Reinforcement Learning for Autonomous Systems Design Value Function Approximation in Reinforcement Learning

Mirco Musolesi

mircomusolesi@acm.org

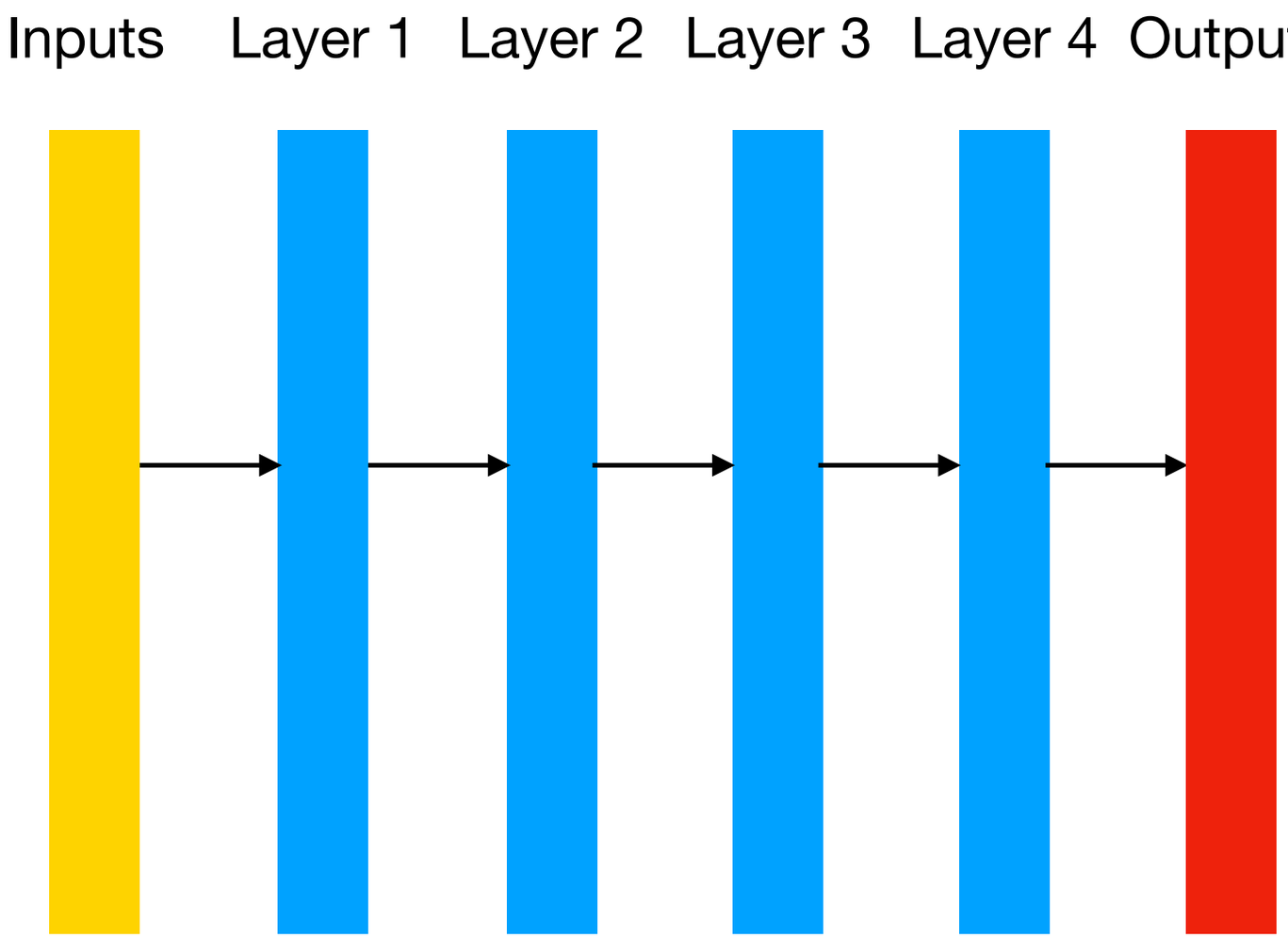
Recap: What is Wrong with Tabular Methods

- Tabular methods suffer from a series of drawbacks:
  - Table with 1 row per state-action entry.
    - What happens if you can’t fit all the state-action entry in a table?
    - We need *function approximation* rather than tables.

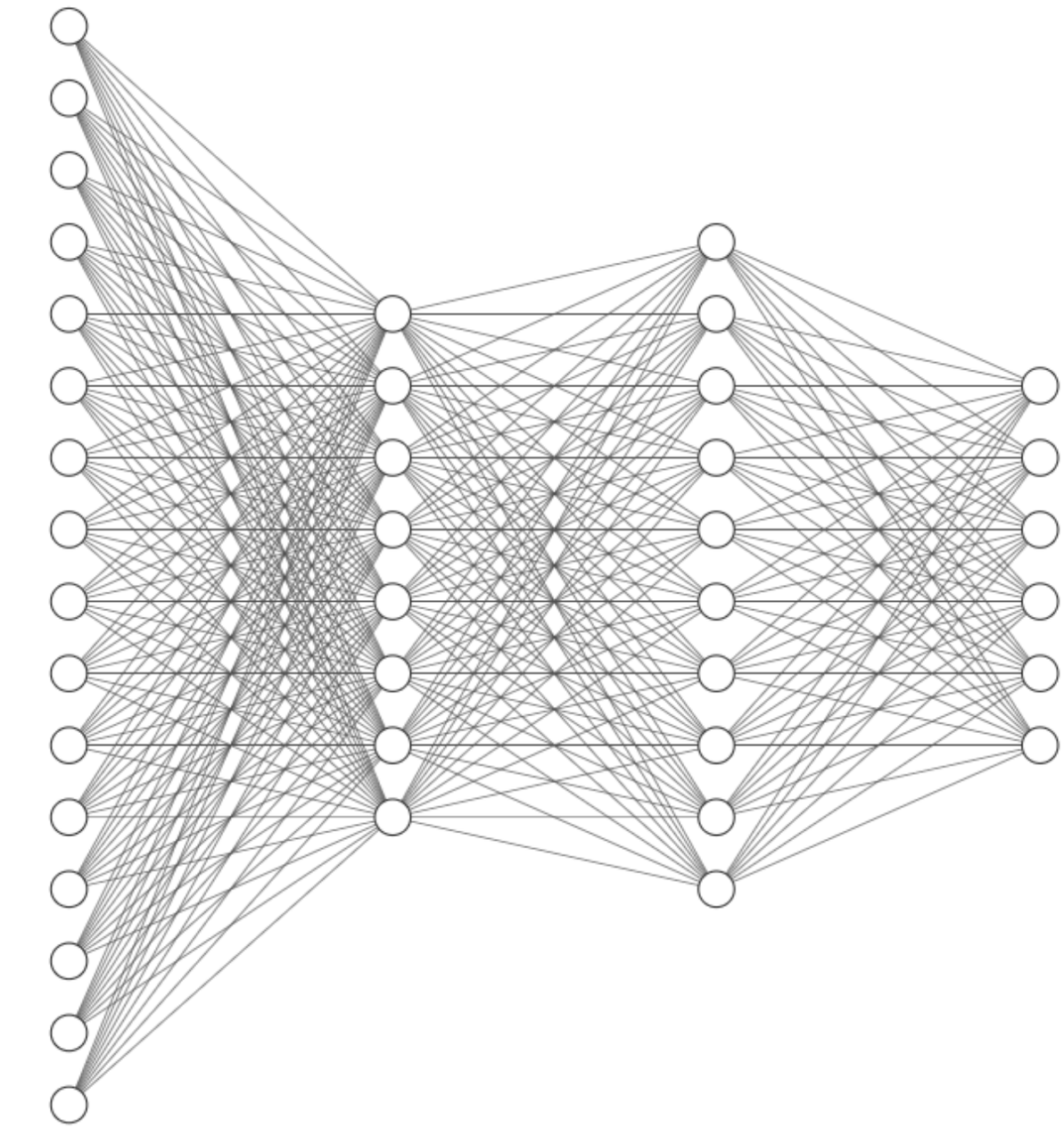
Alternative: Function Approximation

- Function Approximation will provide a mapping between a state or state-action to a value function.
  - Recall: a value-function approximation is a function with in input the state (or the state and action), which gives in output the value function for the state (or the state and action).

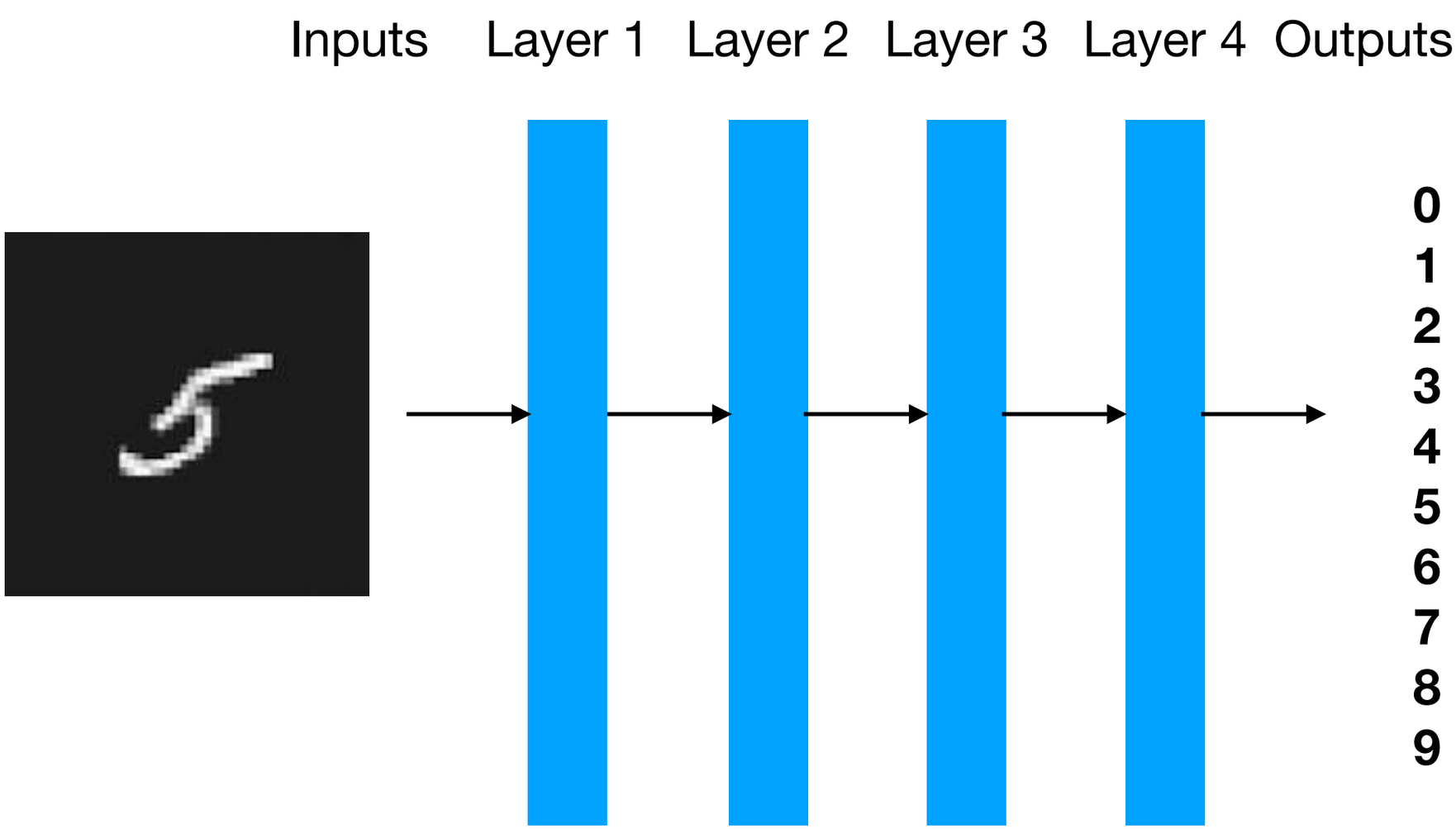
Deep Neural Networks



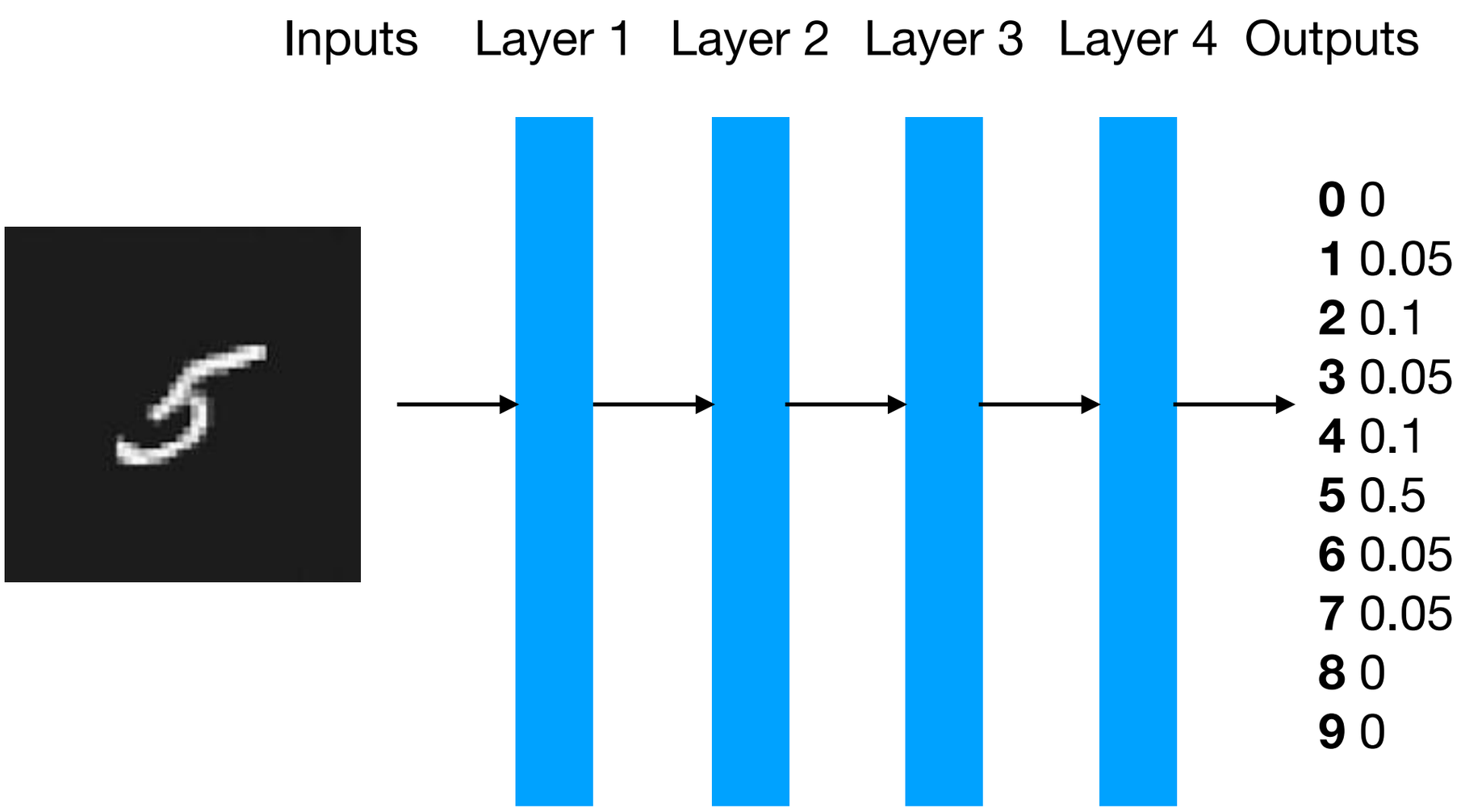
Deep Neural Networks



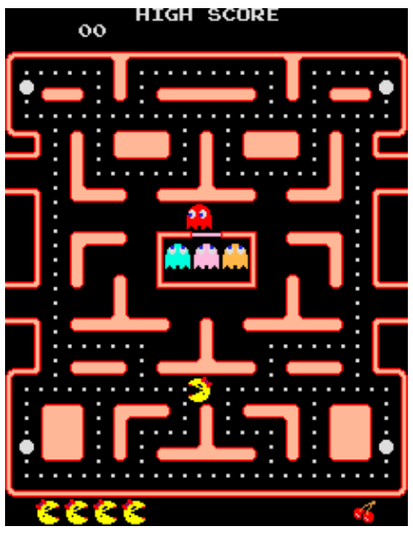
Deep Neural Networks

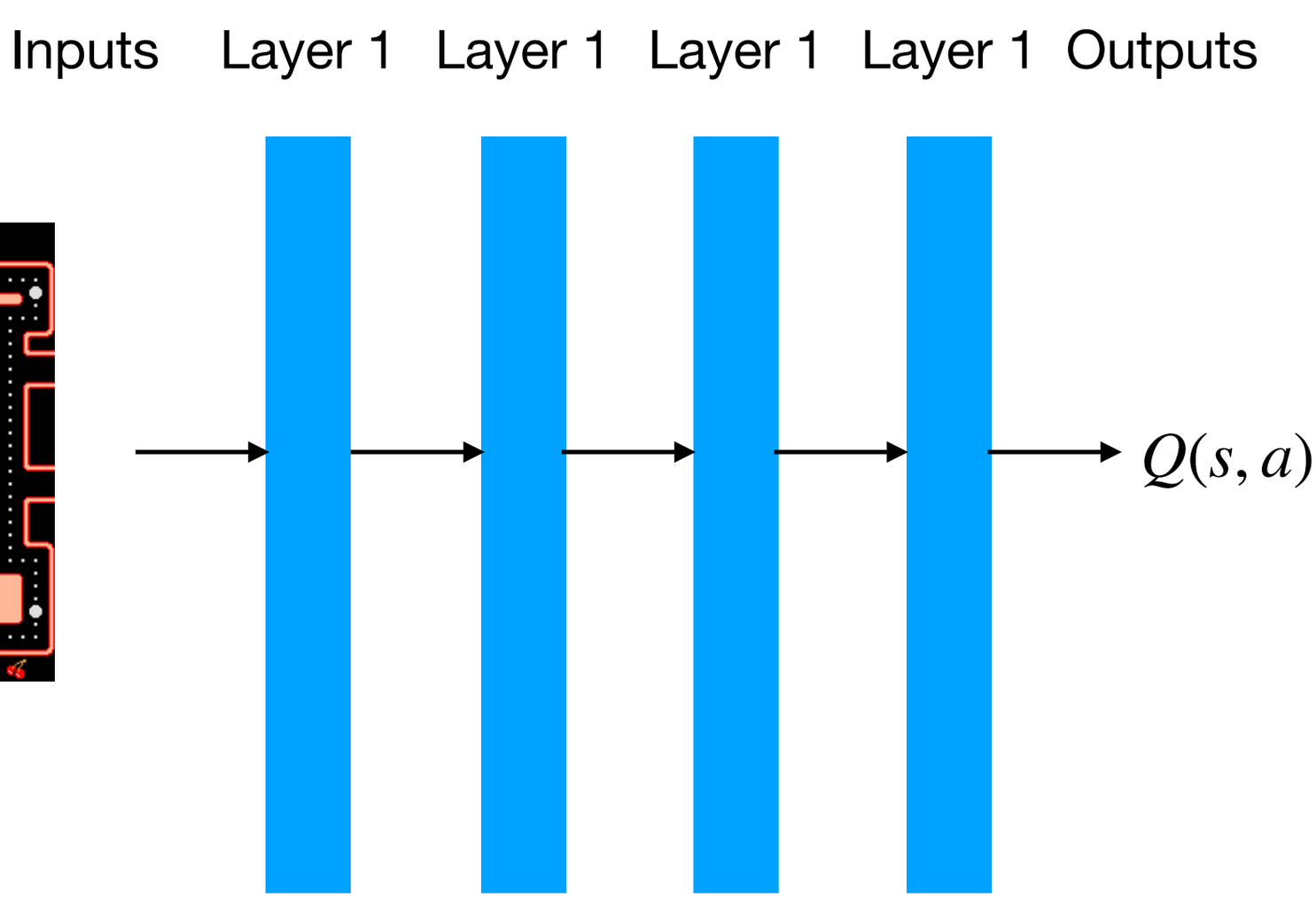


Deep Neural Networks

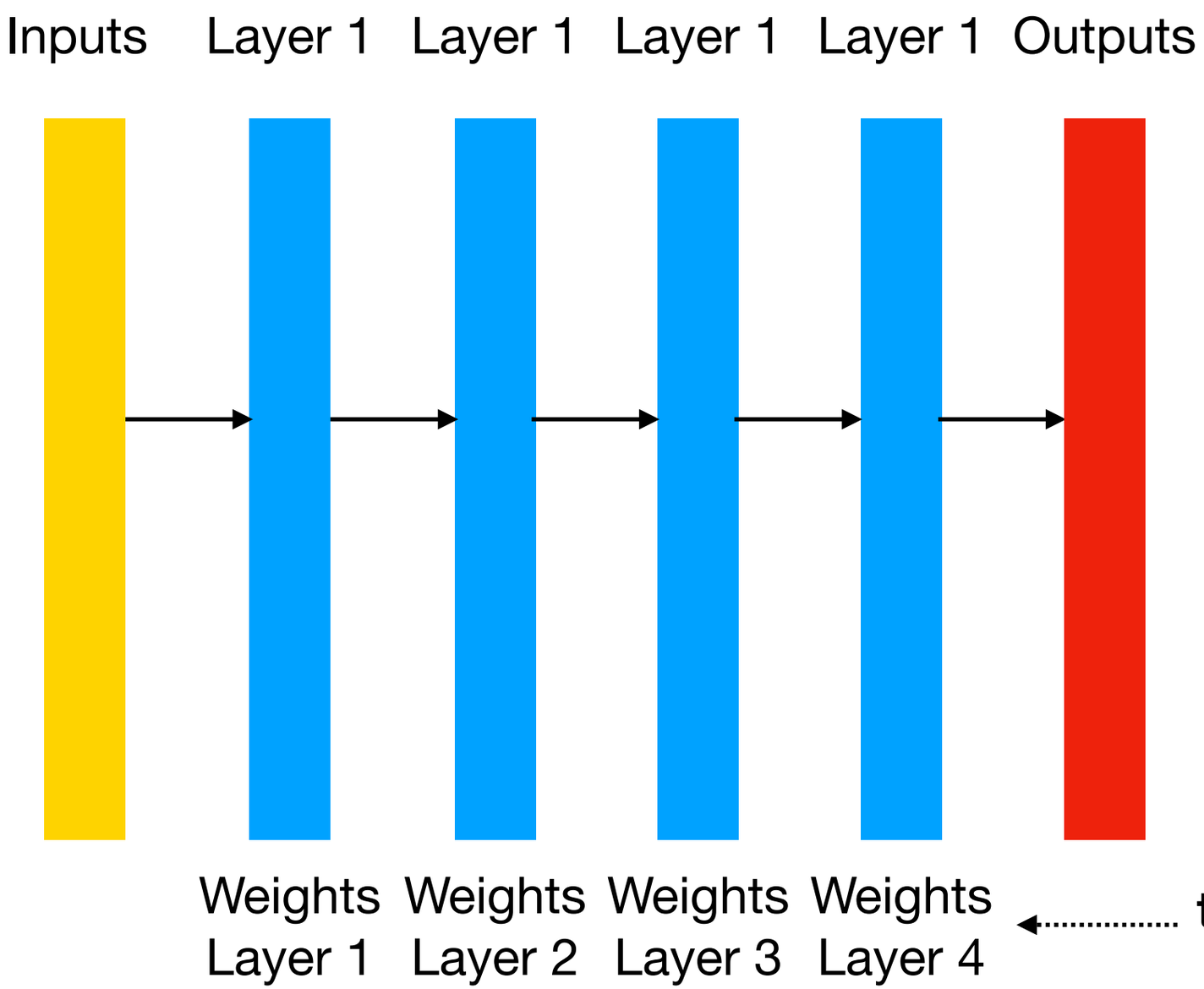


Deep Neural Networks



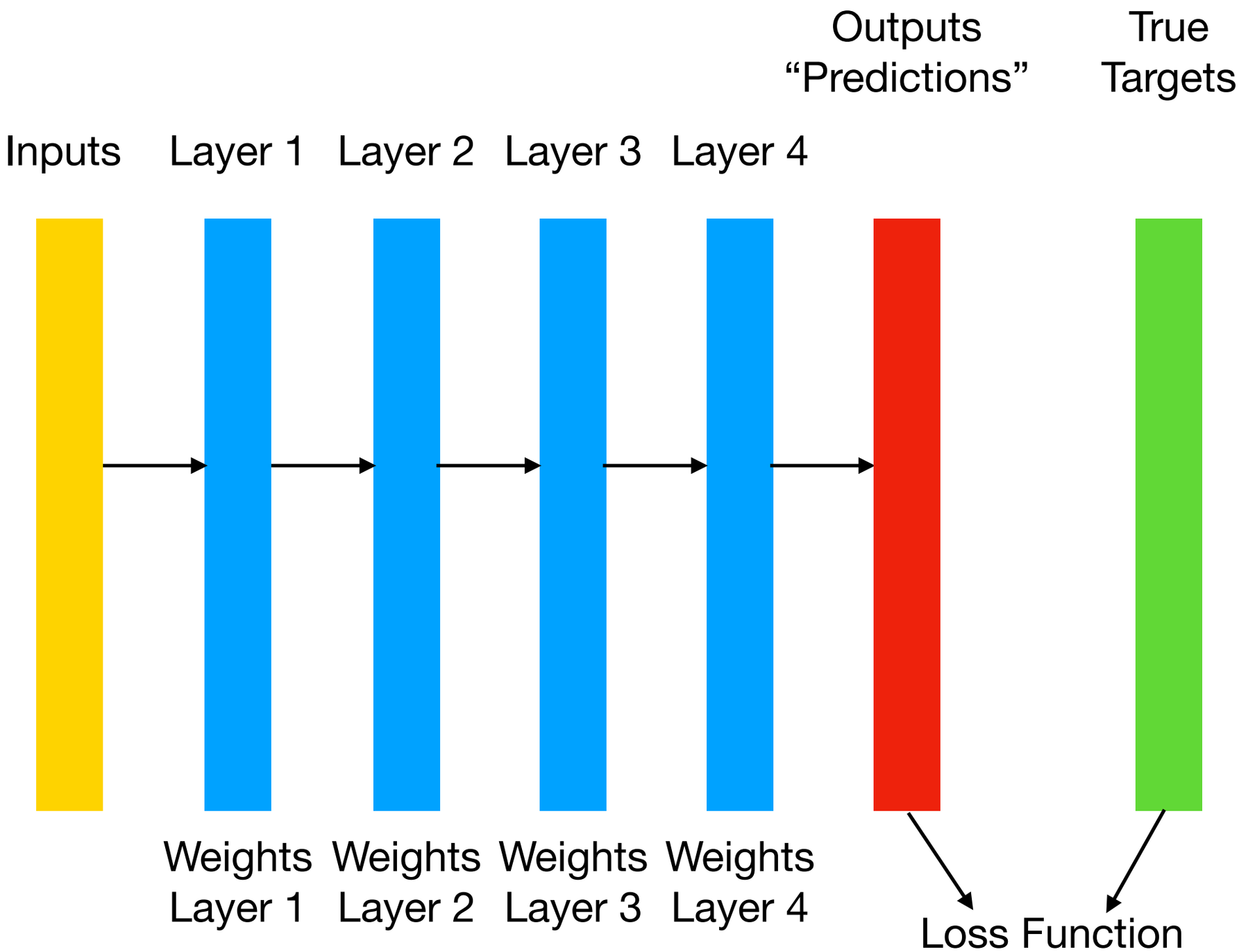


Deep Neural Networks

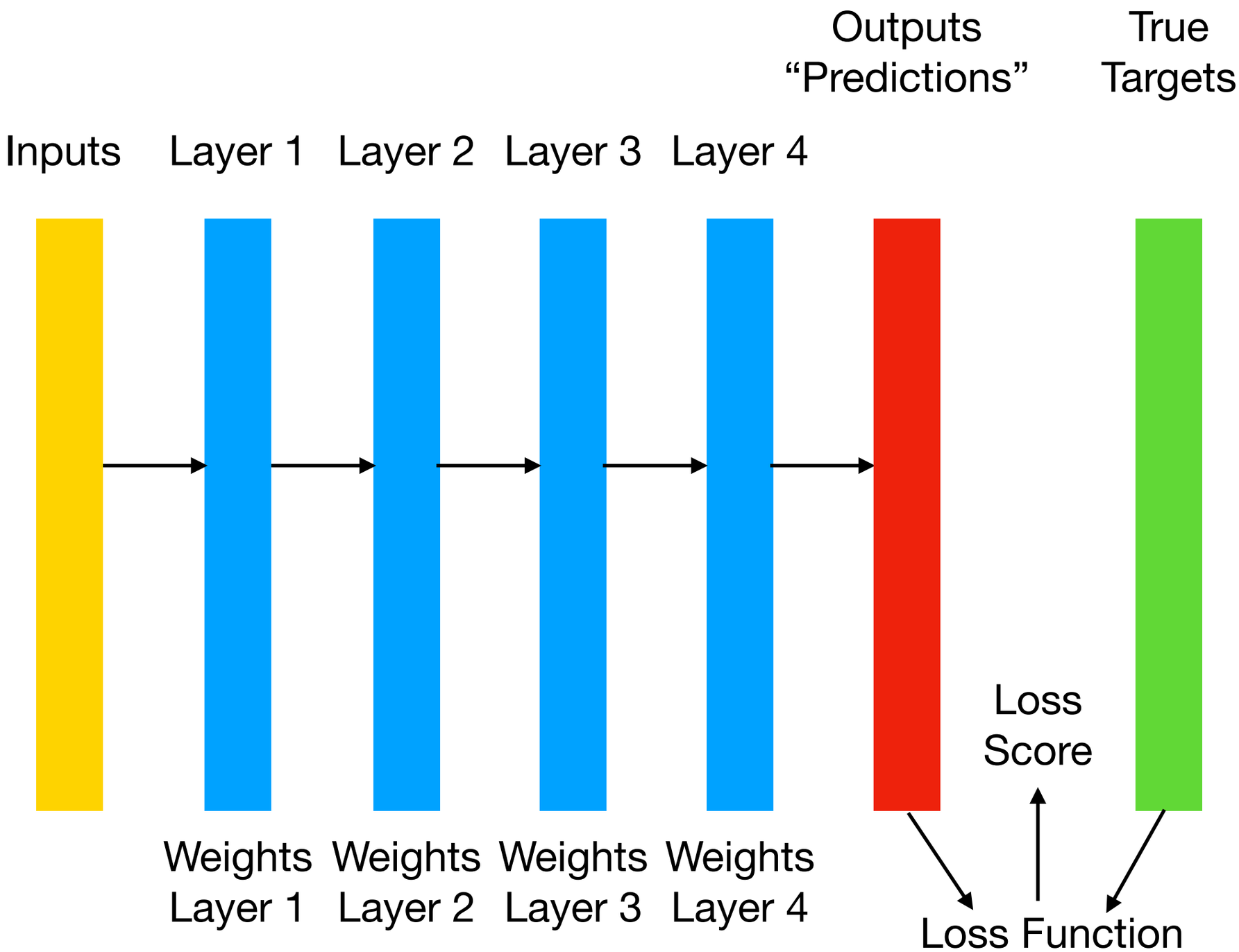


The goal is to find the right values for these weights.

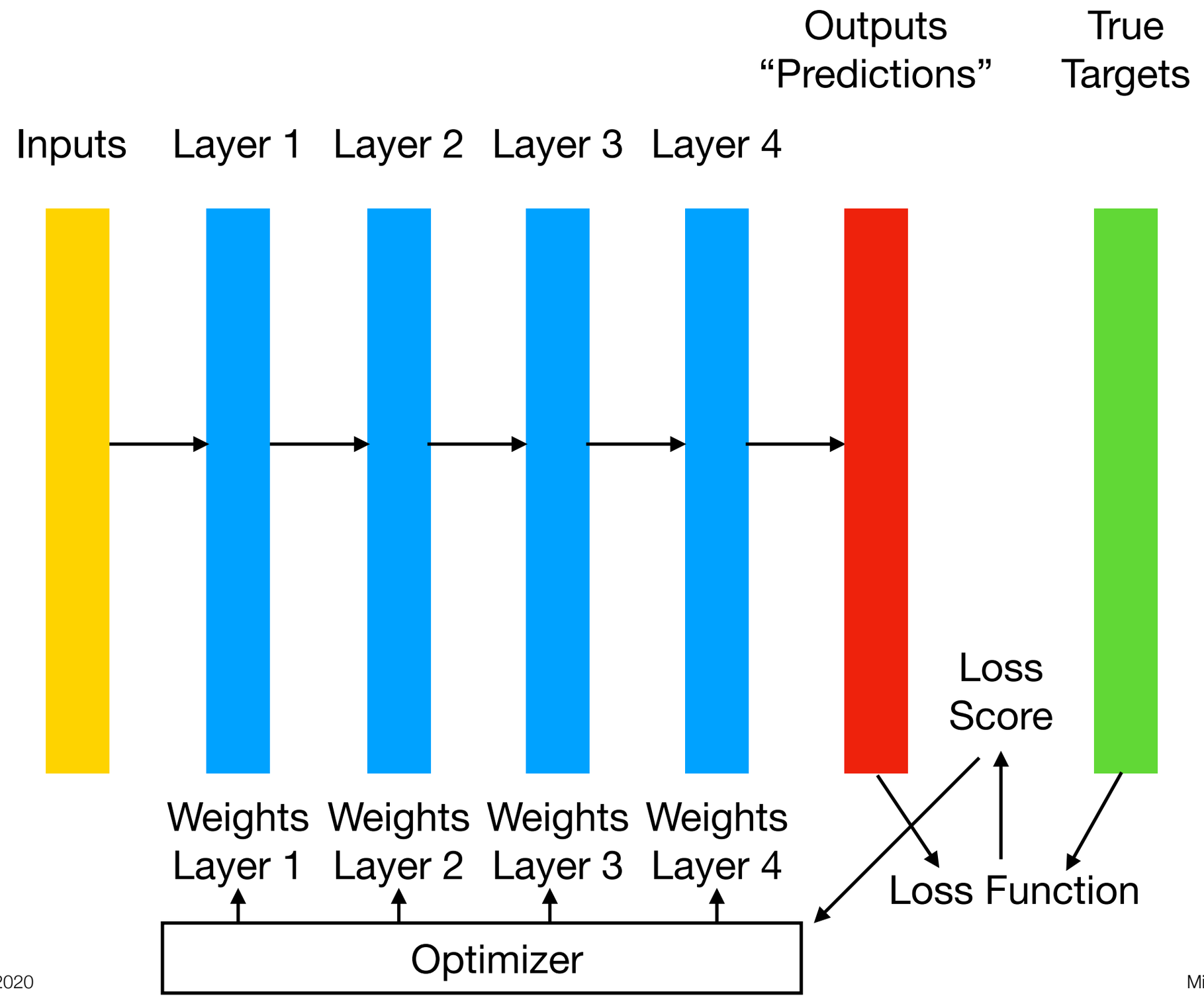
Deep Neural Networks



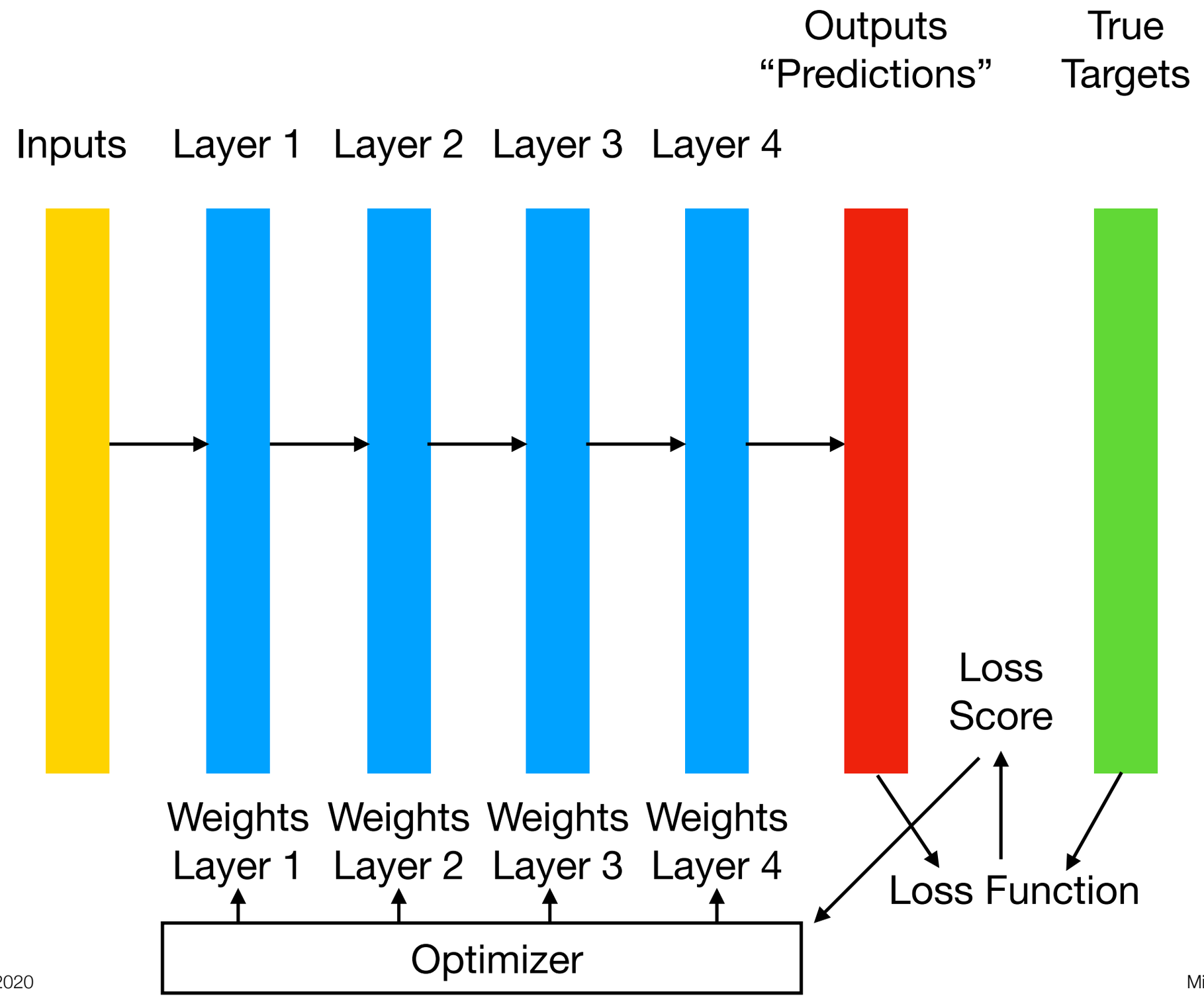
Deep Neural Networks



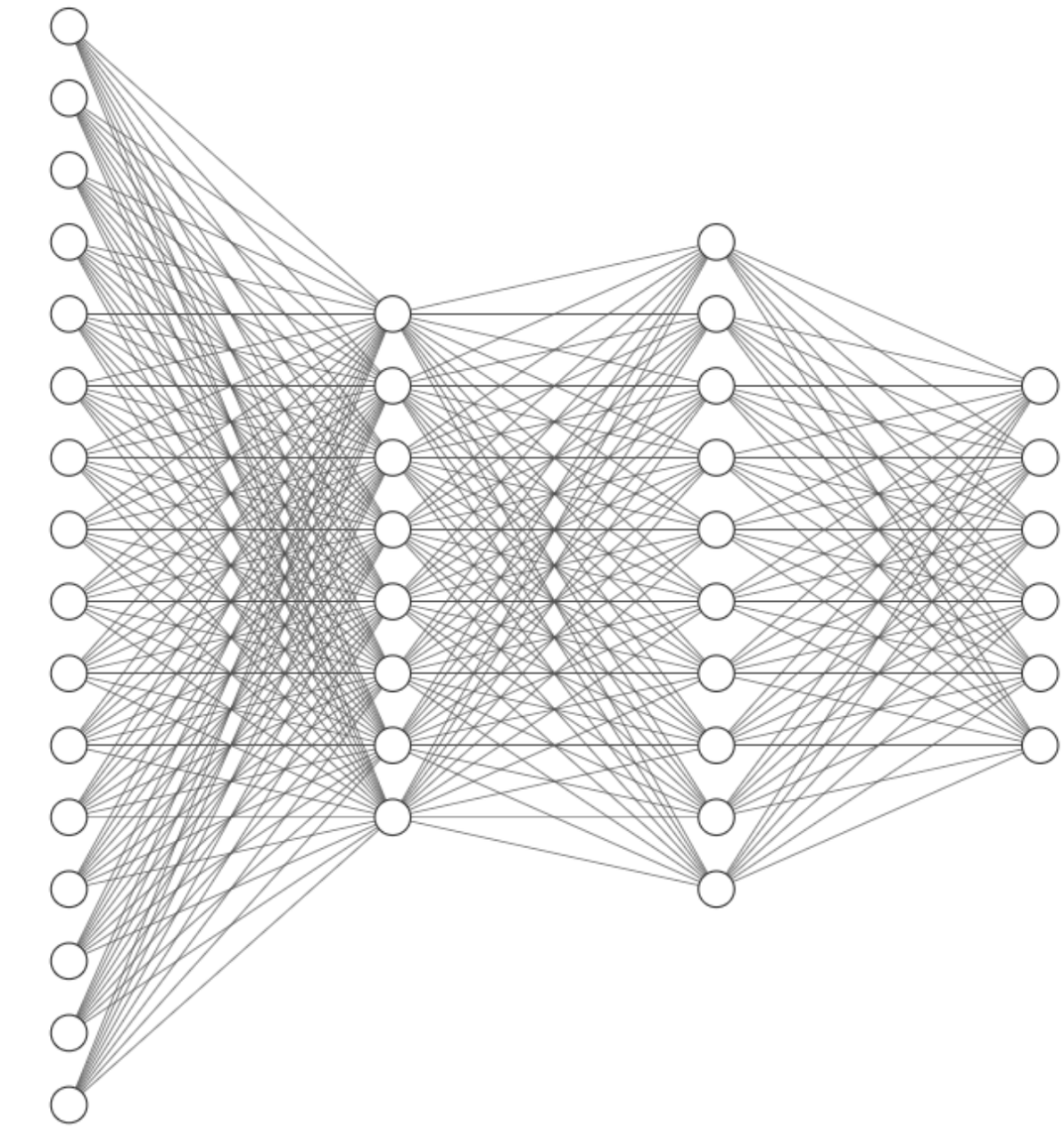
Deep Neural Networks



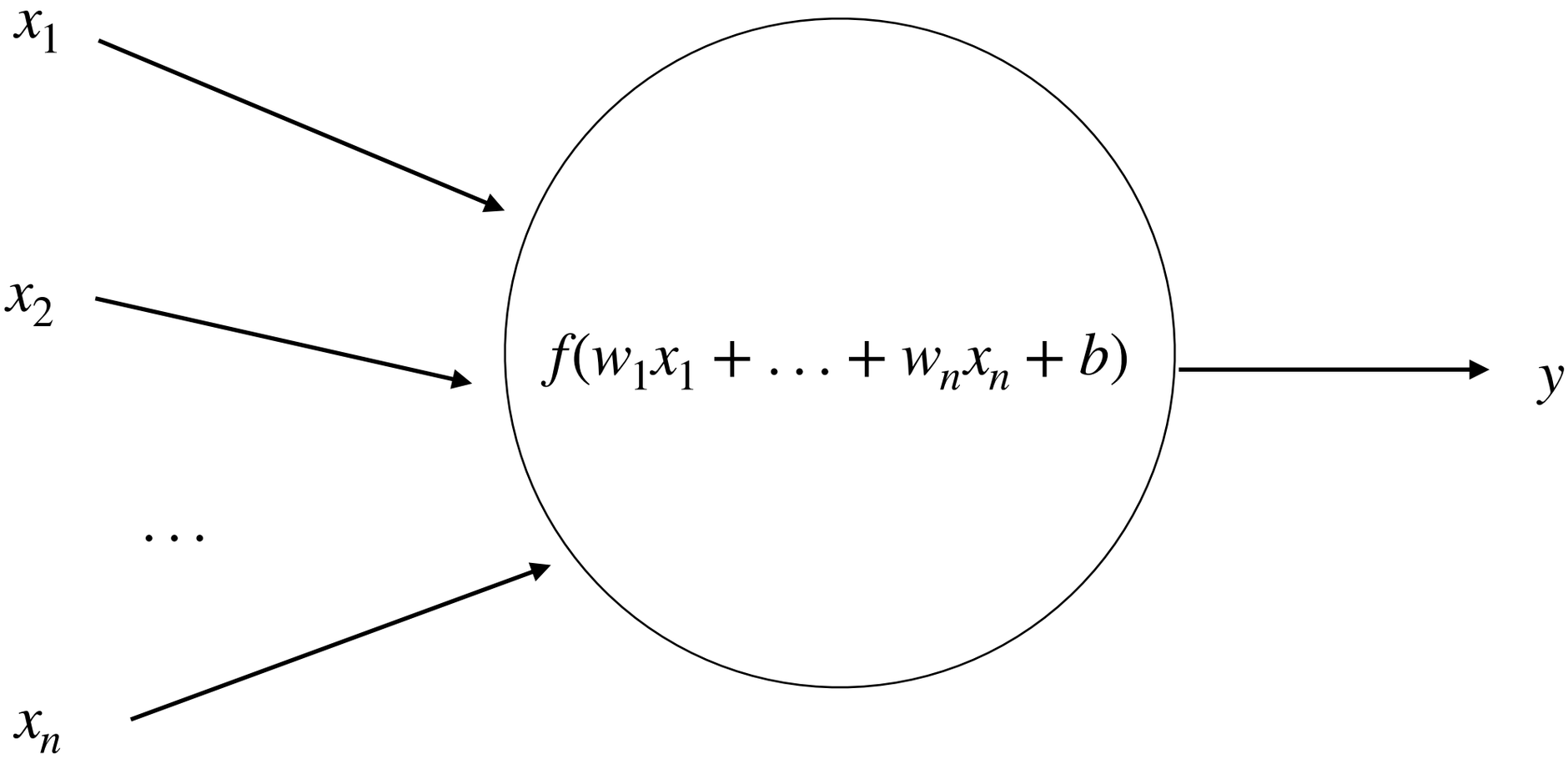
Deep Neural Networks



Deep Neural Networks



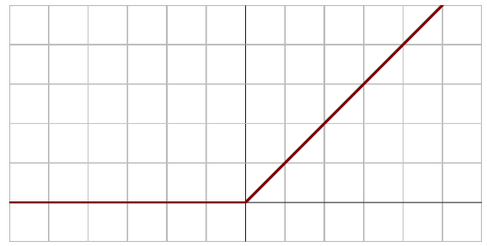
Nodes/Units/Neurons



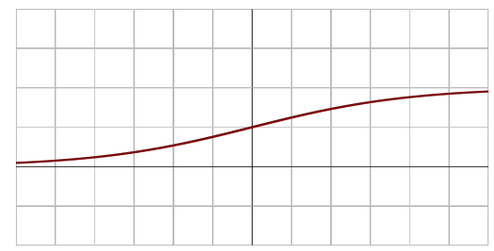
f is called the activation function, b is usually called the bias

Activations Functions

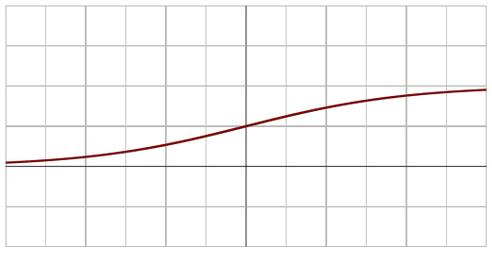
- They are generally used to add non-linearity.
- Examples:
  - *Rectified Linear Unit*: it returns the max between 0 and the value in input. In other words, given the value z in input it returns $max(0,z)$.



  - *Logistic sigmoid*: given the value in input z, it returns $\frac{1}{1+e^{z}}$.

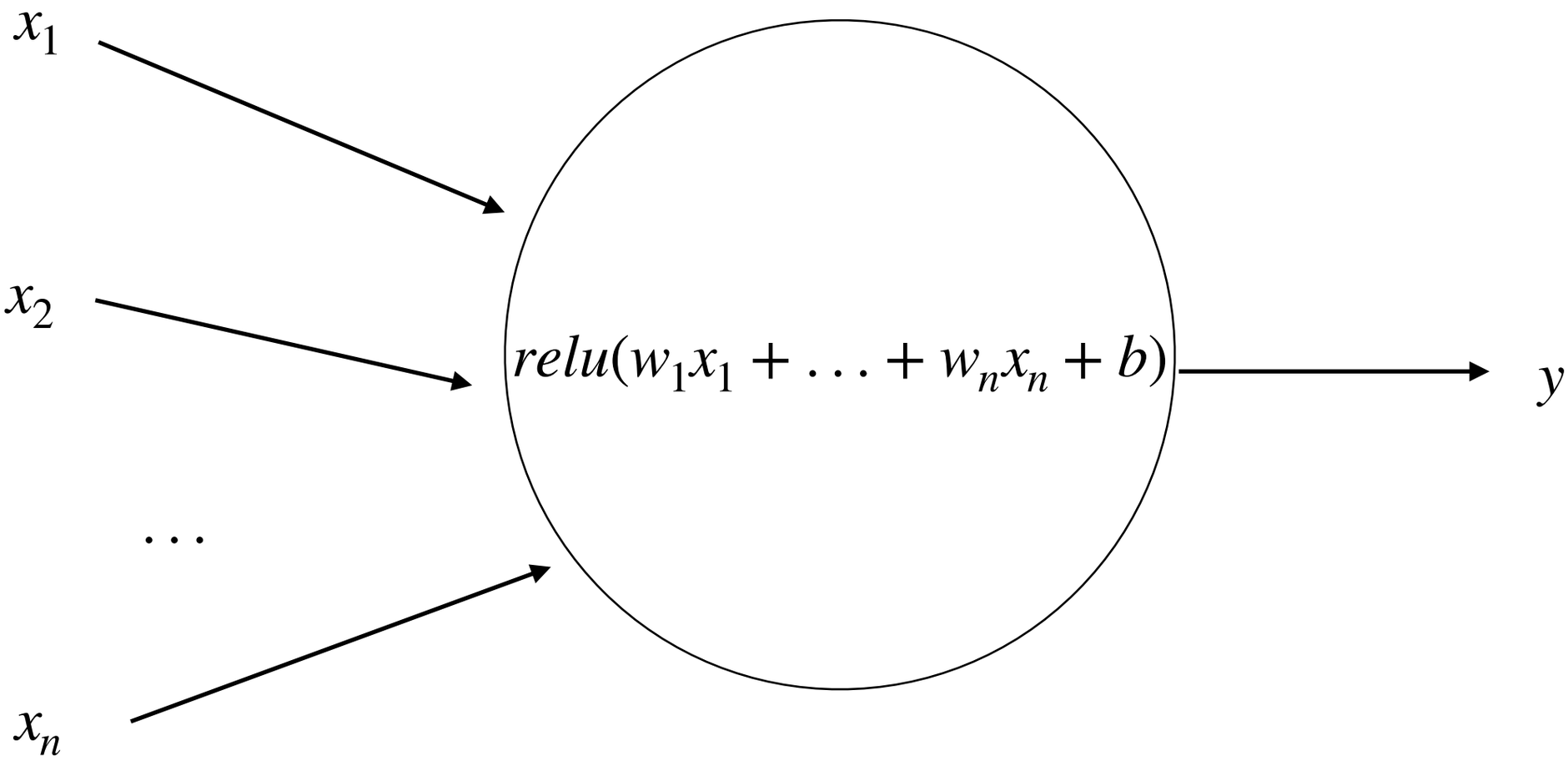


  - *Arctan*: given the value in input z, it returns $tan^{-1}(z)$.



Credit: Wikimedia

Nodes/Units/Neurons



Note that here the function in input of relu is 1-dimensional.

Softmax Function

- Another function that we will use is *softmax*.

- But please note that softmax is not like the activation functions that we discussed before. The activations functions that we discussed before take in input real numbers and returns a real number.

- A softmax function receives in in inputs a vector of real numbers of dimension n and returns a vector of real numbers of dimension n.

- *Softmax*: given a vector of real numbers in input $\mathbf{z}$ of dimension n, it normalises it into a probability distribution consisting of n probabilities proportional to the exponentials of each element z_i of the vector $\mathbf{z}$. More formally,

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \text{ for } i = 1,..n.$$

Gradient-based Optimization

- We will now discuss a high-level description of the learning process of the network, usually called *gradient-based optimization*.

- Each neural layer transforms his input layer as follows:

$$output = f(w_1 x_1 + \ldots + w_n x_n + b)$$

- And in the case of a relu function, we will have

$$output = relu(w_1 x_1 + \ldots + w_n x_n + b)$$

- Note that this is a simplified notation for one layer, it should be $w_{1,i}$ for layer i.

Gradient-based Optimisation

- The learning is based on the gradual adjustment of the weight based on a feedback signal, i.e., the loss described above.

- The training is based on the following training loop:
  - Draw a batch of training examples $\mathbf{x}$ and corresponding targets $\mathbf{y}_{target}$..
  - Run the network on $\mathbf{x}$ (forward pass) to obtain predictions $\mathbf{y}_{pred}$.
  - Compute the loss of the network on the batch, a measure of the mismatch between $\mathbf{y}_{pred}$ and $\mathbf{y}_{target}$..
  - Update all weights of the networks in a way that reduces the loss of this batch.

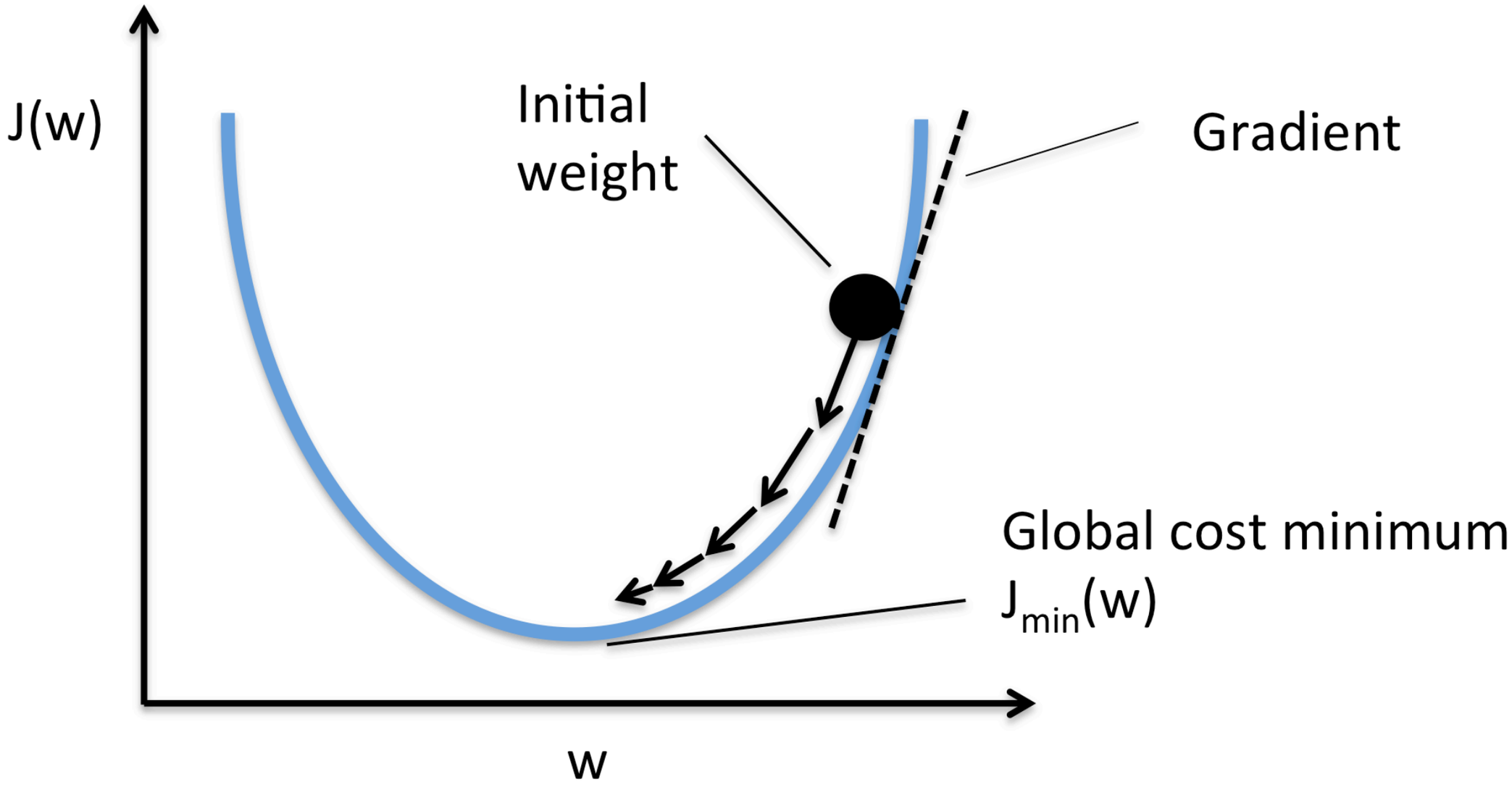
Stochastic Gradient Descent

- Given a differentiable function, it's theoretically possible to find its minimum analytically.

- However, the function is intractable for real networks. The only way is to try to approximate the weights using the procedure describe above.

- More precisely since it's a differentiable function we can use the gradient, which provide an efficient way to perform the correction mention before.

Gradient-based Optimisation



Credit: Sebastian Raschka

Stochastic Gradient Descent

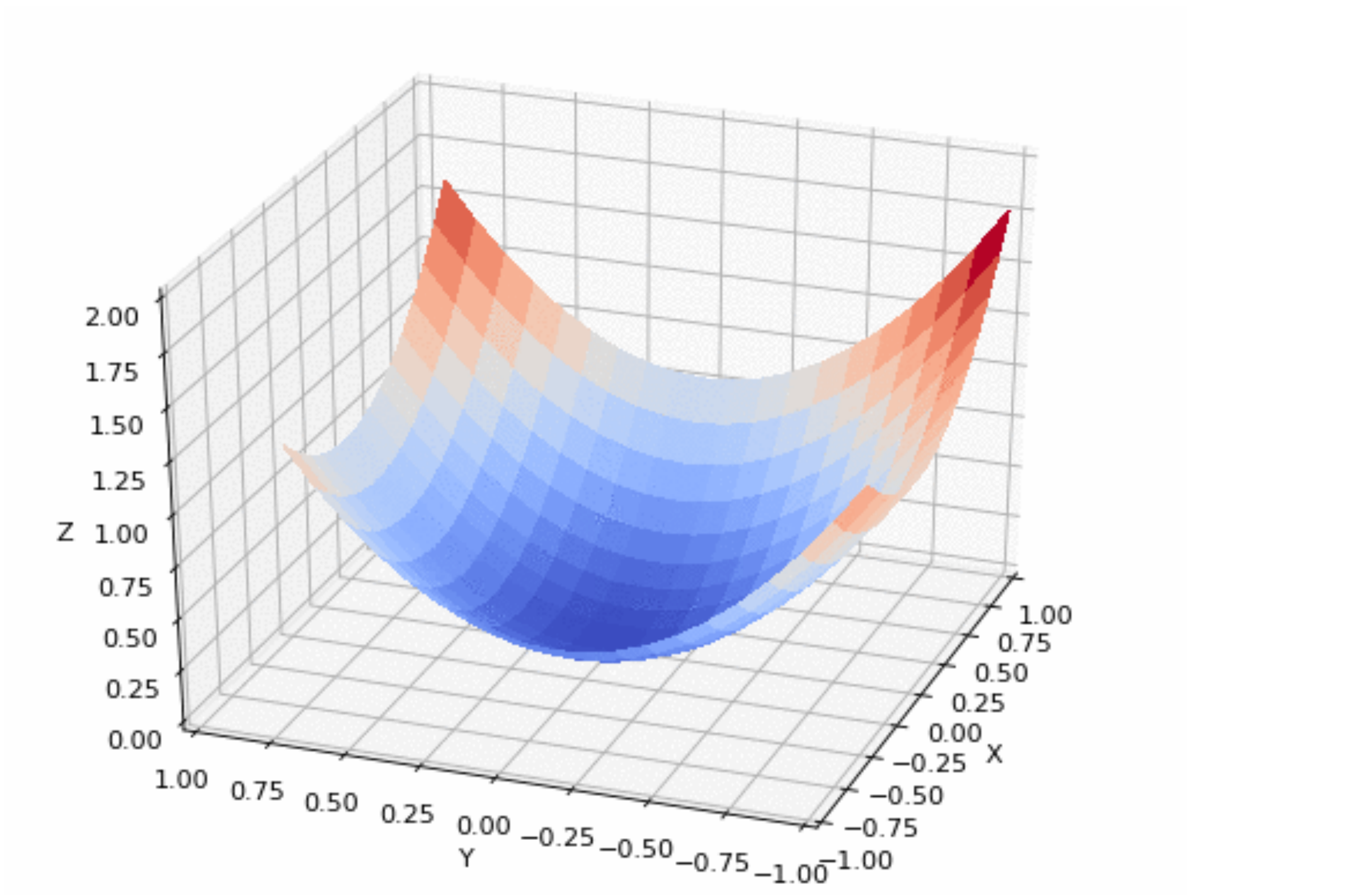
- More formally:
  - Draw a batch of training example $\mathbf{x}$ and corresponding targets $\mathbf{y}_{target}$.
  - Run the network on $\mathbf{x}$ (forward pass) to obtain predictions $\mathbf{y}_{pred}$.
  - Computer the loss of the network on the batch, a measure of the mismatch between $\mathbf{y}_{pred}$ and $\mathbf{y}_{target}$.
  - Compute the gradient of the loss with regard to the network's parameters (backward pass).
  - Move the parameters in the opposite direction from the gradient with: $w_j \leftarrow w_j + \Delta w_j = w_j - \eta \frac{\partial J}{\partial w_j}$

    where J is the loss (cost) function.
  - If you have a batch of samples of dimension k:

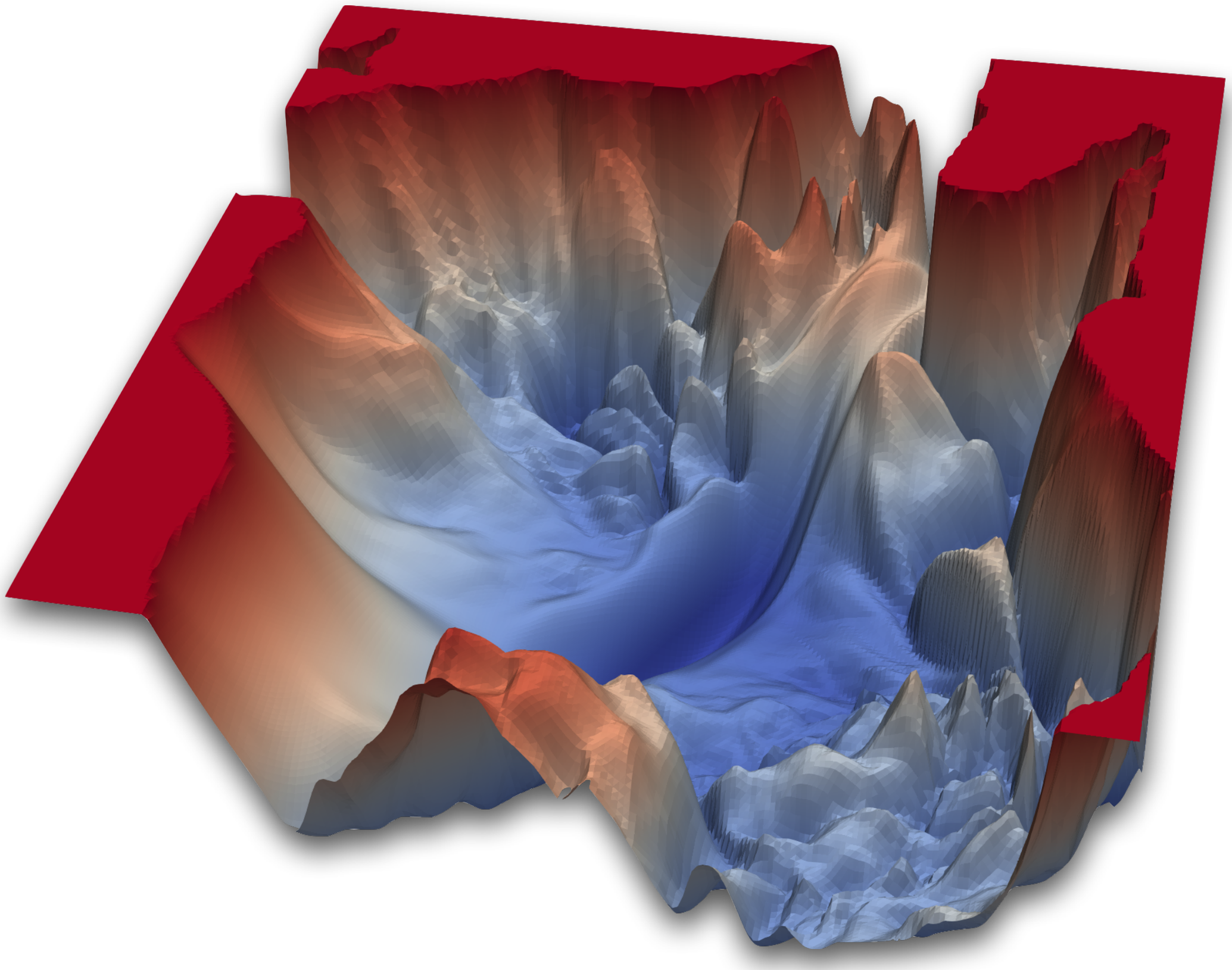
$w_j \leftarrow w_j + \Delta w_j = w_j - \eta \; average(\frac{\partial J_k}{\partial w_j})$ for all the k samples of the batch.

Stochastic Gradient Descent

- This is called the mini-batch stochastic gradient descent (mini-batch SGD).
- The loss function J is a function of $f(\mathbf{x})$, which is a function of the weights.
  - Essentially you calculate the value $f(\mathbf{x})$, which is a function of the weights of the network.
    - Therefore, by definition, the derivative of the loss function that you are going to apply will be a function of the weights.
- The term *stochastic* refers to the fact that each batch of data is drawn randomly.
- The algorithm describe above was based on a simplified model with a single function in a sense.
- You can think about a network composed of three layers, e.g., three tensor operations on the network itself.



https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/



https://www.cs.umd.edu/~tomg/projects/landscapes/

Backpropagation Algorithm

- Suppose that you have three tensor operations/layers f, g, h with weights $\mathbf{W}^1$, $\mathbf{W}^2$ and $\mathbf{W}^3$ respectively for the first, second, third layer. You will have the following functon:

$$y_{pred} = f(\mathbf{W}^1, \mathbf{W}^2, \mathbf{W}^3) = f(\mathbf{W}^1, g(\mathbf{W}^2, h(\mathbf{W}^3)))$$

with $f()$ the rightmost function/layer and so on. In other words, the input layer is connected to $h()$, which is connected to $g()$, which is connect to $f()$, which returns the final result.

- A network is a sort of chain of layers. You can derive the value of the "correction" by applying the chain rule of the derivatives backwards.
  - Remember the chain rule $f(g(x)) = f'(g(x))g'(x)$.

Backpropagation Algorithm

- The update of the weights starts from the right-most layer *back* to the left-most layer. For this reason, this is called *backpropagation* algorithm.

- More specifically, backpropagation starts with the calculation of the gradient of final loss value and works backwards from the right-most layers to the left-most layers, applying the chain rule to compute the contribution that each weight had in the loss value.

- Nowadays, we do not calculate the partial derivates manually, but we use framework like TensorFlow that supports symbolic differentiation for the calculation of the gradient.

- TensorFlow supports the automatic updates of the weights described above.

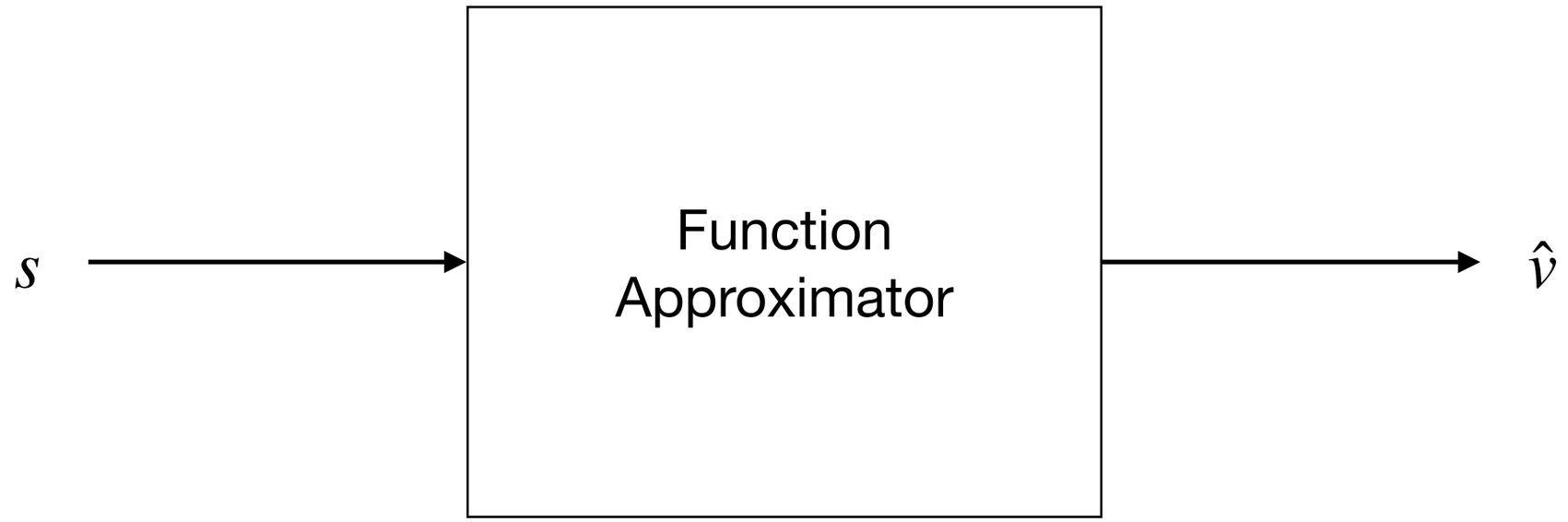
- More theoretical details can be found in:

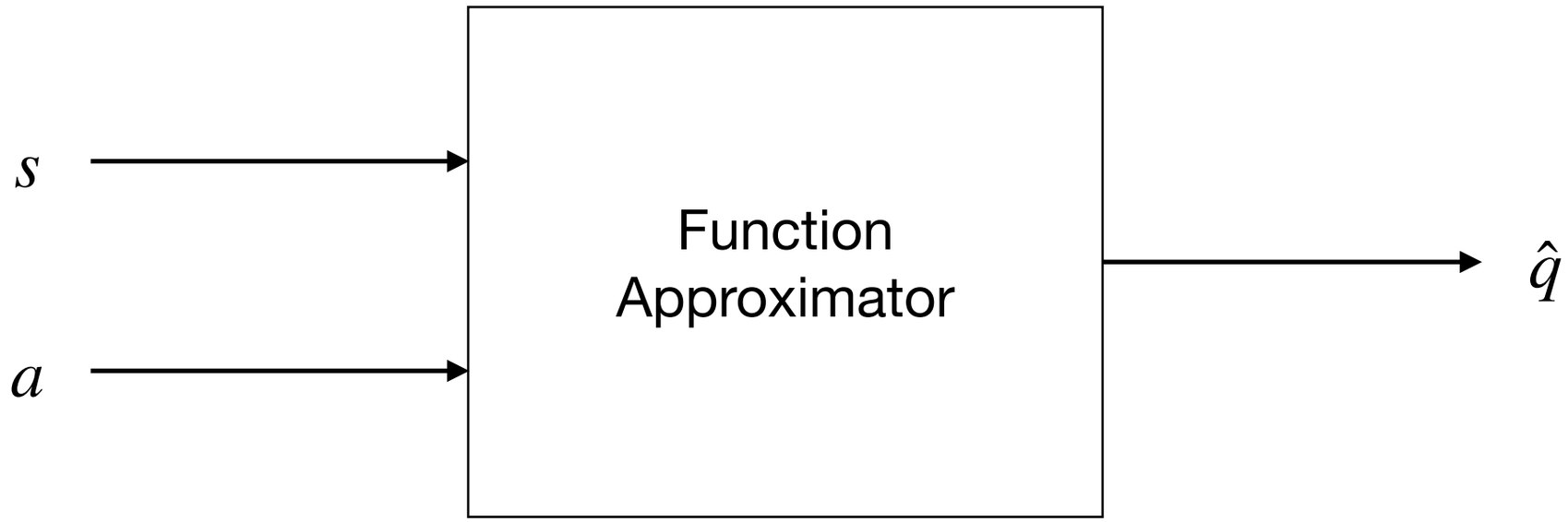
Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press. 2016.

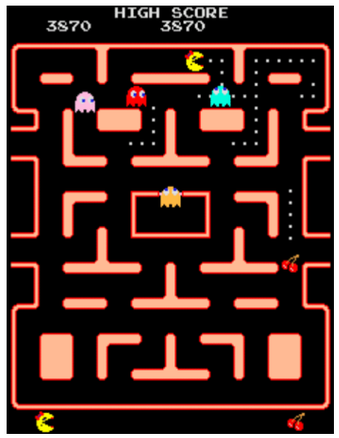
References

- Chapter 1 of Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press. 2016.

- Chapter 2 of Francois Chollet. Deep Learning with Python. Manning 2018.

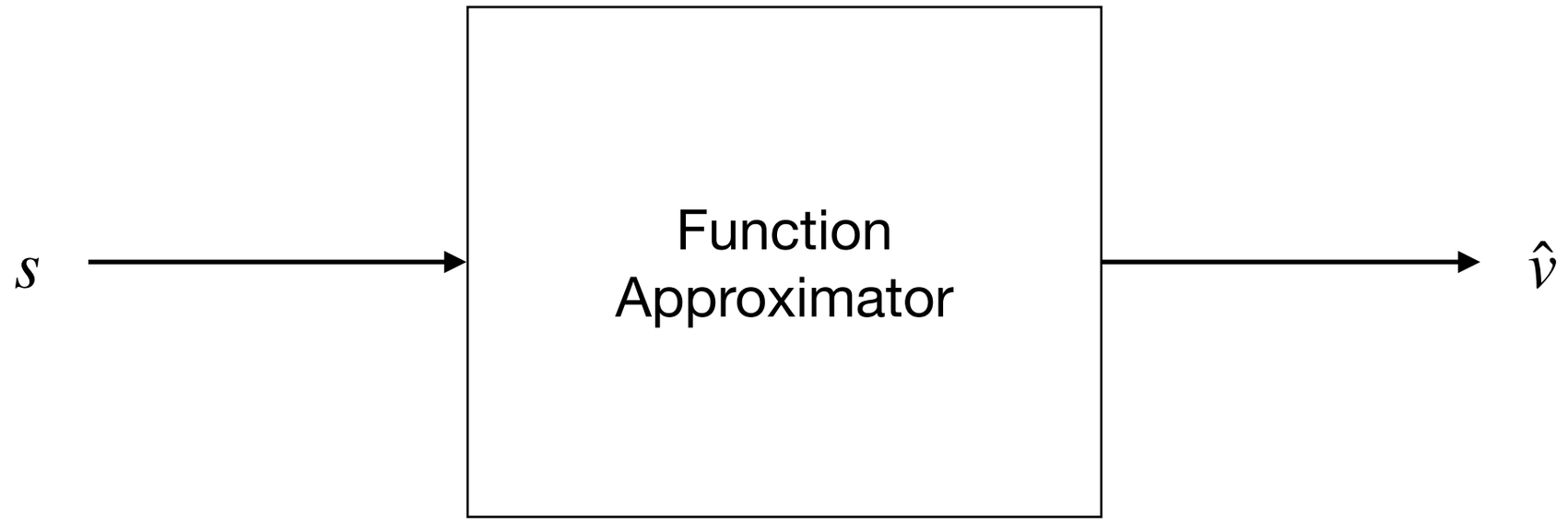

Function
Approximator
s
$\hat{v}$

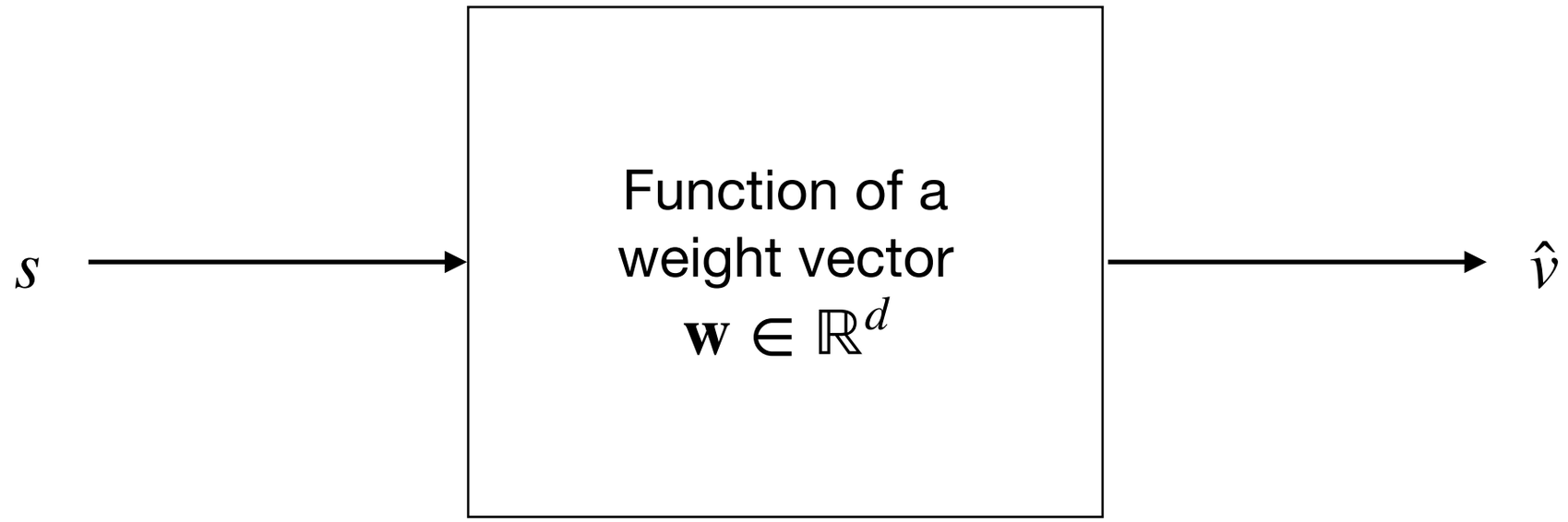

s
a
Function
Approximator
$\hat{q}$



Function Approximator

$\hat{q}$


Function
Approximator
s
$\hat{v}$


s
Function of a
weight vector
$\mathbf{w} \in \mathbb{R}^d$
$\hat{v}$

Approximation Methods

- In this lecture we will focus on the study of function approximation for estimating state-value functions.

- In particular, we will consider an approach where the approximate value function is represented not as a table but as a parametrised functional form with weight vector $w \in \mathbb{R}^d$.

- We will start by considering how to approximate v_π from experience generated using a policy π.

Number of Moves in Go

Board size n×n	3^{n^2}	Percentage of Legal Positions	Number of Legal Positions
1×1	3	33.33%	1
2×2	81	70.37%	57
3×3	19,683	64.40%	12,675
4×4	43,046,721	56.49%	24,318,165
5×5	847,288,609,443	48.90%	414,295,148,741
9×9	$4.43426488243\times10^{38}$	23.44%	$1.03919148791\times10^{38}$
13×13	$4.30023359390\times10^{80}$	8.66%	$3.72497923077\times10^{79}$
19×19	$1.74089650659\times10^{172}$	1.20%	$2.08168199382\times10^{170}$

Data from Wikimedia and the On-line Encyclopedia of Integer Sequences (see A094777)

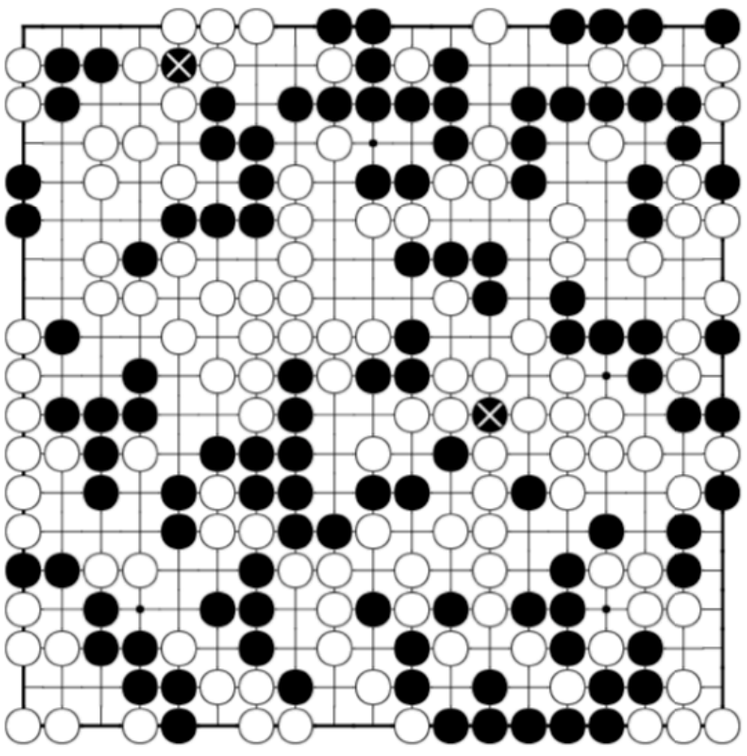
Combinatorics of Go

John Tromp Gunnar Farnebäck

January 31, 2016

Abstract

We present several results concerning the number of positions and games of Go. We derive recurrences for $L(m, n)$, the number of legal positions on an $m \times n$ board, and develop a dynamic programming algorithm which computes $L(m, n)$ in time $O(m^3 n^2 \lambda^m)$ and space $O(m\lambda^m)$, for some constant $\lambda < 5.4$. We used this to compute $L(n, n)$ up to the standard board size $n = 19$. In ternary (mapping 0,1,2 to empty,black,white)

$L(19, 19) =$ 

For even larger boards, we prove existence of a *base of liberties*

$$L = \lim_{m,n\to\infty} \sqrt[mn]{L(m,n)} = 2.975734192043357249381\ldots$$

Based on a conjecture about vanishing error-terms, we derive an asymptotic formula for $L(m, n)$, which is shown to be highly accurate.

A094777 Number of legal positions in Go played on an n X n grid (each group must have at least one liberty). 8

1, 57, 12675, 24318165, 414295148741, 62567386502084877, 836778478479842876285 95, 990966953618170260281935463385, 10391914879129383431898309043879879346 9, 964984285019096545896308879788350980881481778 57, 793474866816582266820936671790189132321673383112185151899, 577742584895132389982379703074839993272872107569911896559426513311 69, 372497923076863964422949047670245176742491579482087175332547995509705958752377 05, 212667732900366224249789357650440598098805861083269127196623872213228196352455447575029701325

(list; graph; refs; listen; history; text; internal format)

OFFSET 1,2

COMMENTS John Tromp wrote a small C program to compute the number for boards up to size 4 X 5, given in the rec.games.go posting below. Gunnar Farnebaeck (gunnar(AT)lysator.liu.se) wrote a pike script to compute the number by dynamic programming, which handles sizes up to 12 X 12 (available upon request).

LINKS John Tromp, Table of n, a(n) for n = 1..19
British Go Association, Go
Sandy Harris, Number of Possible Outcomes of a Game
John Tromp, Complexity of Chess and Go
John Tromp, Number of legal Go positions
John Tromp and Gunnar Farnebäck, Combinatorics of Go (2016)

FORMULA 3^(n*n) is a trivial upper bound.
Tromp & Farnebäck prove that a(n) = (1 + o(1)) * L^(n^2), and conjecture that a(n) ~ A * B^(2n) * L^(n^2) * (1 + O(n*p^n)) for some constants A, B, L, and p < 1. - Charles R Greathouse IV, Feb 08 2016

EXAMPLE The illegal 2 X 2 positions are the 2^4 with no empty points and the 4*2 having a stone adjacent to 2 opponent stones that share a liberty. That leaves 3^4-16-8 = 57 legal positions.

CROSSREFS Sequence in context: A219077 A091749 A218425 * A218662 A331015 A093257
Adjacent sequences: A094774 A094775 A094776 * A094778 A094779 A094780

KEYWORD nonn

AUTHOR Jan Kristian Haugland, Jun 09 2004

EXTENSIONS More terms from John Tromp, Jan 27 2005
a(10)-a(13) from John Tromp, Jun 23 2005
a(14)-a(15) from John Tromp, Sep 01 2005.
a(16) from John Tromp, Oct 06 2005
Michal Koucky should be credited for carrying most of the computational load for computing the n=14, 15 and 16 results with his file-based implementation.
a(17)-a(18) from John Tromp, Mar 08 2015

Approximation Methods

- We will write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$.

- $\hat{v}$ in theory might be a linear function of the weights $\mathbf{w}$, but more generally it can be a non-linear function.

- A widely used technique is to use a multi-layer artificial neural network to compute $\hat{v}$, with $\mathbf{w}$ the vector of the connection weights in all the layers.

  - By adjusting the values of the weights $\mathbf{w}$ a different range of functions can be computed by the network.

  - Typically, the number of weights $\mathbf{w}$ is much less than the number of states ($d \ll |\mathcal{S}|$) where $|\mathcal{S}|$ is the cardinality of the states of the system under consideration.

Approximation Methods

- There are various interesting aspects that need to be considered:
  - Function approximation can be used also for partially observable problems.
  - With an approximation we might build a function that does not depend on certain aspects of the state and we can just build a function assuming that they are not observable.

Value Function Approximation

- Classic methods are based on the idea of an update “towards” an (update) target. For example:
  - In the Monte Carlo method the update target for the value prediction is G_t.
  - In the TD(0) the update target is $R_{t+1} + \gamma \hat{v}(S_{t+1}\mathbf{w}_t)$.
- In classic methods, what we do is to change the current estimation value “towards” the update.
- The estimation of value function approximation is different:
  - Possibly very complex methods are possible.
  - We will use supervised learning (i.e., we will learn through sets of inputs-outputs).
    - Since we are trying to learn how to link the inputs to real values we can this as a function approximation problem.

Nonlinear Function Approximation: Artificial Neural Networks

- Artificial neural networks (ANN) are widely used for nonlinear function approximation.

- We have results that guarantees that an ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy (see Cybenko 1989).

- Nowadays, we tend to use networks with more than one hidden layer (we usually use the term deep learning for that reason).

- Learning is based on stochastic gradient descent.

Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of

Stochastic Gradient Descent

- Stochastic gradient descent methods (SGD) adjust the weights after each example by a small value in the direction that would most reduce the error on that example.

- We will discuss here the case for one example, but this can be done for multiple examples at a time (remember mini-batch gradient descent methods).

- Remember the formula we used for the update of the weight for a deep learning network (valid for a "shallow" artificial neural network as well):

$$w_j \leftarrow w_j + \Delta w_j = w_j - \eta \frac{\partial J}{\partial w_j}$$

- We will indicate time here, since we are updating w_j at each of a series of discrete time steps $t = 0,1,2,...$ as follows:

$$w_{j,t+1} \leftarrow w_{j,t} + \Delta w_{j,t} = w_{j,t} - \eta \frac{\partial J}{\partial w_{j,t}}$$

Stochastic Gradient Descent

- Assuming that we have a loss function that represents the mean squared error (which we call mean square error as follows:

$$J(\mathbf{w}) = \sum_{s \in \mathcal{S}} (U_t - \hat{v}(S_t, \mathbf{w}_t))^2$$

- We will obtain the following

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \frac{1}{2}\eta \nabla [U_t - \hat{v}(S_t, \mathbf{w}_t)]^2$$
$$= \mathbf{w}_t + \eta [U_t - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

where U_t is the target and η is the step-size parameter.

Stochastic Gradient Decent

- Recall the definition of gradient:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)$$

- Stochastic gradient methods are “gradient descent” methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example’s squared error.
  - Intuition: that is the direction in which the error falls more rapidly.
  - Recall: they are stochastic since they are based on samples that are randomly chosen.

Semi-gradient One-step Sarsa Prediction

- We can extend the mathematical framework we discussed for state value functions to state-action value functions.
- In this case the target U_t is an approximation of $q_\pi(S_t, A_t)$.
- The general gradient-descent update for action-value prediction is

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

- For example, the update for the *(episodic) semi-gradient one-step Sarsa method* is the following:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \eta[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

- It is called semi-gradient because we are not really taking the “real” gradient. The “real” gradient will require to calculate the gradient di U_t.
  - Why are we not doing it? Essentially complexity and convergent is guaranteed also in this case: for a constant policy, this method converges in the same way that TD(0) does.

Semi-gradient One-step Sarsa Control

- But, as usual, we are interested in the problem of *control,* i.e. we are interested finding the optimal policy.

- The method is essentially the same as for tabular methods.

- For each possible action a available in the current state S_t, we can compute $\hat{q}(S_t, a, \mathbf{w}_t)$ and then find the greedy action using

$$A_t^* = \underset{a}{\operatorname{argmax}}\, \hat{q}(S_t, a, \mathbf{w}_t)$$

- Policy treatment is then done by changing the estimation policy to a soft approximation of the greedy policy (e.g., ϵ-greedy policy).

- Actions are selected according to this same policy (this is *on-policy control*).

Semi-gradient One-step Sarsa Control

Input: a differentiable action-value function parameterisation $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step size $\eta > 0$ and small $\epsilon > 0$

Initialise value-function weights $\mathbf{w} \in \mathbb{R}$ arbitrarily

Loop for each episode:

$S, A \leftarrow$ initial state and action of episode (e.g., ϵ-greedy)

Loop for each step of episode:

Take action A, observe R, S'

If S' is terminal:

$\mathbf{w} \leftarrow \mathbf{w} + \eta[R - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$

Go to next episode

else:

Choose A' as a function of $\hat{q}(S', ., \mathbf{w})$ (e.g., ϵ-greedy)

$\mathbf{w} \leftarrow \mathbf{w} + \eta[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$

$S \leftarrow S'$ and $A \leftarrow A'$

Semi-gradient Q-Learning

- The basic procedure is similar for the “classic” Q-learning.
- The update is as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

where $\mathbf{w}_t$ is the vector of the network’s weights, A_t is the action selected at time t, and S_t and S_{t+1} are respectively the preprocessed image stacks input to the network at time steps t and $t + 1$.

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the

LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

The theory of reinforcement learning provides a normative account[1], deeply rooted in psychological[2] and neuroscientific[3] perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems[4,5], the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms[3]. While reinforcement learning agents have achieved some successes in a variety of domains[6–8], their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks[9–11] to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games[12]. We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a pro-

agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s,\ a_t = a,\ \pi\right],$$

which is the maximum sum of rewards r_t discounted by γ at each time-step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a) (see Methods)[19].

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function[20]. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_{a'} Q(s', a')$. We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay[21–23] that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution (see below for details). Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target.

While other stable methods exist for training neural networks in the

Credit: Wikimedia

210 x 160 pixel image frames with 128 colours at 60Hz

Deep Q-Network (DQN)

- Semi-gradient Q-learning is at the basis of the Deep-Q-Network (DQN) used for implementing the agent. You can actually use the two terms interchangeably.
- It combines Q-learning with a deep (convolutional) multi-layer (deep) network.
- The gradient is based on back propagation.
- DQN's reward is calculated as follows:
  - +1 whenever it is increased;
  - -1 whenever it is decreased;
  - 0 otherwise.
- In other words the positive (and negative) values were capped to +1 (and -1).

States and Actions

- The Atari games have a resolution of 210x160 pixel image frames with 128 colours at 60 Hz.

- Data were pre-processed in order to obtain a 84x84 array of luminance values.

- Because the full states of many of the Atari games are not completely observable from the images frames and to provide more information to the agent for the decision, the authors “stacked” the four most recent frames so that the inputs of the network has dimension 84x84x4.

  - Otherwise you would have ended up with a 1-order Markov system.

States and Actions

- The activation levels of DQN's outputs are the estimated optimal values of the corresponding state-value pairs given the stack of images in input.

- The assignment of output units to a game's actions varied from game to game (from 4 to 18).

  - Not in all the games the output values are actually considered (but they are still in the network).

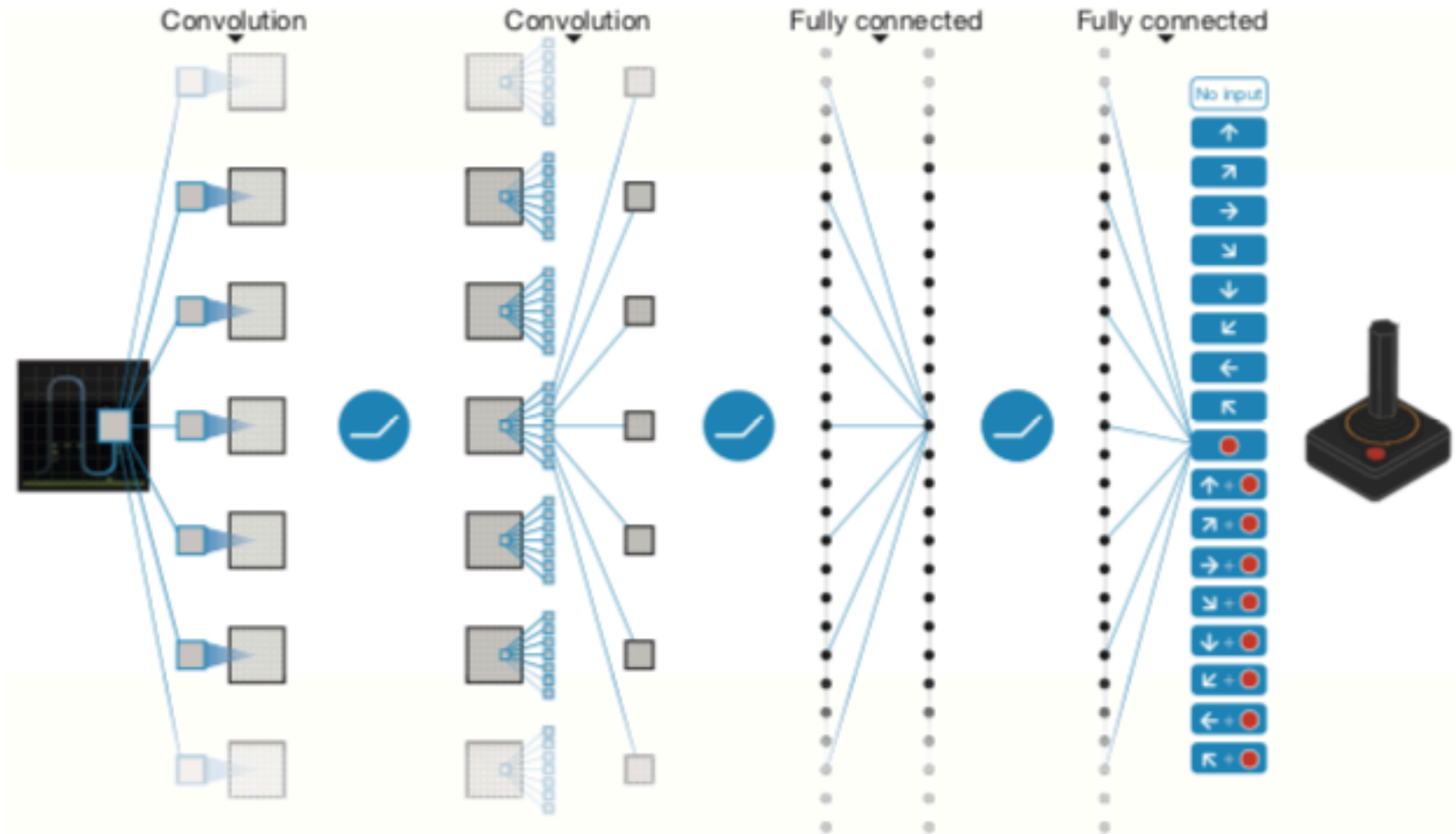


Figure 1 | Schematic illustration of the convolutional neural network. The details of the architecture are explained in the Methods. The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0,x)$).

Deep Q-Network (DQN)

- DQN uses an ϵ-greedy policy, with ϵ decreasing linearly over the first million frames and remaining at a low value for the rest of the learning session.
- The network is trained using a *mini-batch stochastic gradient descent* after processing 32 images (and previous 3 for each image).
  - Changes proportional to the running average to the magnitudes of recent gradients for that weight (RSMProp).
- A key modification to standard Q-learning is the use of *experience replay*.

LiveSlides web content

To view

Download the add-in.
liveslides.com/download

Start the presentation.

Experience Replay

- In DQN the authors made a series of changes to the standard basic Q-learning.

- First they used a method called *experience replay* in order to deal with the convergence problem of Q-learning.

- In Q-learning, if the updates are highly correlated, the learning process converges very slowly.

- The key idea of experience reply is to store agent's experience in a replay memory that is accessed to perform the weight updates.

Machine Learning, 8, 293–321 (1992)

Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching

LONG-JI LIN ljl@cs.cmu.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213

Abstract. To date, reinforcement learning has mostly been studied solving simple learning tasks. Reinforcement learning methods that have been studied so far typically converge slowly. The purpose of this work is thus two-fold: 1) to investigate the utility of reinforcement learning in solving much more complicated learning tasks than previously studied, and 2) to investigate methods that will speed up reinforcement learning.

This paper compares eight reinforcement learning frameworks: *adaptive heuristic critic* (*AHC*) learning due to Sutton, *Q-learning* due to Watkins, and three extensions to both basic methods for speeding up learning. The three extensions are experience replay, learning action models for planning, and teaching. The frameworks were investigated using connectionism as an approach to generalization. To evaluate the performance of different frameworks, a dynamic environment was used as a testbed. The environment is moderately complex and nondeterministic. This paper describes these frameworks and algorithms in detail and presents empirical evaluation of the frameworks.

Keywords. Reinforcement learning, planning, teaching, connectionist networks

Experience Replay

- The training is done through an emulator.

- After the emulator executes action A_t in a state represented by image stack S_t and returned reward R_{t+1} and the following image stack S_{t+1}, it added the tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ to the replay memory (also called replay buffer).

- The experience replay memory accumulates experiences over many plays of the same game.

- Each Q-learning update is performed by randomly sampling from the experience replay buffer.

Experience Replay

- Instead of S_{t+1} becoming the new S_t for the next update, as it would be in the usual Q-learning, a new unconnected experience is drawn from the replay memory.
  - Important point: since Q-learning is an off-policy algorithm, it does not need to learn from “connected” trajectories.
- The random selection of uncorrelated “experiences” reduces the variance of the update. This allows Q-learning to converge more quickly.
  - Dealing with the variance of the update is a key issue for RL and it is an active area of research.

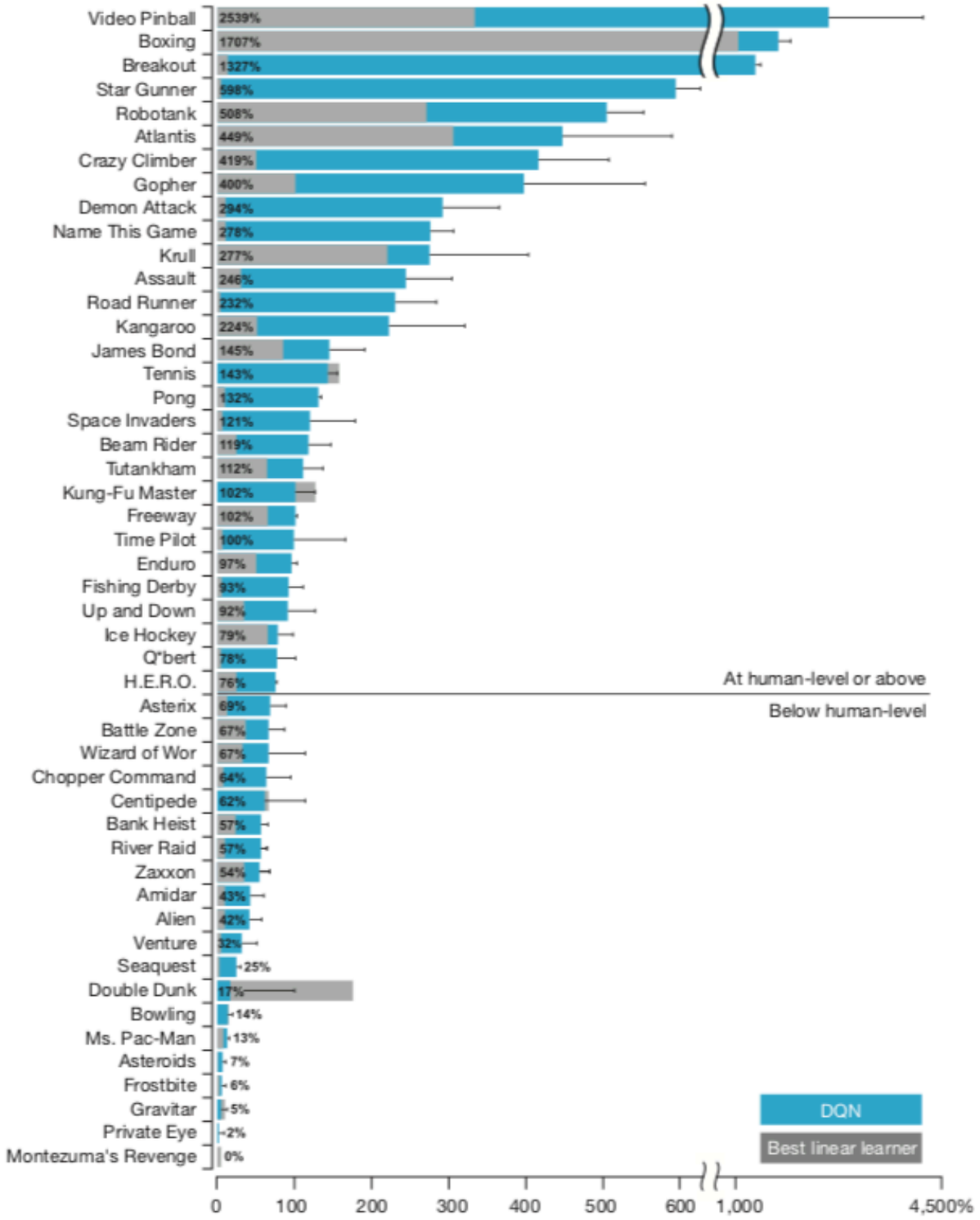


Figure 3 | Comparison of the DQN agent with the best reinforcement learning methods[15] in the literature. The performance of DQN is normalized with respect to a professional human games tester (that is, 100% level) and random play (that is, 0% level). Note that the normalized performance of DQN, expressed as a percentage, is calculated as: 100 × (DQN score − random play score)/(human score − random play score). It can be seen that DQN outperforms competing methods (also see Extended Data Table 2) in almost all the games, and performs at a level that is broadly comparable with or superior to a professional human games tester (that is, operationalized as a level of 75% or above) in the majority of games. Audio output was disabled for both human players and agents. Error bars indicate s.d. across the 30 evaluation episodes, starting with different initial conditions.

References

- Chapters 9 and 16 of Barto and Sutton. Introduction to Reinforcement Learning. Second Edition. MIT Press. 2018.