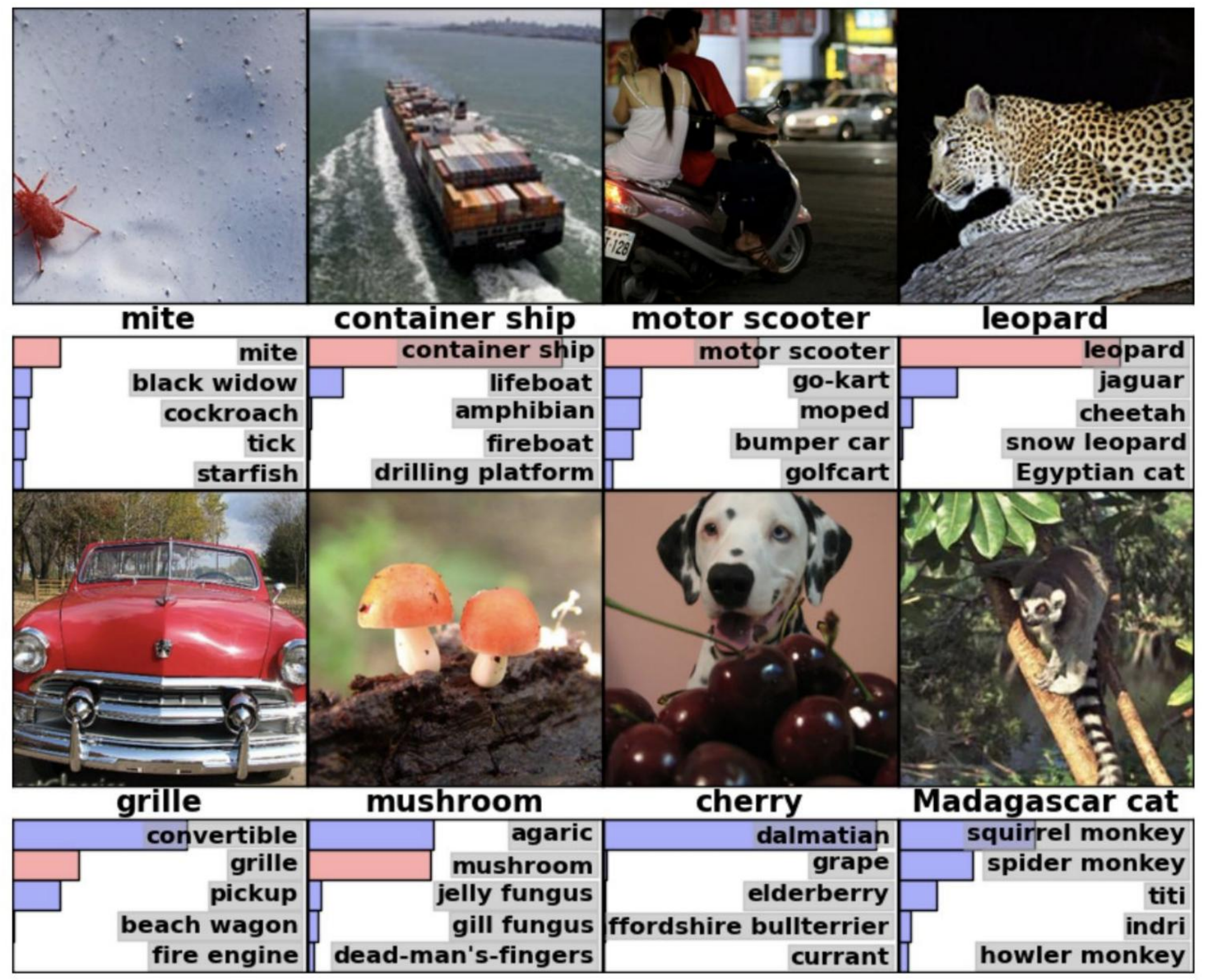
Module 9

Deep Learning

Why

ImageNet challenge

IMAGENET

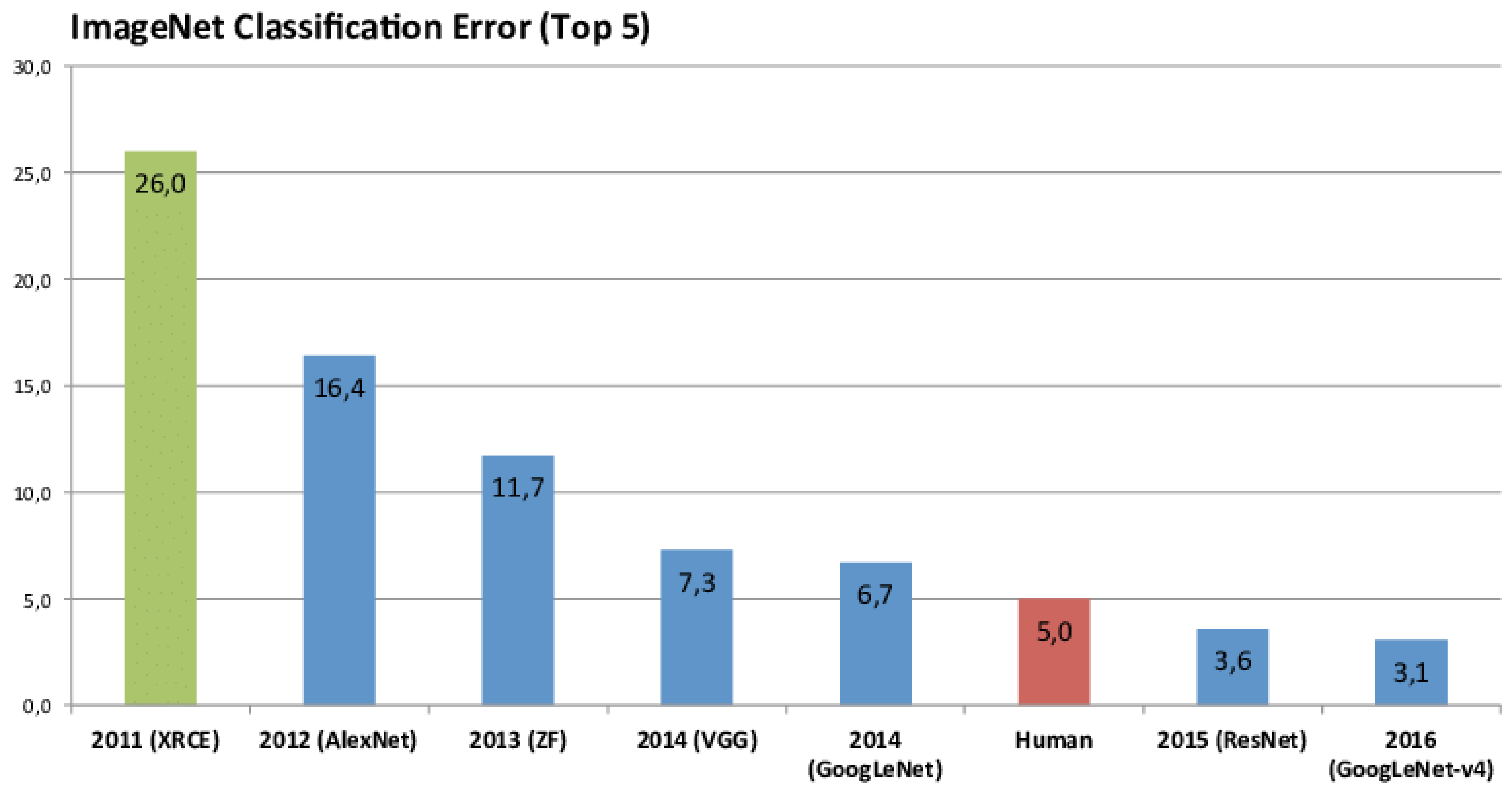


(Source: [Xavier Giro-o-Nieto])

- 1000 object classes
- Images
    - 1.2 Million to train
    - 100K to test

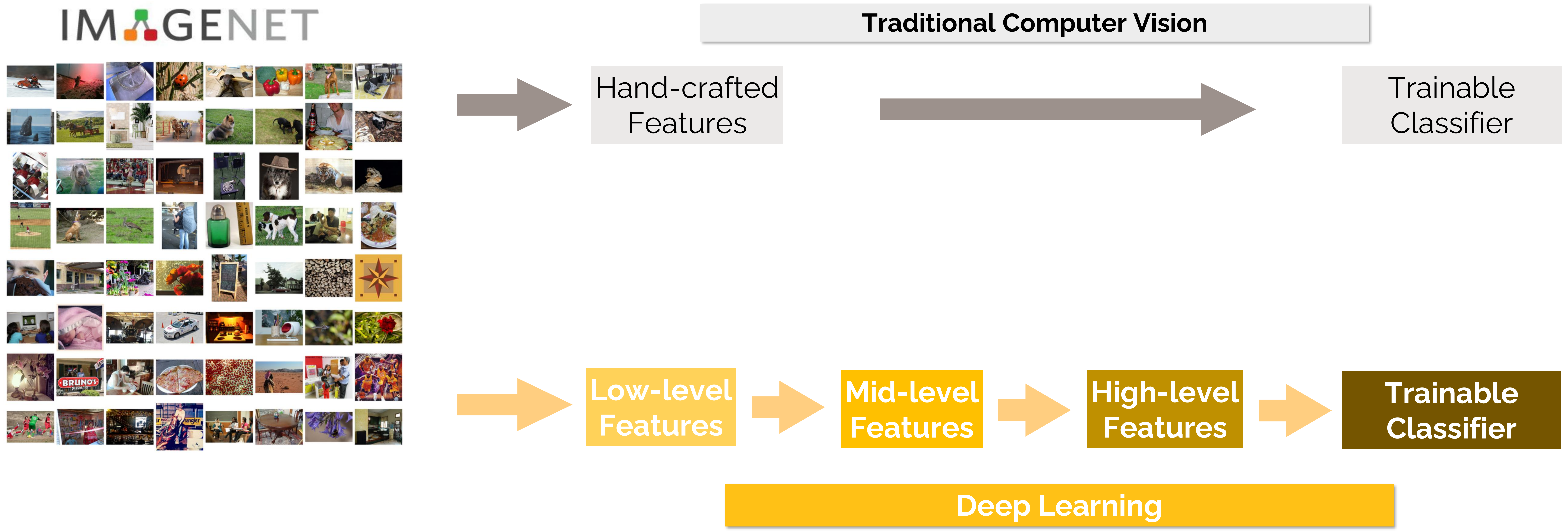
Computer vision took a quantum leap in 2012 with AlexNet

Why?

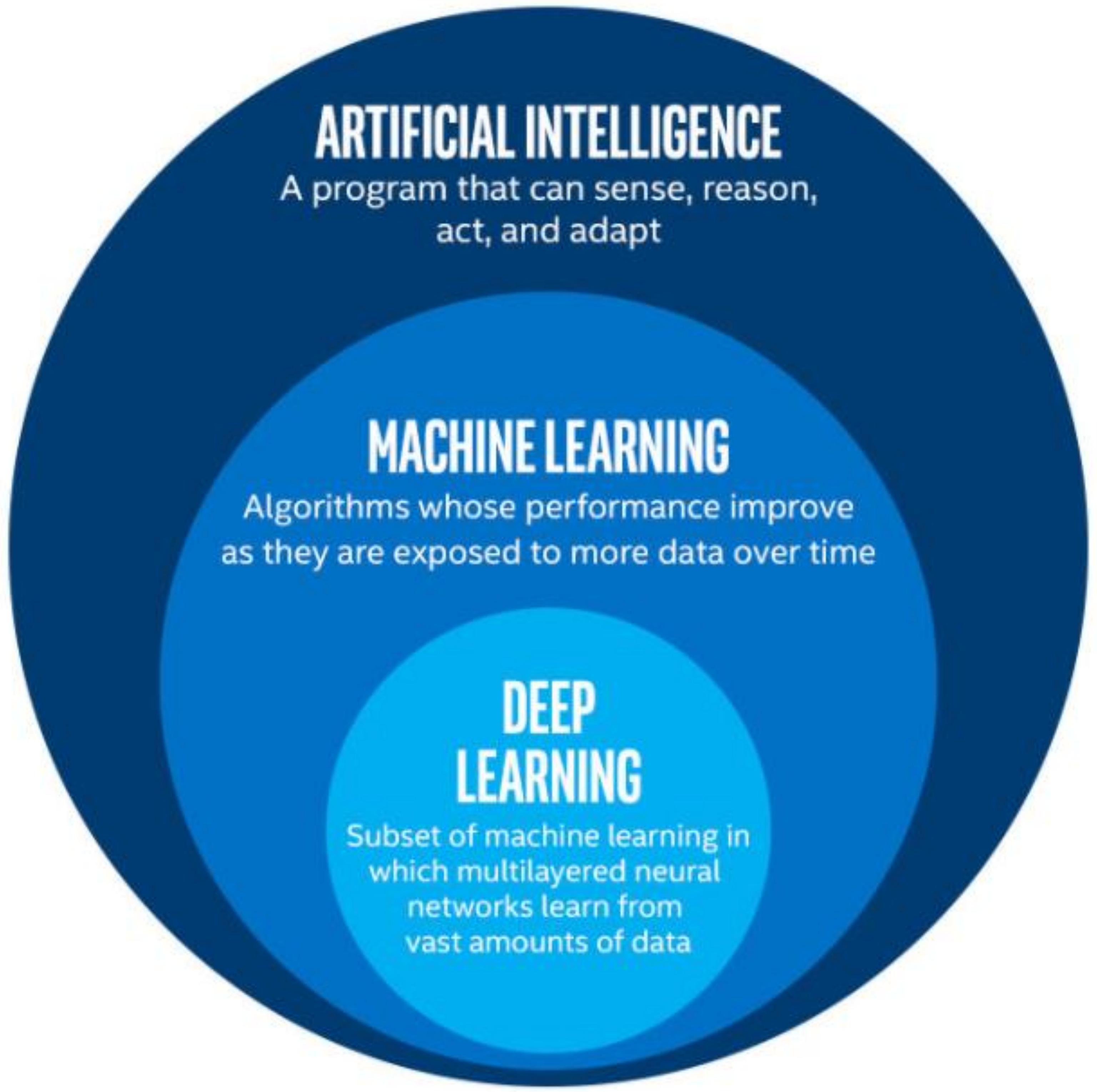


https://www.researchgate.net/figure/Winner-results-of-the-ImageNet-large-scale-visual-recognition-challenge-LSVRC-of-the_fig7_324476862

What is Deep Learning?



What is Deep Learning?


ARTIFICIAL INTELLIGENCE
A program that can sense, reason, act, and adapt
MACHINE LEARNING
Algorithms whose performance improve as they are exposed to more data over time
DEEP LEARNING
Subset of machine learning in which multilayered neural networks learn from vast amounts of data

In today's context a subset of the field of AI

https://towardsdatascience.com/cousins-of-artificial-intelligence-dda4edc27b55

Recap of learning architectures

Category 1:

- Here you know what the answer should look like. There is a teacher-signal, a gold standard, ***labeled-data*** that you are comparing against. This is referred to as ***"Supervised Learning".*** (SVM, Regression,)

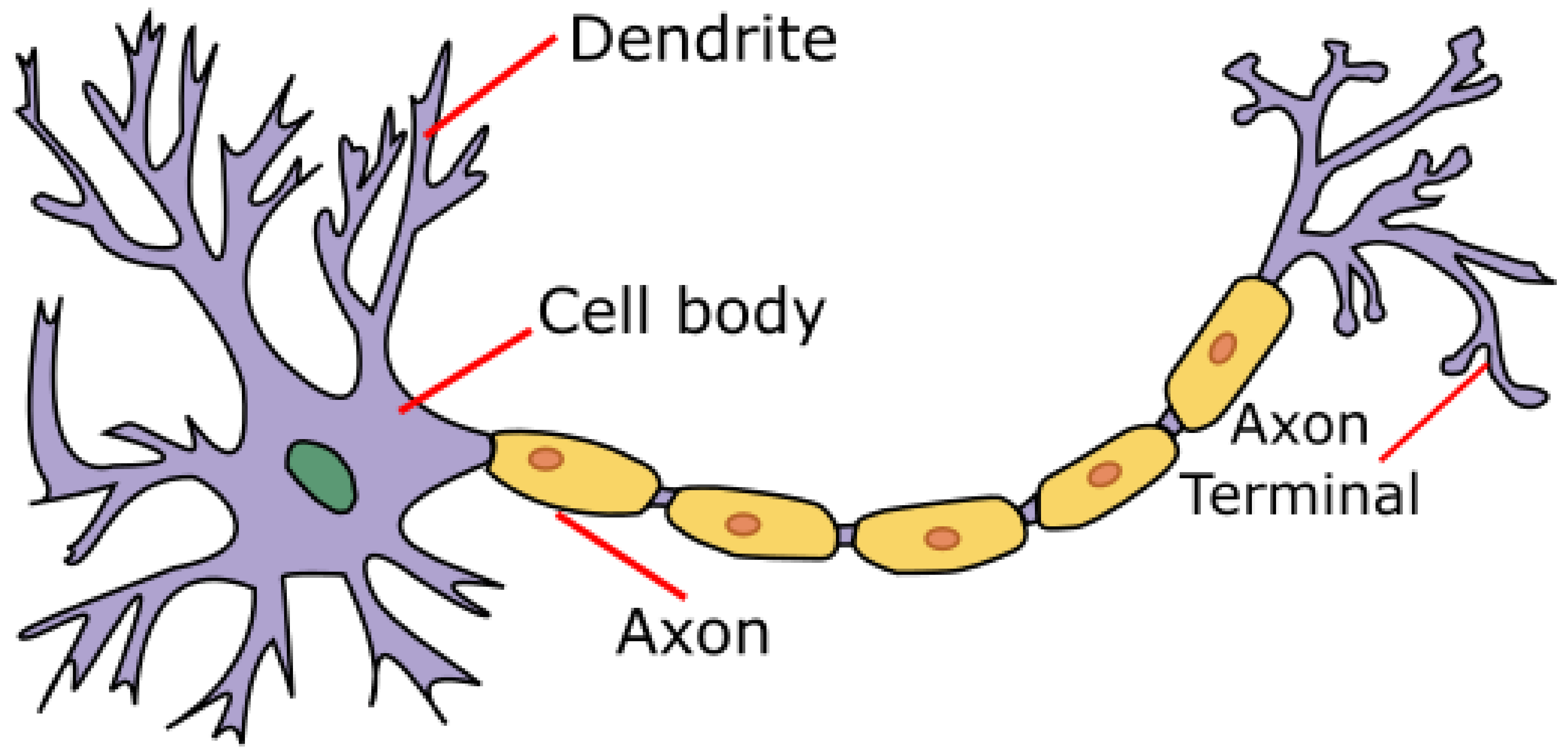
Category 2:

- Here you do not have much a priori knowledge, i.e., no clearly labeled data of the answer. We search for structure in the data. This is referred to as "***Unsupervised Learning".***

Neural Networks

The Biological Neuron

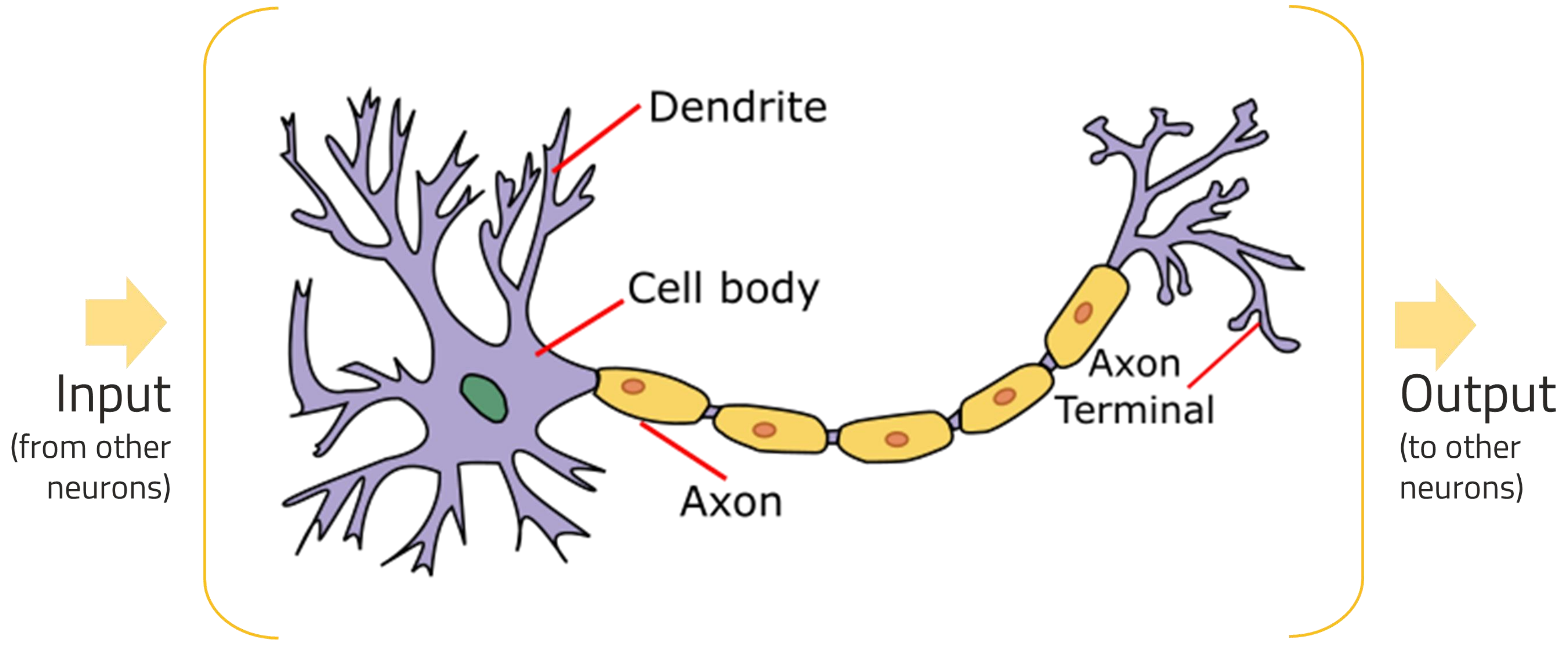
Input
(from other
neurons)

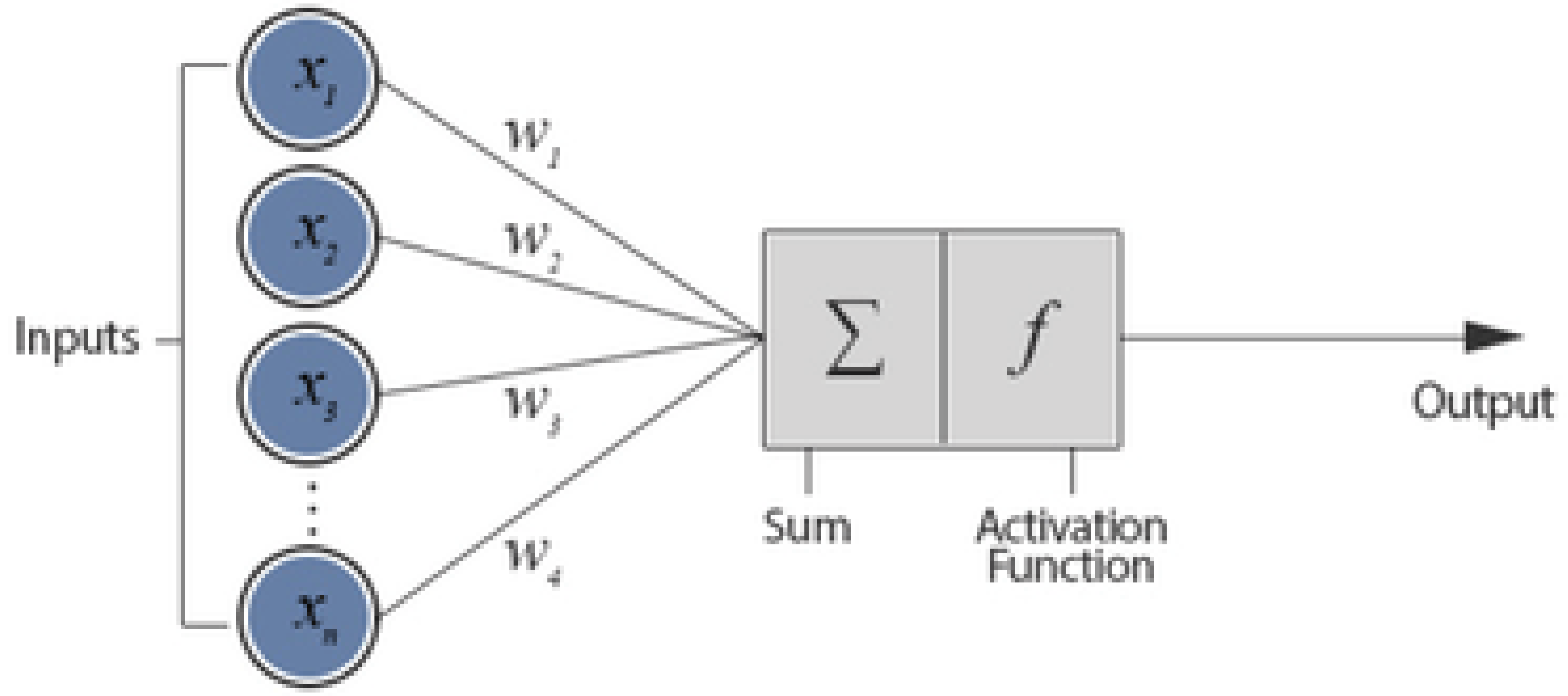


Output
(to other
neurons)

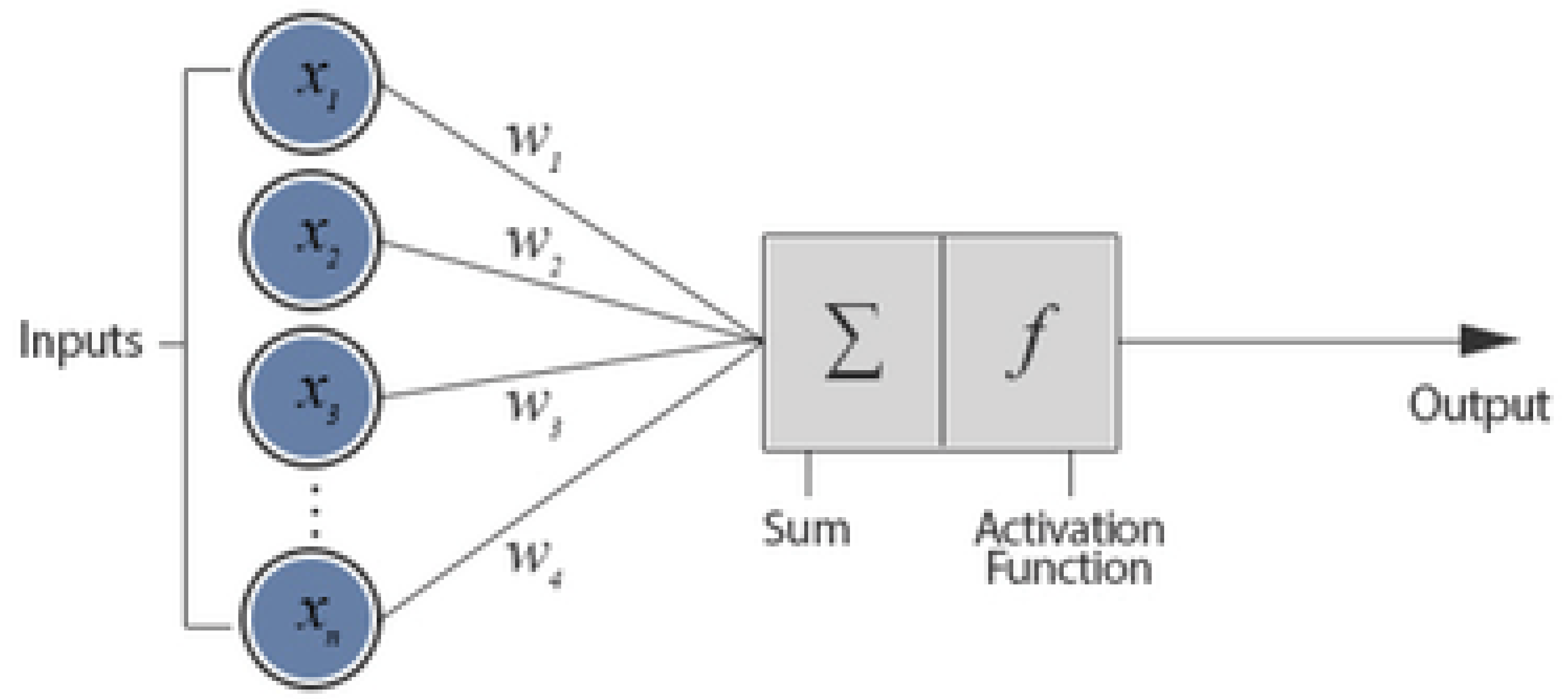
https://en.wikipedia.org/wiki/Neuron

The Artificial Neuron





The Artificial Neuron



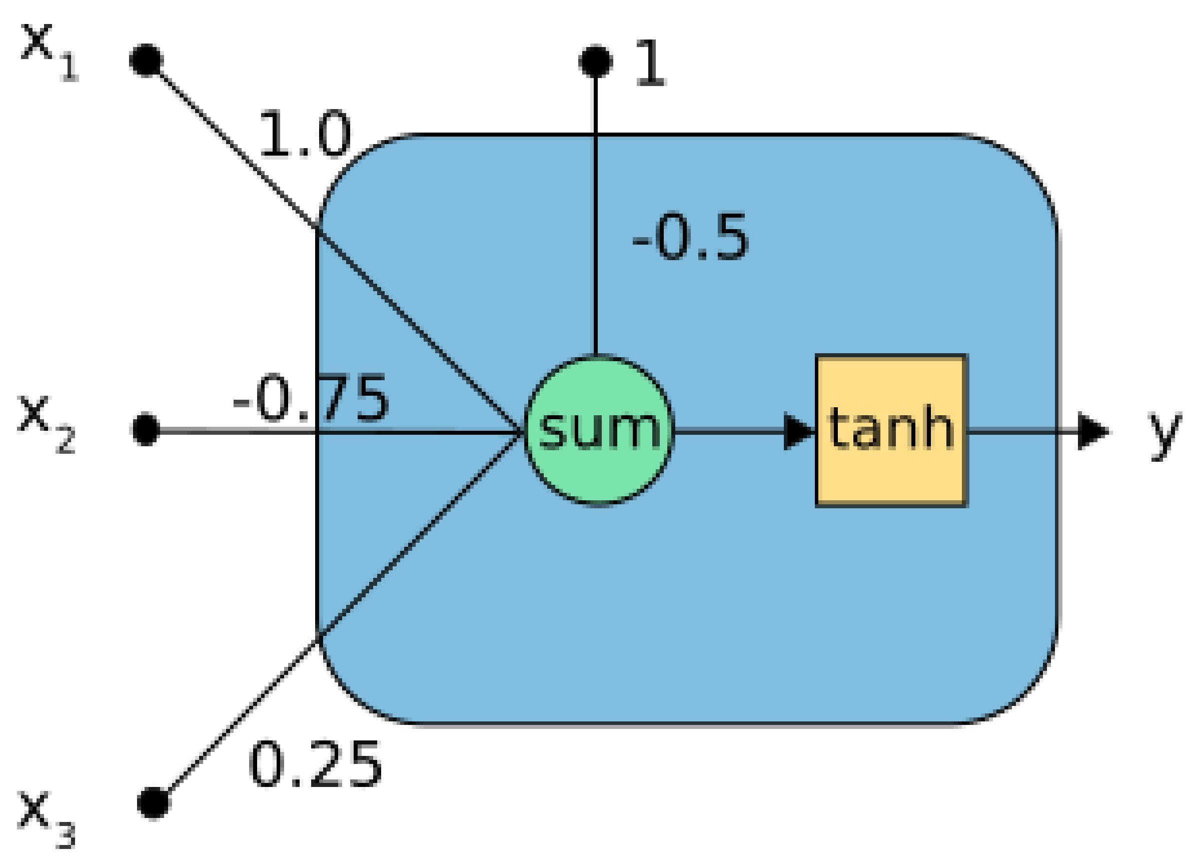
Encapsulates three main aspects of a biological neuron

- Primarily a threshold element (‘firing’)
- Nonlinear response (i.e. thresholding is not a straight line)
- Adaptive (can change its connection strengths to other neurons)

i.e. abstracts the integrate and fire paradigm.

Most used artificial neuron model is the “Perceptron” (Minksy & Papert, 1969) model (an iteration of the McCulloh-Pitts (1943) model). *Perceptron is the mainstay ‘neural model’ for modern AI*

Perceptron

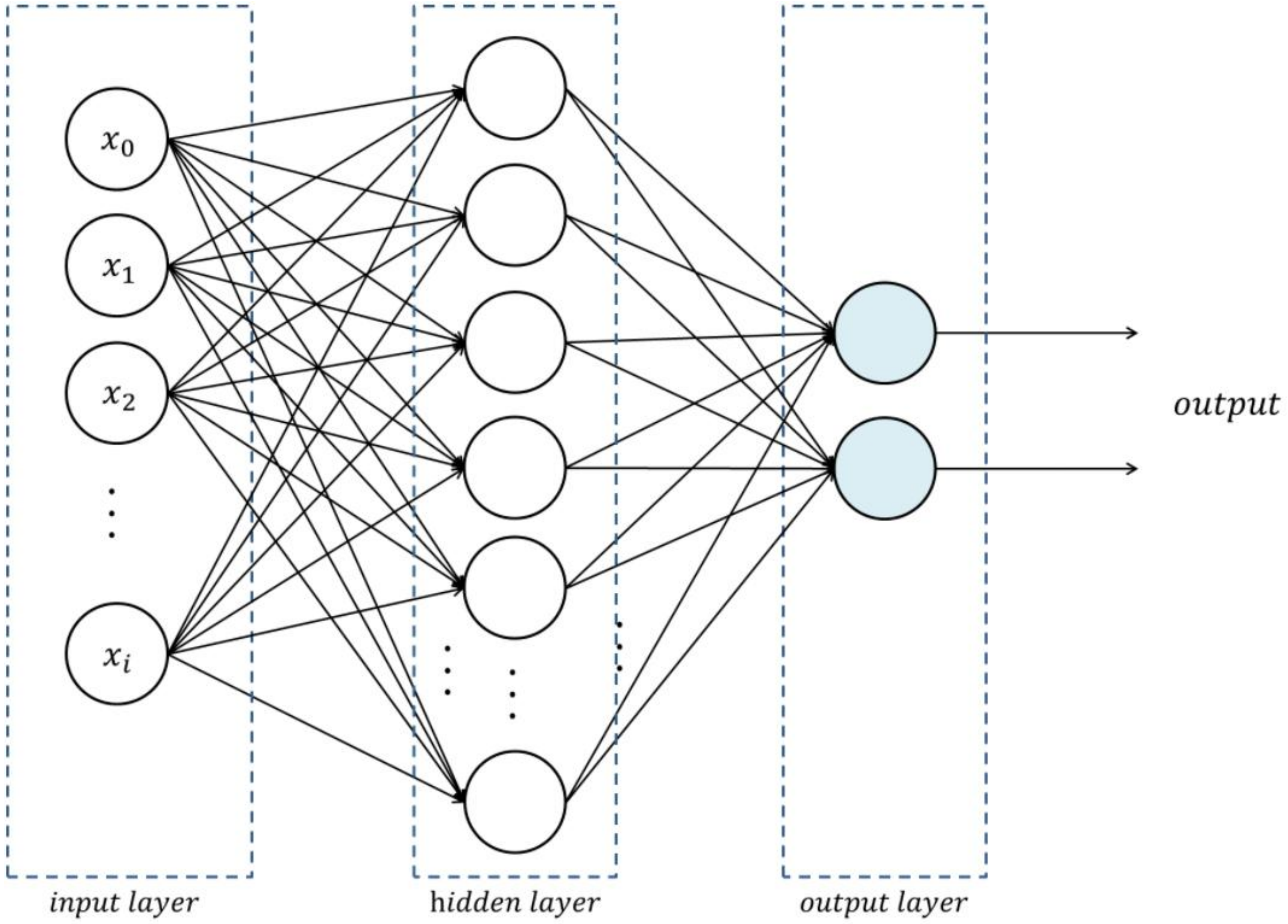


Has:

1. Inputs (***x_1 x_2 x_3***)
2. Weights, **w** (1,-0,75,0.25)
3. Combination/integration function: Typically summation (***sum***)
4. Activation function (threshold function, here ***tanh***.)
5. Output (***y***)
6. There is also a bias **b** (1) (state of neuron without input)

Individual Perceptrons have limited computational capacities. The computational power comes from stitching a lot of these together as networks with layered architecture

Multi-layer Perceptron

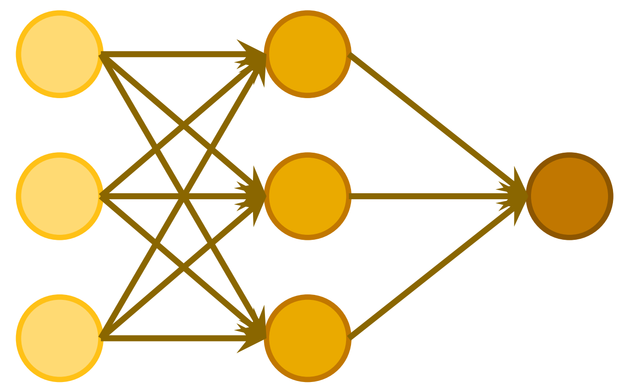


https://www.cc.gatech.edu/~san37/post/dlhc-fnn/

Basic operation of a neural network

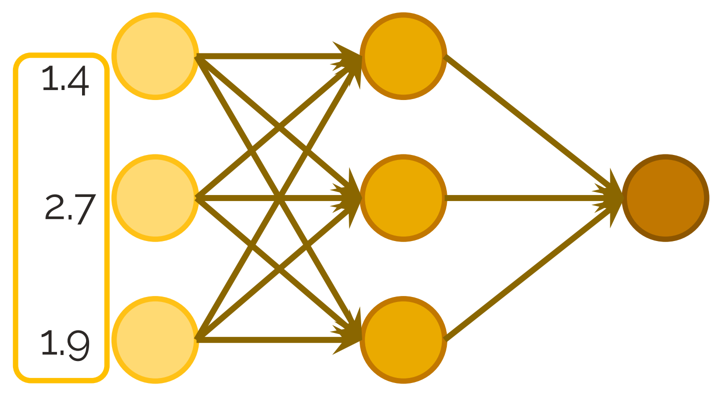
Initialize with random weights

Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0



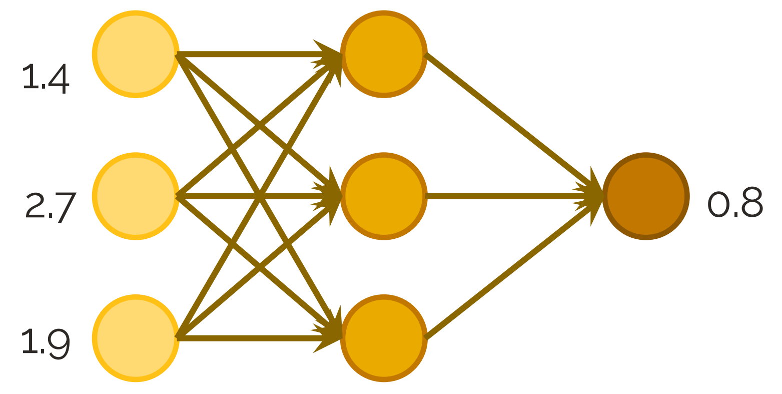
Present a training pattern

Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0



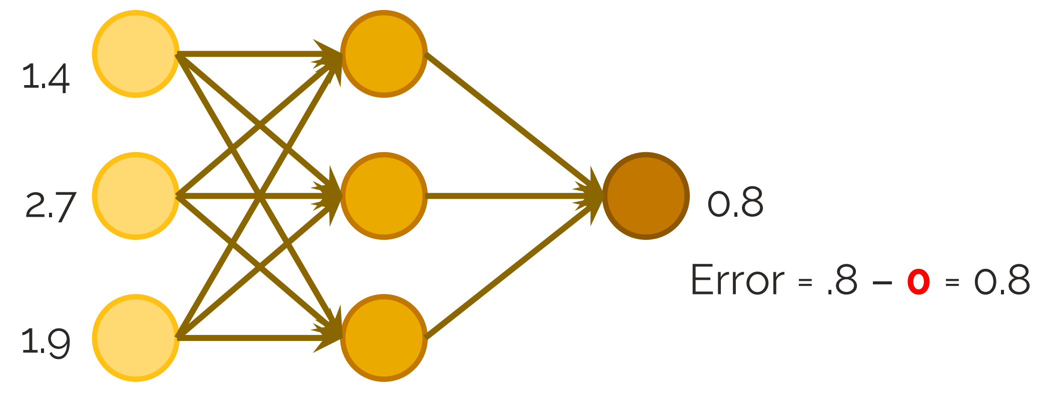
Feed through to output

Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0



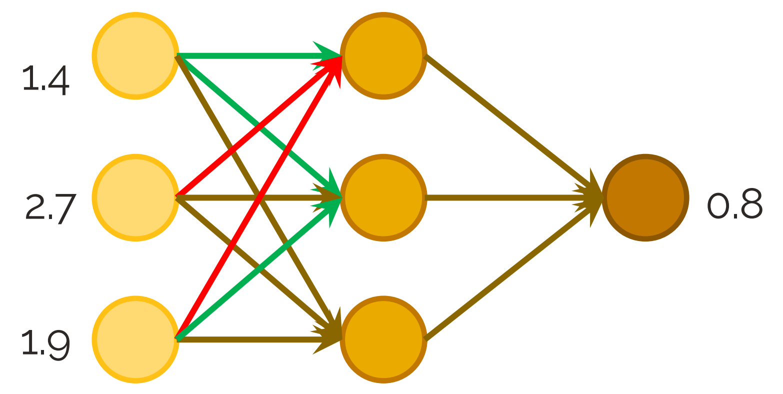
Compare to target, calculate Error

Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0



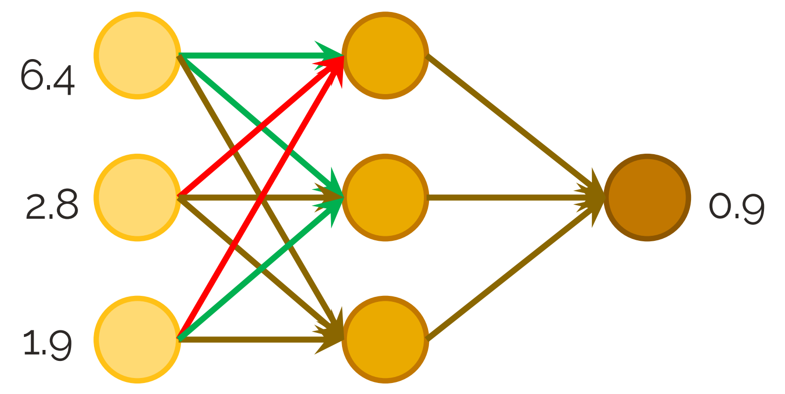
Modify weights based on Loss function

Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0



Present new pattern, repeat process

Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

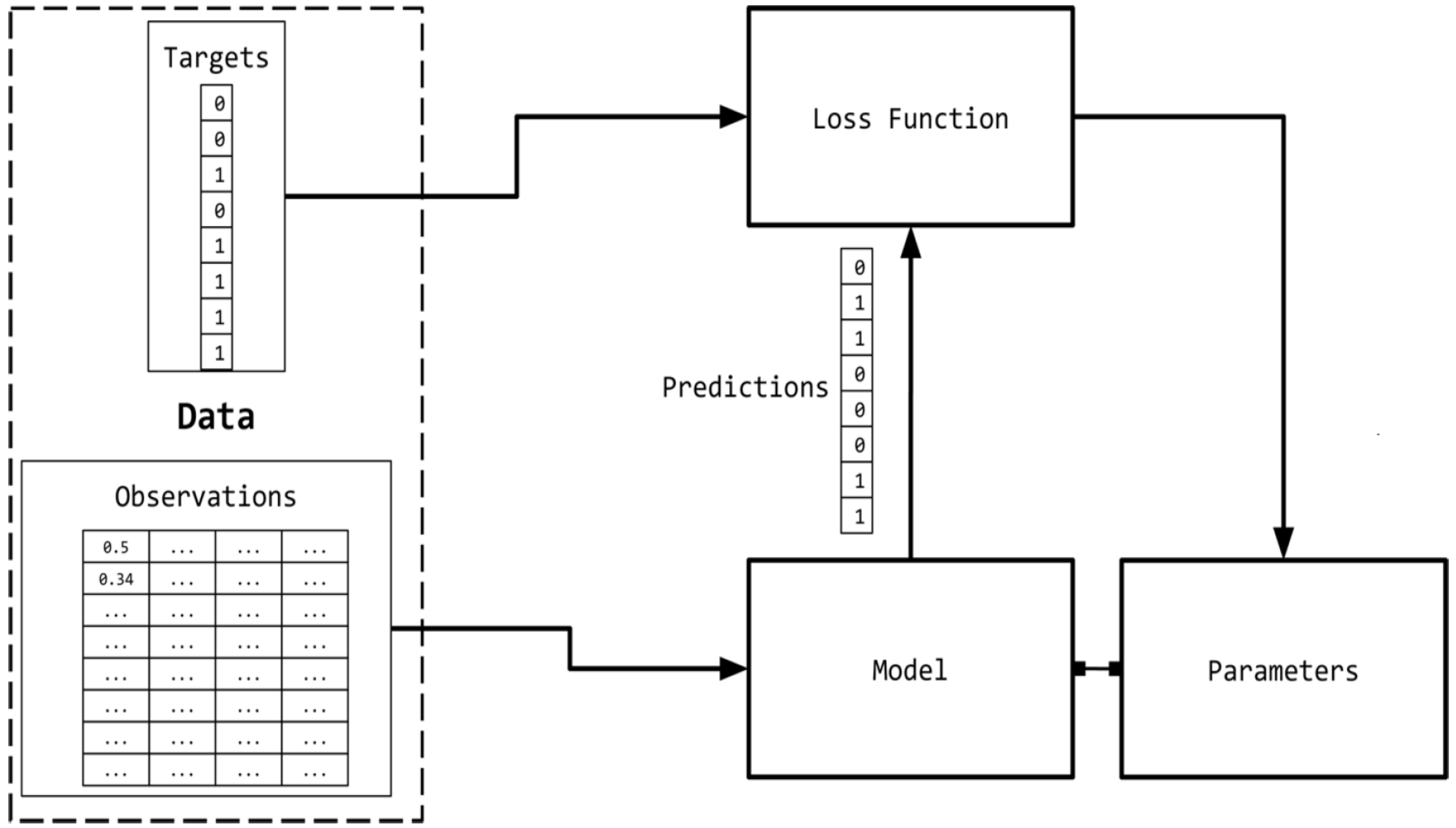


Do this thousands of times till the network learns the correct associations.

Neural Network Demo

https://playground.tensorflow.org/

Recap: Supervised Learning



Observations are items about which we want to predict something. Also referred to as *inputs*.

Targets are ***labels*** corresponding to an observation. These are usually the things being predicted.

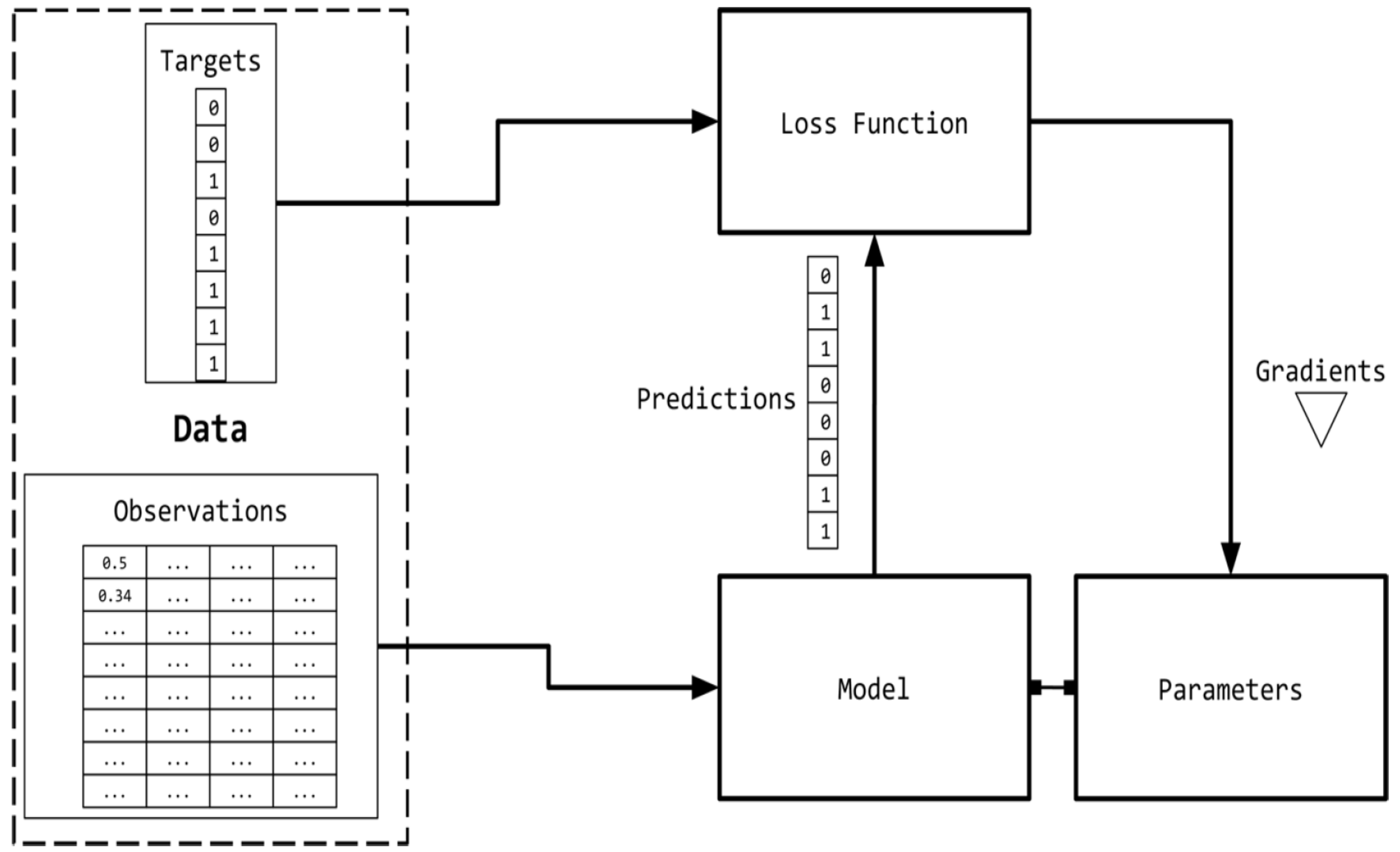
Model is a mathematical expression or a function that takes an observation, ***x***, and predicts the value of its *target label*.

Parameters, also called *weights*, these parameterize the model. It is standard to use the notation ***w*** (for weights) or $\hat{w}$.

Predictions, also called *estimates*, are the values of the targets guessed by the model. The prediction of a target ***y*** is denoted as $\hat{y}$.

Loss function compares how far off a prediction is from target for observations in the training data. The lower the value of the loss, the better the model is at predicting the target.

Recap: Supervised Learning



The ***general supervised modeling paradigm*** involves training the model on existing labeled data set and then applying it to new data to make predictions.

- start with lots of ***correctly labeled*** training ***data/observations***.
- pass it through our ***model*** and calculate the ***error*** (loss, cost etc) by comparing it to the correct label values
- We then use another function known ***tweak the model's parameters*** so that the cost or error is low.

The goal of supervised learning model is to pick values of the parameters that minimize the loss function for a given dataset. i.e., equivalent to finding roots of an equation.

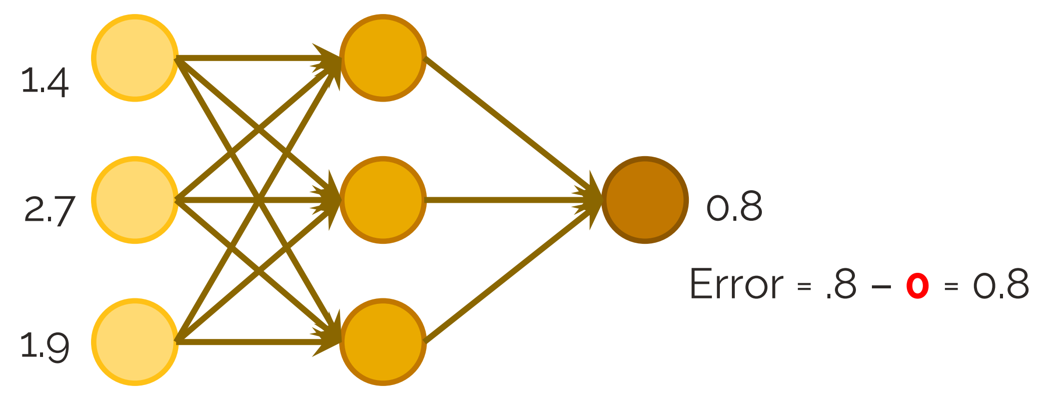
Most of the work centers around figuring out

- What model to use e.g. linear, non-linear, neural networks
- What is the best way to calculate error (cost function)
- What is the best way to minimize that error. E.g. gradient descent, simulated annealing etc. What algorithm can do the minimization fastest/efficiently. Depends on choice of #3, the error function

Finding the right parameters

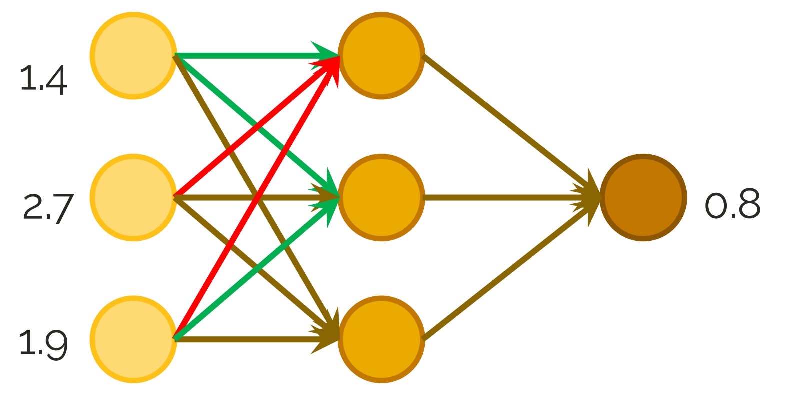
Compare to target, calculate Error

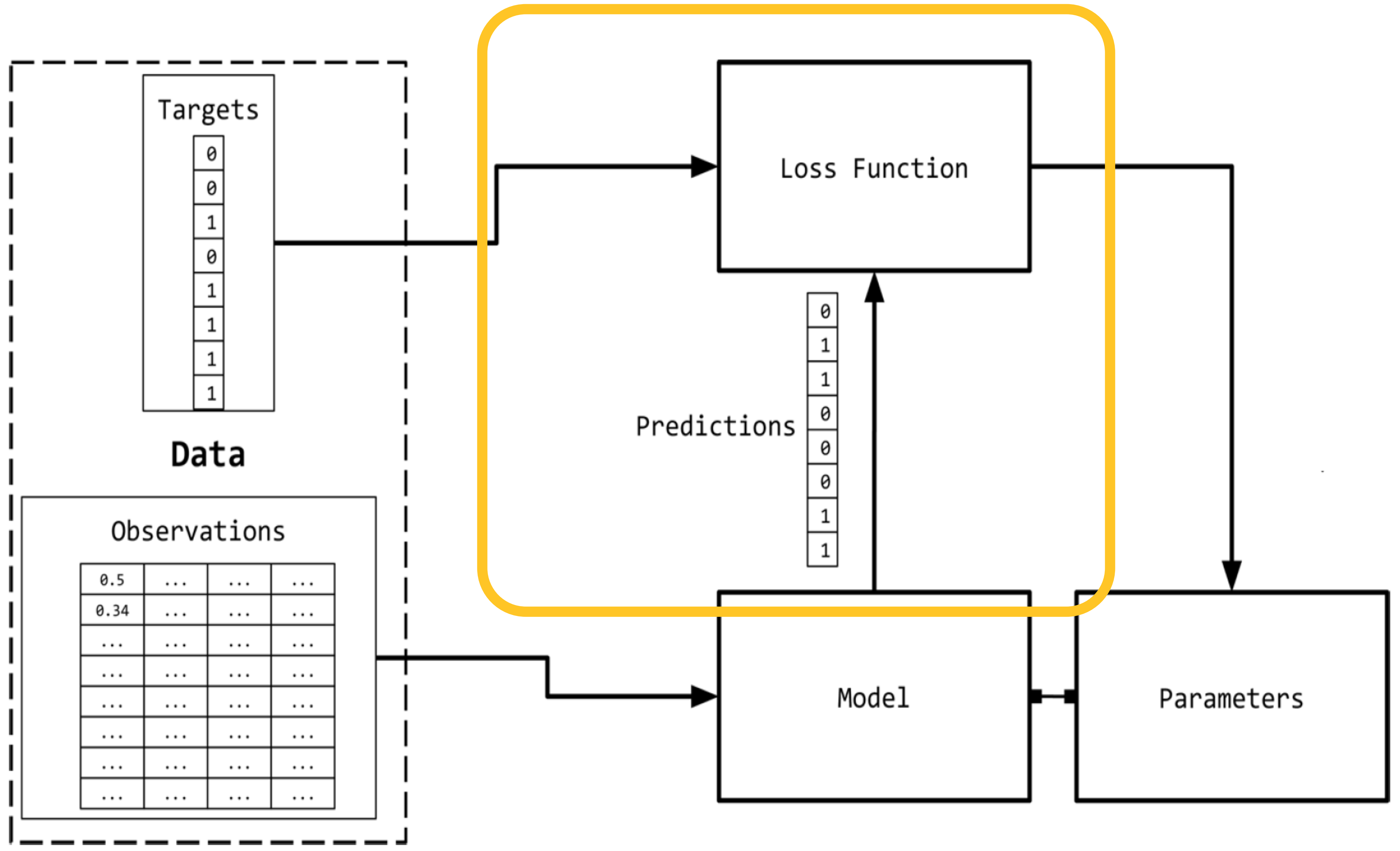
Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0



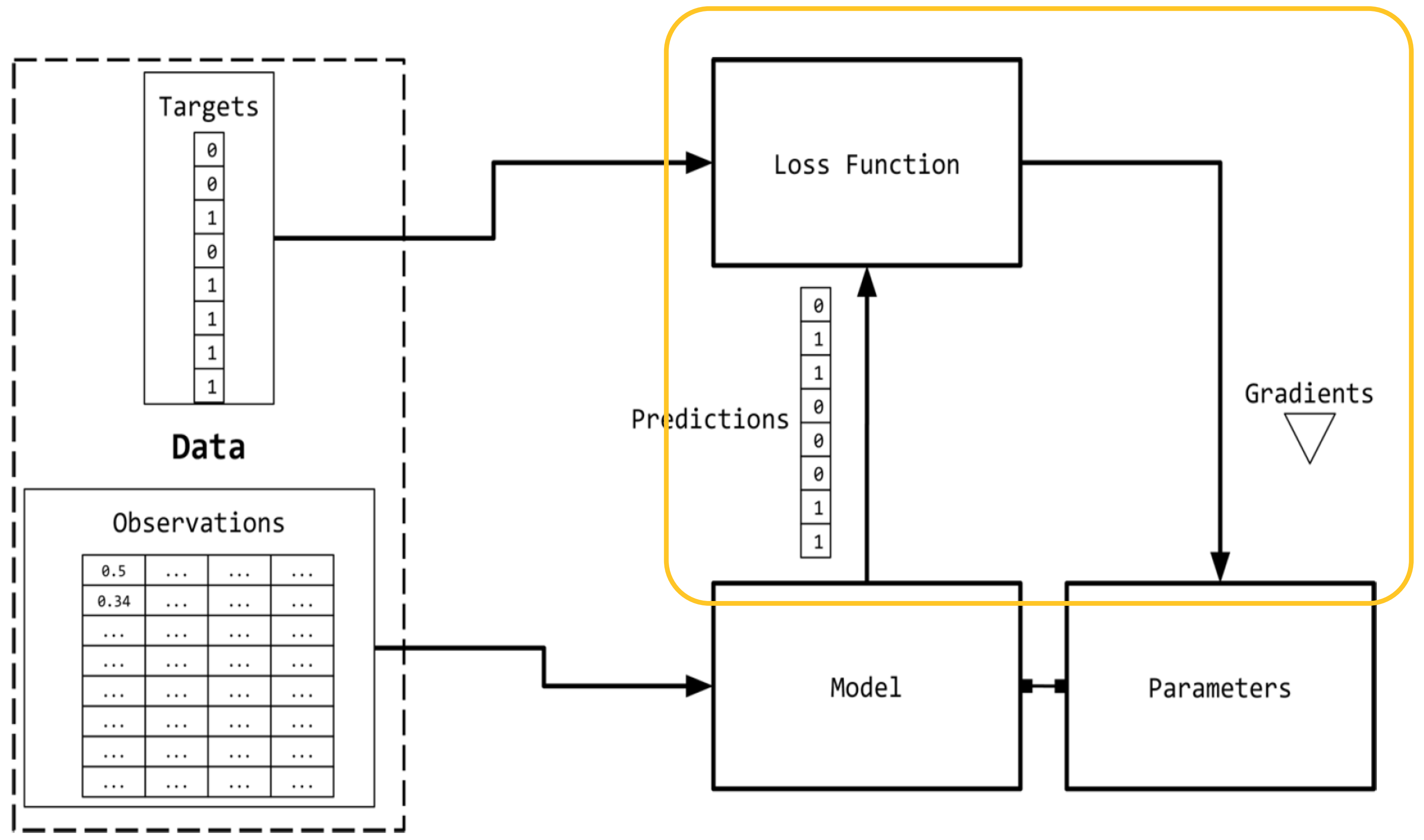
Modify weights based on Loss function

Observations			*Targets*
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0





Finding the right parameters: Loss function



Cost/Loss Function

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Minimizing Cost: e.g., Least mean squares algorithm

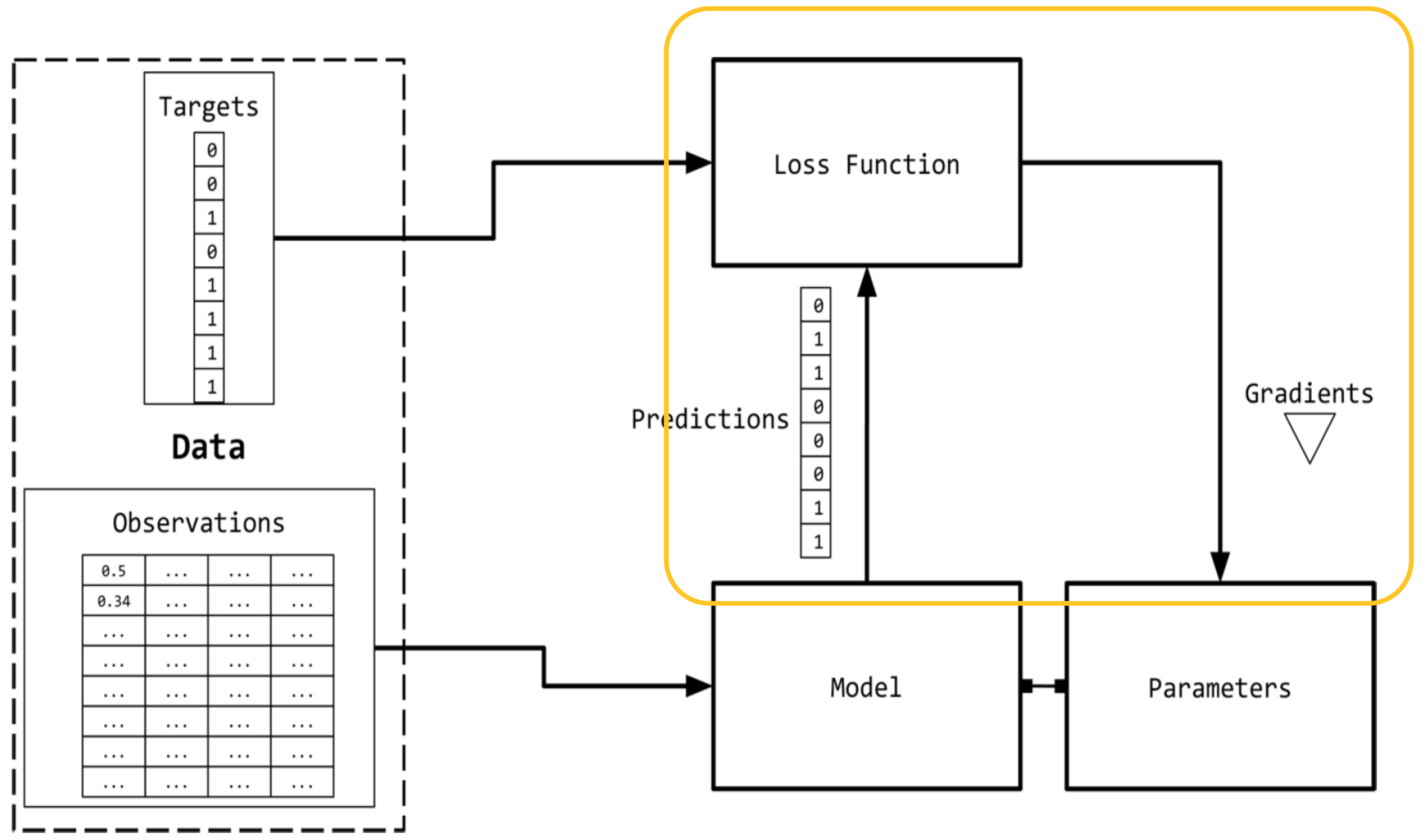
We want to choose θ so as to minimize J(θ). Start with initial guess for θ and iteratively change θ to minimize J(θ). We will use another algorithm called ***Gradient Descent*** to achieve this.

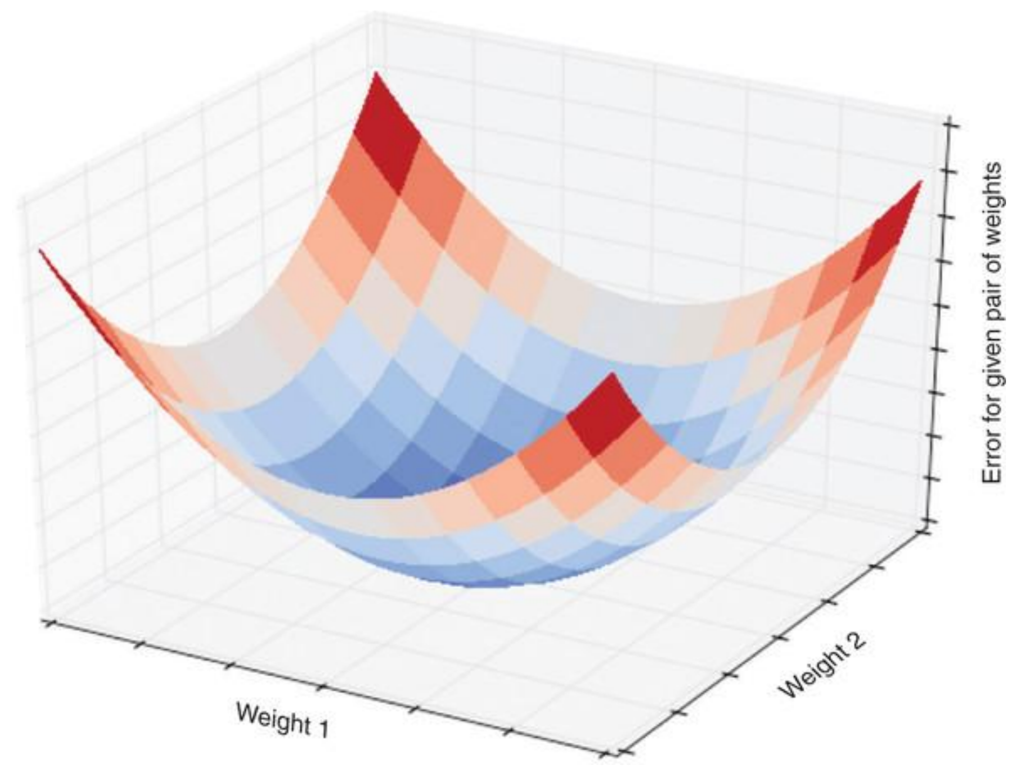
Conceptually ***Gradient Descent*** works by finding the direction of the largest change around a location. We then move along this direction by a step and do our calculation again and find the direction of greatest change at that point, move another step and so on. Mathematically this is represented by:

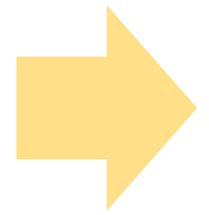
$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)$$

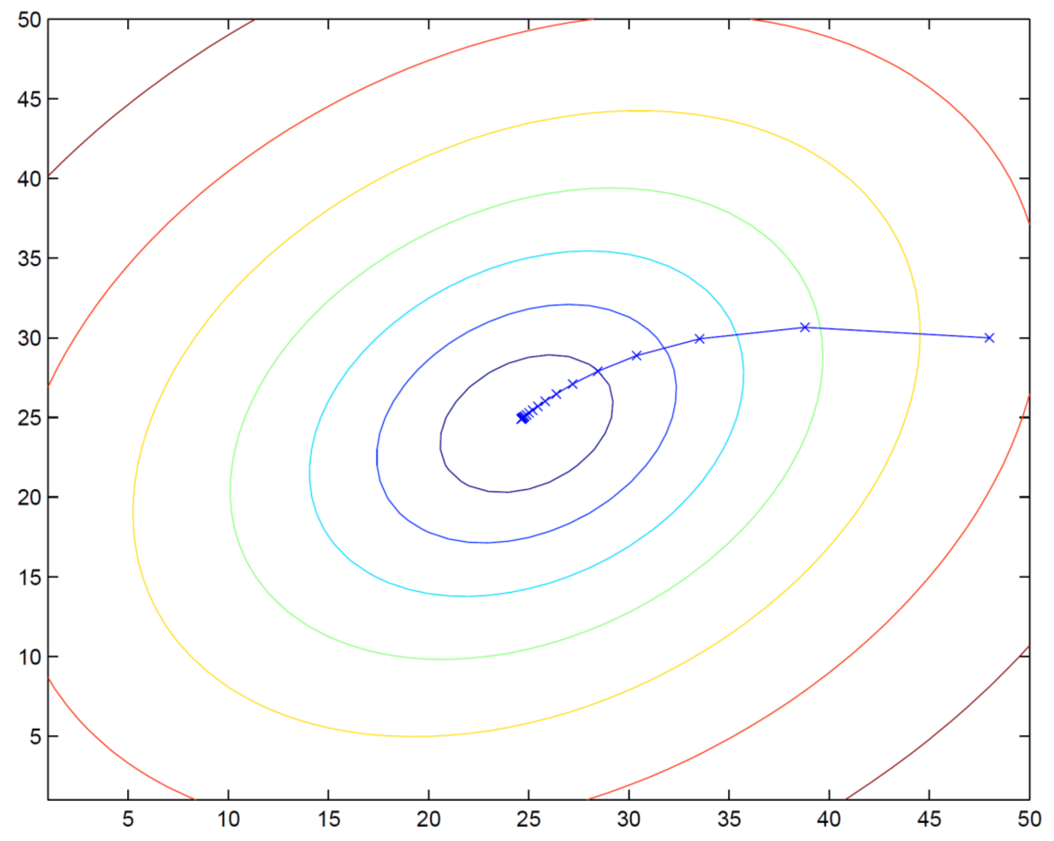
α is the ***learning rate*** that determines how big a step is. This is set by hand or before the training begins. Such parameters are also referred to as ***hyperparameters***

Finding the right parameters: Gradient Descent





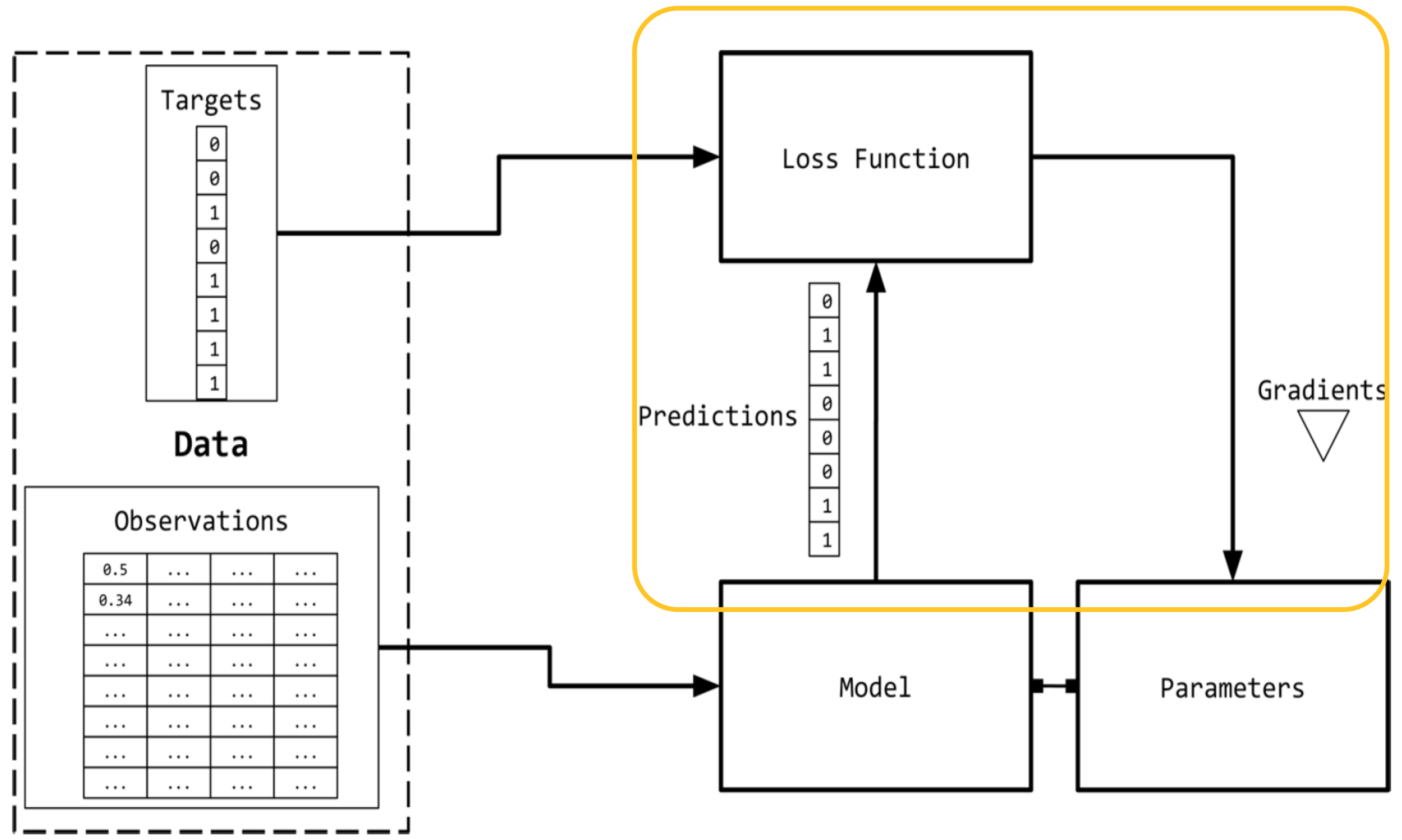




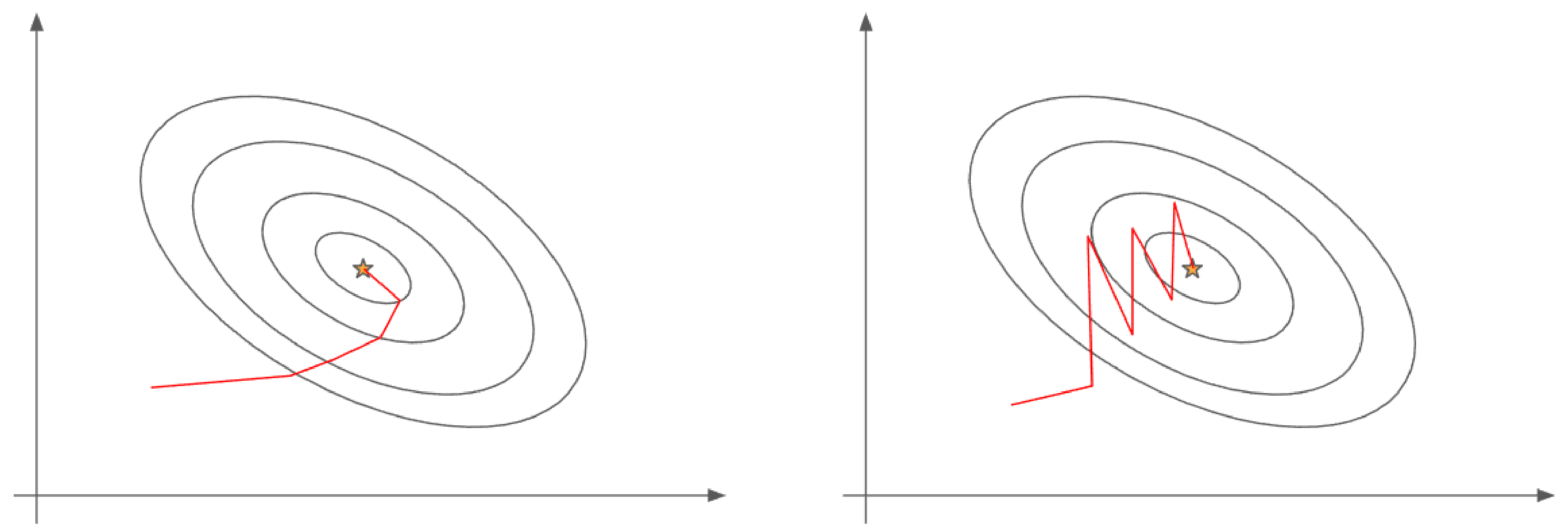
Example trajectory using batch gradient descent. Ellipses are contours of our loss function:

Once the batch gradient descent is done we have our **θs, i.e., the parameters of our model**

Finding the right parameters: Gradient Descent



Stochastic Gradient Descent: Alternative quicker version



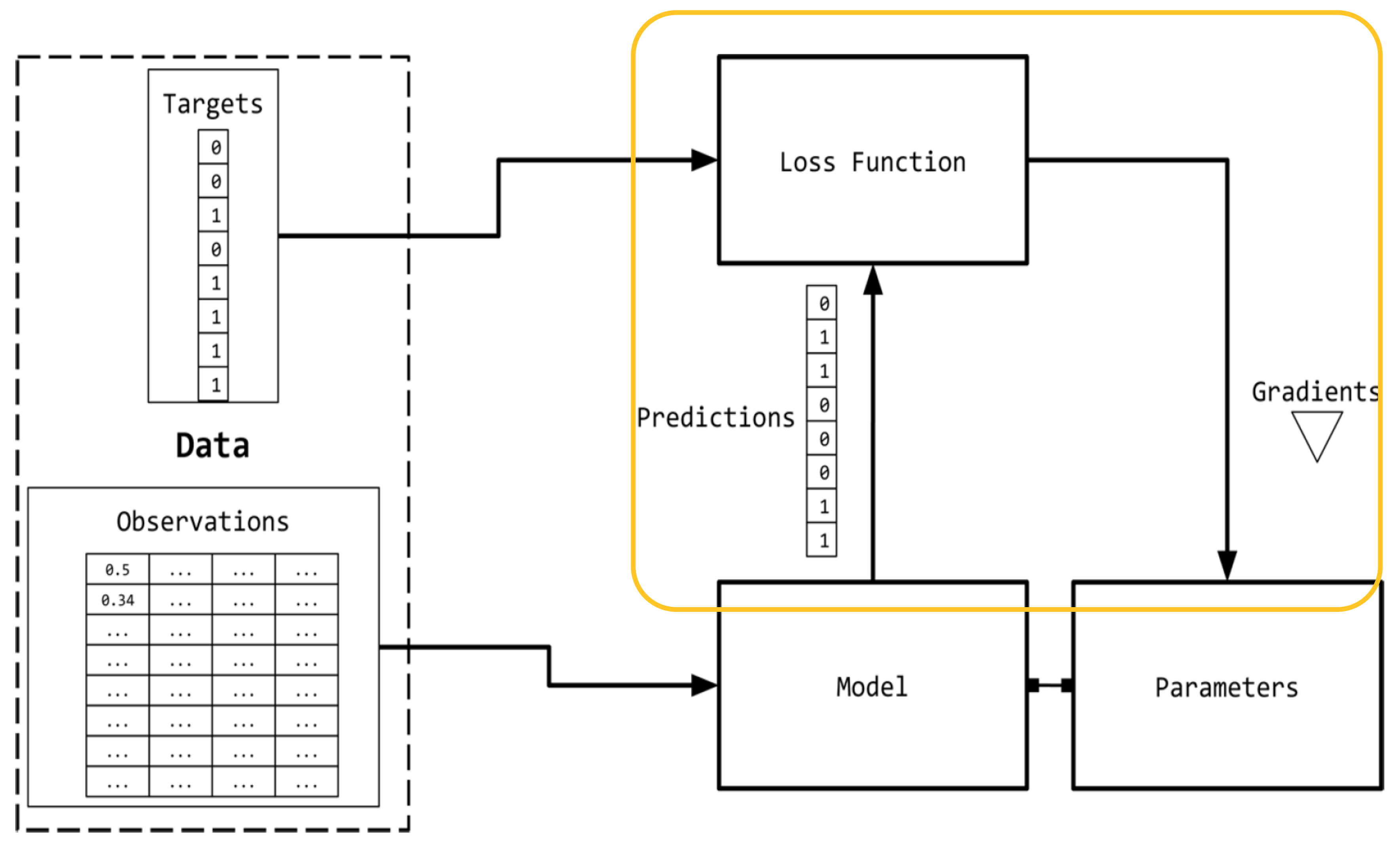
Gradient Descent

Stochastic Gradient Descent

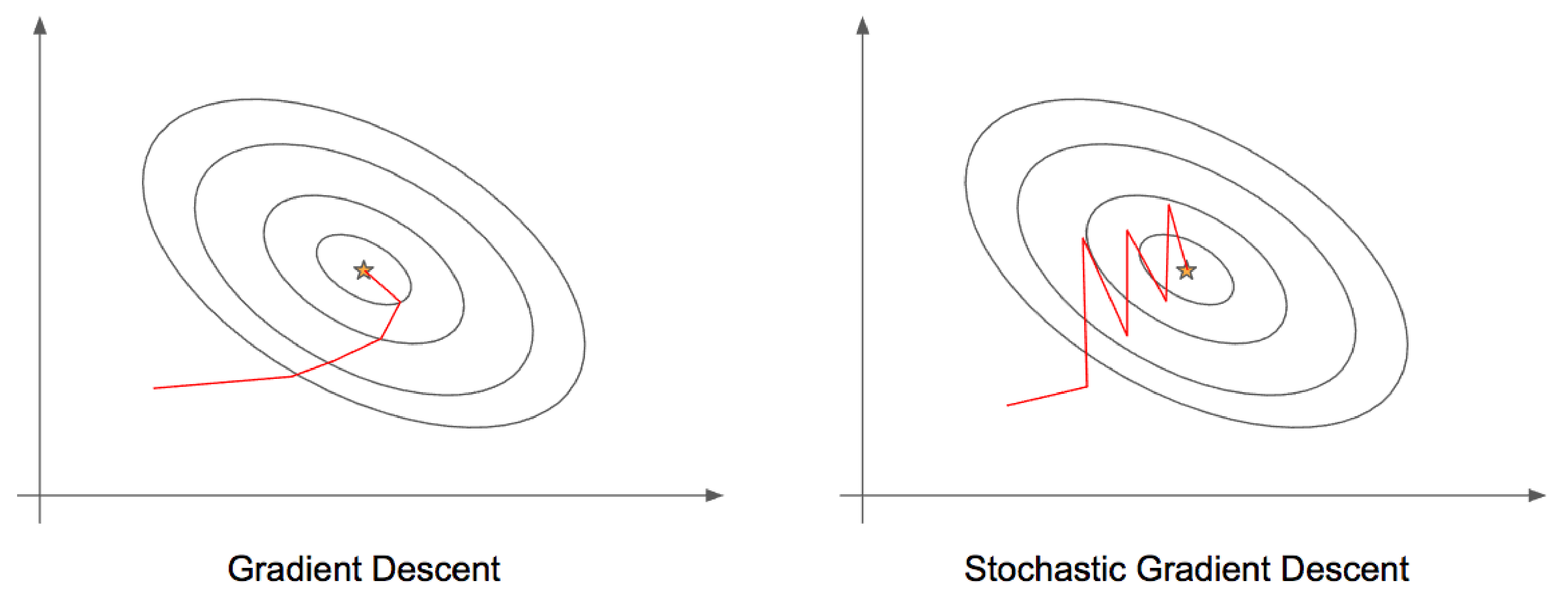
Update against each example as you go through the training set. Takes a more zig-zag path to minimum, but gets their faster. More commonly used in machine learning (ML)

Loss functions and optimization algorithms are an area of active research. Progress is being made in multiple disciplines and lots of new developments. But the basic premises are similar. Traverse the error surface fast to a minima that is acceptable for the data/task at hand.

Finding the right parameters: Gradient Descent



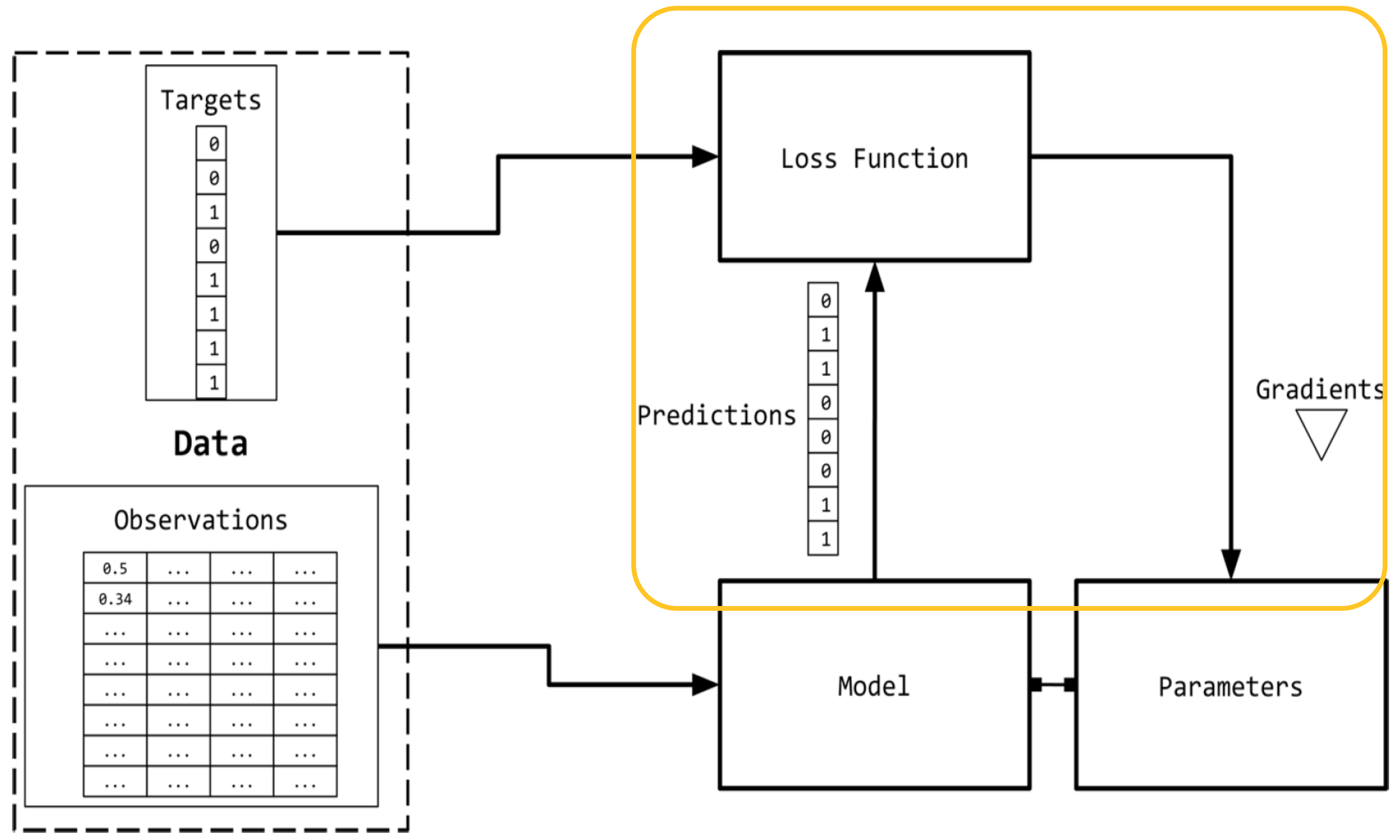
Stochastic Gradient Descent: Alternative quicker version



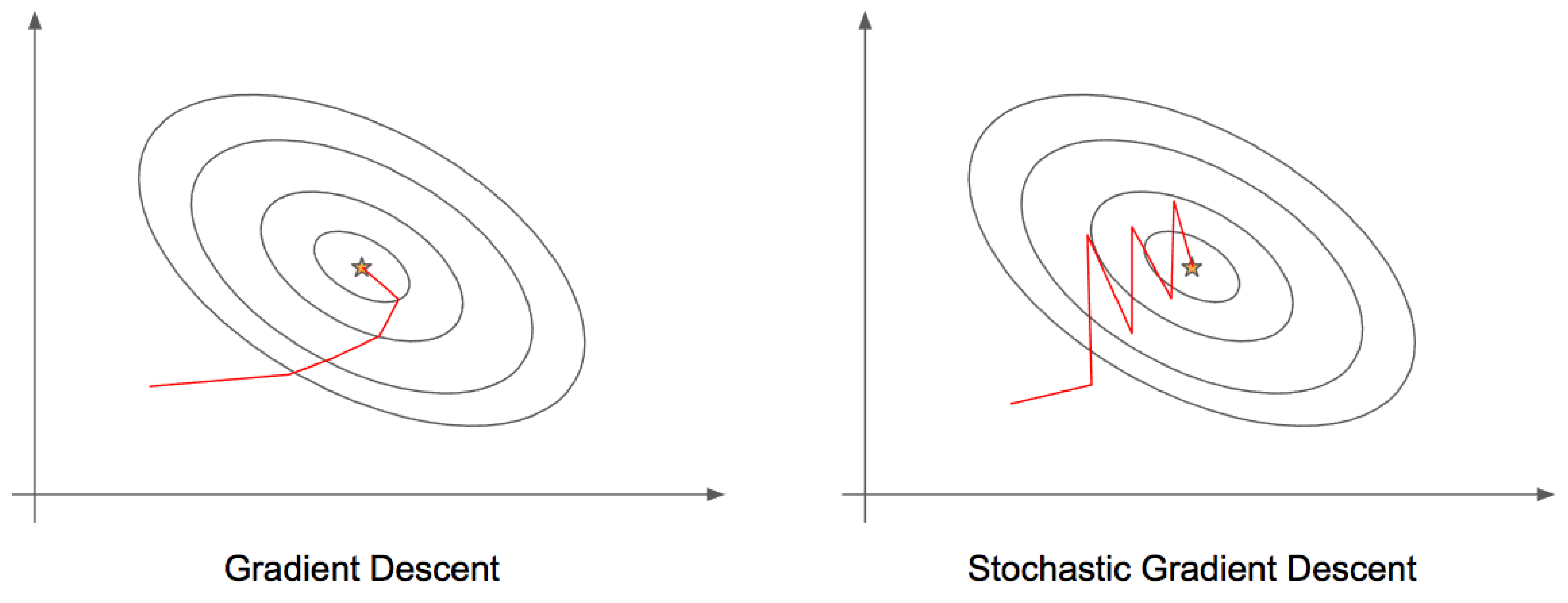
Update against each example as you go through the training set. Takes a more zig-zag path to minimum, but gets their faster. More commonly used in machine learning (ML)

Loss functions and optimization algorithms are an area of active research. Progress is being made in multiple disciplines and lots of new developments. But the basic premises are similar. Traverse the error surface fast to a minima that is acceptable for the data/task at hand.

Finding the right parameters: Gradient Descent



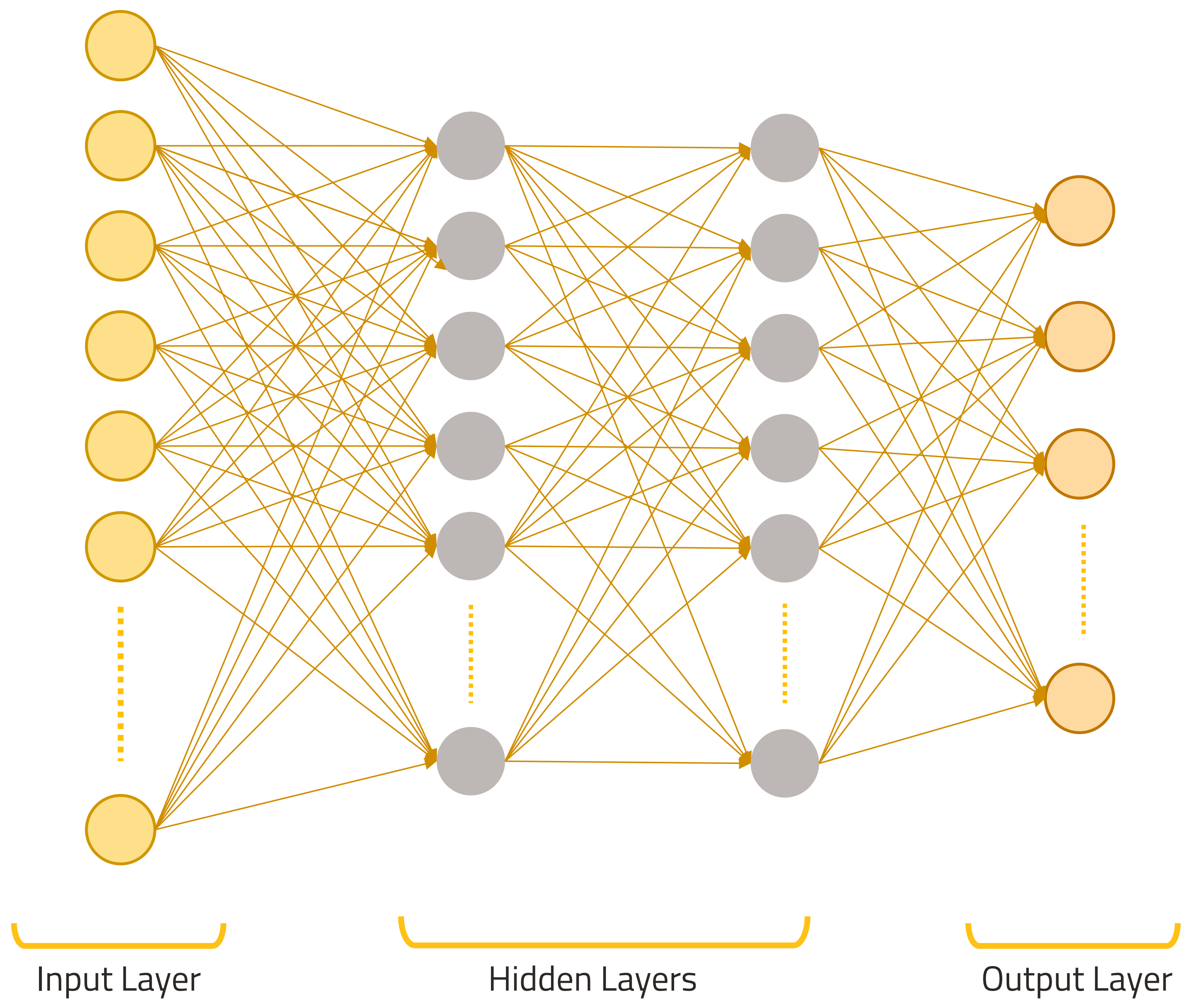
Stochastic Gradient Descent: Alternative quicker version



Update against each example as you go through the training set. Takes a more zig-zag path to minimum, but gets their faster. More commonly used in machine learning (ML)

Loss functions and optimization algorithms are an area of active research. Progress is being made in multiple disciplines and lots of new developments. But the basic premises are similar. Traverse the error surface fast to a minima that is acceptable for the data/task at hand.

Finding the right parameters efficiently: Backpropagation



Finding gradients is a computationally intensive process

It used to be complicated and difficult, but in 1986 Rumelhart et.al adapted an algorithm (from late 60s early 70s) called ***backpropagation*** that made parameter finding more efficient and tractable in neural networks with lots of parameters.

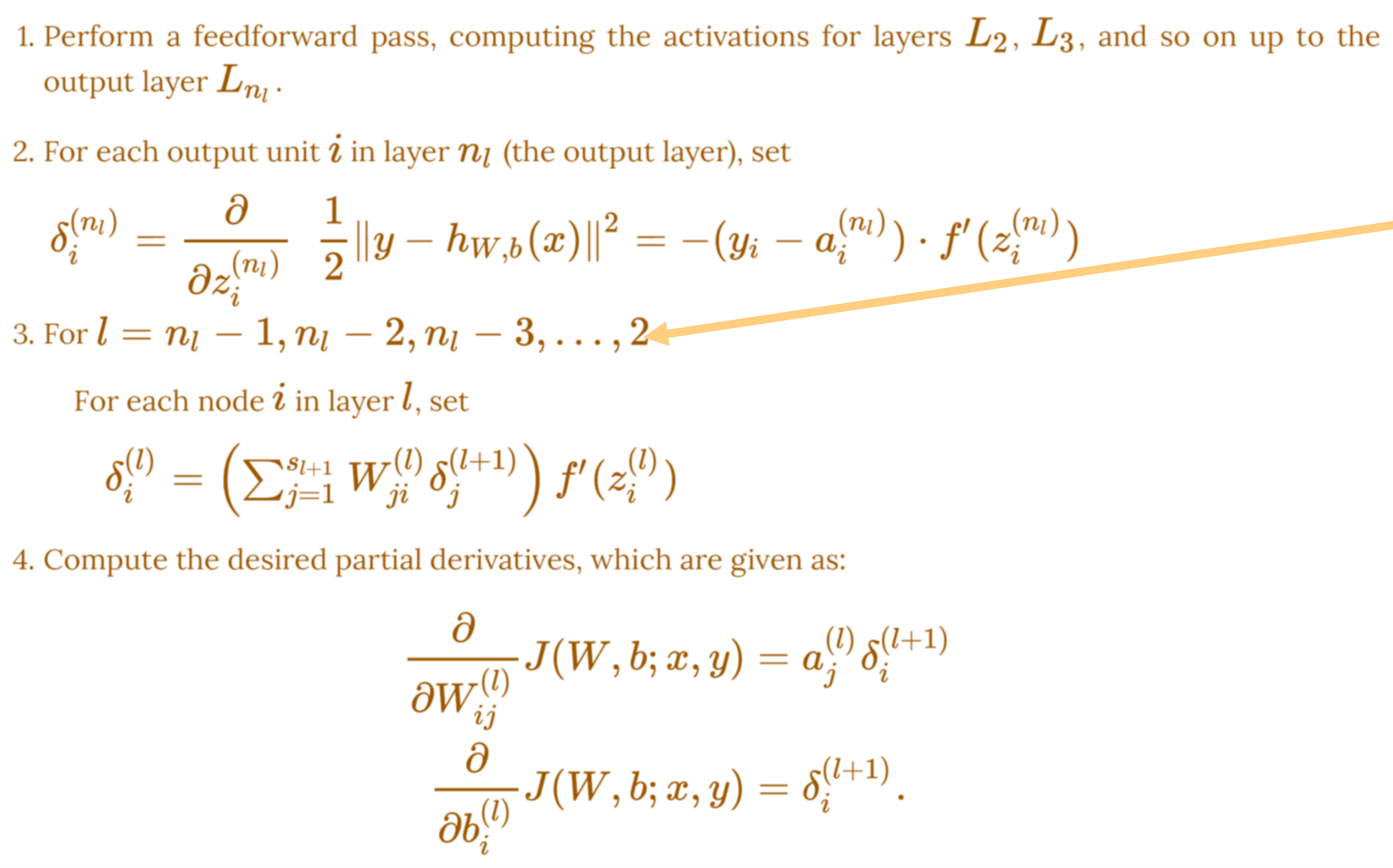
Backpropagation (backprop) is a special case of a type of algorithm called *automatic differentiation*. It calculates exact values of derivatives for given values of a function.

Backprop overview:

- Calculate the activations of the network for one given input/training example. This is the ***forward pass (from Input layer -> Output layer)***
- Calculate error using error function and minimize error.
- Error is sent back to each neuron, i.e., ***backward pass*** (from Output layer -> Input layer)

Finding the right parameters efficiently: Backpropagation

The backpropagation algorithm for a single training input



1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \; \frac{1}{2}\|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

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

Go from output towards input hence: ***backpropagation***

Backpropagation has been around for quite some time. What has changed that allows it be more widely adapted now?

Availability of powerful and flexible software libraries (that implements backpropagation and more general automatic differentiation algorithms) based on the notion of ***computational graphs***.

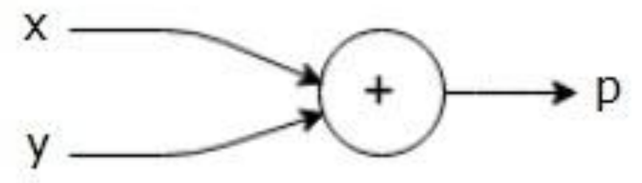
Efficient Backprop with Computational Graphs

A computational graph turns an equation or mathematical expression into a series of sequential **nodes** and **edges** representing the operations and values of each of the elements in the equation/expression. Traversing the graph in order is equivalent to calculating the mathematical expression.

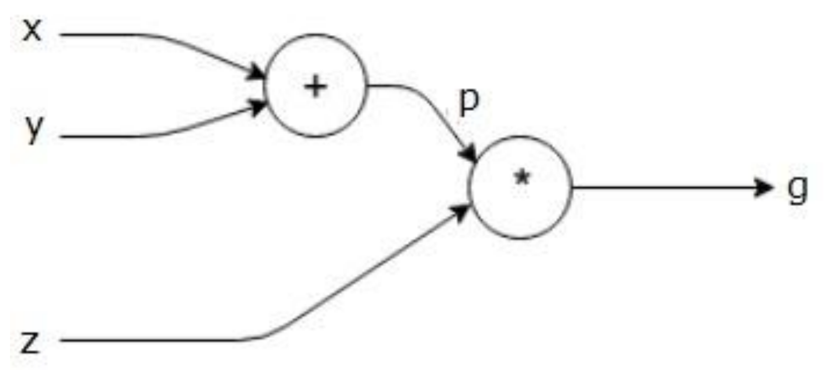
Consider an expression as follows:

$$p = x + y$$

The computational graph for it is:



$$g = (x + y) * z$$

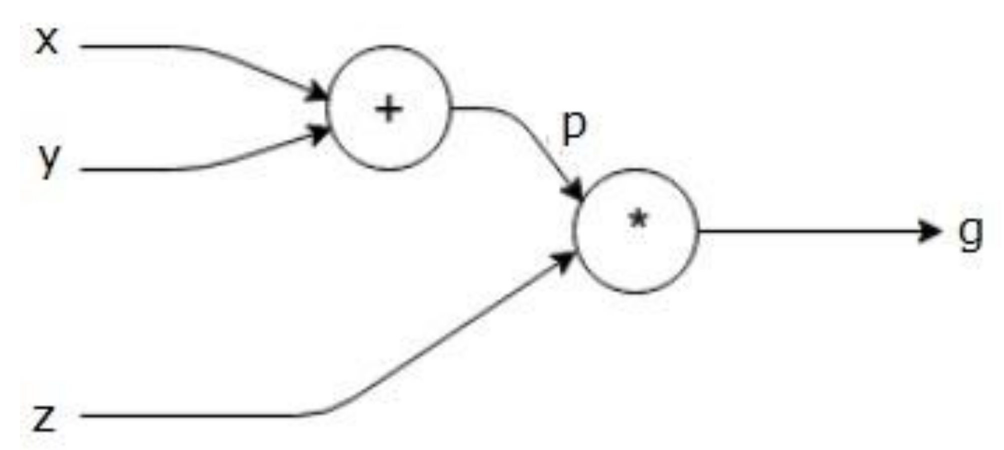


Just like in chain rule we define that *intermediate* variables that helps with the computation of the graph

Backpropagation: Computational Graphs Perspective.

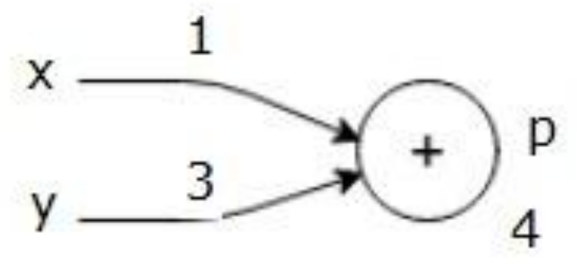
Forward pass for: *x*=1,*y*=3,*z*=−3

$$g = (x+y) * z$$



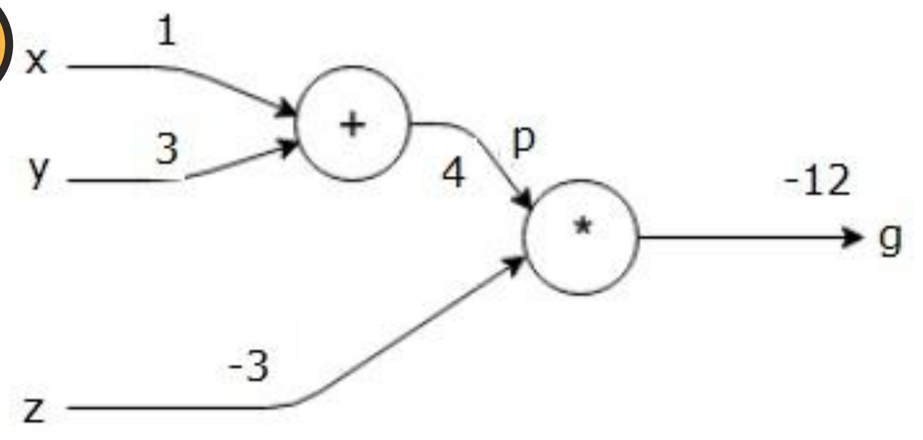
Go from left to right, forward

a



use the value of x = 1 and y = 3, to get ***p* = 4**

b



Then p = 4 and z = -3 to get ***g* = -12**

Backward pass

In the backward pass, our intention is to compute the gradients for each input with respect to the final output.

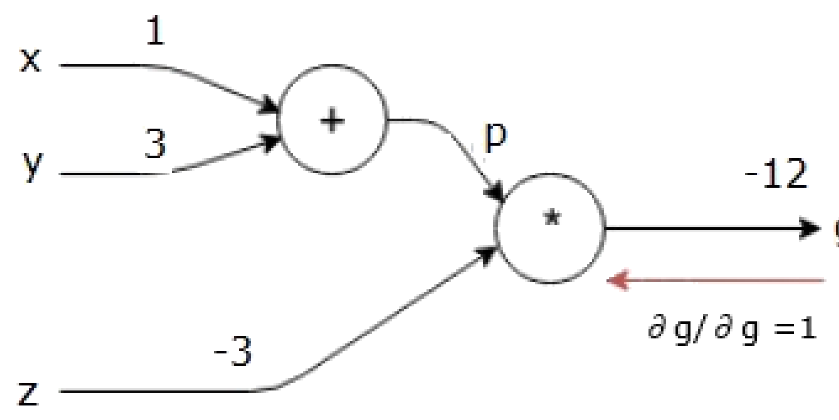
Desired gradients:

$$\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}, \frac{\partial g}{\partial z}$$

Go from right to left, backward pass

1

$$\frac{\partial g}{\partial g} = 1$$



2

Next, backward pass through the "*" operation. Calculate the gradients at p and z.

Since ***g* = *p**z** $\frac{\partial g}{\partial z} = p$ and $\frac{\partial g}{\partial p} = z$

From forward pass we know p = 4 and z = -3. Plug them in:

$$\boxed{\frac{\partial g}{\partial z} = p = 4} \qquad \frac{\partial g}{\partial p} = z = -3$$

3

We want to calculate the gradients at x and y. By chain rule

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial y} \text{ and } \frac{\partial g}{\partial x} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial x}$$

We know the **dg/dp = -3. *p*** depends on ***x*** and **y.** So **dp/dx** and **dp/dy** are direct.

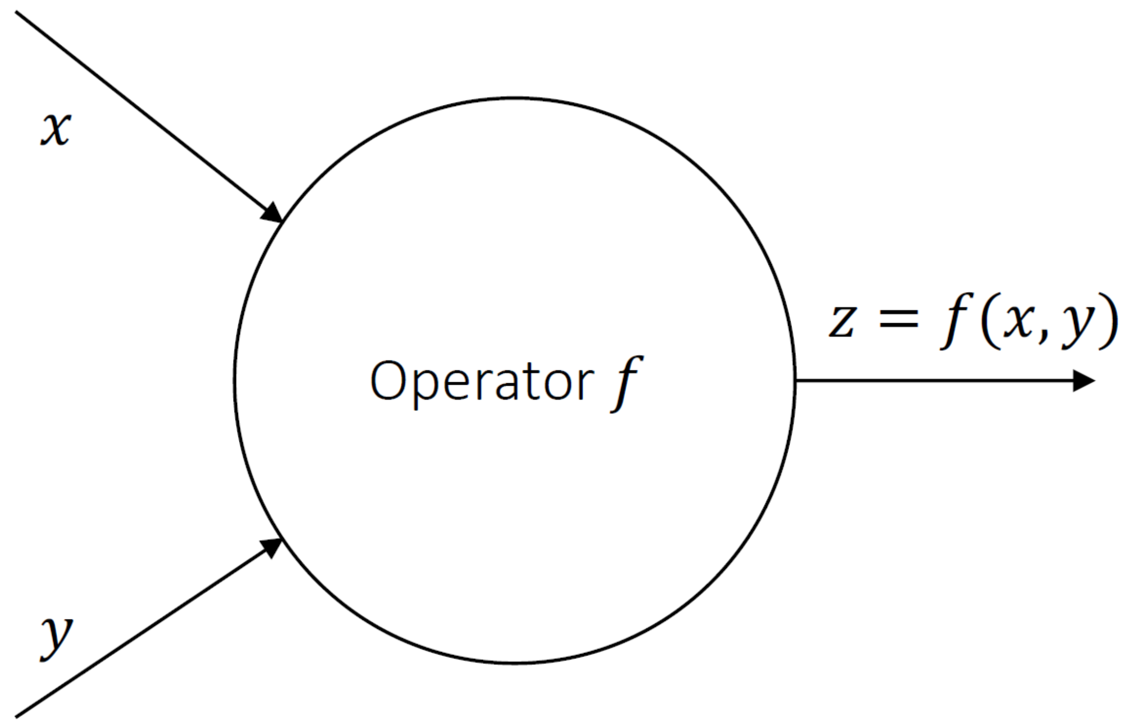
$$p = x + y \rightarrow \frac{\partial p}{\partial x} = 1, \frac{\partial p}{\partial y} = 1$$

$$\boxed{\frac{\partial g}{\partial x} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial x} = (-3)\,.1 = -3 \quad \text{and} \quad \frac{\partial g}{\partial y} = \frac{\partial g}{\partial p} * \frac{\partial p}{\partial y} = (-3)\,.1 = -3}$$

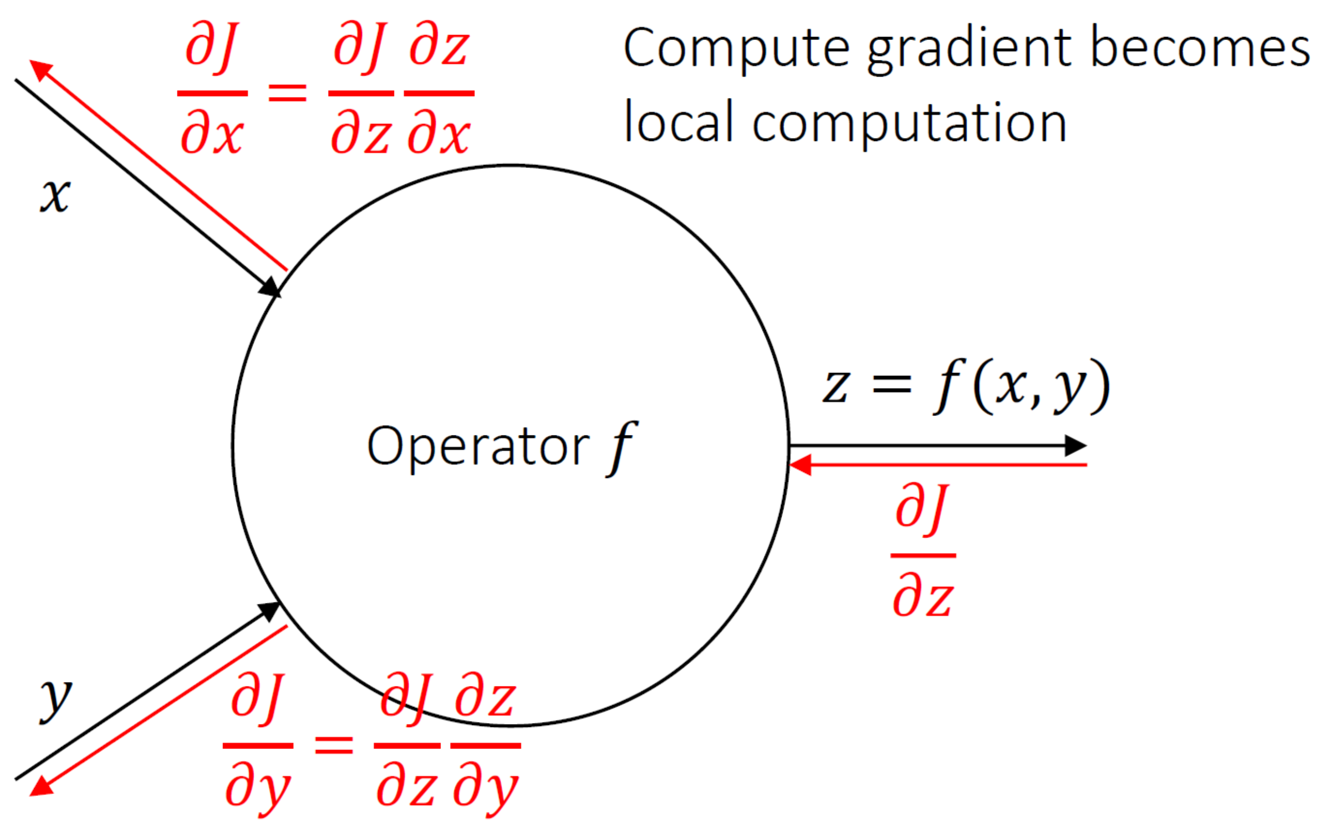
Backpropagation: Computational Graphs Perspective.

Computation at each graph node in 2 stages:

Stage 1: Forward pass



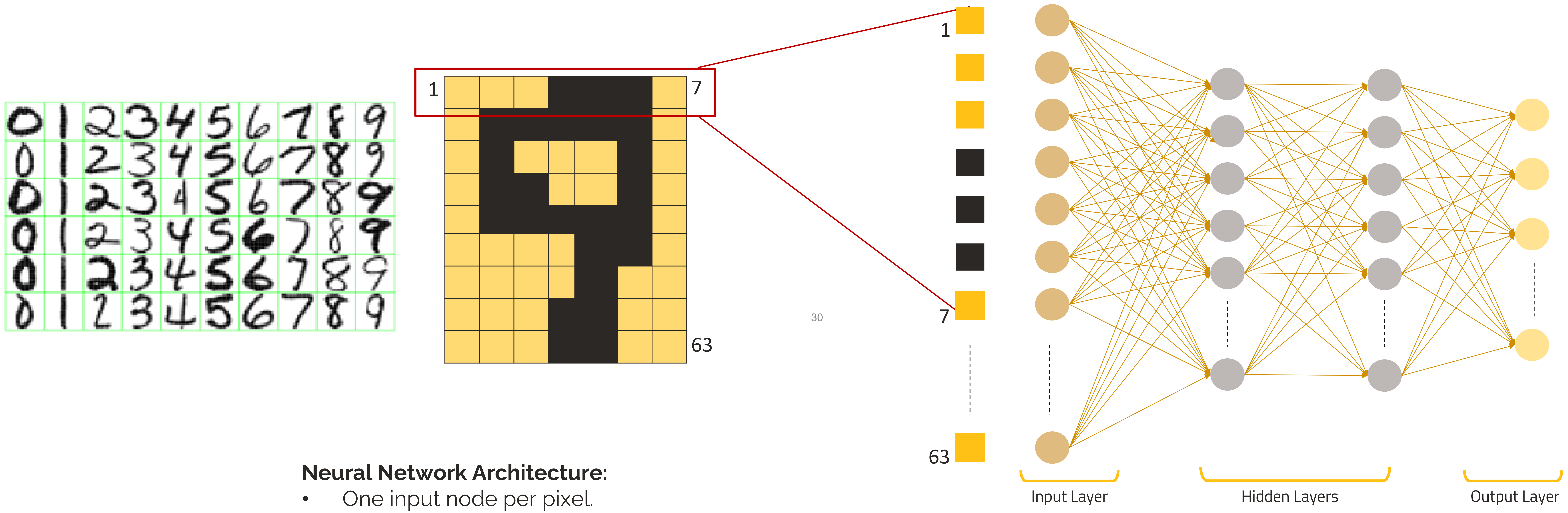
Stage 2: Backward pass



$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial z}\frac{\partial z}{\partial x}$
Compute gradient becomes local computation
x
Operator f
$z = f(x, y)$
$\frac{\partial J}{\partial z}$
y
$\frac{\partial J}{\partial y} = \frac{\partial J}{\partial z}\frac{\partial z}{\partial y}$

Neural Networks: Architectures

Network Architecture and Features

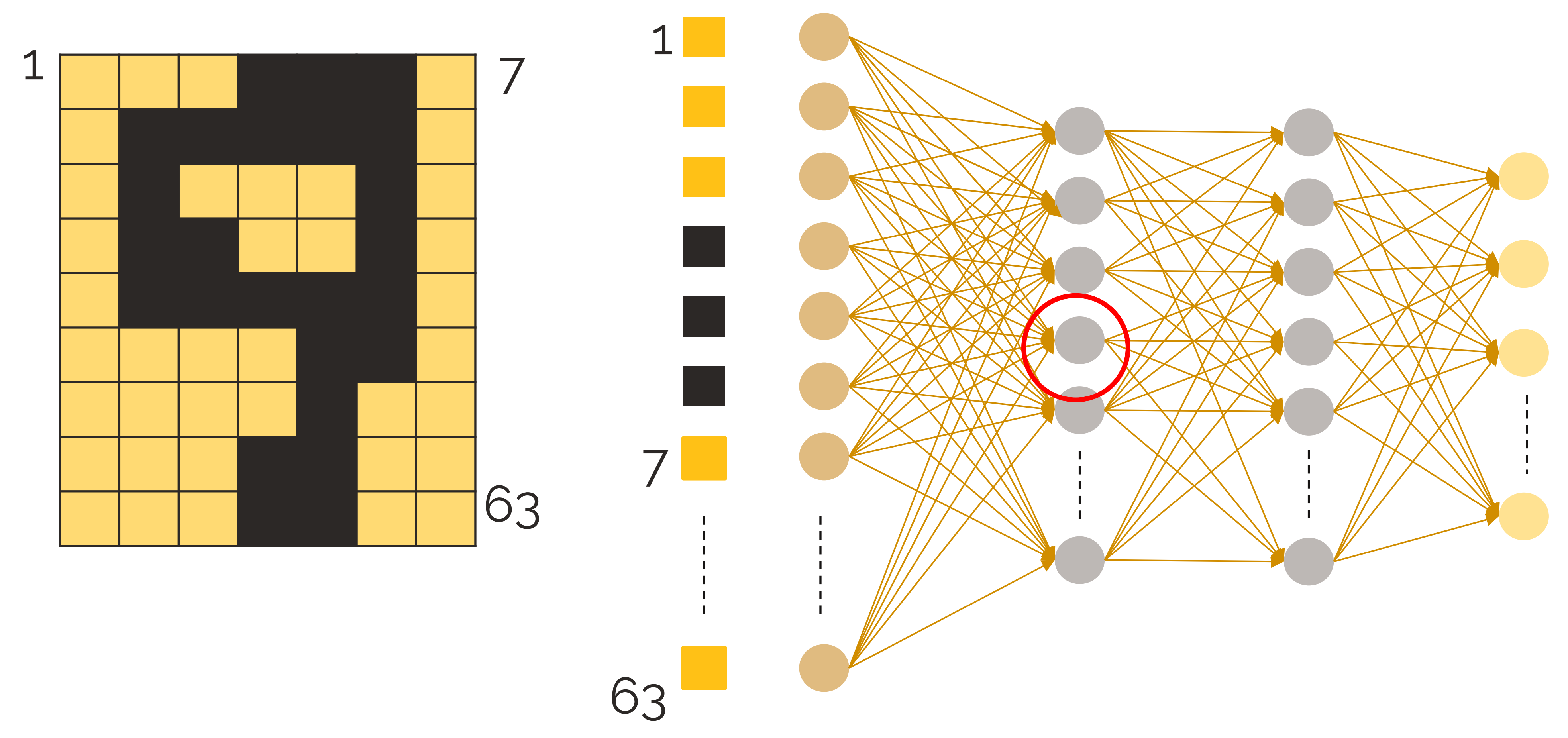


Neural Network Architecture:

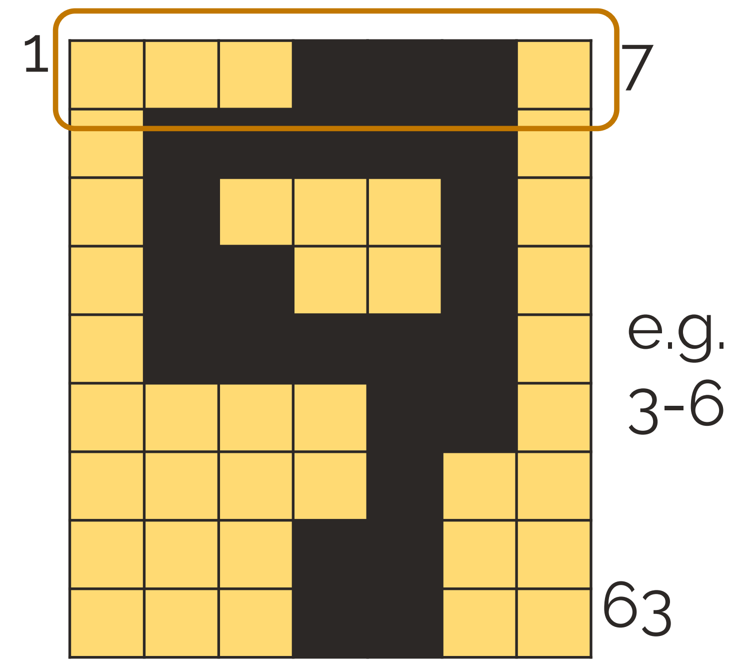
- One input node per pixel.
- Two hidden layers
- One output layer
- Nodes in one layer is only connected to nodes in next layer ***(feed forward).*** No backwards connections or layer jumping connections.
- Every node is connected to all nodes in the next layer i.e. ***fully connected***

ANNs: Network Connection Weights

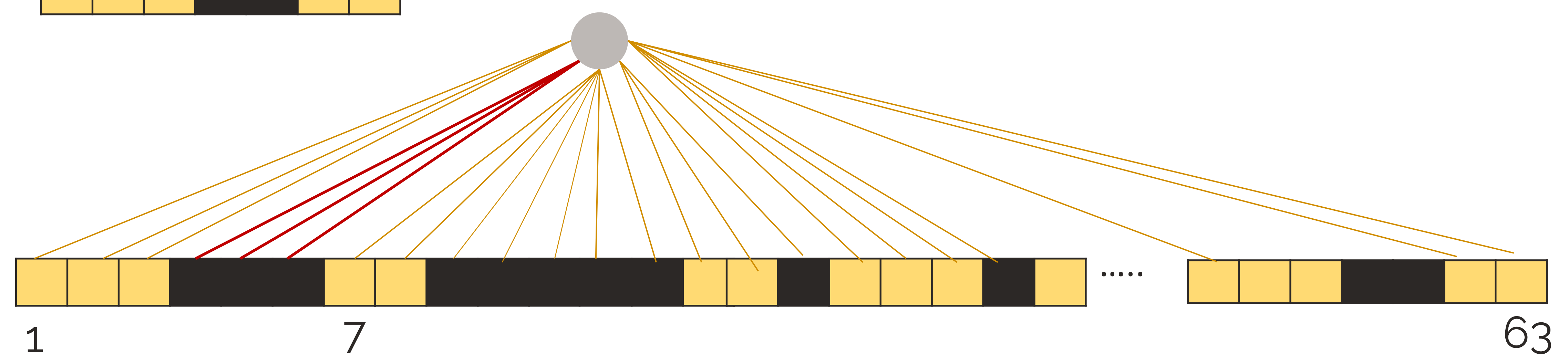
How does the AI learn this is a 9 or g?

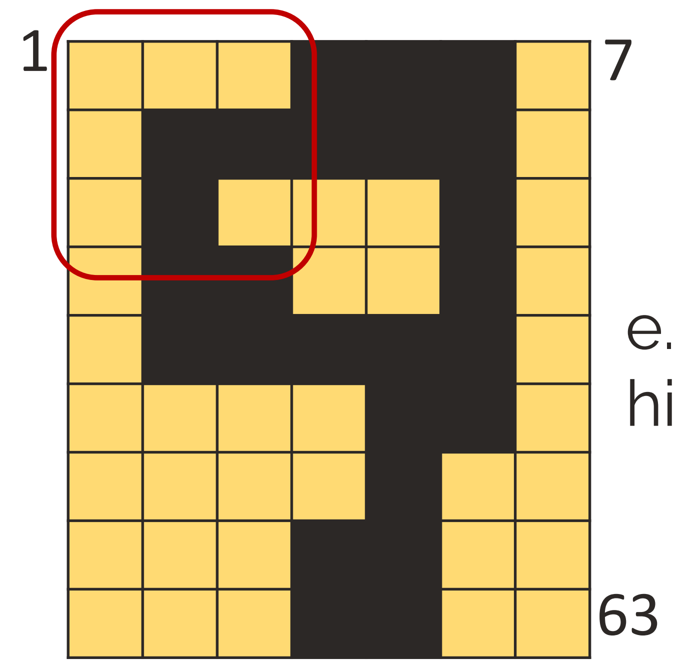


Each hidden unit has pattern of connection/weights that abstracts a certain aspect of the input. They act as self-organized pattern detectors.

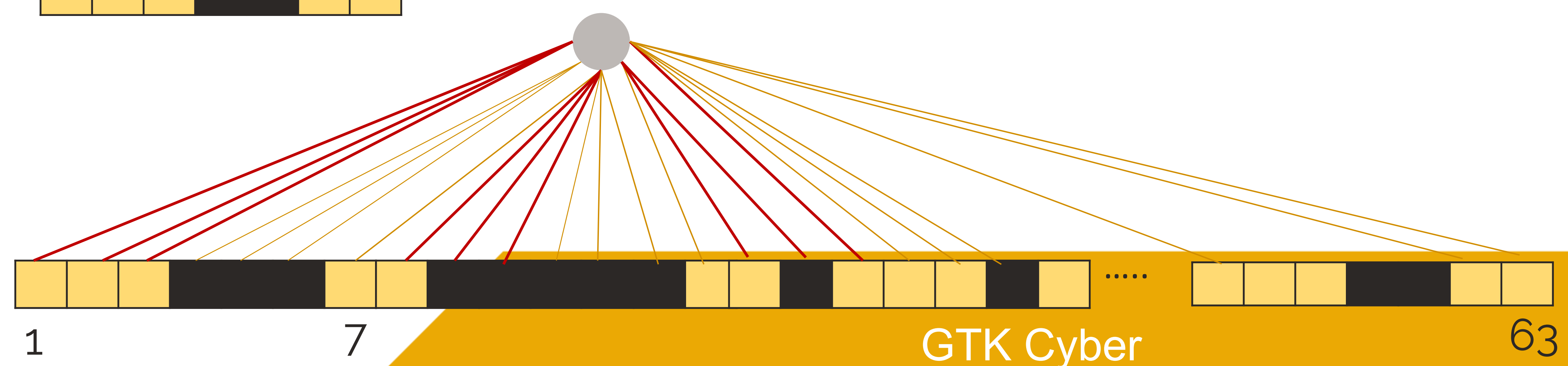


e.g. This particular unit has strong connection for inputs 3-6 (Red lines). Will send strong signal out for first row





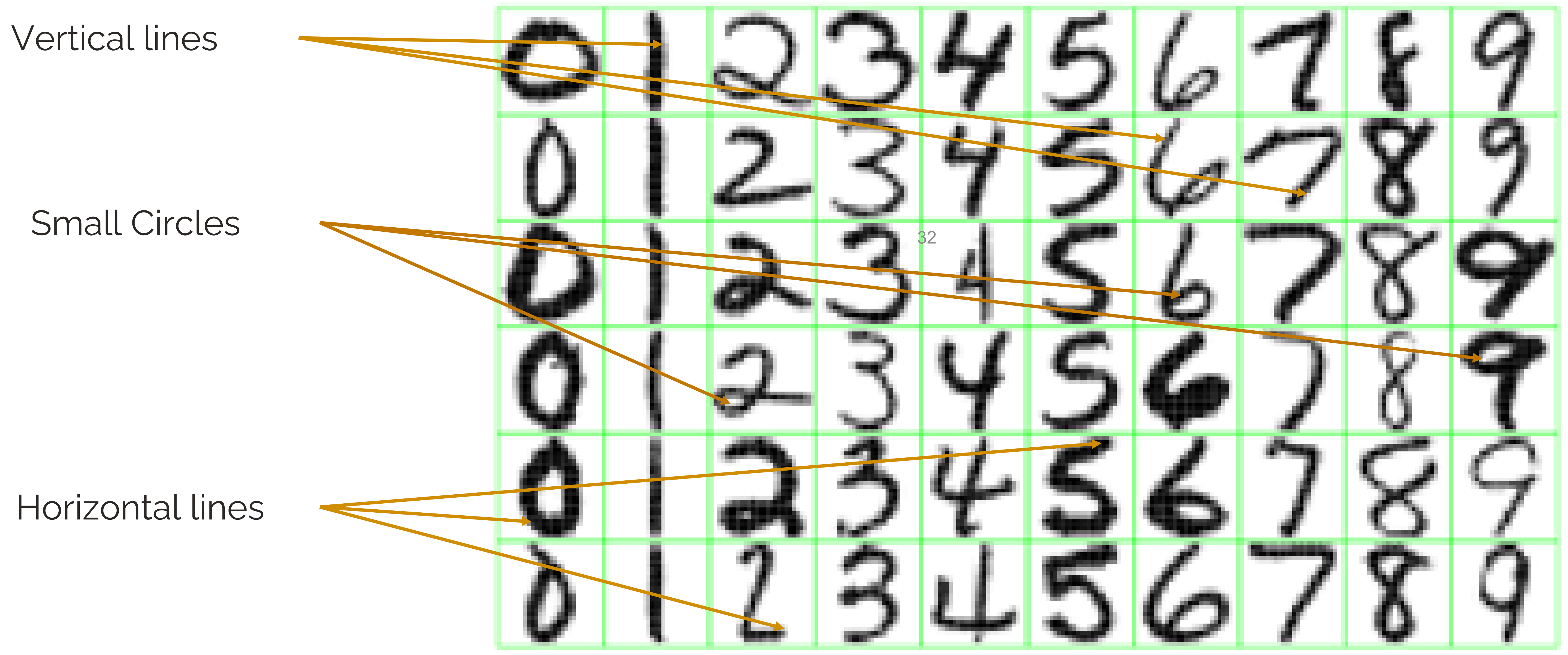
e.g. This particular unit will send strong signal out for the highlighted area.



ANNs: Layers

What type of feature are recognized by the groups of units or layers ?

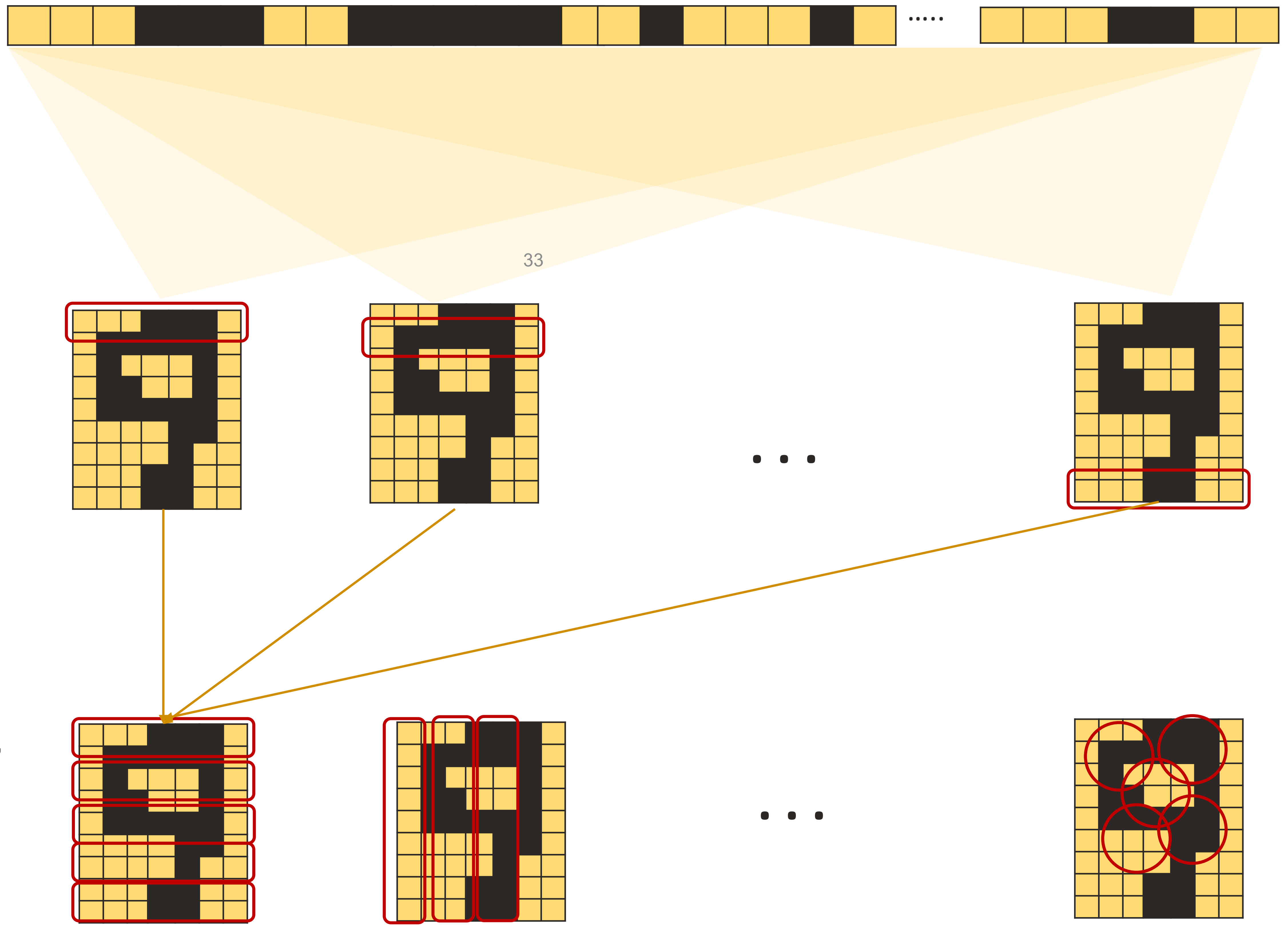
Basic features detected by initial layers.



ANNs: Feature Detection in Layers

What type of feature are recognized by the units ?

Higher order features detected by subsequent layers



Initial layer units: e.g. detect lines in specific positions

Higher level detectors (horizontal line, 'upper loop', etc...

Deep Neural Networks: Two Architectures

- We'll work on two types of networks
  - Convolutional neural networks (CNNs)
    - Mostly used for images and static data
  - Recurrent Neural Networks (RNNs)
    - Mostly used for sequences and time stamped data.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks:

- Developed primarily for image processing; ***takes in hi-dimensional images, creates low-dimensional intermediate representations*** for classification

Handling hi-dimensional patterns in CNNs: Two main ideas

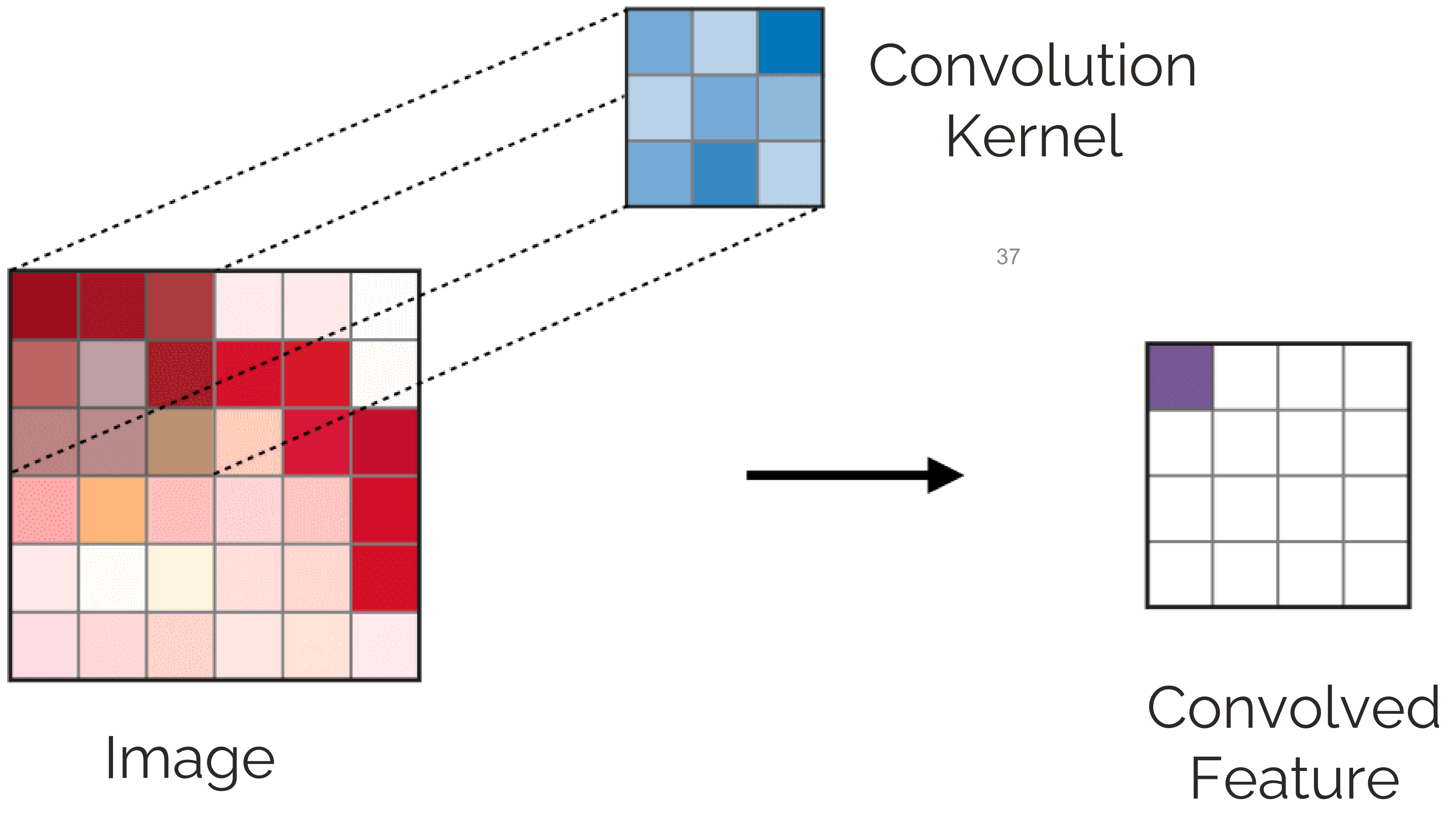
1. ***Filter the high-dimensional information in space.***
   a) Information squeezing/extraction. Typically, **convolution** + **pooling** in space.

2. ***Have a subset on 'neurons' learn that part of the input.***
   a) Divide and conquer. Move away from fully connected architecture. Have a subset focus on the spatially filtered information.

CNNs Idea 1 : Handling hi-dimensional patterns space

1. Filter the high-dimensional information in space: Convolution + Pooling
   i. **Convolution (feature extraction) +**

Convolution Example:

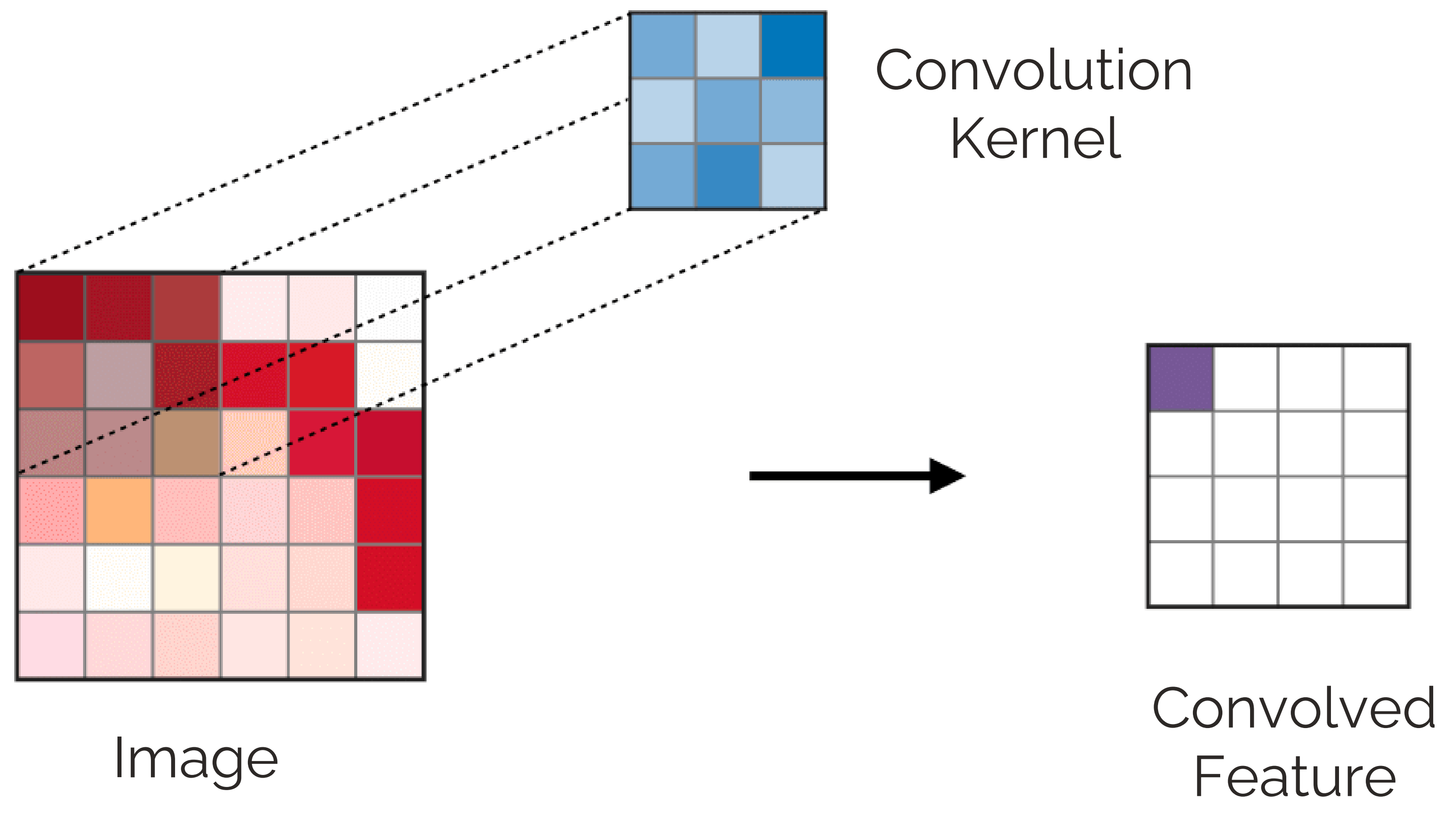


https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

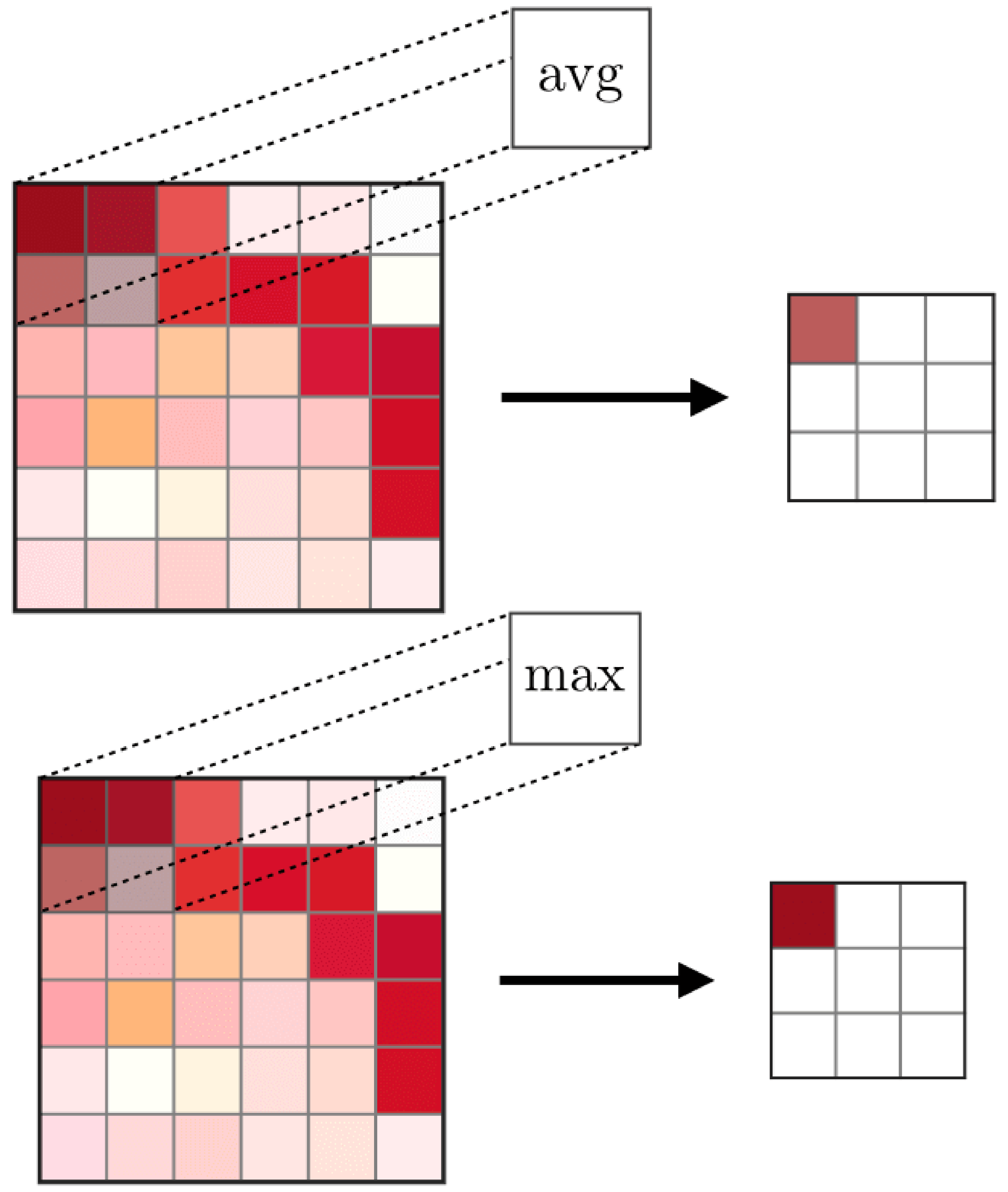
CNNs Idea 1 : Handling hi-dimensional patterns space

1. Filter the high-dimensional information in space: Convolution + Pooling
   i. **Convolution (feature extraction) +**

Convolution Example:



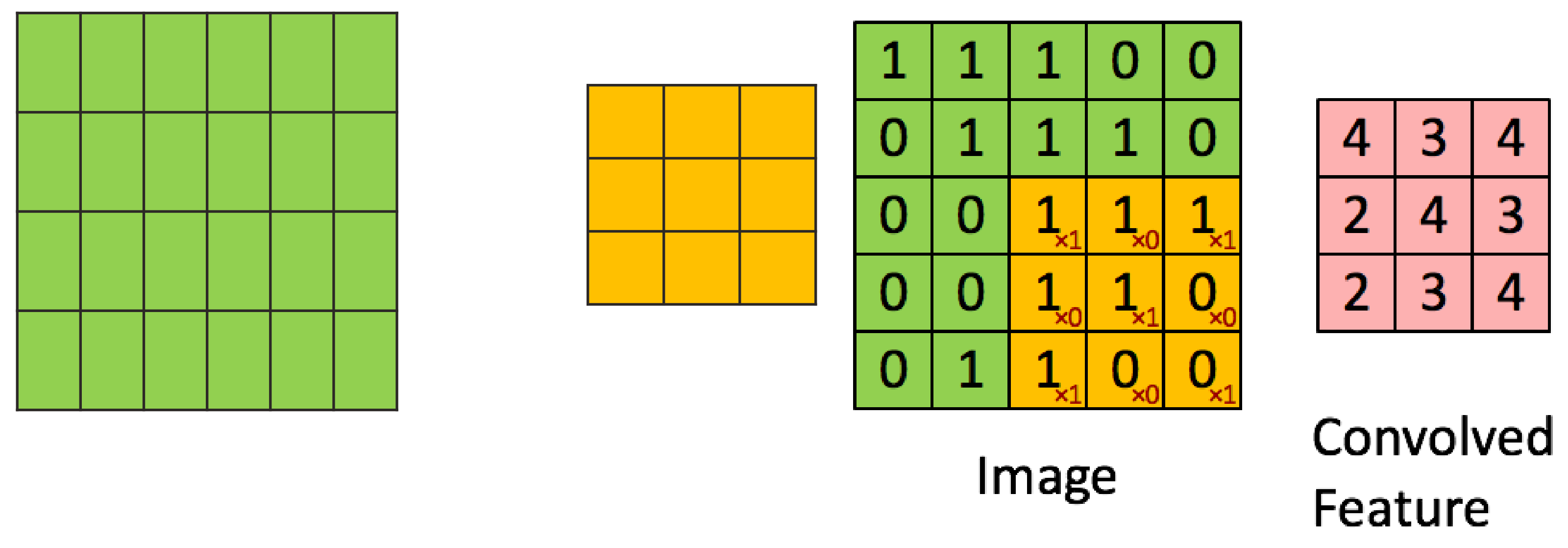
ii. **Pooling (reduce information further).**
   a) Filter the output of the convolution further.
   b) Typically uses a spatial kernel that calculates maximum or average

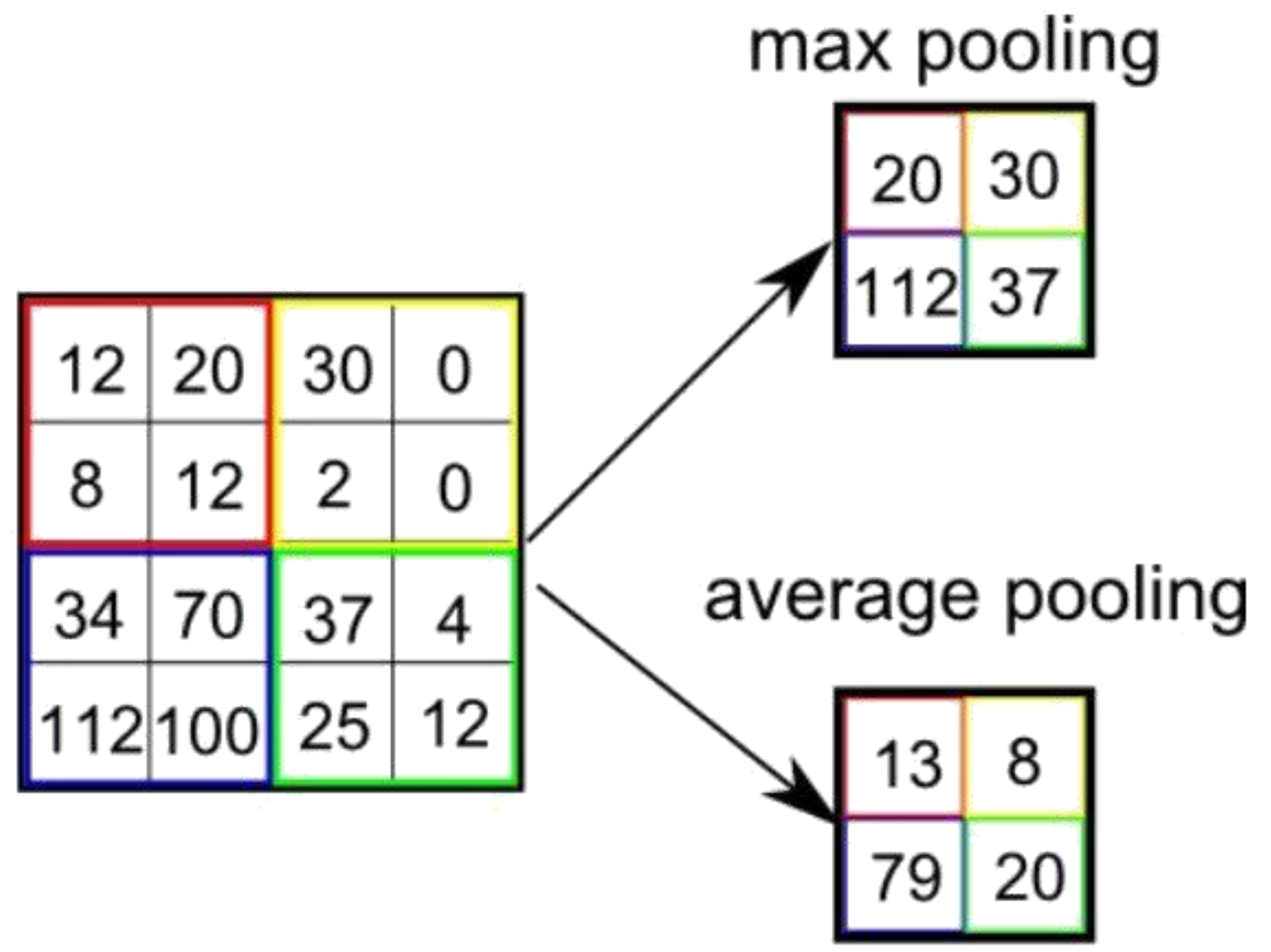


https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

CNNs Idea 1 : Handling hi-dimensional patterns space

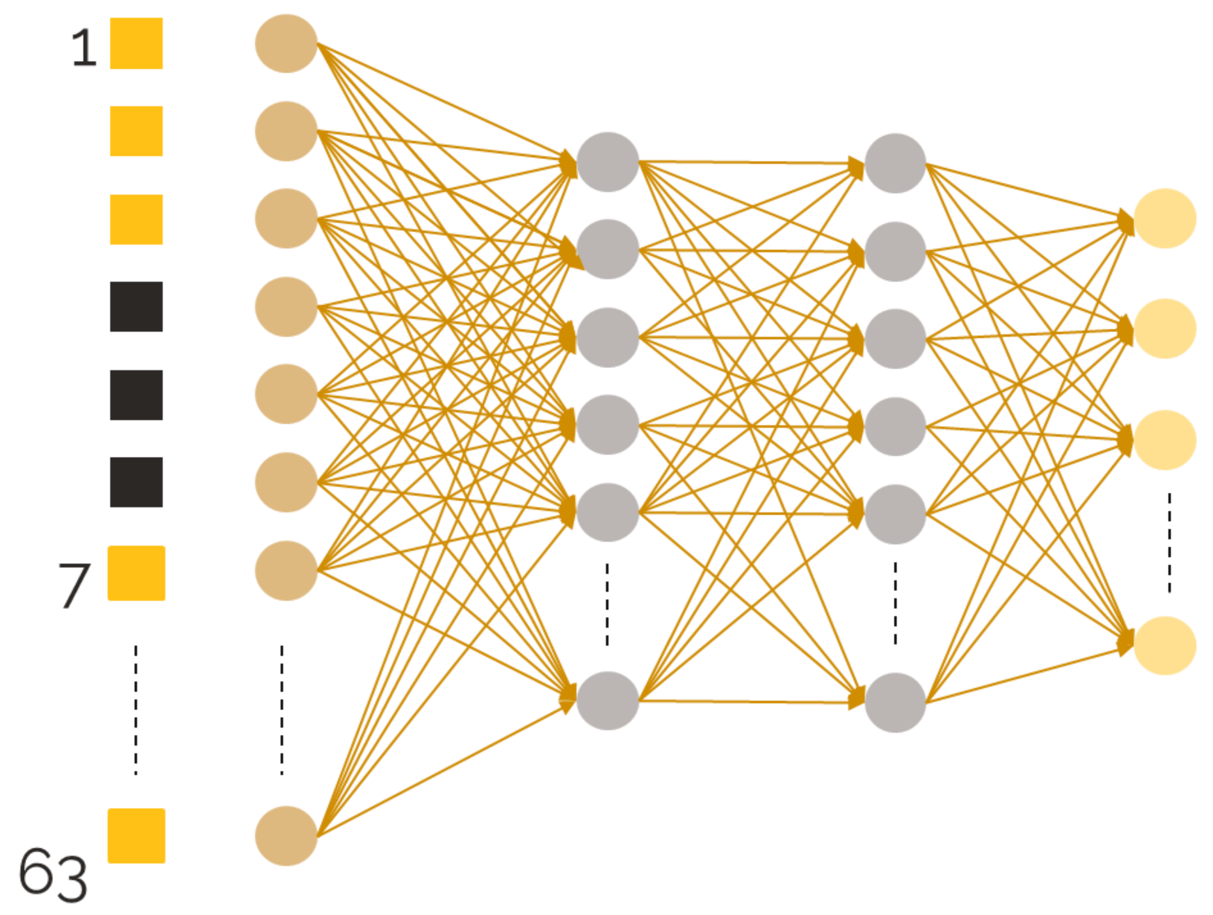
1. Filter the high-dimensional information in space: Convolution + Pooling
   - **i. Convolution (feature extraction) +**
   - **ii. Pooling (reduce information further).**
     - a) Filter the output of the convolution further.
     - b) Typically uses a spatial kernel that calculates maximum or average





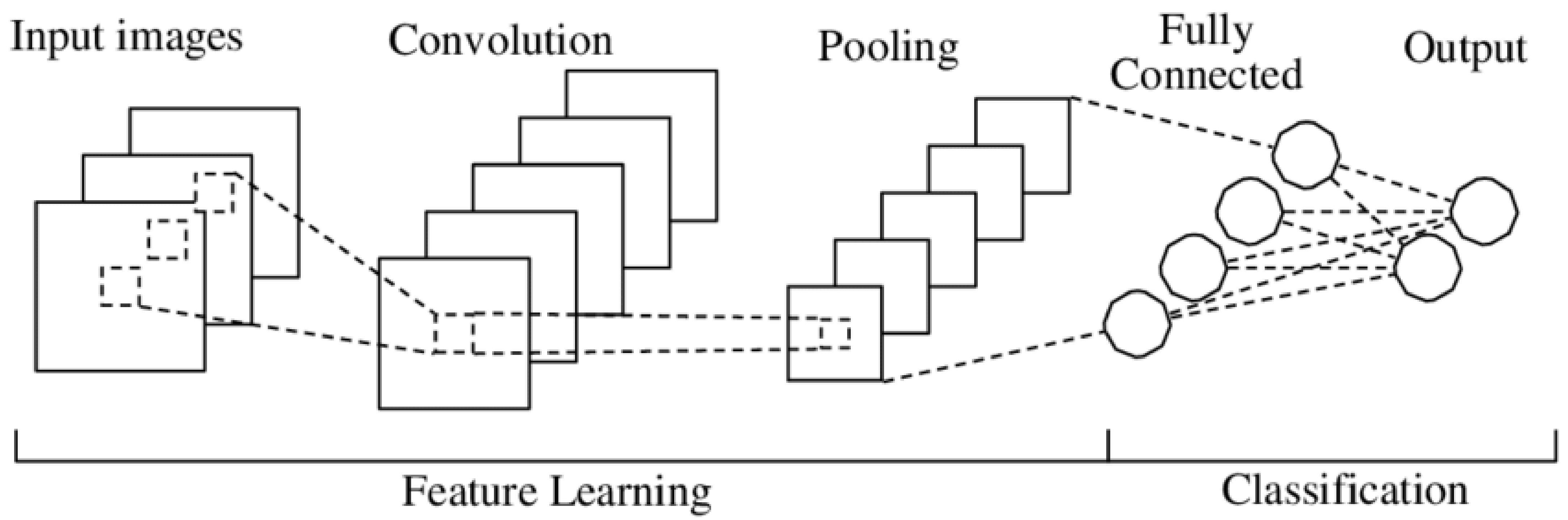
Handling hi-dimensional patterns in CNNs: Idea 2

CNN architecture: Putting it all together



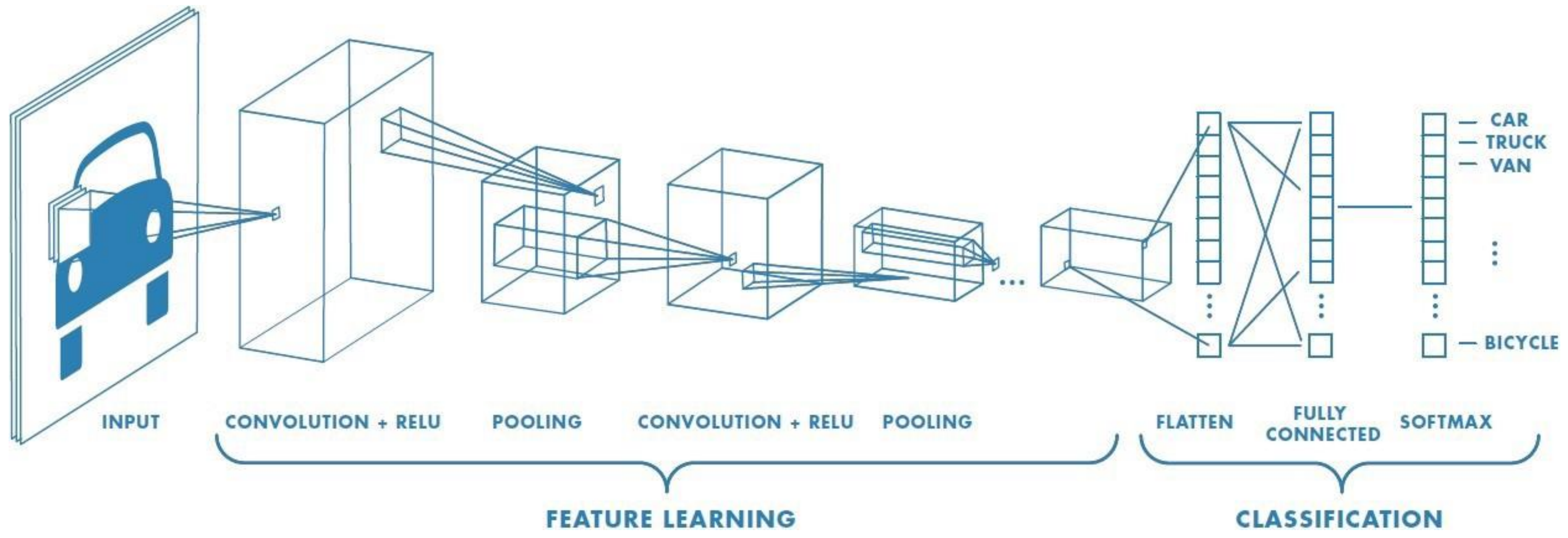
This is what we were looking at before; i.e, it is feedforward, fully connected ANN

But in CNNs we want a subset of 'neurons' learn a part of the input. Divide and conquer. Move away from fully connected architecture.



Start with image, filter (convolution, pooling), then feed that to a fully connected (FC or dense) neural net. This can be chained or repeated as necessary.

Convolutional Neural Architecture



https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

https://medium.com/@curiousily/tensorflow-for-hackers-part-iv-neural-network-from-scratch-1a4f504dfa8

CNN Lab

Deep Learning for Everyone!

Nice to have!
Lots of GPUs!!!

Backend C - Frontend Python
(Released 2015 by Google)

Keras - Python
(High-level wrapper API)

Alternative backend (theano, torch) and high-level APIs (Lasagne, Caffe, Pylearn2, Chainer)

Terminology

Deep Learning is performed in an iterative fashion! This means only part of all training data is used at a time.

- **Batch Size:** number of training examples in one forward/backward pass aka how many URLs are we training on here at a time (limited by RAM).
- **Iterations:** number of passes, where for each pass "batch size" number of training examples are used.
- **Epoch**: one forward and backward pass of "all" training data.

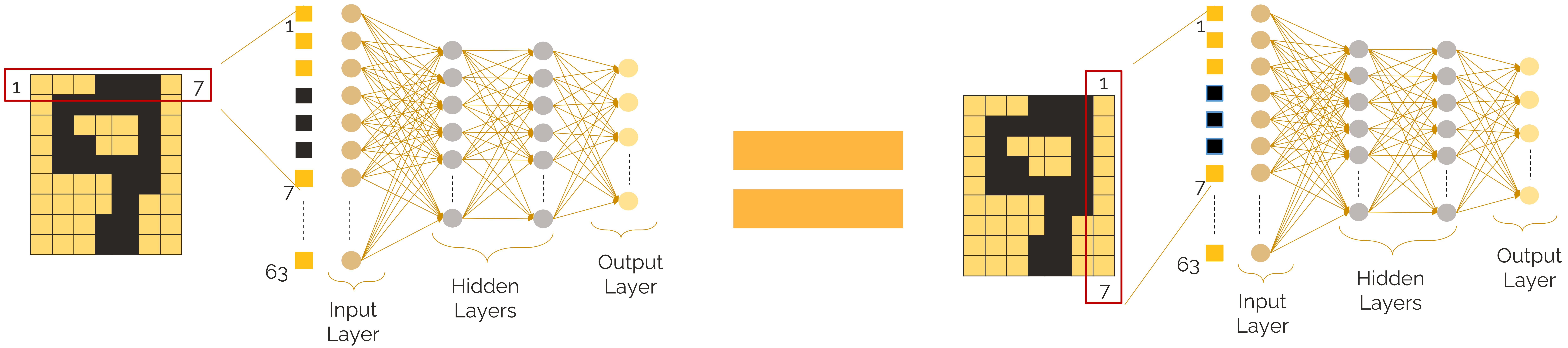
Example: For 1000 training examples (here URLs) and batch size of 500 it will take 2 iterations to complete 1 epoch. For Keras you have to provide batch size and number of epochs!!! Batch size of 32 is commonly used!

Recurrent Neural Networks

Data sequences and data in time

So far, we have been dealing with data that is temporally/sequentially invariant. i.e. no specific relationship in time But many data are time-dependent (e.g., stock-market, logins, server-logs) or sequences (e.g. language, videos).

For image processing/classification the following two networks are equivalent.
i.e. any sequential relationship is not captured.

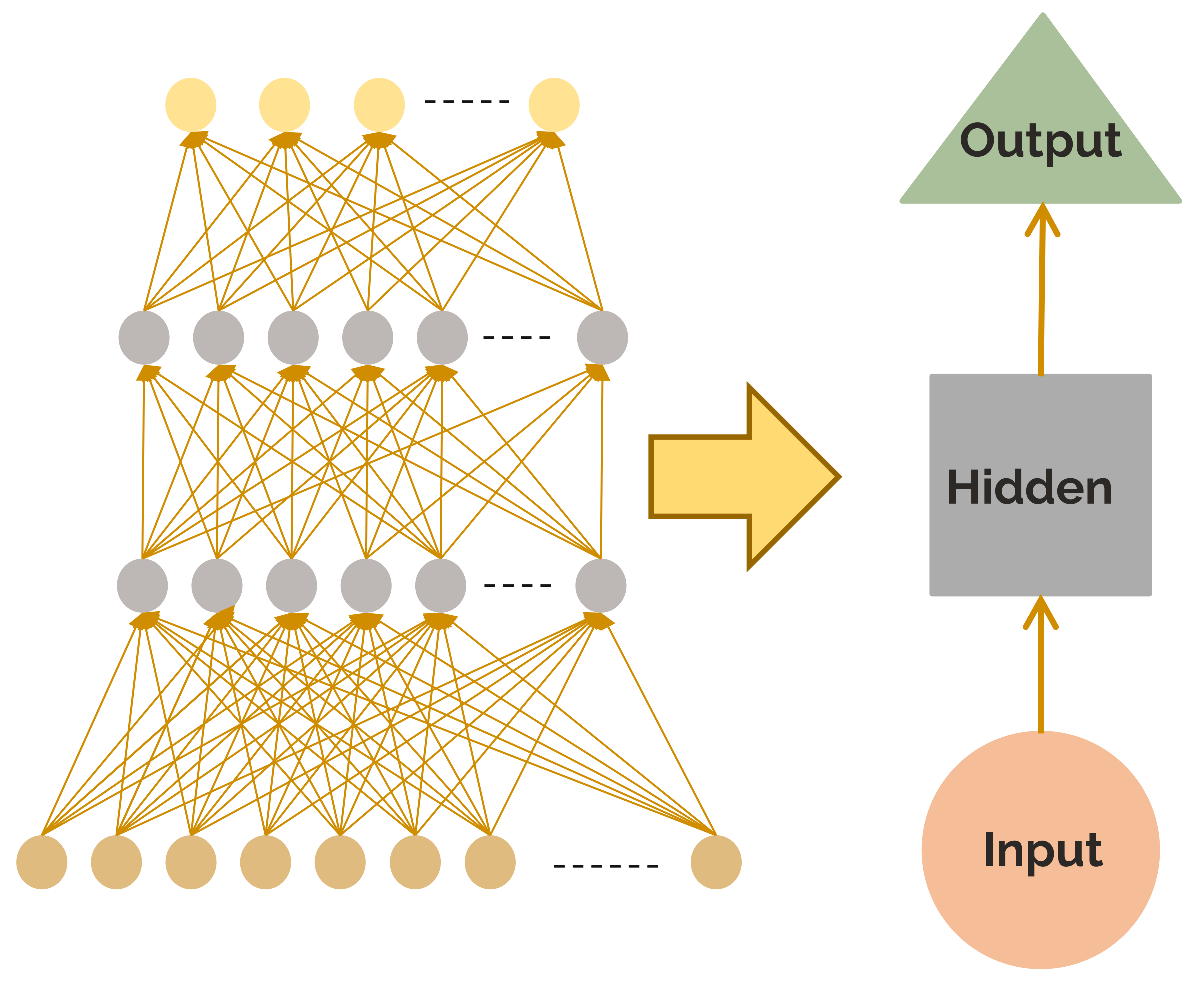


The class of neural networks that can parse sequential data is called ***recurrent neural networks (RNNs)***

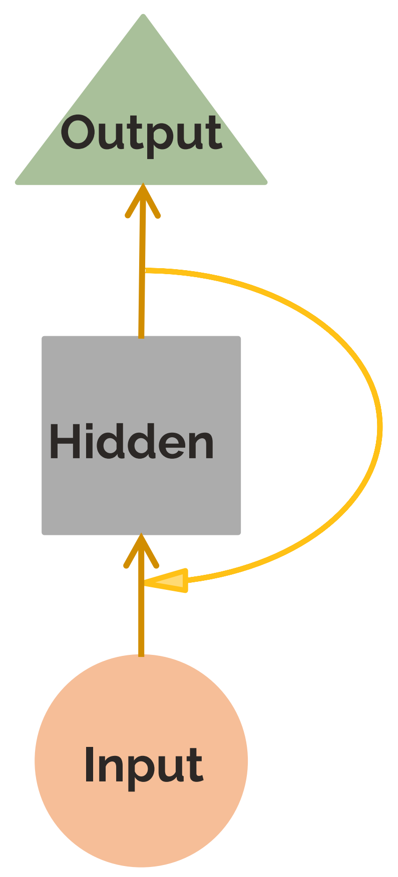
Recurrent neural networks (RNNs)

Recurrent neural networks (RNNs) are a class of neural networks that have both ***feedforward*** and ***feedback*** connections.

Feedforward Neural Networks



Recurrent Neural Networks



Information flow is unidirectional. Progresses from input layer to output layer

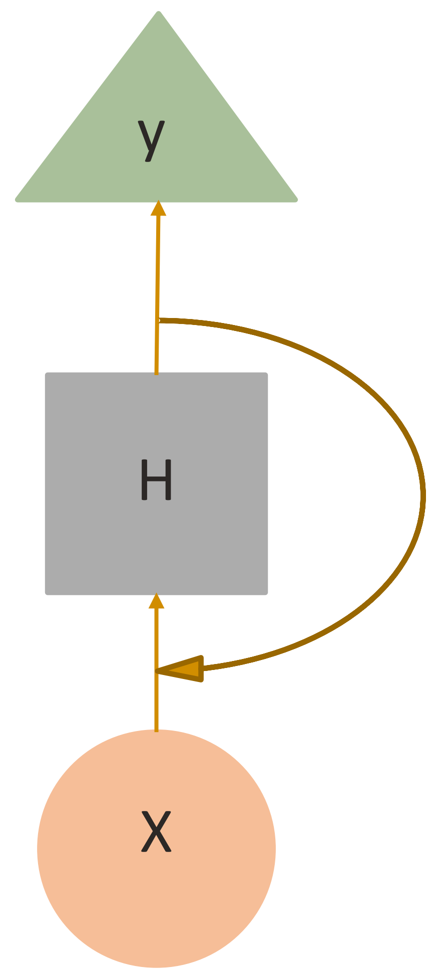
Cannot handle sequences or variable length inputs. Example text processing

To overcome this we can introduce loops in the hidden layers.

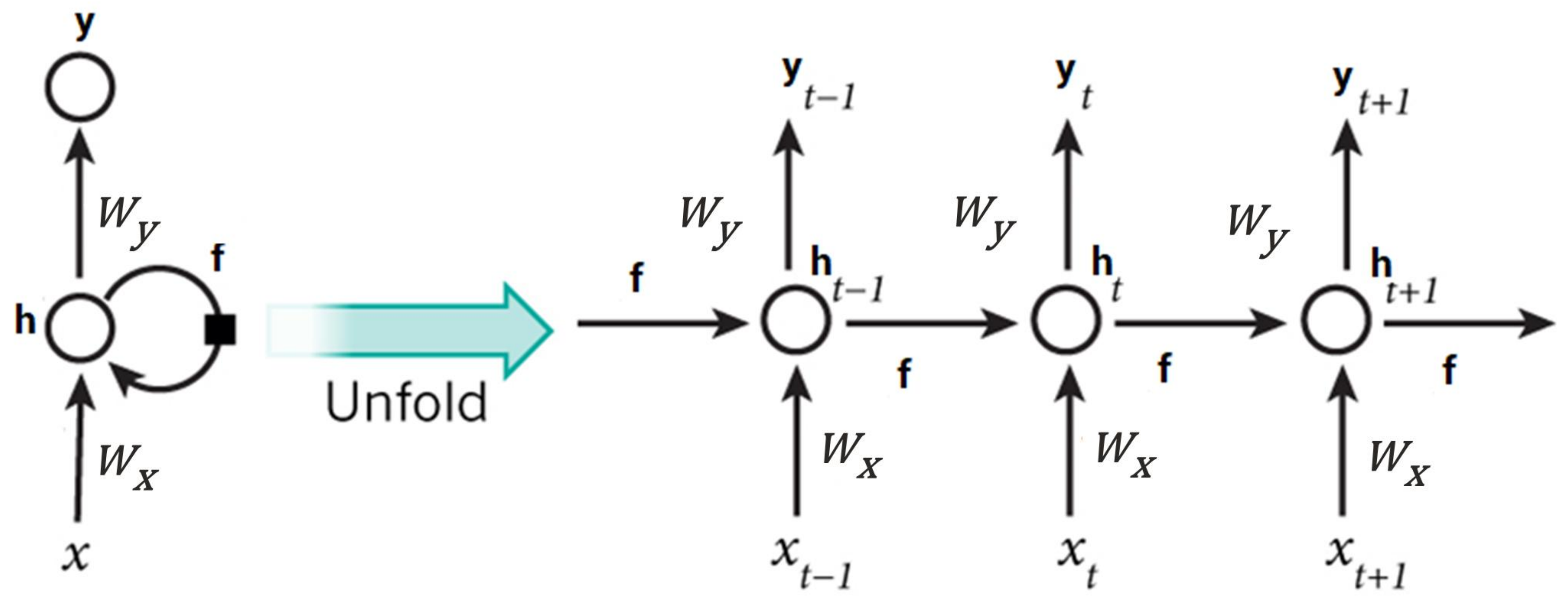
At each step in the sequence the hidden layer can feedback its intermediate output as new input to itself till the sequence is done.

So hidden layer input at step ***t*** is dependent upon step ***t-1*** and so on. This type of relation is called recurrence relation. Hence recurrent network.

Recurrent neural networks (RNNs)



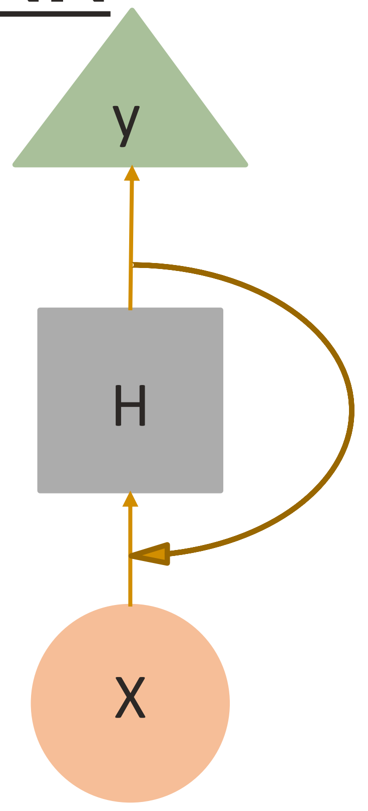
We can create an "unfolded" feedforward version of basic RNN

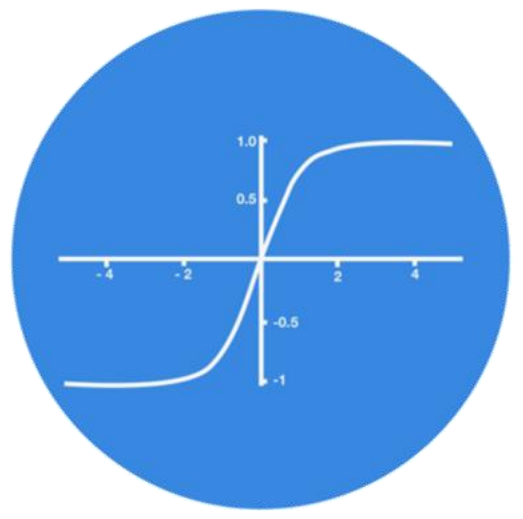


- Each step/loop can be thought of as "memory" of the network (by modifying the weights). *Memory* is meant as the influence of history of the sequence on the current step.
- How many steps to loop over decides what the memory length is (or how much back in steps/time the current step is influenced by).
- Basic RNNs become unstable as the memory length increases. This is a technical problem. With each loop you have to calculate the gradient. This can introduce numerical issues during training.

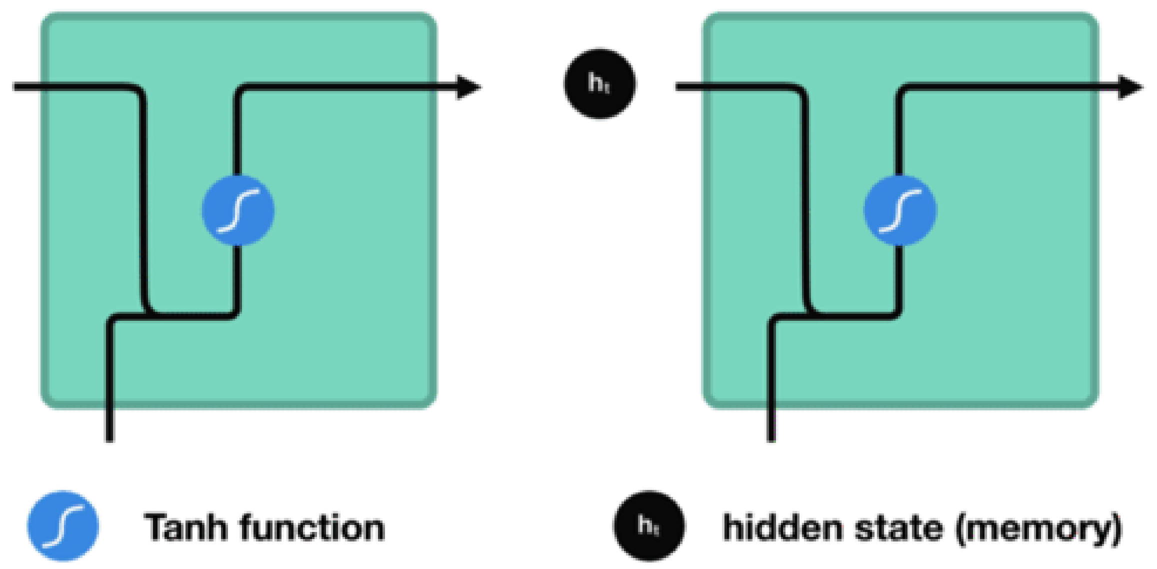
RNNs : Gates

Basic RNN



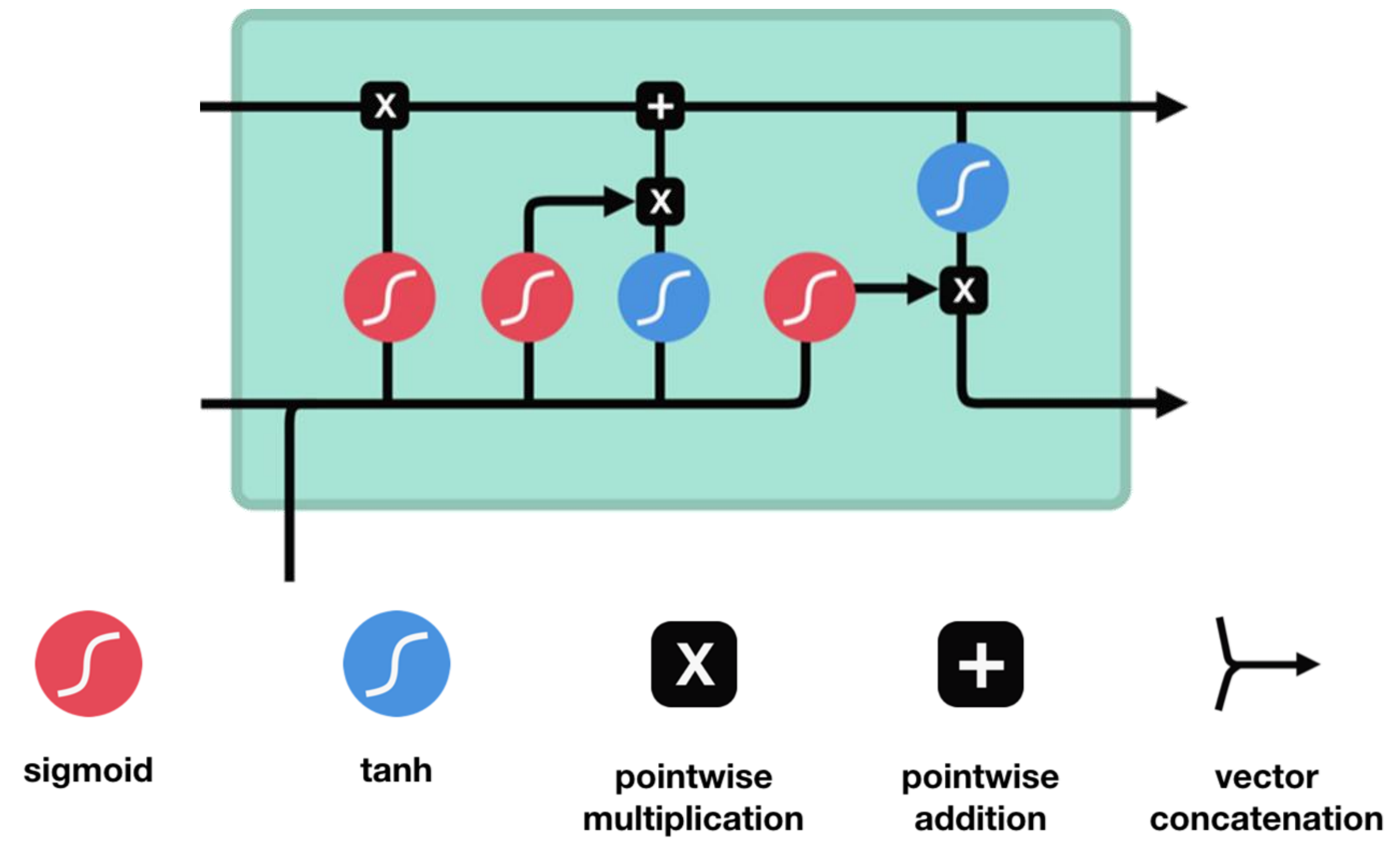


tanh activation



- In a basic RNN each layer and unit is made up of artificial neuron with ***one activation function*** e.g. tanh
- Each node processes its input and passes the result of the the hidden state to node in next layer (step in loop)

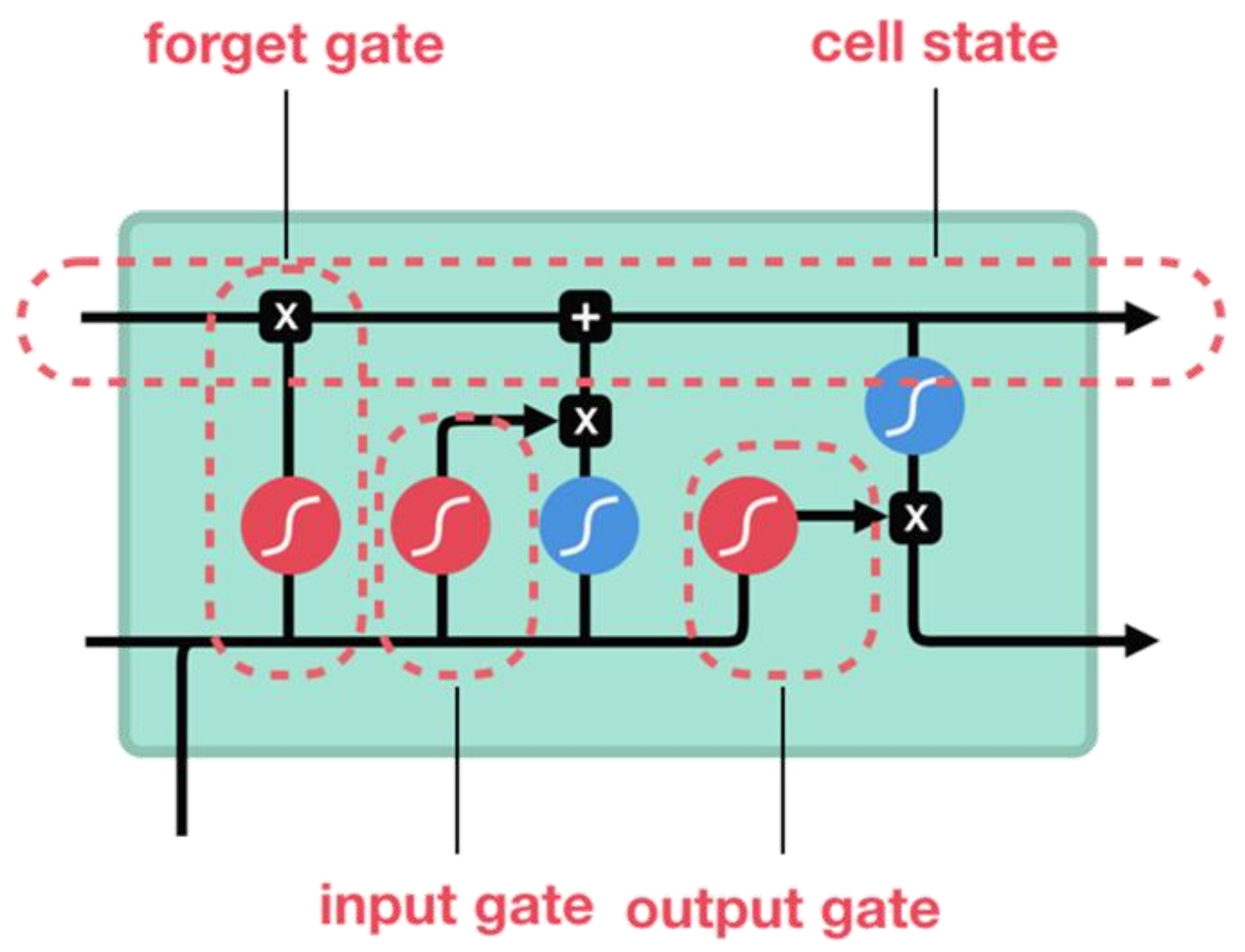
Modified RNN for longer steps/memory

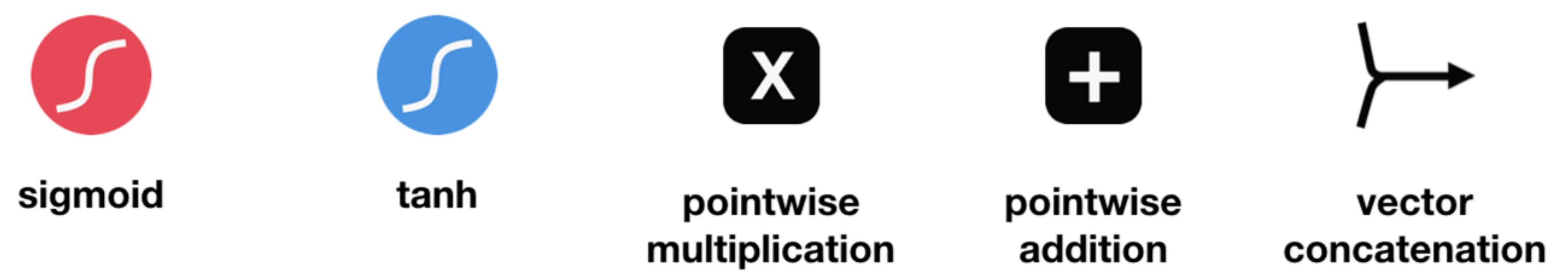


- In RNNs modified for longer memory the ***basic unit has multiple functions*** within each neuron/cell (e.g. sigmoid, tanh etc)
- These functions and their outputs are combined algebraically (addition, multiplication etc) to produce the output of the neuron/cell
- Since each cell in now more complex there are more functions and parameters and hence more complexity. But the complexity gives more flexibility and control at each cell.
- The functions of a given cell selectively modulate/scale the input from the previous cell and thus ***act as gates for how the information from a previous state is allowed to go forward.***

Common RNNs with Gates

Long short-term memory (LSTM)



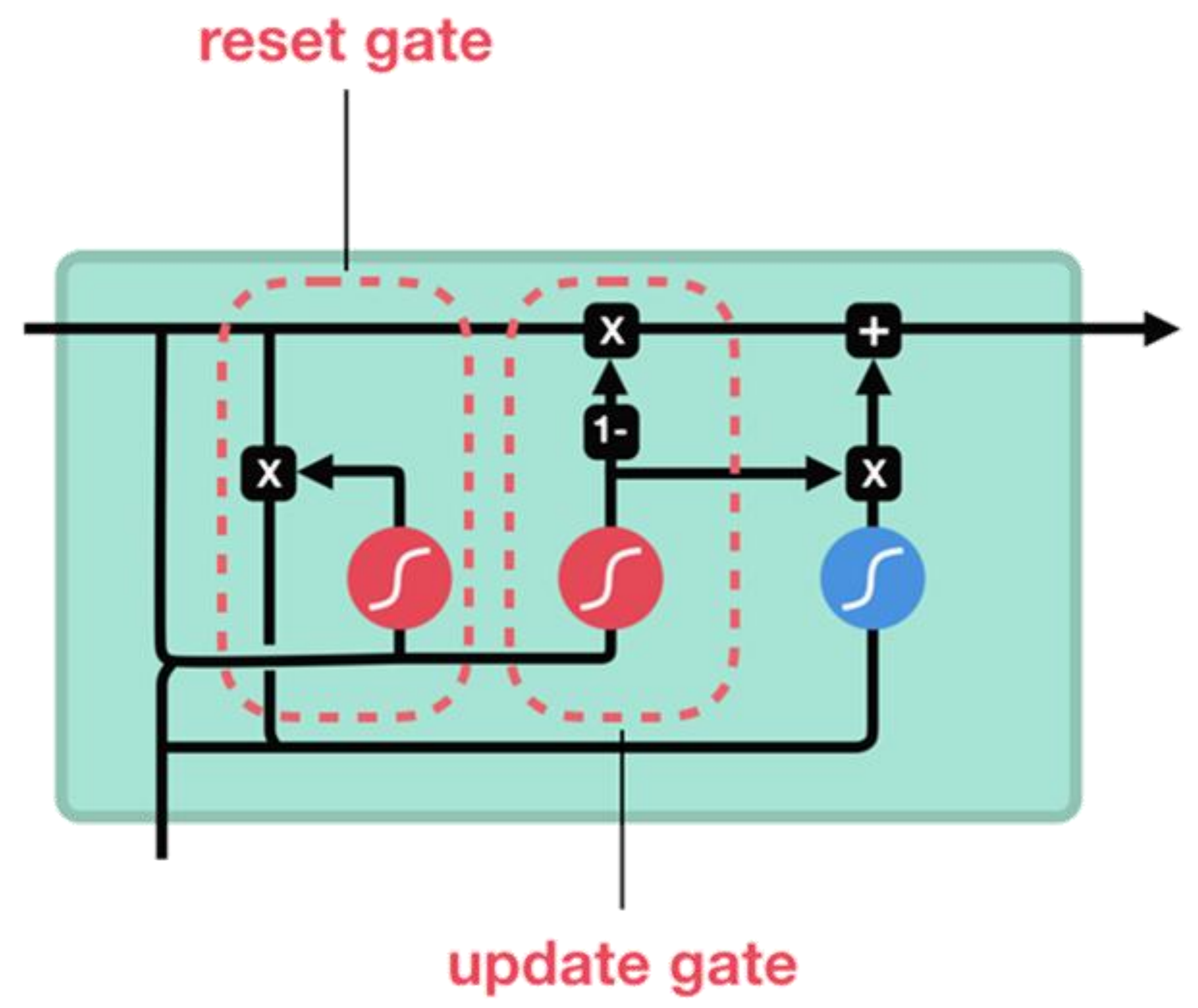


LSTMs utilize for gates:

1. Forget gate: decides what information should be thrown away or kept from previous step.
2. Input gate: decides which values will be updated from current input
3. Cell gate: generate *candidate* current cell state
4. Output gate: Generate final current cell state and hidden state.

The four gates together with its parameters provide flexibility for modulating the values and keeping track of the inputs for very long periods/steps

Gated Recurrent Units (GRUs)



sigmoid
tanh
pointwise multiplication
pointwise addition
vector concatenation

GRUs are a modified/simplified version of LSTMs that combines the four gates into two and is widely used.

RNN Lab

Deep Learning Takeaways

- Deep learning isn't magic
  - The size of our datasets is what's really magic
- Certain problems and problem areas are perfect for deep learning
  - Others are not
- Deep learning is hard to implement
  - But the tools are getting easier every day
- RNNs probably have the most application in Cybersecurity

Questions?