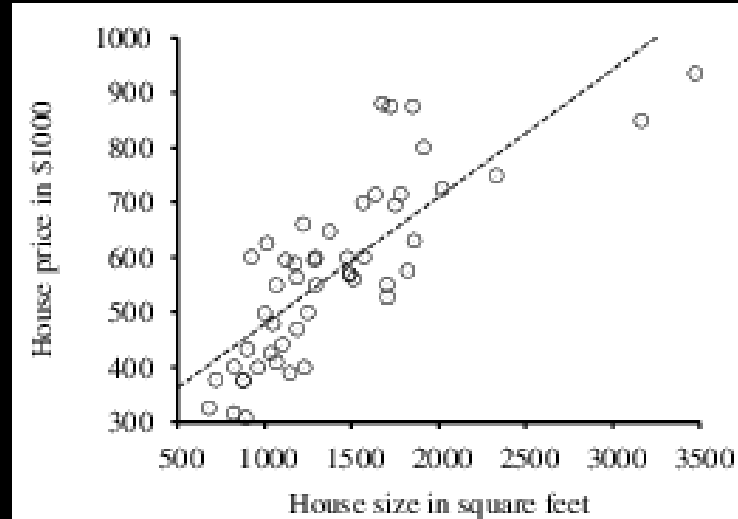


Sigmoid Function: 
$$\frac{1}{1 + e^{-\frac{in_j}{p}}}$$

- $in_j = \sum_{i=0}^n x_i w_i$  is the activation of the neuron
- $p$  is a parameter which controls the shape of the curve (usually 1)

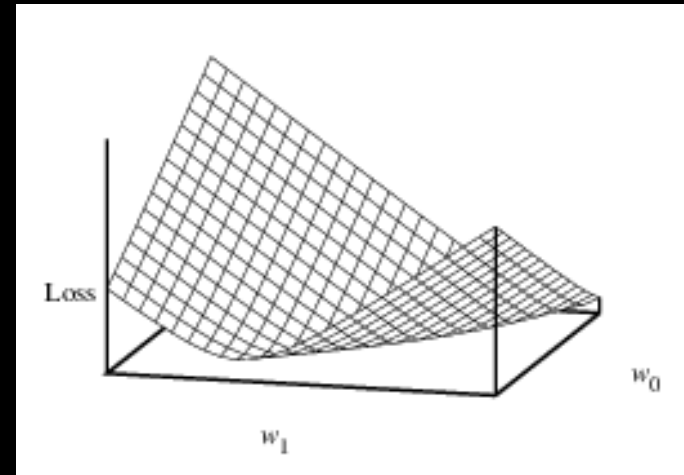
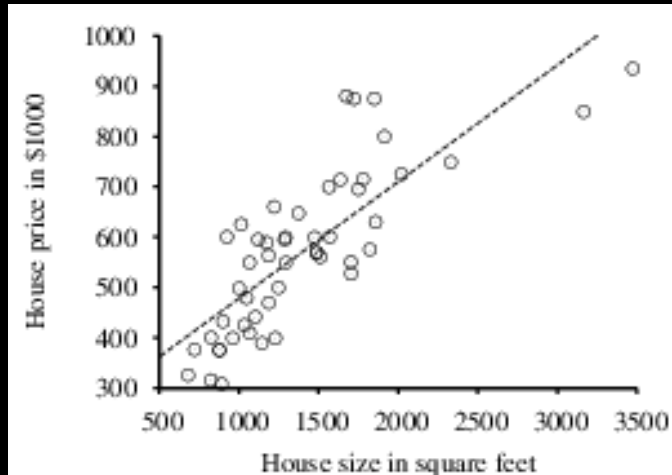
# Example: Linear Regression using a Perceptron

- Linear regression:



- Find a linear function  $f(x) = w_1x + w_0$  that best predicts the continuous-valued output.

# Linear Regression As an Optimization Problem

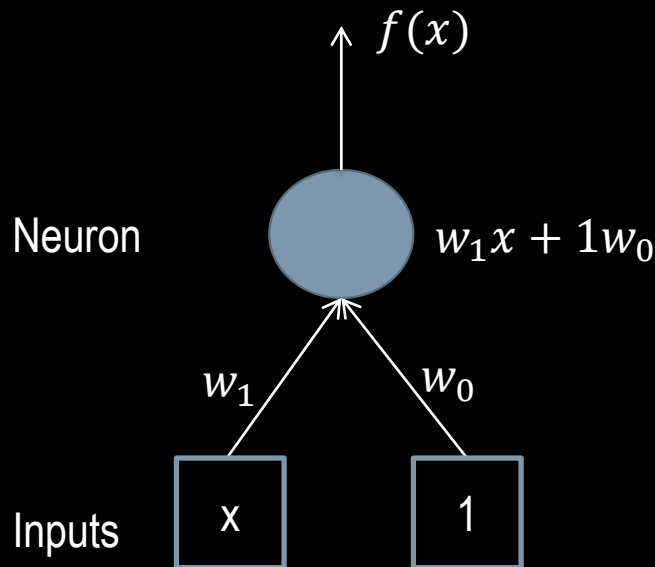


- Finding the optimal weights could be solved through:
  - Least-squares
  - Gradient descent
  - Simulated annealing
  - Genetic algorithms
  - ... and now Neural Networks

# Linear Regression Modeled as a Perceptron

- Ignoring the thresholding, we can get continuous valued output

$$f(x) = w_1x + w_0$$



# The Bias Term

- So far we have defined the output of a perceptron as controlled by a threshold

$$x_1w_1 + x_2w_2 + x_3w_3 \dots + x_nw_n \geq t$$

- But just like the weights, this threshold is a parameter that needs to be adjusted

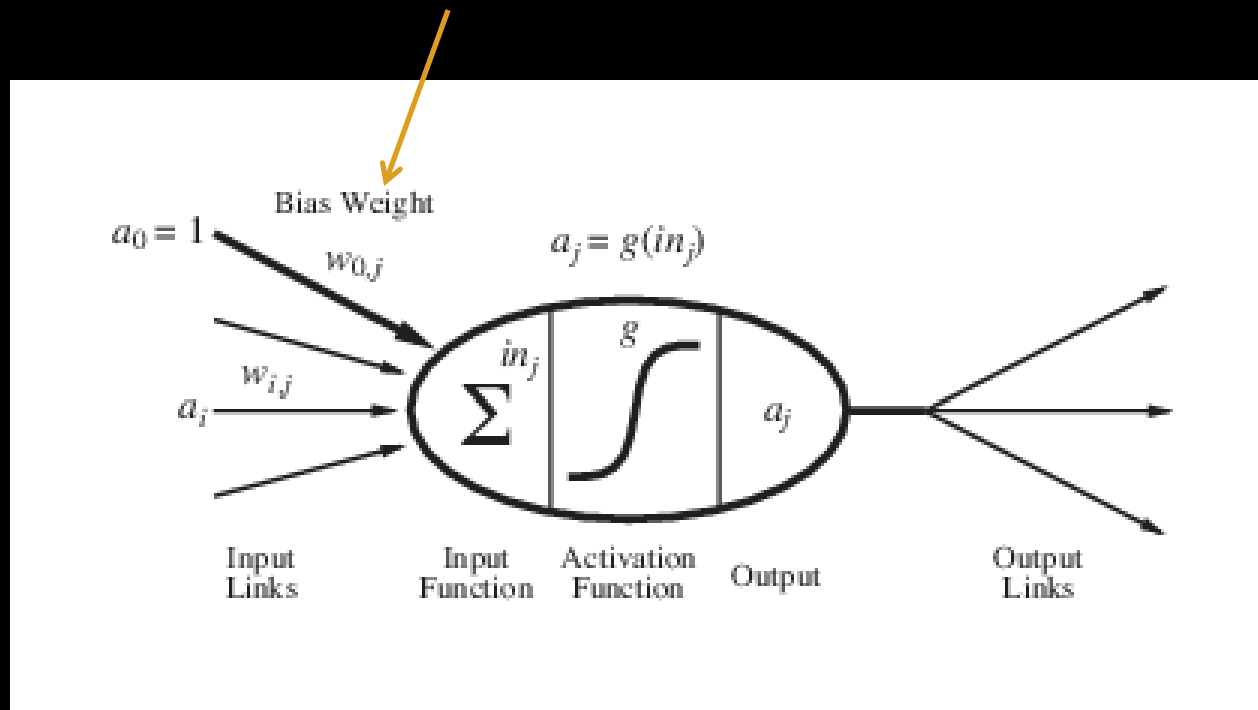
- Solution: make it another weight

$$x_1w_1 + x_2w_2 + x_3w_3 \dots + x_nw_n + (1)(w_0) \geq 0$$



The bias term.

# A Neuron with a Bias Term

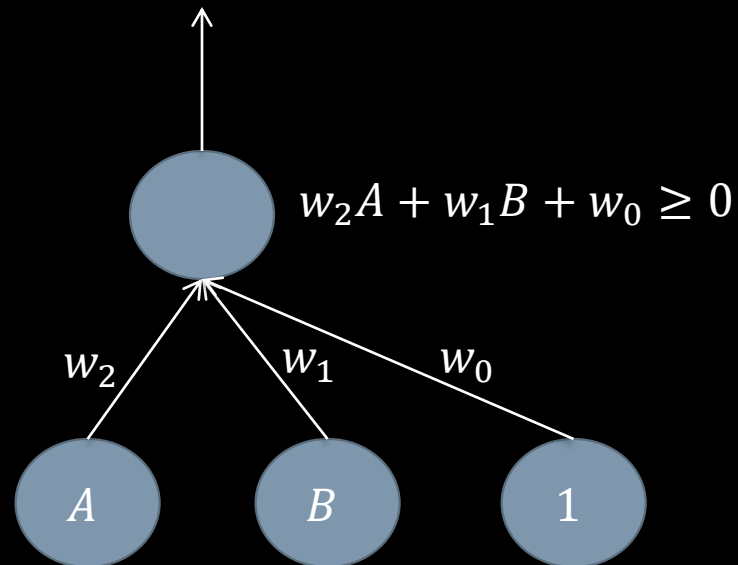


# Perceptron Example

- Assign weights to perform the logical OR operation.

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

**OR**



$$w_0 = ?$$

$$w_1 = ?$$

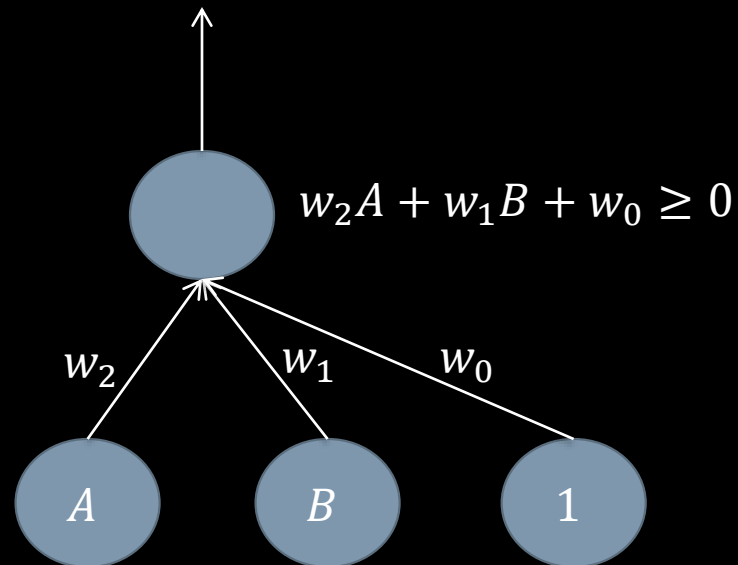
$$w_2 = ?$$

# Perceptron Example

- Assign weights to perform the logical OR operation.

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

**OR**



$$w_0 = -0.5$$

$$w_1 = 1$$

$$w_2 = 1$$



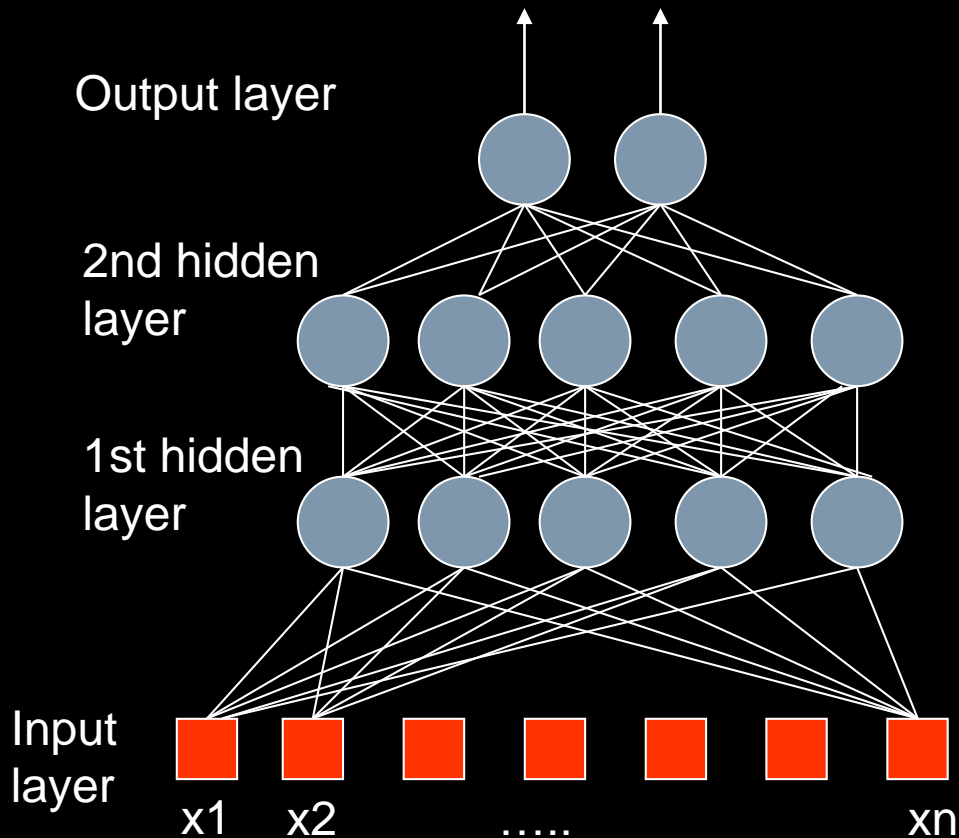
# Artificial Neural Network (ANN)

- A group of connected artificial neurons
  - Many variants
- A mathematical model to solve engineering problems
  - Standard neuron models are too simple to model biology!
- Common ANN Tasks:
  - Classification
  - Regression
  - Estimation

# General ANN Structure

- Finite number of continuous/discrete inputs
- Zero or more hidden layers
- One or more continuous/discrete outputs
- All nodes at the hidden and output layers contain a bias term

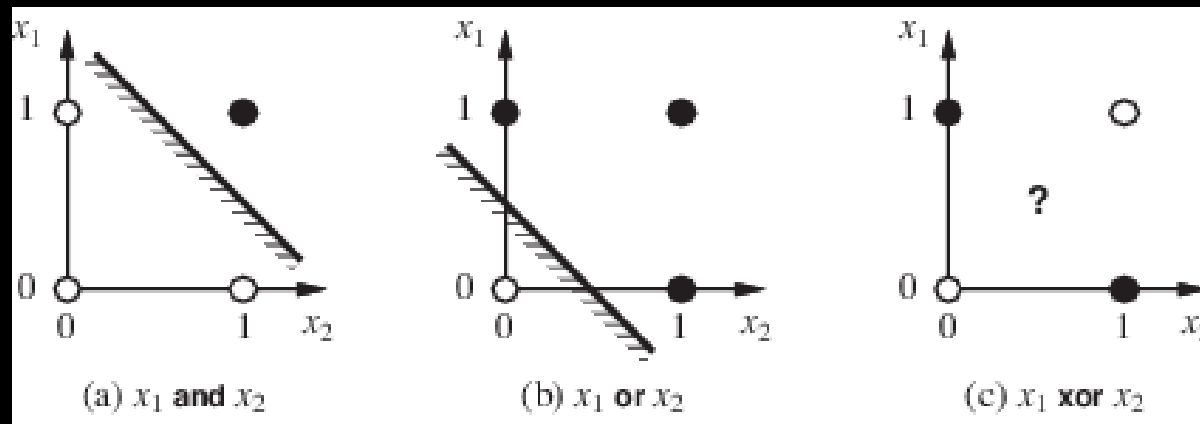
# Common type of ANN: Feed-Forward Neural Networks



- The information is propagated from the inputs to the outputs
- There are **no cycles** between outputs and inputs
  - the state of the system is not preserved from one iteration to another

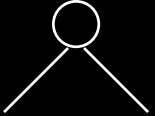
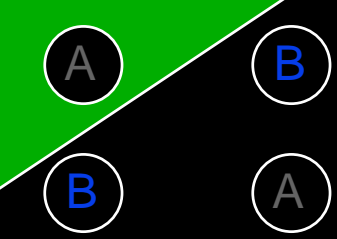
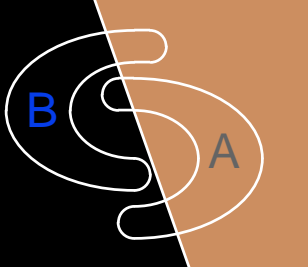
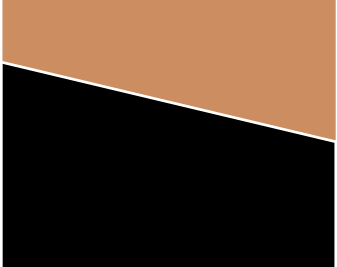
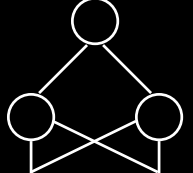
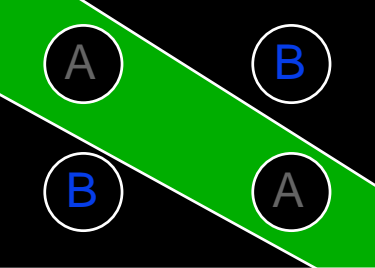
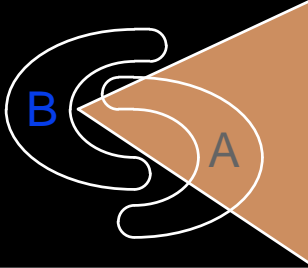
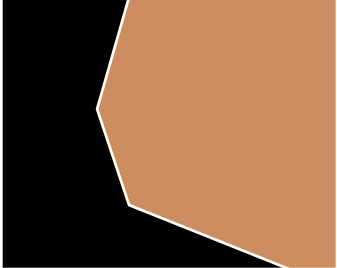
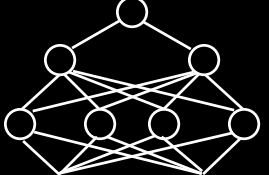
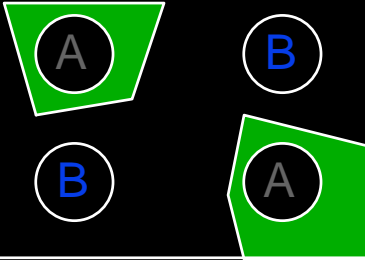
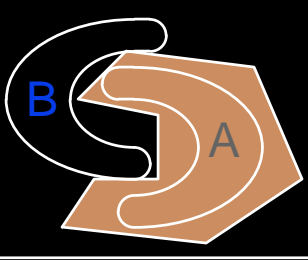
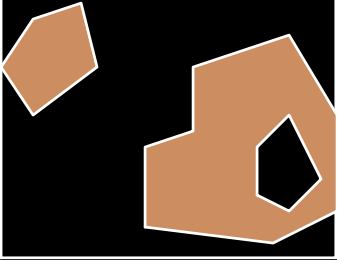
# Linear Separability

- A single perceptron can classify any input that is linearly separable



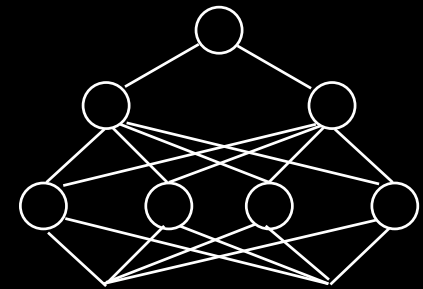
- For more complex problems we need a more complex model

# Different Non-Linearly Separable Problems

<i>Structure</i>	<i>Types of Decision Regions</i>	<i>Exclusive-OR Problem</i>	<i>Classes with Meshed regions</i>	<i>Most General Region Shapes</i>
<i>Single-Layer</i> 	<i>Half Plane Bounded By Hyperplane</i>			
<i>Two-Layer</i> 	<i>Convex Open Or Closed Regions</i>			
<i>Three-Layer</i> 	<i>Arbitrary (Complexity Limited by No. of Nodes)</i>			

# How many layers should we use?

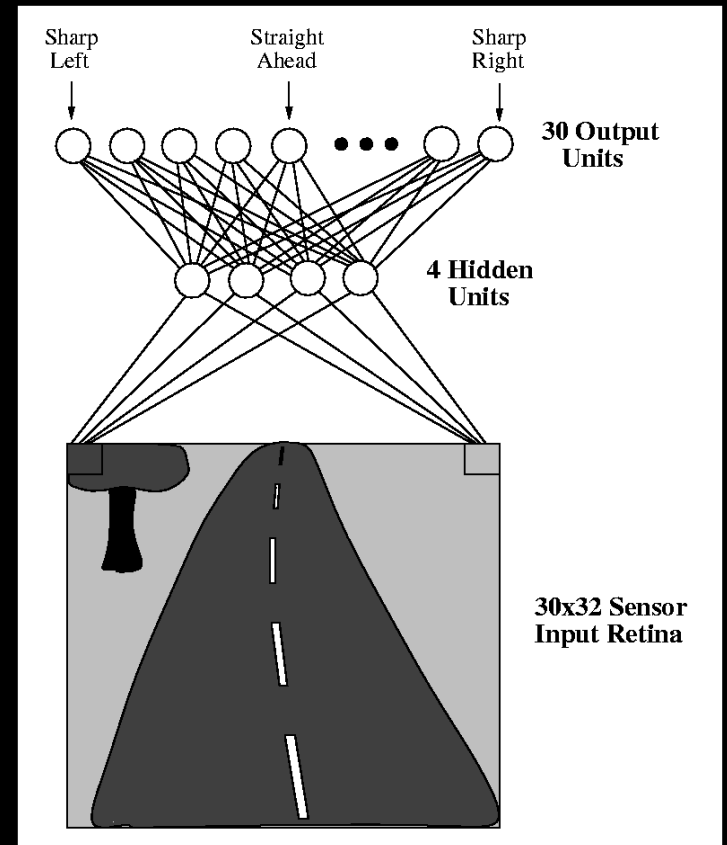
- It was proved that feed-forward ANNs with a single hidden layer (i.e. ANNs with input, hidden, and output layer) *can approximate any continuous real-valued function*
  - Proved independently in three papers: Cybenko (1989), Hornik et al. (1989), and Funahashi (1989)
- But proof **does NOT** tell us
  - How many neurons to use
  - How to set the weights



*Three-Layer Feed-Forward ANN*

# ALVINN

## Neural Network controlled AGV (1994)



$$961 * 4 + 5 * 30 = 3994 \text{ weights}$$

# How I built a neural network controlled self-driving (RC) car!

Tweet < 275

+1 96 people +1'd this

**Updated January 2nd:** the source code is now open source and available on github.

Recently, I have been refreshing my knowledge of Machine Learning by taking [Andrew Ng's](#) excellent Stanford [Machine Learning course](#) online. The lecture module on [Neural Networks](#) ends with an intriguing motivating [video](#) of the [ALVINN](#) autonomous car driving itself along normal roads at [CMU](#) in the mid 90s.

I was inspired by this video to see what I could build myself over the course of a weekend. From a [previous project](#) I already had a [cheap radio controlled car](#) which I set about trying to control.



<http://blog.davidsingleton.org/nnrccar>



# Learning ANN Weights

---

# Learning weights

- Want to learn a set of weights that allows ANN to accomplish a given task
- The Learning process (supervised)
  1. Present the network a number of inputs and their corresponding outputs
  2. See how closely the actual outputs match the desired ones
  3. Modify the parameters to better approximate the desired outputs
  4. Repeat 1-3 until convergence

# Single **Perceptron** Learning Algorithm

1. Initialize the weights to some random values (or 0)
2. For each sample  $(x_j, y_j)$  in the training set
  1. Calculate the current output of the perceptron,  $h_{x_j}$
  2. Update the weights

$$w_i = w_i + \alpha (y_j - h_{x_j}) x_{j,i}$$

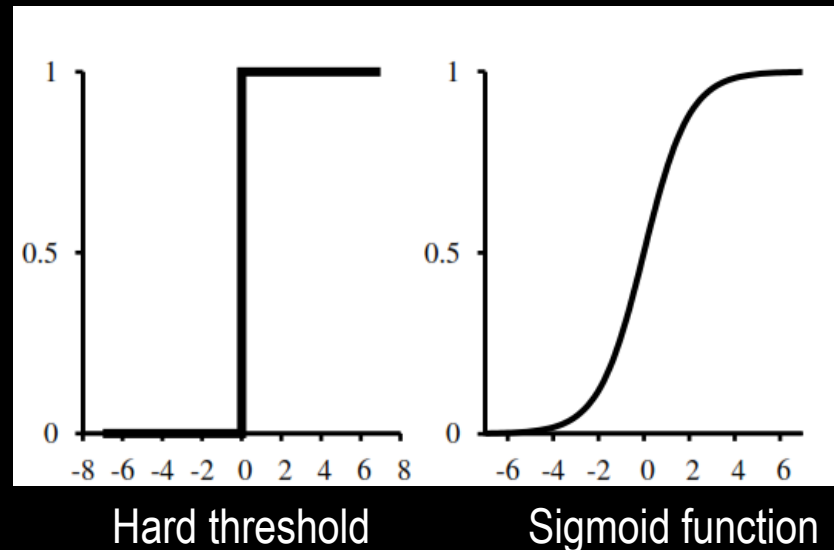
$\alpha$  is the learning rate, usually 0.1

3. Repeat until the error  $(y_j - h_{x_j})$  is smaller than some predefined threshold

# Building a network

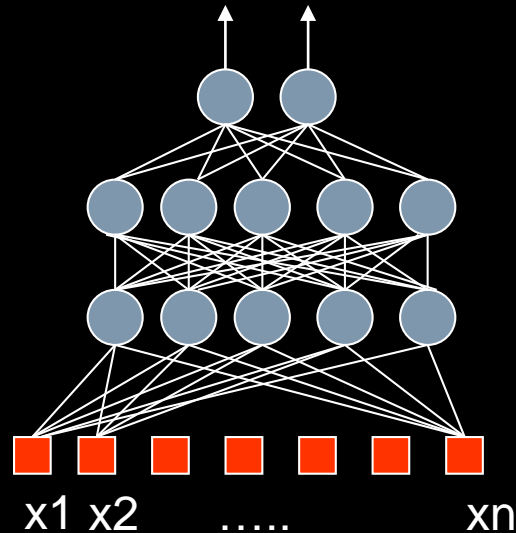
- Hard threshold creates problems: no derivative at boundary
- Instead, use sigmoid function (can use others, too):

$$h_w(\mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\frac{\mathbf{w} \cdot \mathbf{x}}{p}}} \leftarrow \text{Usually } p = 1$$



# Finding the Weights for a Network of Neurons

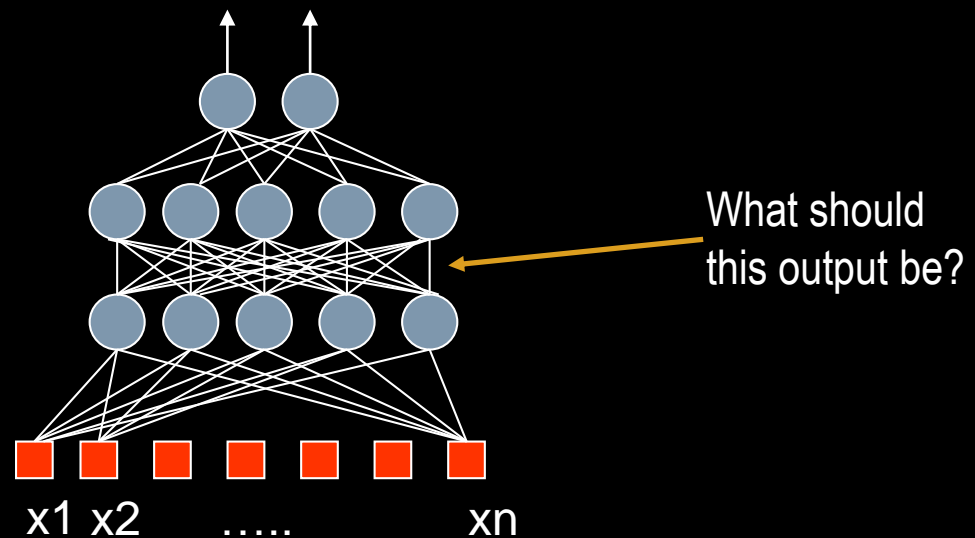
- Need to find the optimal weights for inputs to every neuron



- Could be solved by:
  - Simulated annealing
  - Genetic algorithms
  - Gradient descent ←

# Learning the Weights in a Neural Network

- Gradient descent works to find a local minimum, but it requires us to know the correct output of the node
- Also, we changed the threshold function to sigmoid



- How do we know the correct output of a given **hidden** node?

# Backpropagation Algorithm

- Gradient descent over *entire network's* set of weights
- Easily generalized to arbitrary directed graphs
- Will find a local optimum (not necessarily global error minimum)
  - in practice often works well (can be invoked multiple times with different initial weights)

# Intuition

- **General idea:** hidden nodes are “responsible” for some of the error at the output nodes it connects to
- The change in the hidden weights is proportional to the strength (magnitude) of the connection between the hidden node and the output node
- Derivative of sigmoid is easy to compute:

$$g'(\cdot) = g(\cdot)(1 - g(\cdot))$$



# Backpropagation Algorithm

1. Initialize the weights to some random values (or 0)
2. For each sample  $(x_j, y_j)$  in the training set

- a. Calculate the current output of the node,  $h_{x_j}$
- b. For each output node  $k$ , update the weights

$$\Delta_k = (h_{x_k})(1 - h_{x_k})(y_k - h_{x_k})$$

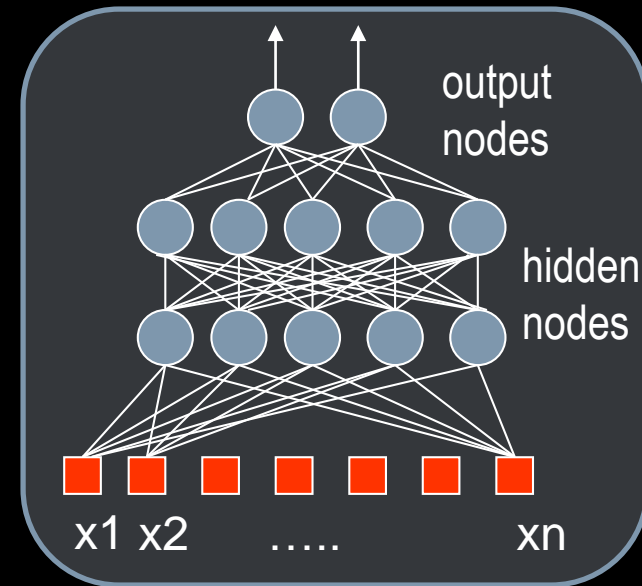
- c. For each hidden node  $j$ , update the weights

$$\Delta_j = (h_{x_j})(1 - h_{x_j}) \sum_k w_{j,k} \Delta_k$$

3. For all network weights do

$$w_{i,j} = w_{i,j} + \alpha \Delta_j x_i$$

4. Repeat until weights converge or desired accuracy is achieved



# Intuition

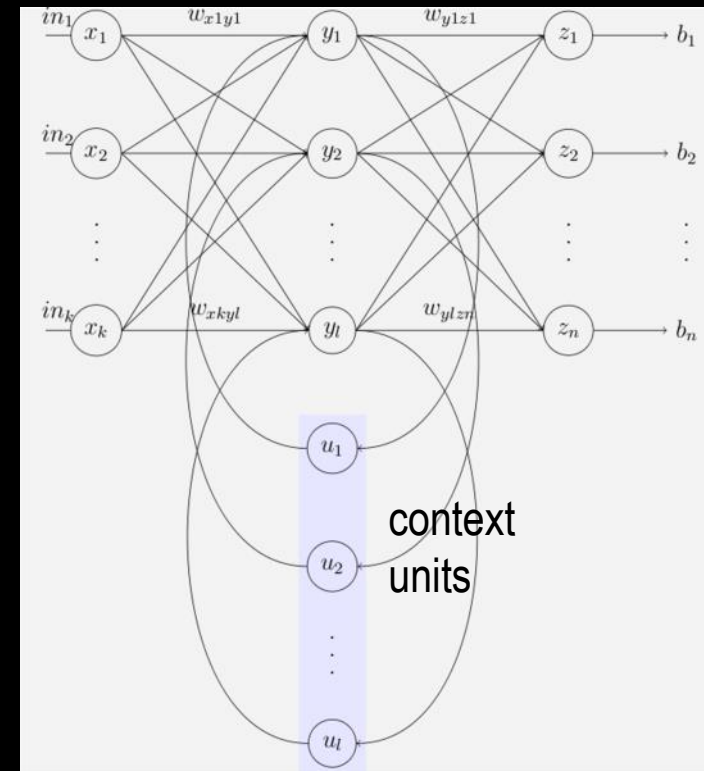
- When expanded, the update to the output nodes is almost the same as the perceptron rule

$$w_i = w_i + \alpha (y_j - h_{x_j}) \underbrace{(h_{x_j})(1 - h_{x_j})}_{\text{Derivative of sigmoid}} x_i$$

- Difference from perceptron rule is that the algorithm uses a sigmoid function instead of a step function
  - (full derivation on p. 726-727 in AI book)

# Beyond feed-forward: Recurrent Neural Networks

- ANNs by default have no “memory”, i.e. no concept of state
  - Difficult to do tasks like sequence prediction
- Recurrent neural networks allow cycles
- Can use them to
  - store previous state of hidden units (Elman SRNs)
  - store previous values of output units (Jordan SRNs)
- Many other RNN variants



Elman Simple Recurrent Network (SRN)

Context units maintain a copy of the previous values of the hidden units

Break

# Deep Learning

---

# A brief history of ANNs

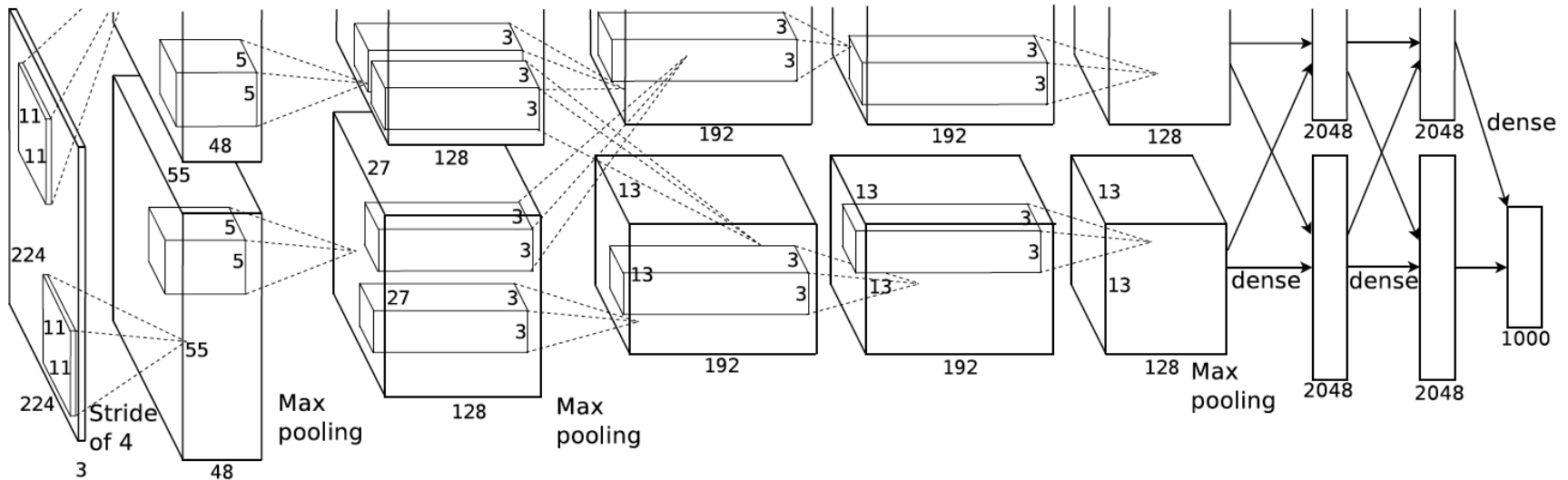
- 1940s and 1950s: Foundations of ANNs came about
  - Alan Turing proposed **unorganized machines** (randomly connected, binary neural networks) in 1948
- 1969: Research stalled when Marvin Minsky and Seymour Papert showed computational limitations of ANNs
  - Main problem was a lack of computation power
  - Single-layer network couldn't represent exclusive-or function
- 1975: Paul Werbos publishes backpropagation algorithm
  - Can now train multi-layer networks
- 1980s and early 1990s: ANNs become popular in ML research
  - Focus shifts to 3-layer networks (recall can prove that 3-layer network can approximate any continuous real-valued function)
- Mid-Late 1990s: ANNs overtaken in popularity by SVMs and simpler methods

# Recent history of ANNs

- Mid-late 2000s: Researchers move away from 3-layer ANNs toward
  - recurrent neural networks (allow cycles in the network)
  - deep feed-forward neural networks
- Recently: “Deep Learning” (i.e. using multi-layer feed-forward ANNs) is now its own sub-field
  - Very popular in ML
  - Deep learning algorithms have won a lot of ML competitions
  - Pretty much all modern computer vision methods use deep learning

# What is Deep Learning?

- The same algorithms as ANN, but more layers and more neurons!

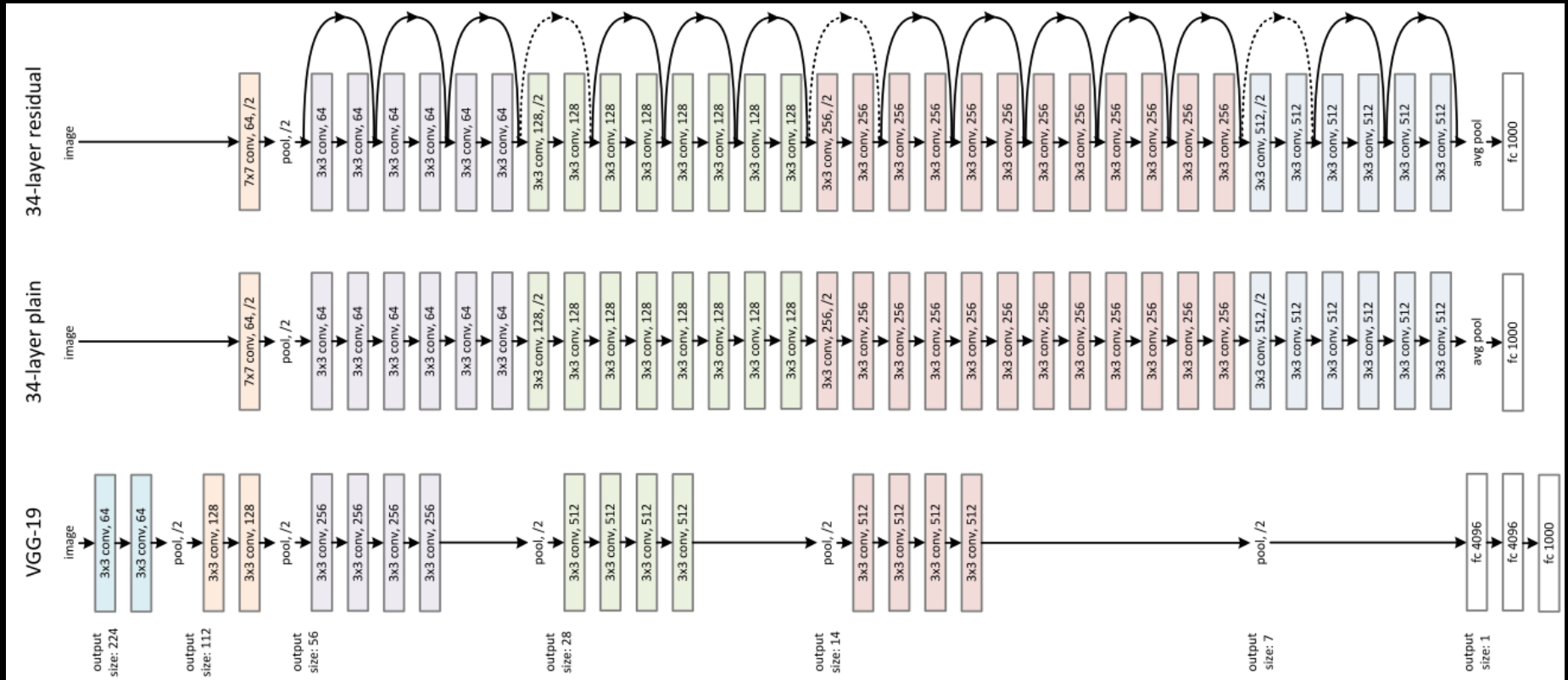


AlexNet

- Having millions of neurons is typical
- Designing the architecture is black magic: lots of manual tweaking



# Deep Learning: Going Deeper!



Resnet-50

# Deep Learning in Action

- Demonstration on handwritten characters [Hinton et al. 2006]
  - <http://www.cs.toronto.edu/~hinton/adi/index.htm>
- Human Tracking: Cao et al. CVPR 2017
  - <https://www.youtube.com/watch?v=pW6nZXeWIGM>
- Neural Task Programming: Xu et al. ICRA 2018
  - <https://www.youtube.com/watch?v=THq7I7C5rkk>

# Deep Learning Tools

- Training on GPUs is much faster than CPUs
  - More GPUS = Faster Meta-parameter search = Better results
- Tools used for Deep Learning
  - Tensorflow (Google)
  - pyTorch (Facebook)
- Keras is a simple wrapper for the above

# Summary

- ANNs are networks of simple thresholding functions
- 3-layer networks can approximate any continuous real-valued function
- Backpropagation is used to learn the weights of a neural network
  - Gives local optimum
- Deep networks are the leading edge of ANN research
- Many robotics applications of deep learning!

# Homework

- Final project!