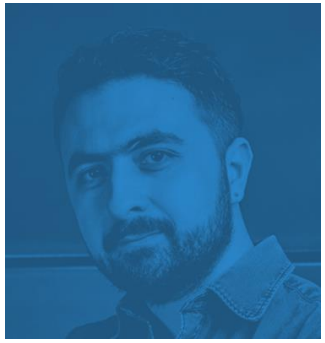
Intelligent Systems Programming

Lecture 4: DeepMind Game Algorithms

DeepMind Founders, acquired by Google 2014, HQ in London

- Mantra: Solve intelligence. Use it to make the world a better place.

Demis Hassabis, CEO



Mustafa Suleyman

Shane Legg

Readings and Project

SSS17 (URL link on Learnit)

ARTICLE

doi:10.1038/nature24270

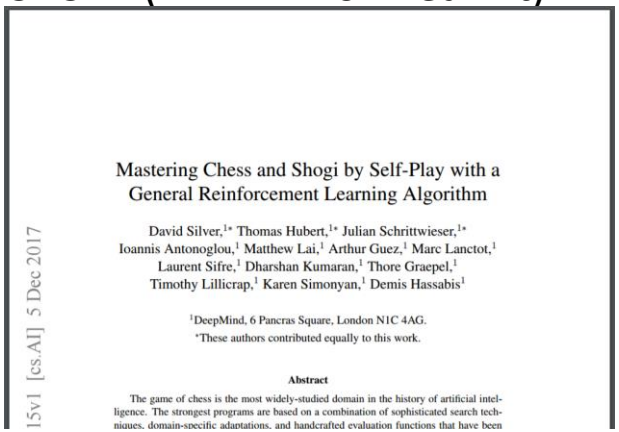
Mastering the game of Go without human knowledge

David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here we introduce an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of the tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program AlphaGo Zero achieved superhuman performance, winning 100–0 against the previously published, champion-defeating AlphaGo.

Much progress towards artificial intelligence has been made using supervised learning systems that are trained to replicate the decisions of human experts[1–4]. However, expert data sets are often expensive, trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it uses only the black and white stones from the board as input features.

SHS17 (PDF Link on Learnit)



15v1 [cs.AI] 5 Dec 2017

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

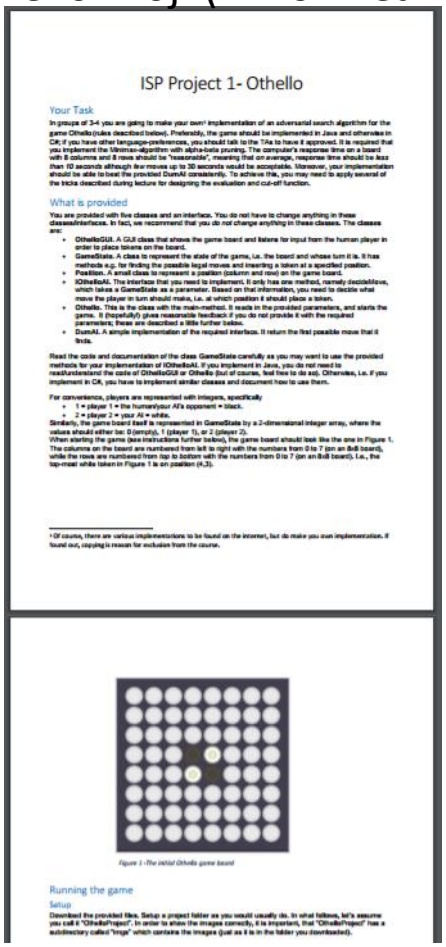
David Silver,[1]* Thomas Hubert,[1]* Julian Schrittwieser,[1]* Ioannis Antonoglou,[1] Matthew Lai,[1] Arthur Guez,[1] Marc Lanctot,[1] Laurent Sifre,[1] Dharshan Kumaran,[1] Thore Graepel,[1] Timothy Lillicrap,[1] Karen Simonyan,[1] Demis Hassabis[1]

[1]DeepMind, 6 Pancras Square, London N1C 4AG.
*These authors contributed equally to this work.

Abstract

The game of chess is the most widely-studied domain in the history of artificial intelligence. The strongest programs are based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions that have been

Othello Proj. (ZIP on Learnit)



ISP Project 1- Othello

Your Task

In groups of 3-4 you are going to make your own* implementation of an adversarial search algorithm for the game Othello (rules described below). Preferably, the game should be implemented in Java and otherwise in C#; if you have other language-preferences, you should talk to the TAs to have it approved. It is required that you implement the Minimax-algorithm with alpha-beta pruning. The computer's response time on a board with 8 columns and 8 rows should be "reasonable", meaning that on average, response time should be less than 10 seconds although few moves up to 30 seconds would be acceptable. Moreover, your implementation should be able to beat the provided DumAI consistently. To achieve this, you may need to apply several of the tricks described during lecture for designing the evaluation and cut-off function.

What is provided

You are provided with five classes and an interface. You do not have to change anything in these classes/interfaces. In fact, we recommend that you *do not change anything* in these classes. The classes are:

- **OthelloGUI**. A GUI class that shows the game board and listens for input from the human player in order to place tokens on the board.
- **GameState**. A class to represent the state of the game, i.e. the board and whose turn it is. It has methods e.g. for finding the possible legal moves and inserting a token at a specified position.
- **Position**. A small class to represent a position (column and row) on the game board.
- **IOthelloAI**. The interface that you need to implement. It only has one method, namely decideMove, which takes a GameState as a parameter. Based on that information, you need to decide what move the player in turn should make, i.e. at which position it should place a token.
- **Othello**. This is the class with the main-method. It reads in the provided parameters, and starts the game. It (hopefully!) gives reasonable feedback if you do not provide it with the required parameters; these are described a little further below.
- **DumAI**. A simple implementation of the required interface. It return the first possible move that it finds.

Read the code and documentation of the class GameState carefully as you may want to use the provided methods for your implementation of IOthelloAI. If you implement in Java, you do not need to read/understand the code of OthelloGUI or Othello (but of course, feel free to do so). Otherwise, i.e. if you implement in C#, you have to implement similar classes and document how to use them.

For convenience, players are represented with integers, specifically

- 1 = player 1 = the human/your AI's opponent = black.
- 2 = player 2 = your AI = white.

Similarly, the game board itself is represented in GameState by a 2-dimensional integer array, where the values should either be: 0 (empty), 1 (player 1), or 2 (player 2).
When starting the game (see instructions further below), the game board should look like the one in Figure 1. The columns on the board are numbered from left to right with the numbers from 0 to 7 (on an 8x8 board), while the rows are numbered from *top to bottom* with the numbers from 0 to 7 (on an 8x8 board). I.e., the top-most white token in Figure 1 is on position (4,3).

*Of course, there are various implementations to be found on the internet, but do make you own implementation. If found out, copying is reason for exclusion from the course.

Figure 1 - The initial Othello game board

Running the game

Setup

Download the provided files. Setup a project folder as you would usually do. In what follows, let's assume you call it "OthelloProject". In order to show the images correctly, it is important, that "OthelloProject" has a subdirectory called "imgs" which contains the images (just as it is in the folder you downloaded).

Today's Program

[10-10:50] Methods

- Reinforcement learning
- Convolutional neural networks

[11-11:50] AlphaGo Zero

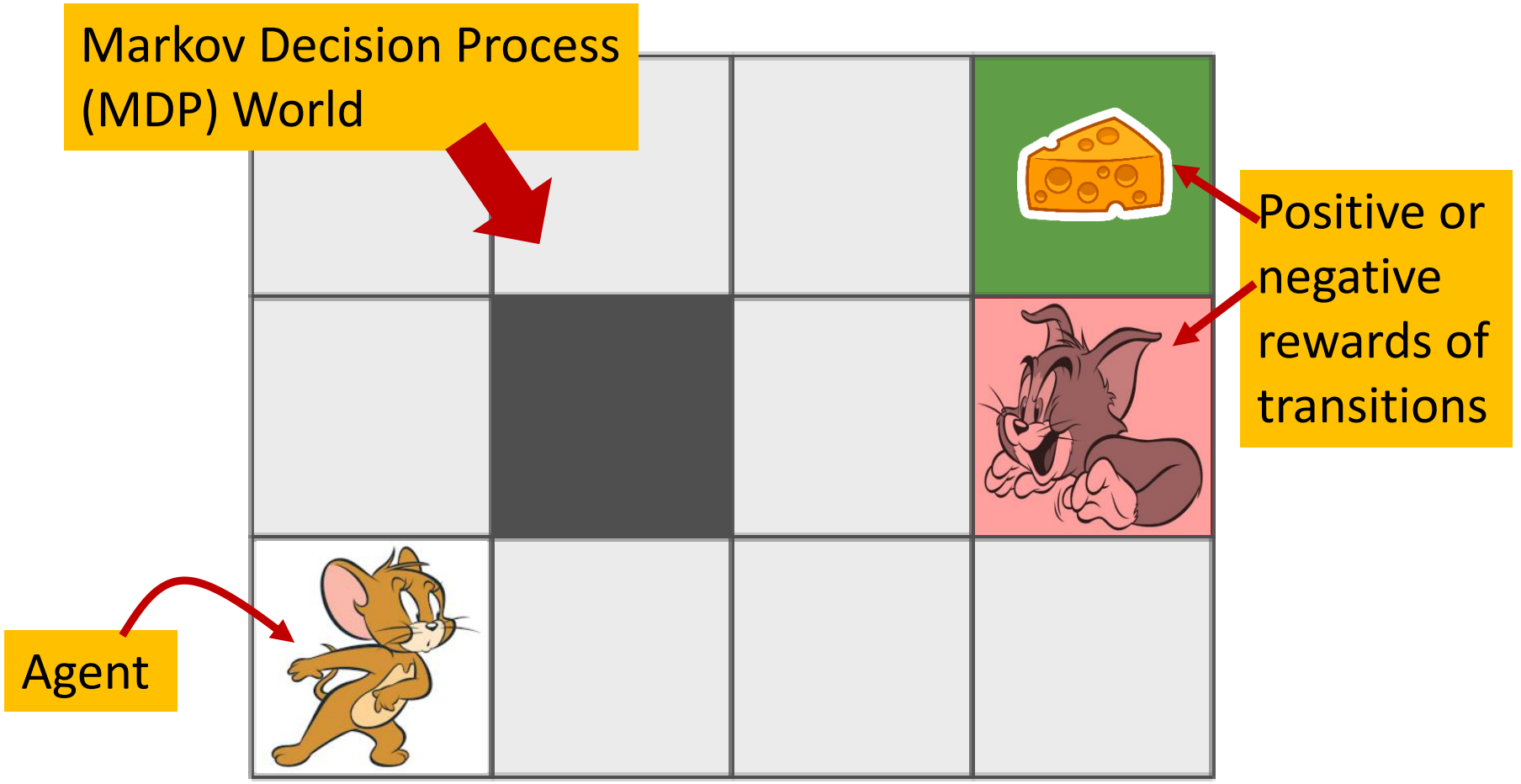
- AlphaGo Zero's neural network
- Self-play reinforcement learning
- Results

[12-14] Exercises

What's the Fuss About DeepMind?

How to Learn a Game from Zero Knowledge

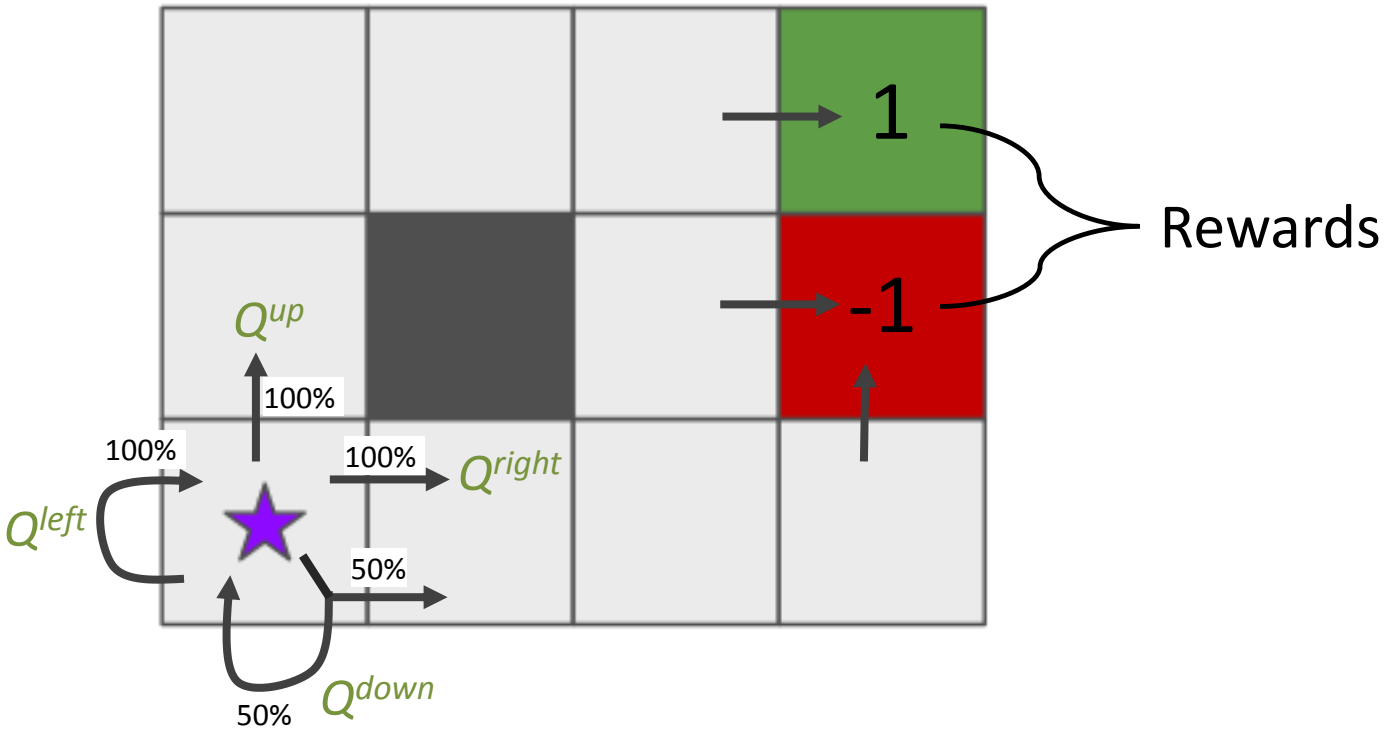
Reinforcement Learning (RL)



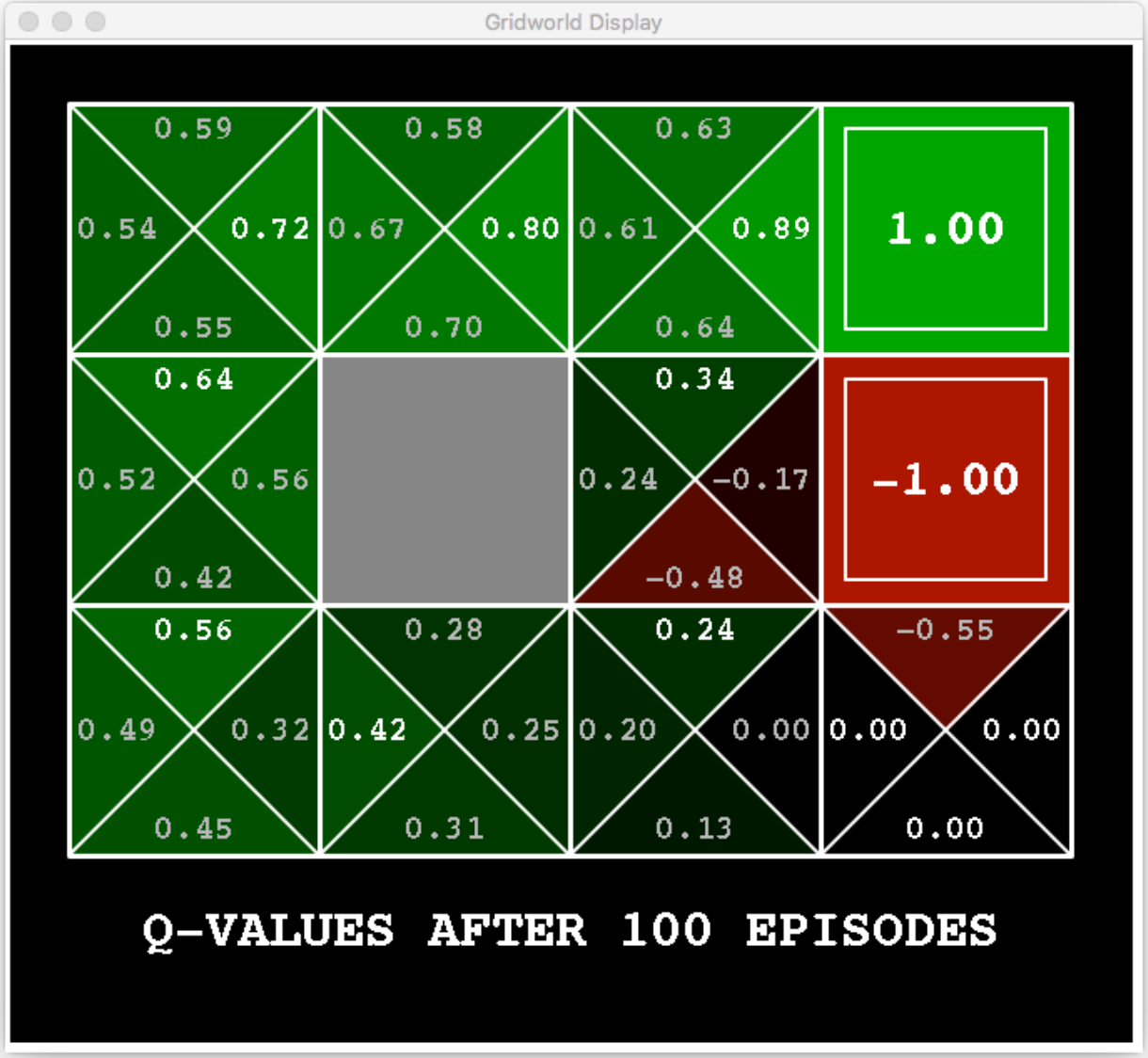
Q-Learning

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

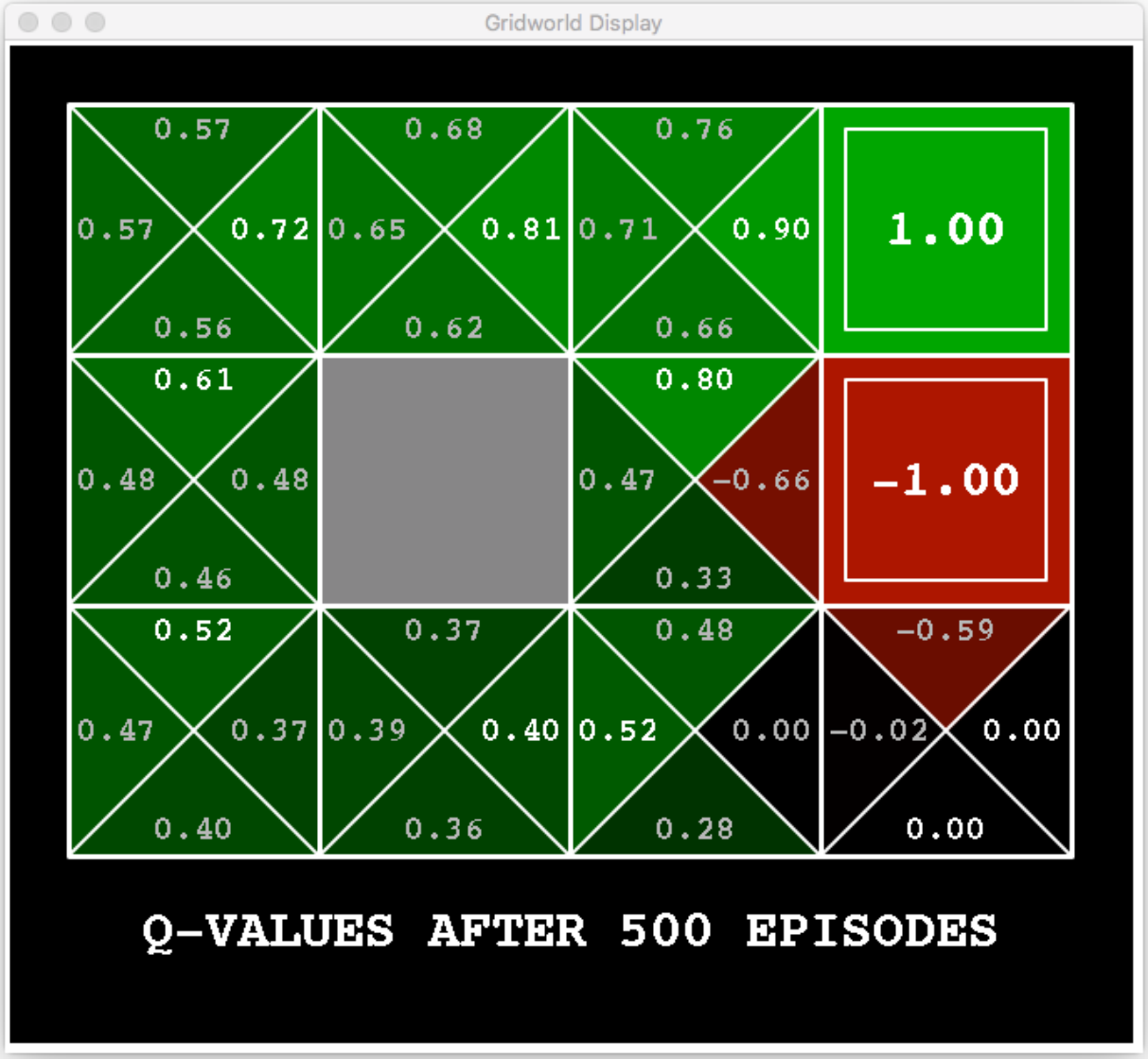
Policy (π):
Take action with max Q-value



Q-Learning Example (1/2)

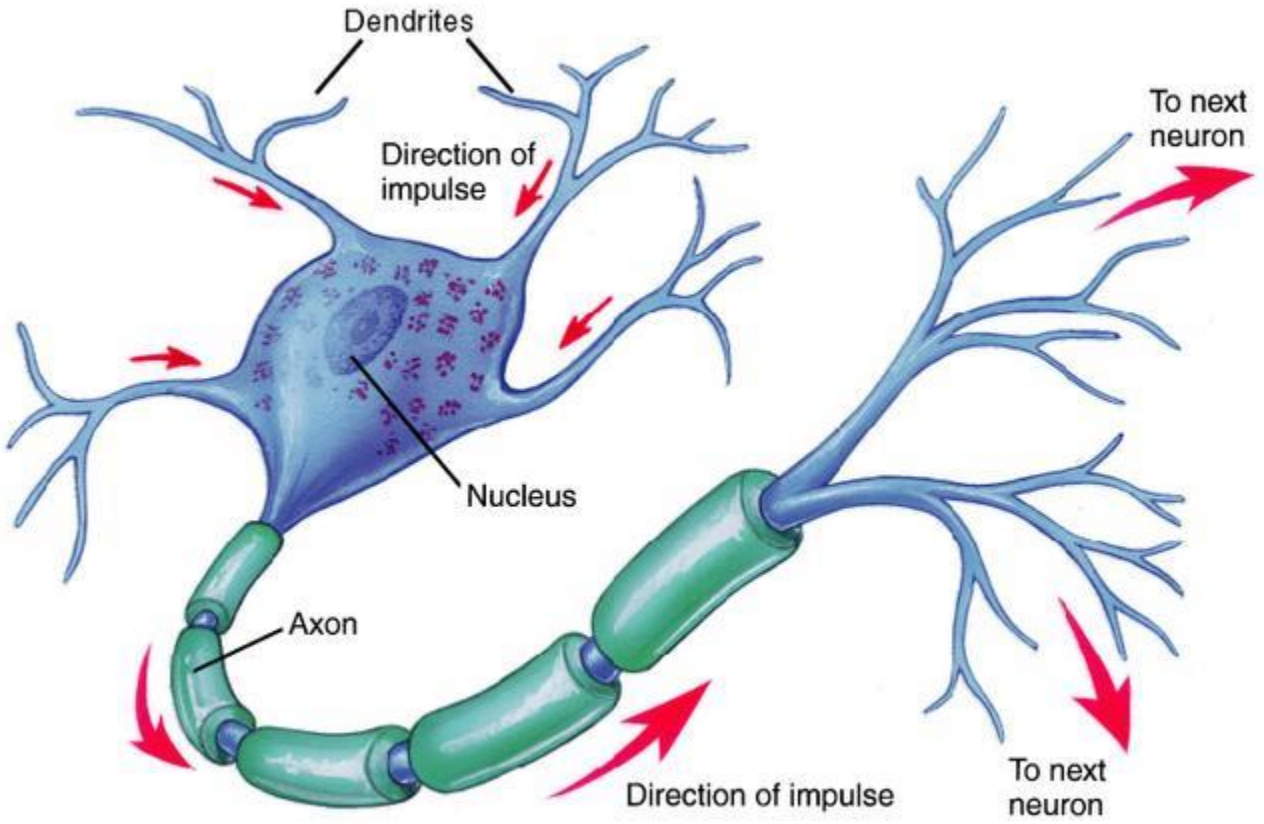


Q-Learning Example (2/2)

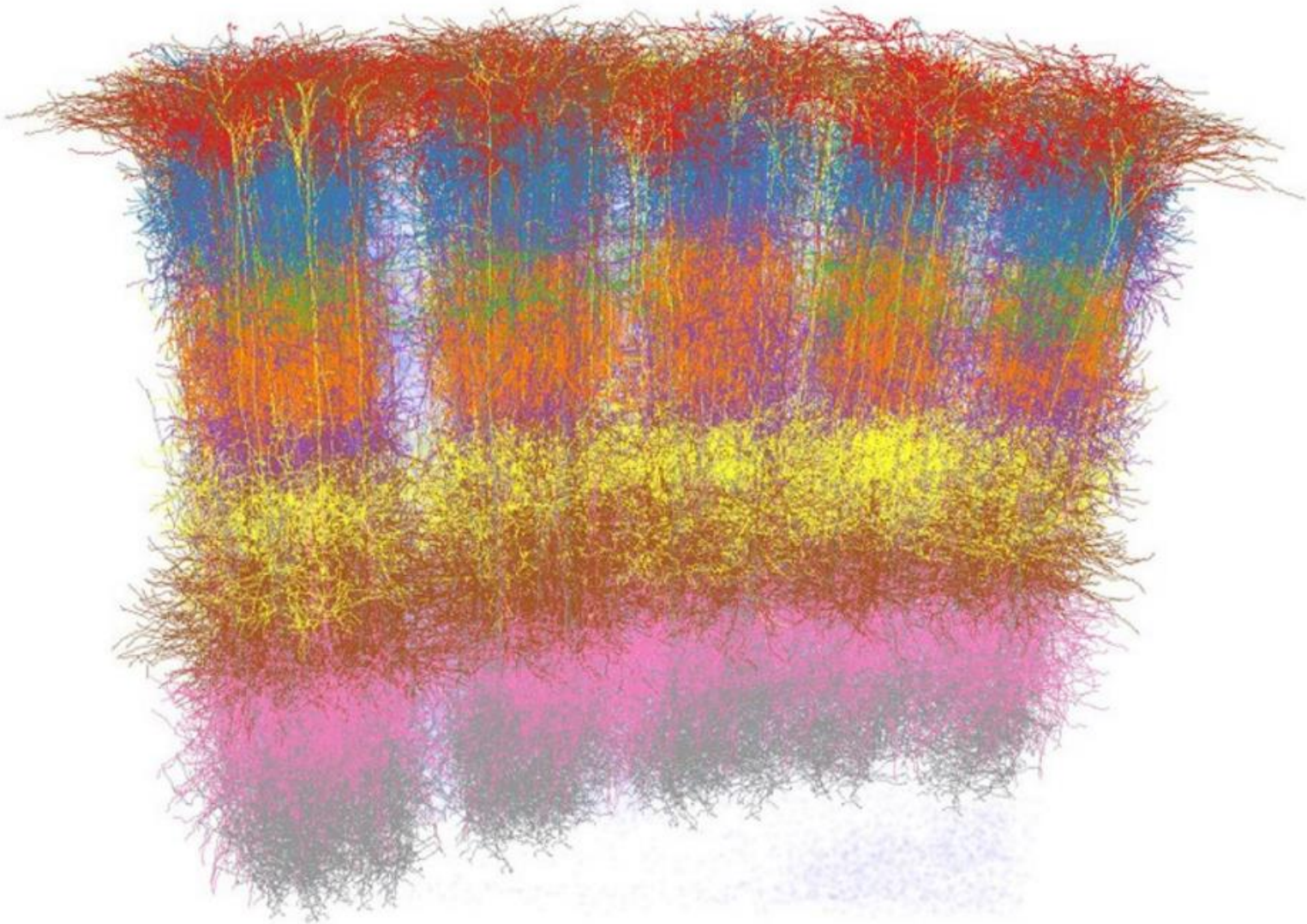


Trick: Use a Neural Network to Represent the Q-Table

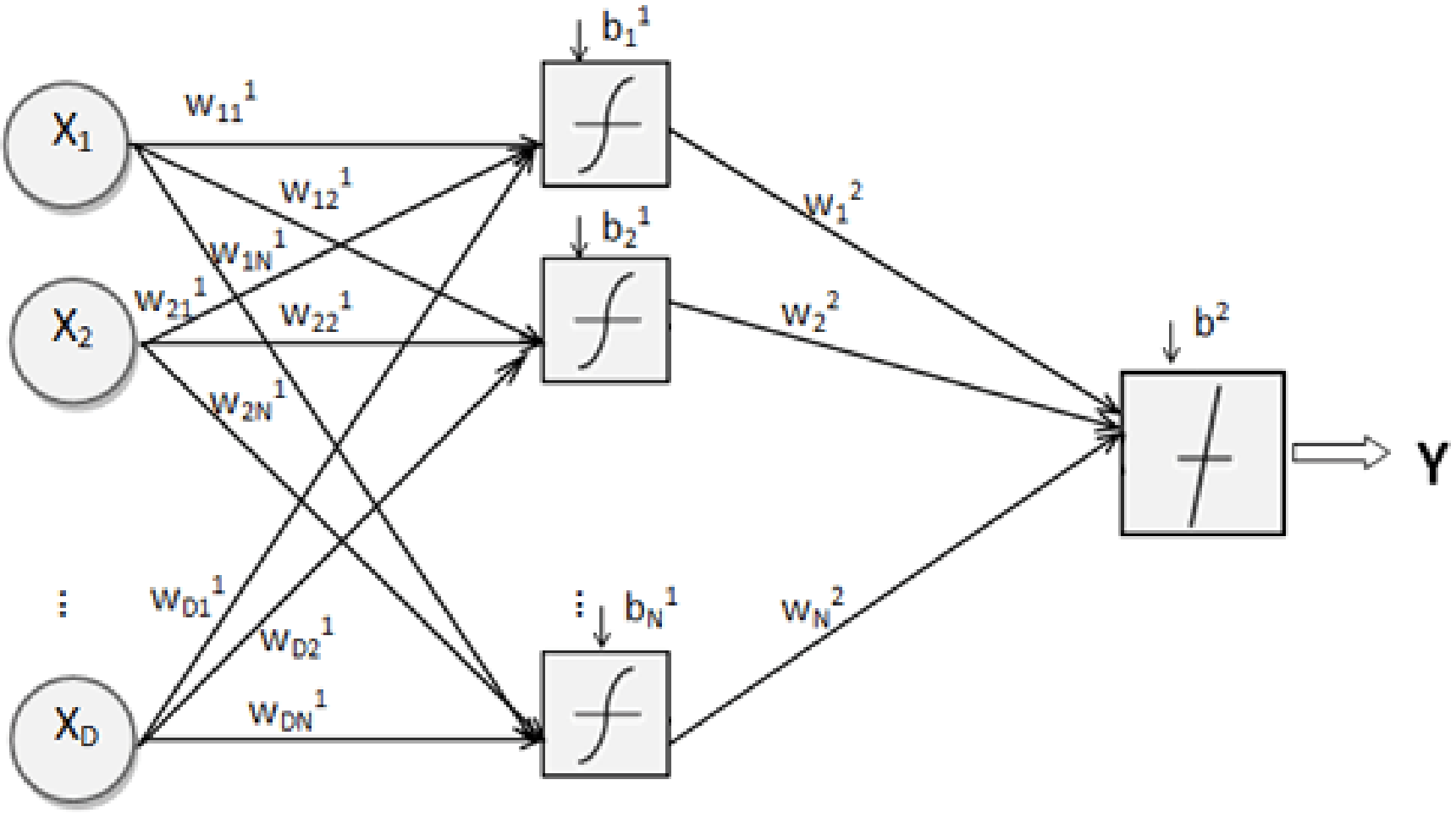
Neurons



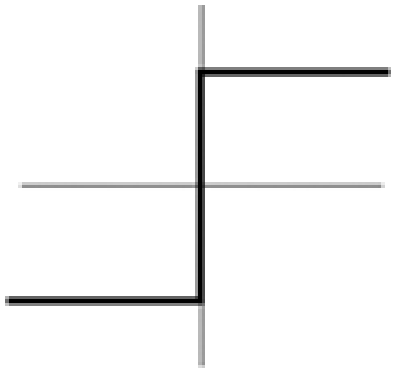
Neuron Network in Cortex



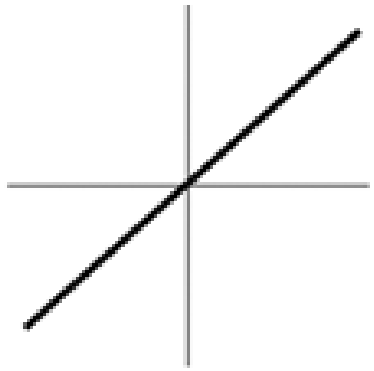
Artificial Neural Networks (ANNs)



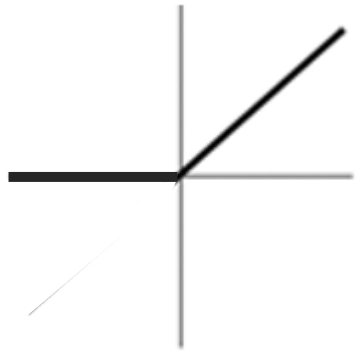
Popular Threshold Units



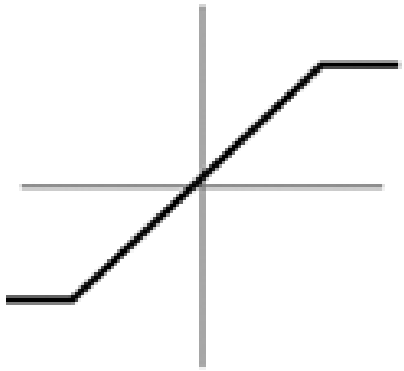
Step Function



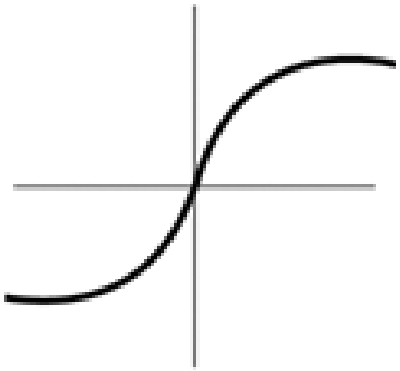
Linear Function



Rectified linear unit (ReLU)



Threshold Logic

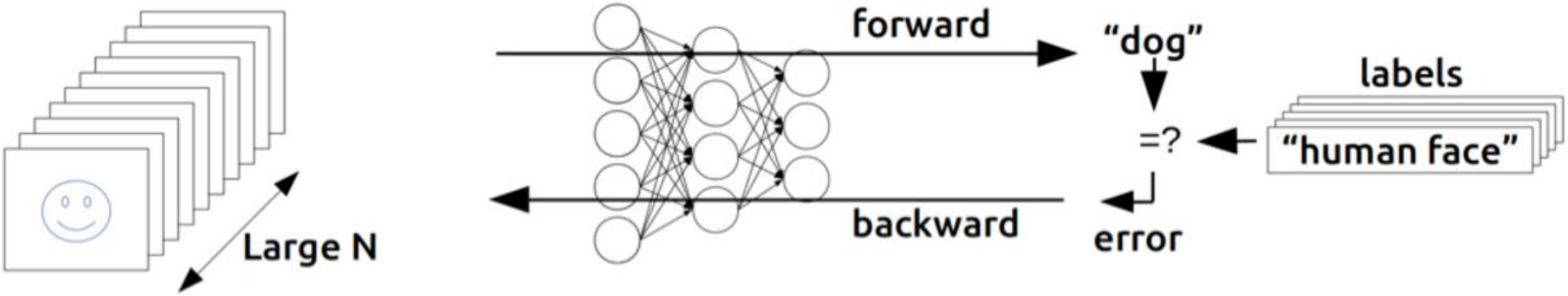


Sigmoid Function

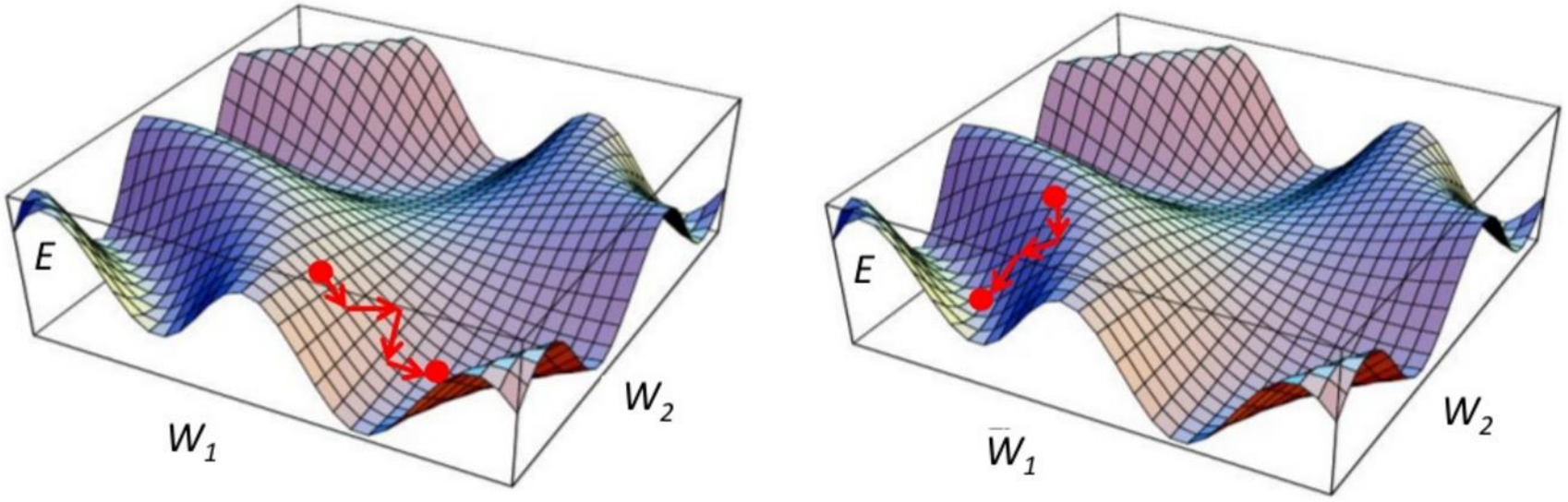
tanh()

http://www.yaldex.com

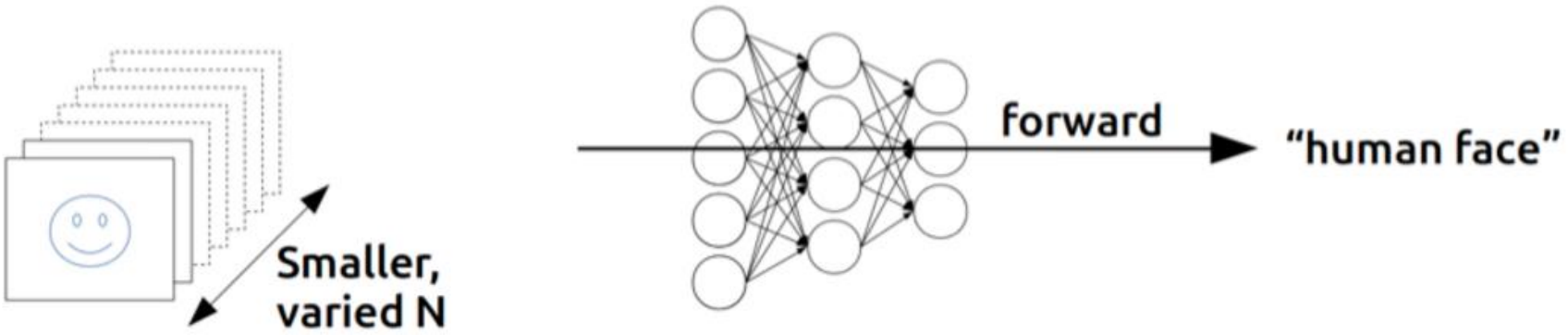
Training an ANN



Gradient Descent



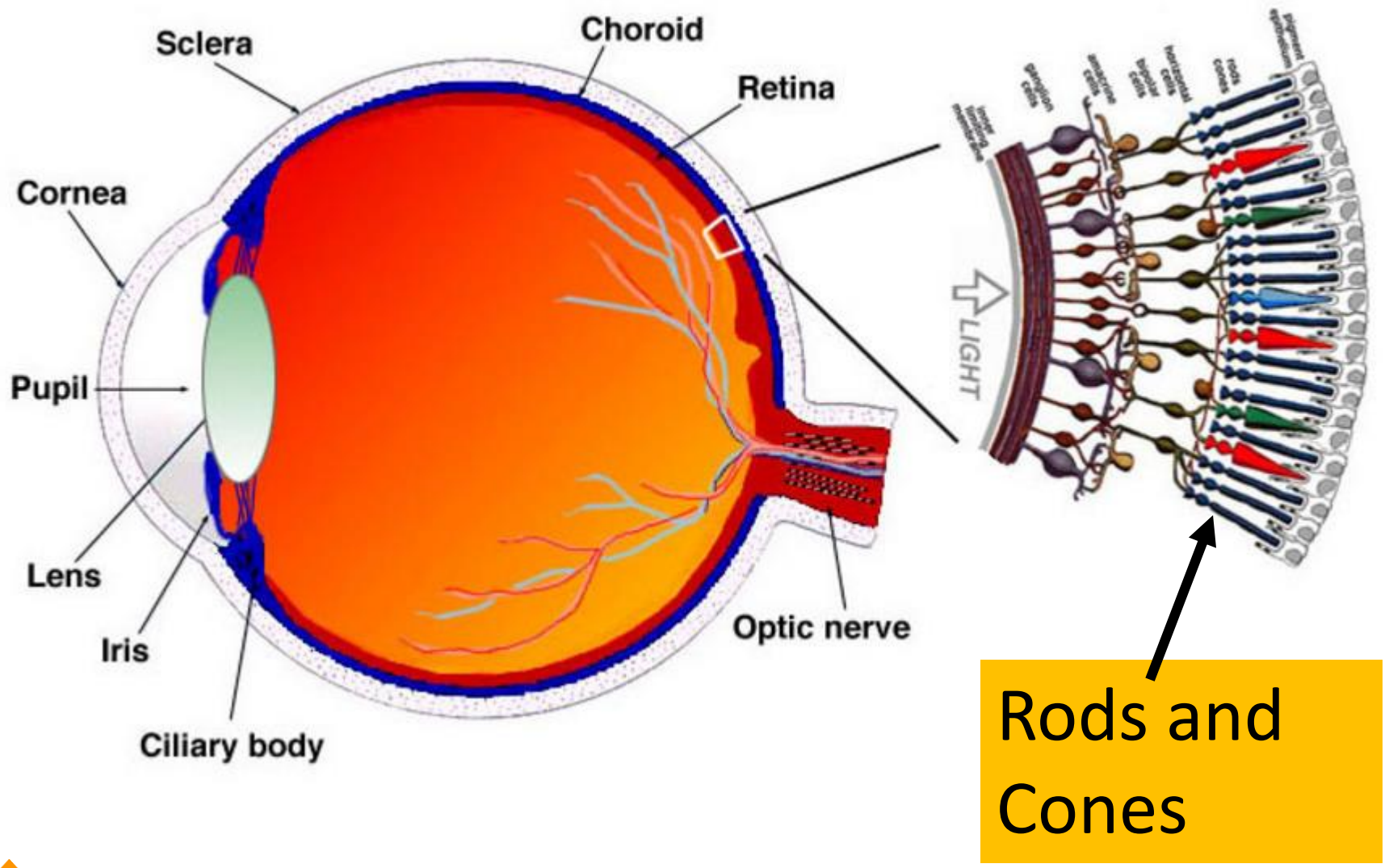
Classifying with an ANN



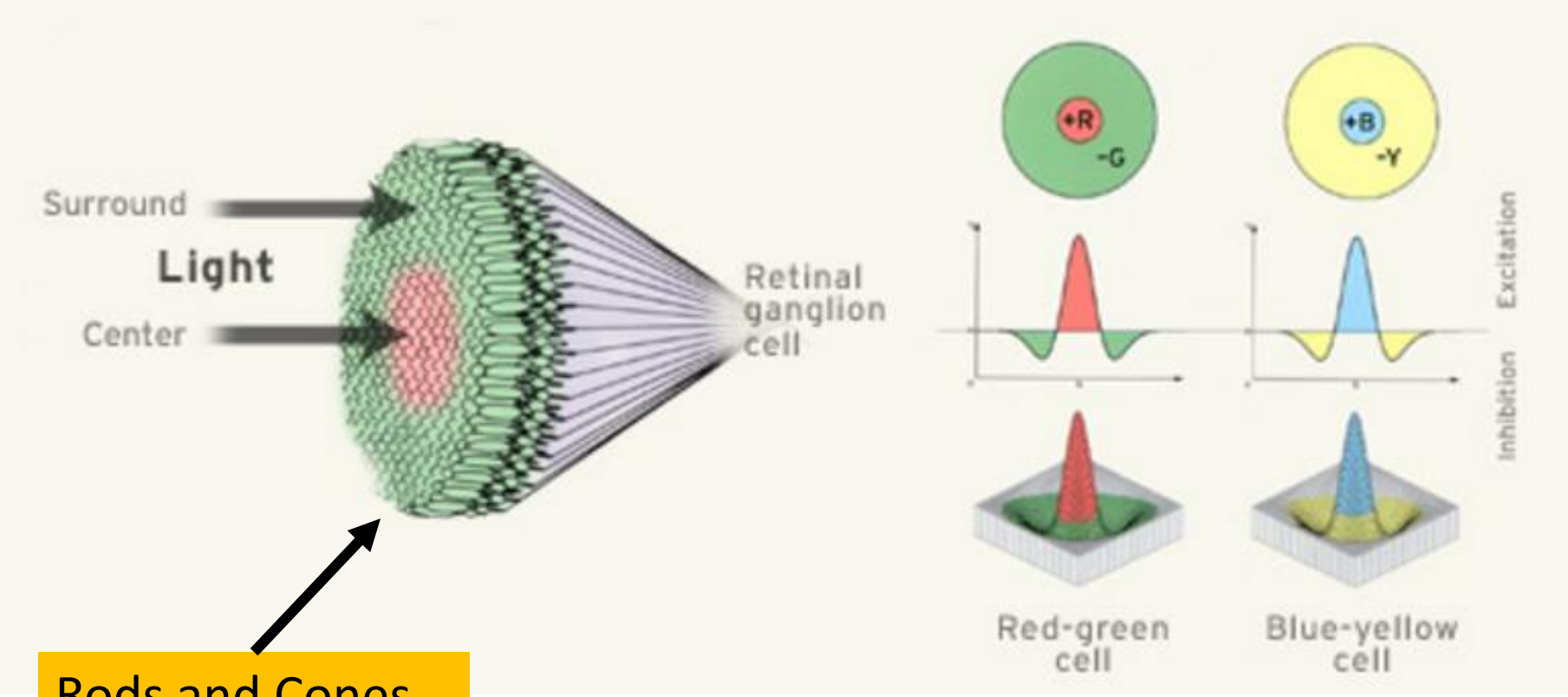
Practical Challenges

- Curse of dimensionality: too many weights to learn
- In image classification, full connectivity overkill
- Idea: reduce number of weights by imitating visual perception

Visual Processing Starts in Retina

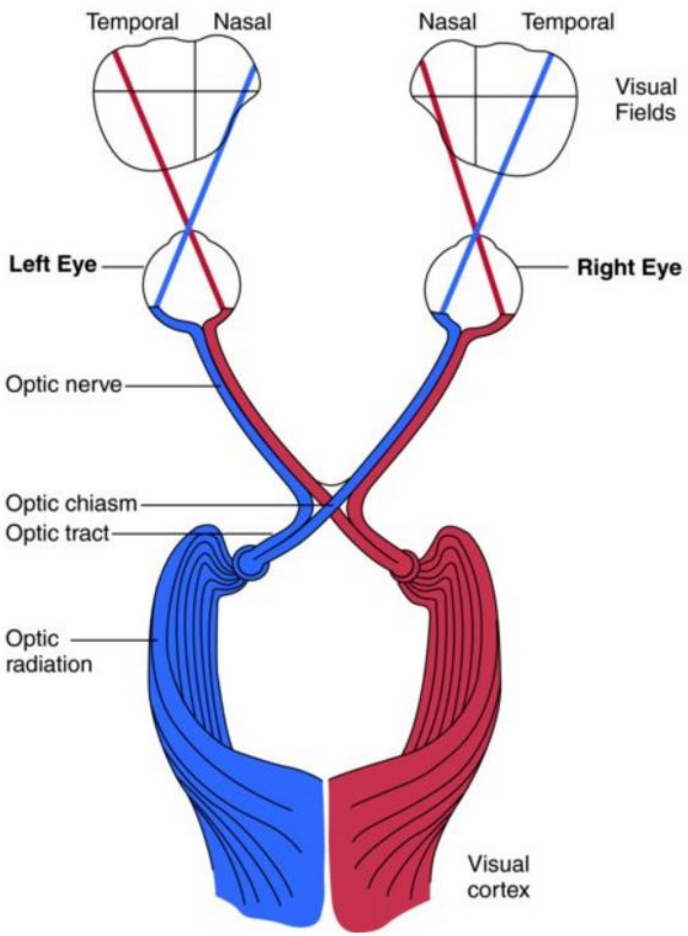


Receptive Field of Ganglion Cells

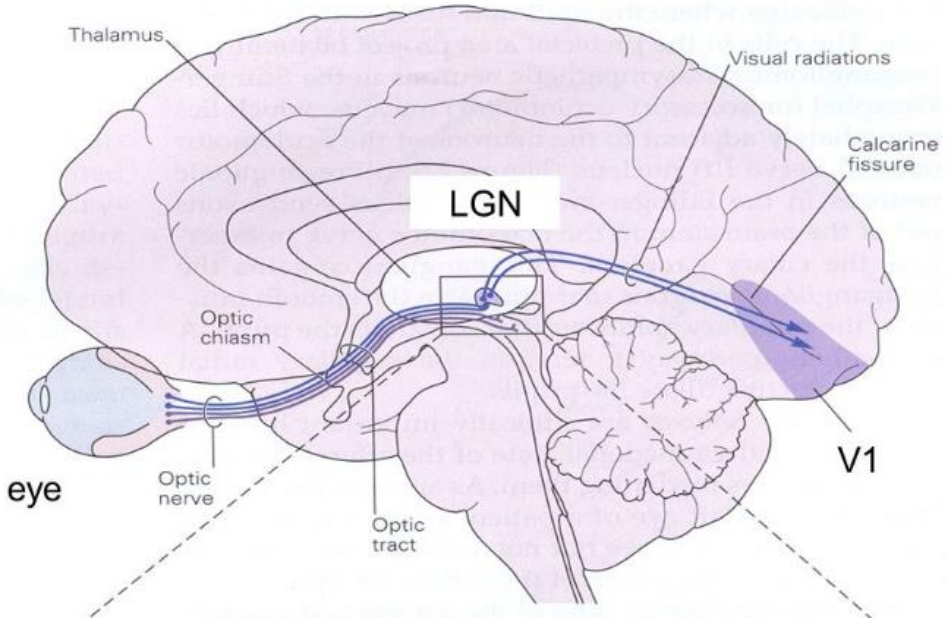


Rods and Cones

Central Visual Pathways

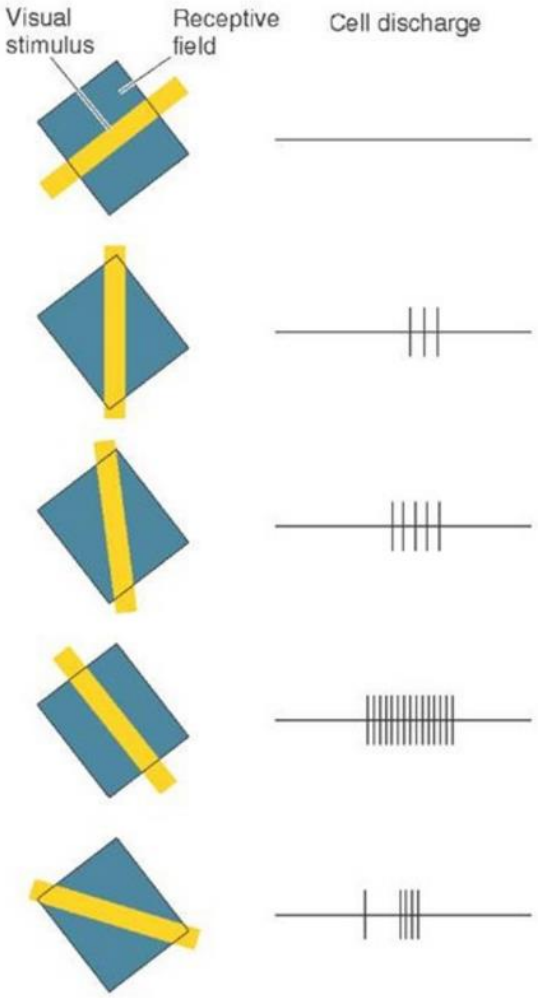


Top View



Side View

Receptive Field of Simple Cells in V1

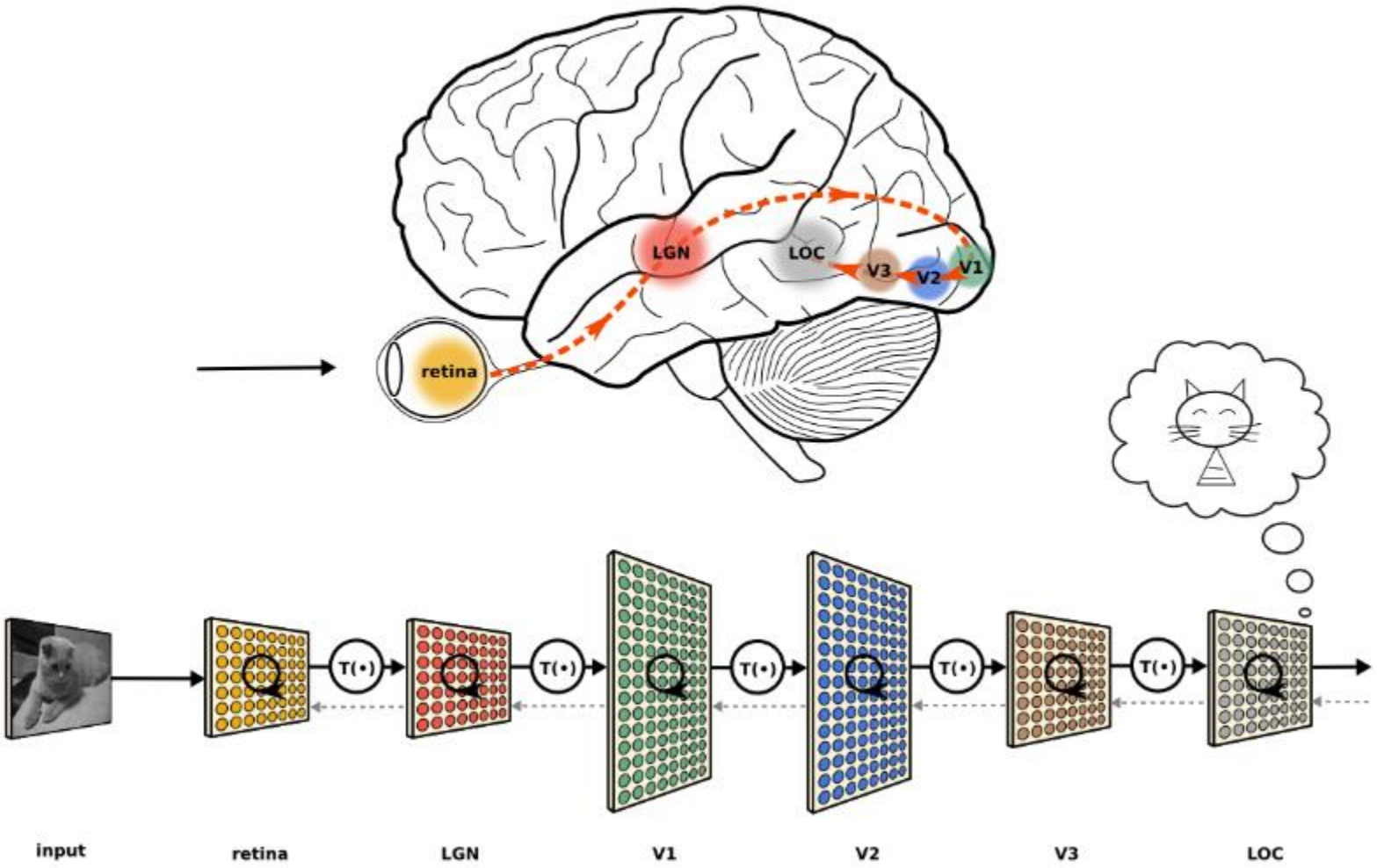


Conclusions

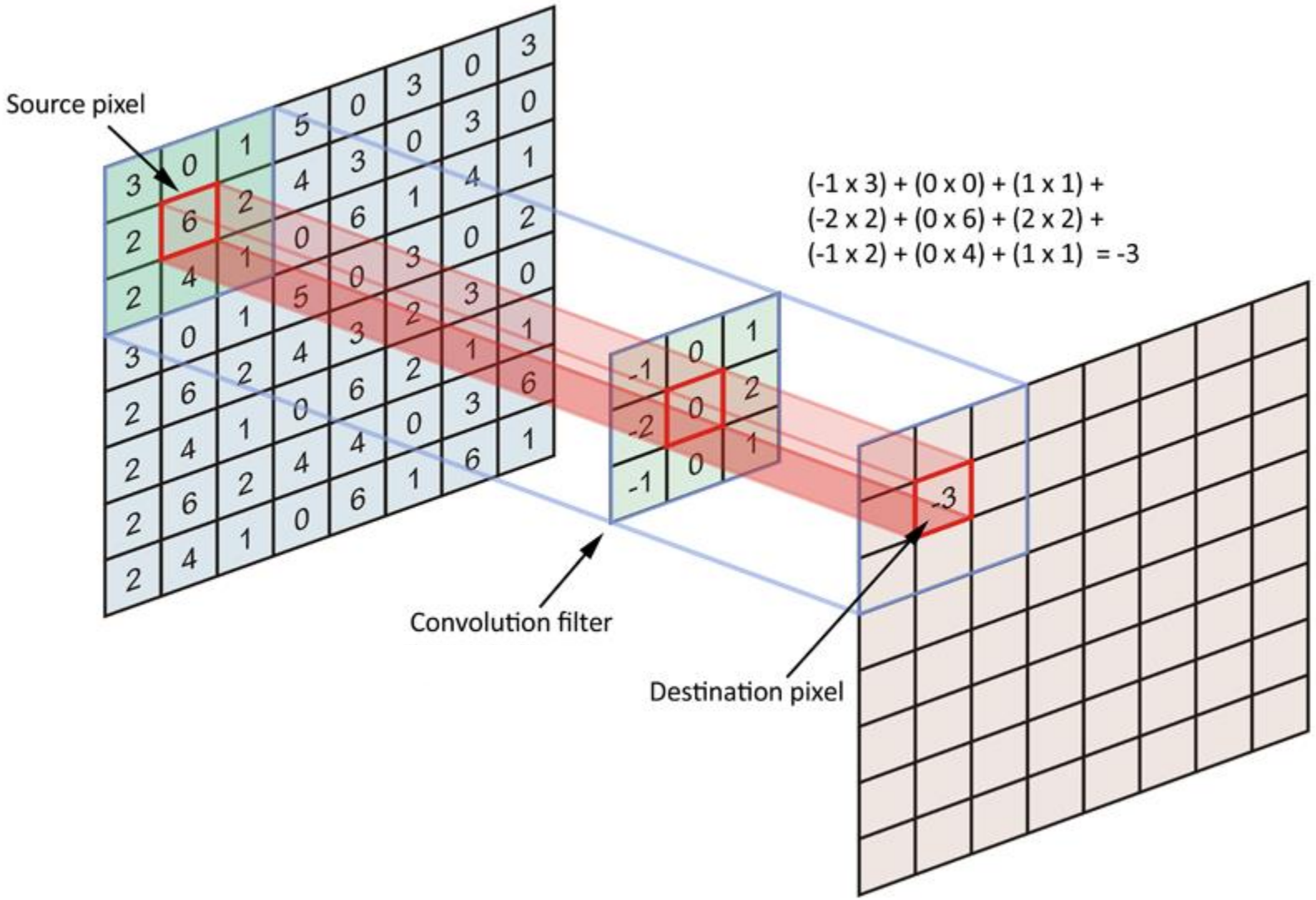
- Retina and V1 are topologically organized
- Ganglion and simple cells have receptive fields equal to **convolution filters**

Modelling the Visual Pathway

neuwritesd.org

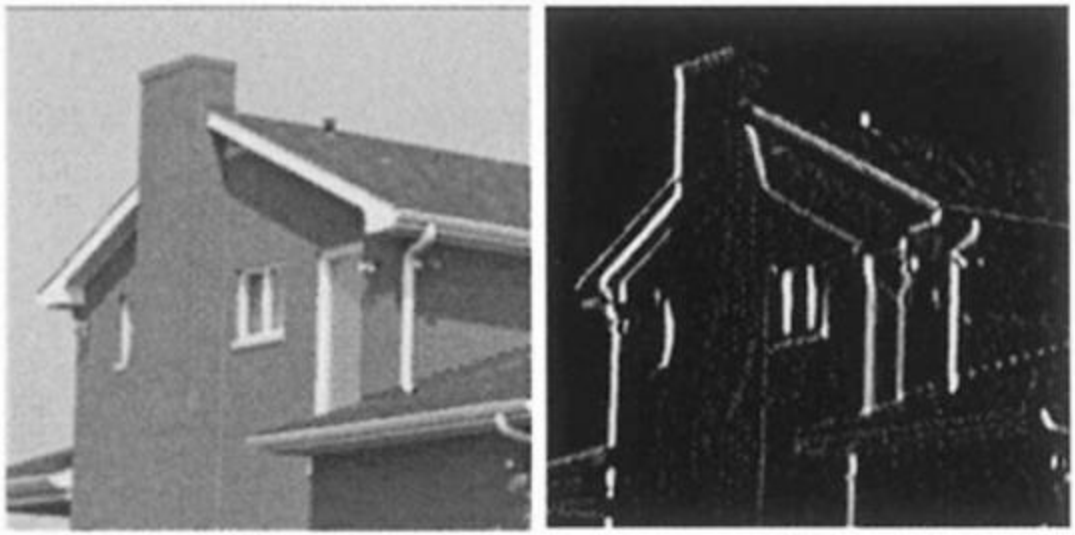


Convolution Filter



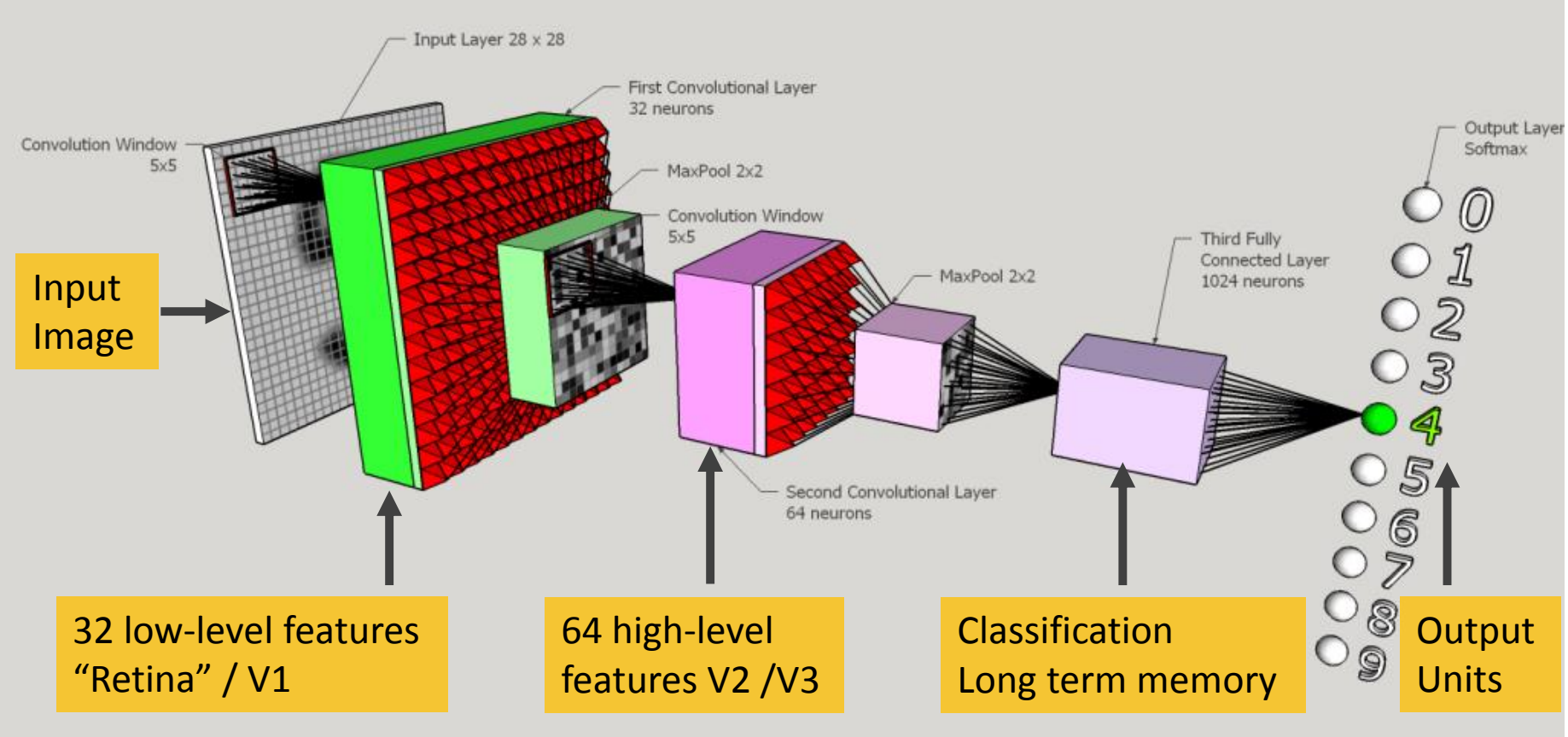
Example: Vertical Edge Detection

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$



what-when-how.com

Convolutional Neural Networks (CNNs)

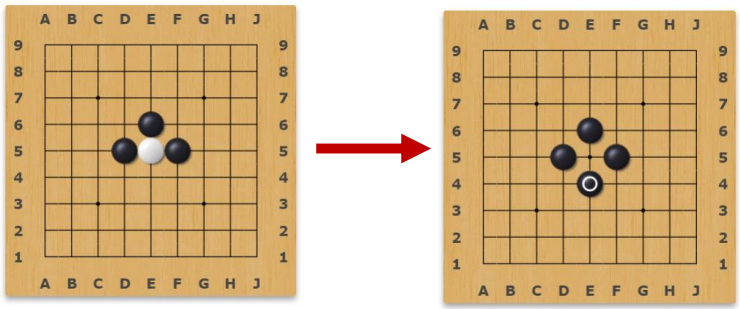


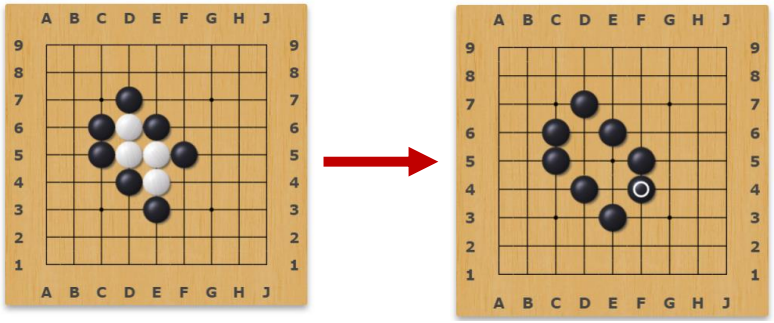
zderadicka.eu

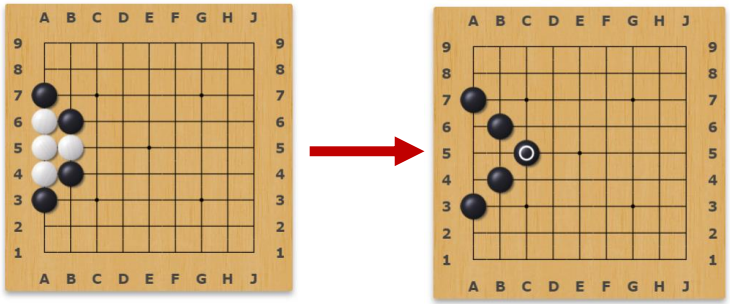
Break

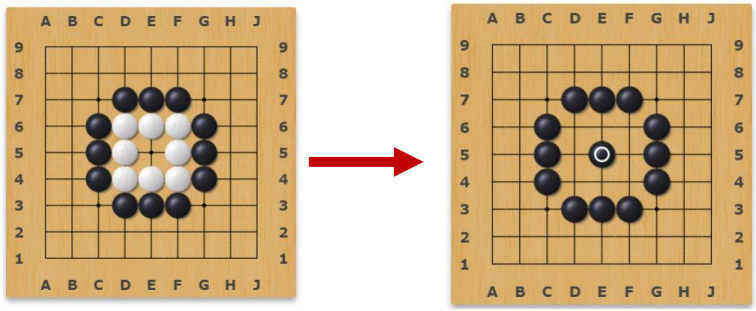
Go: A Zero-Sum Turn-Taking Game

Captures

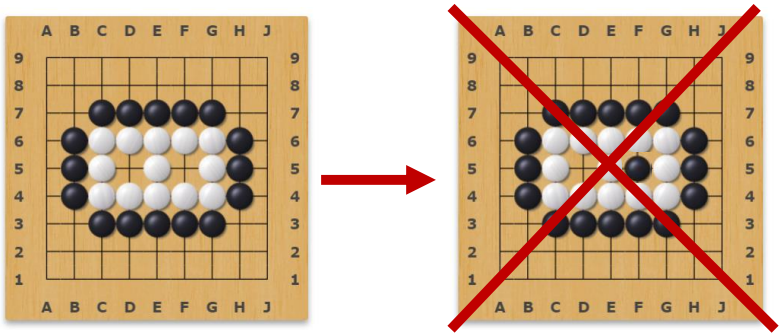




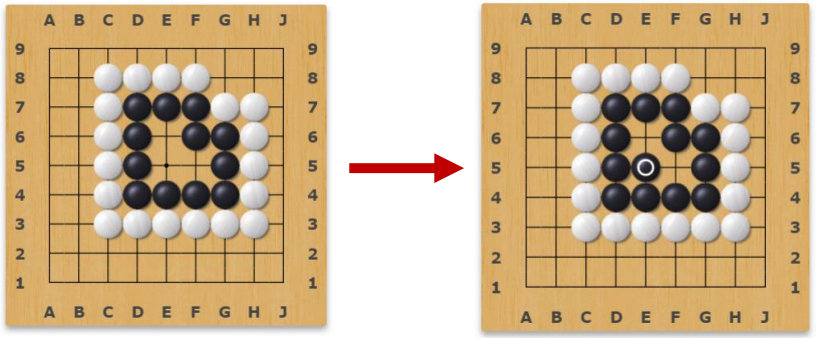




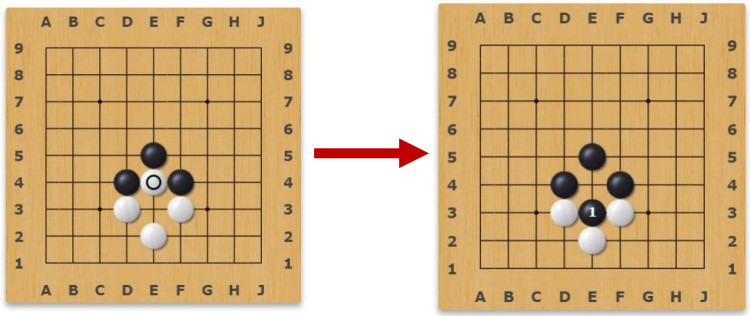
Capture Protection and Ko Rule



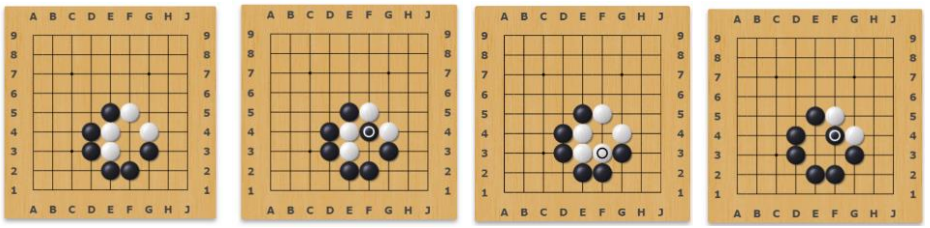
No capture with two holes!
Black F5 will suicide off board…



Capture protection

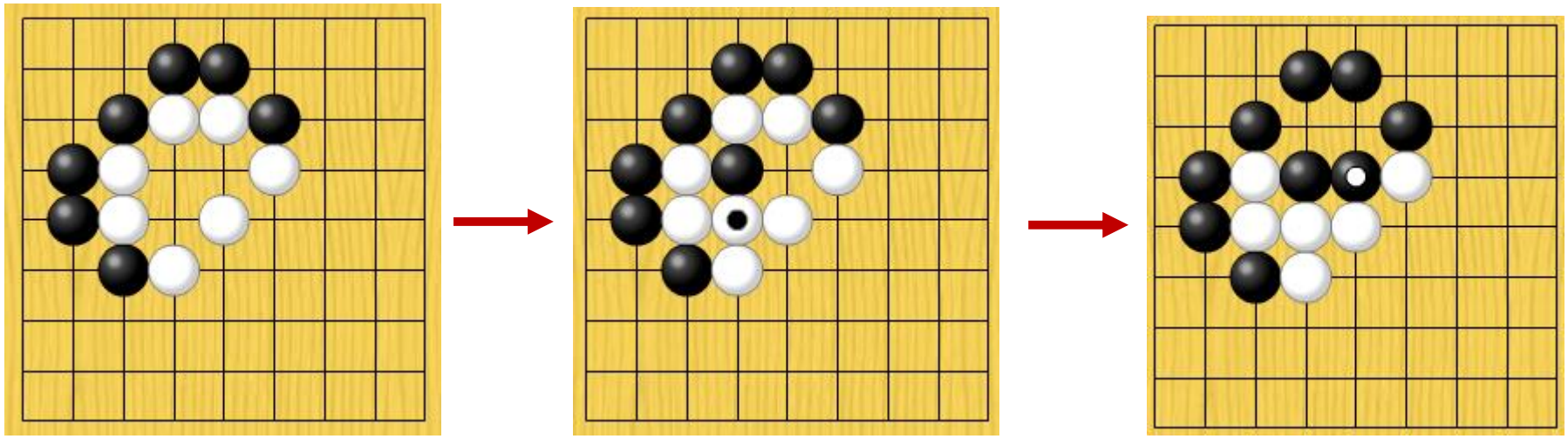


Ko rule: White cannot do E4 in next move, but ok later

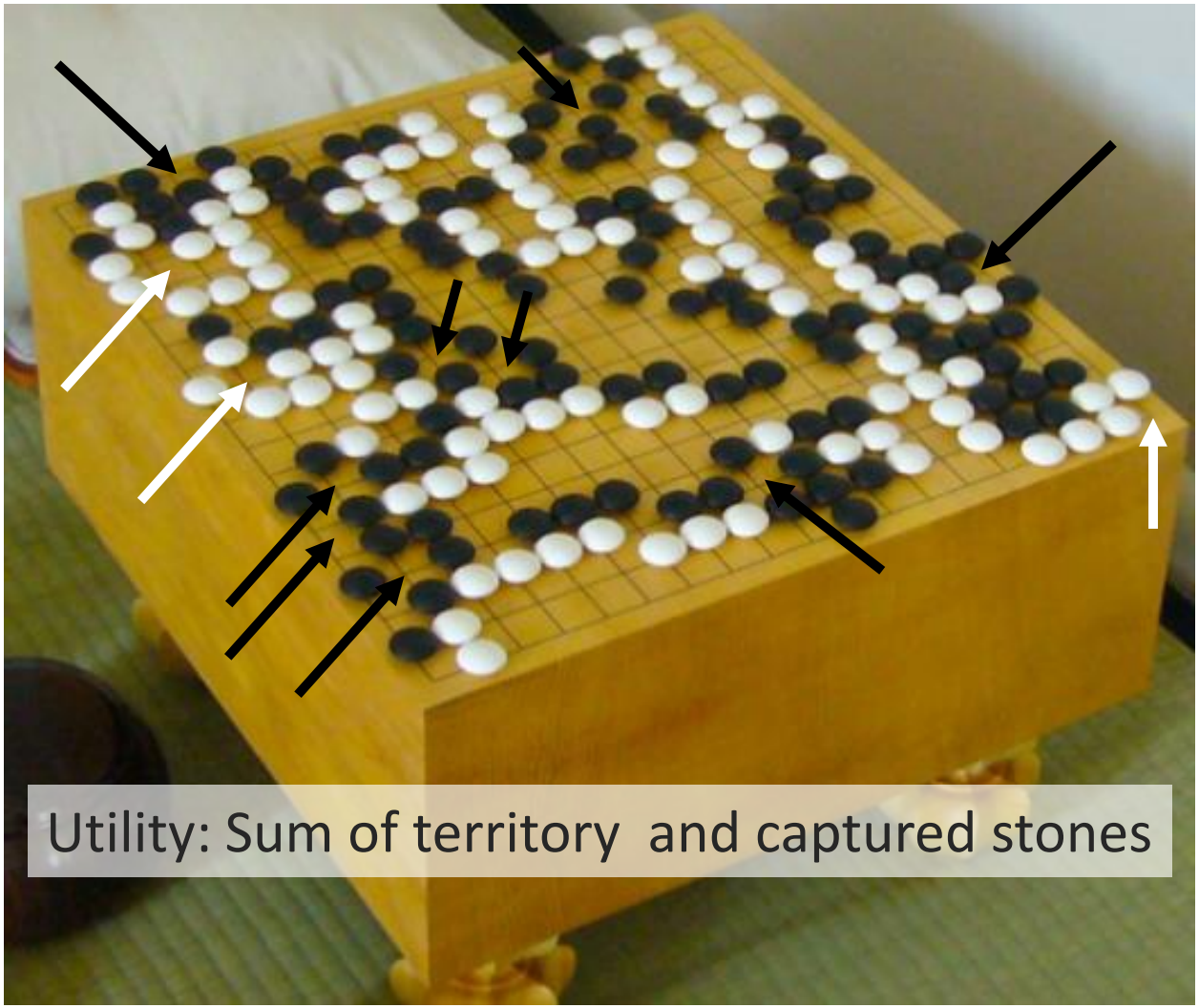


No Ko rule applies, since no direct reverse to a previous state

Double Atari Example



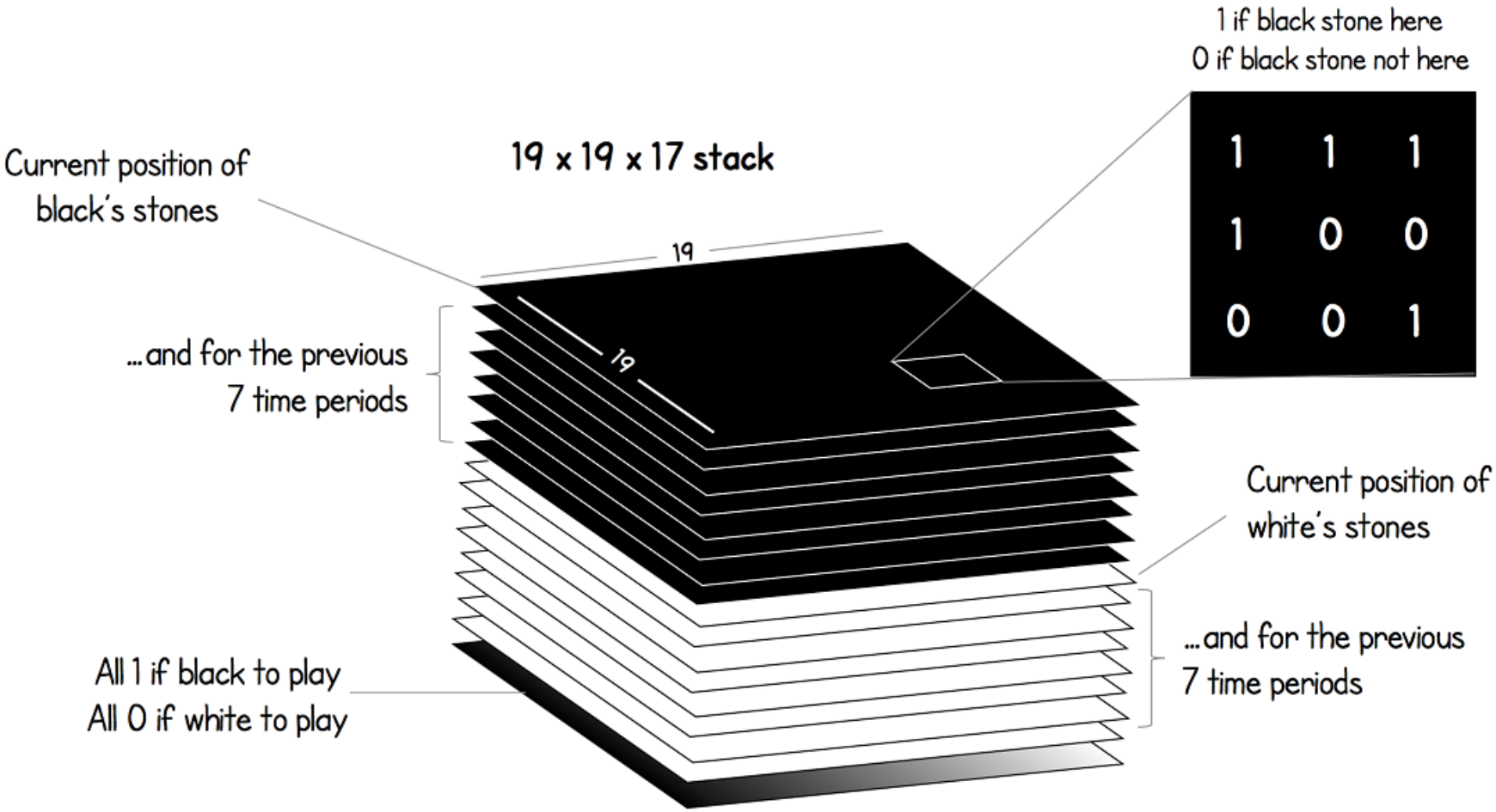
Final State Example



AlphaGo Zero's Neural Network

- Input / Output: $f_\theta(s) \rightarrow (\boldsymbol{p}, v)$
  - s Board state including history (last 7 moves)
  - $\boldsymbol{P}$ Best move probabilities for s
  - v Board value of s
  - θ Arc weights of neural network

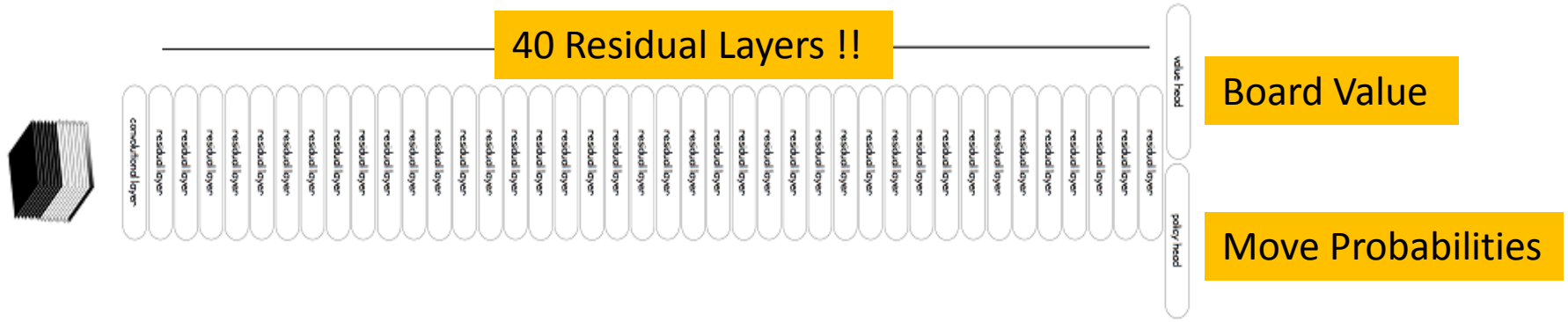
NN Architecture: Input



medium.com

Deep Network

Input



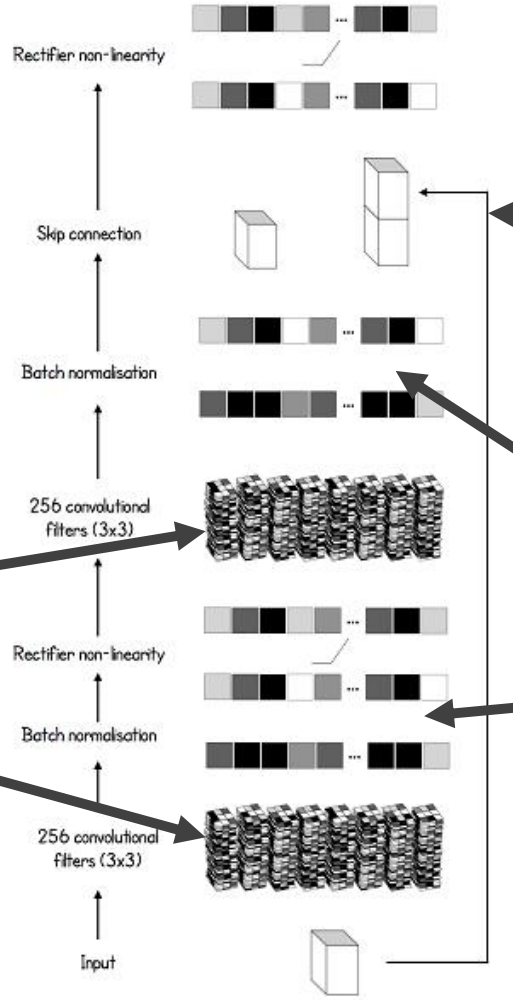
Output

Board Value

Move Probabilities

medium.com

A Residual Layer



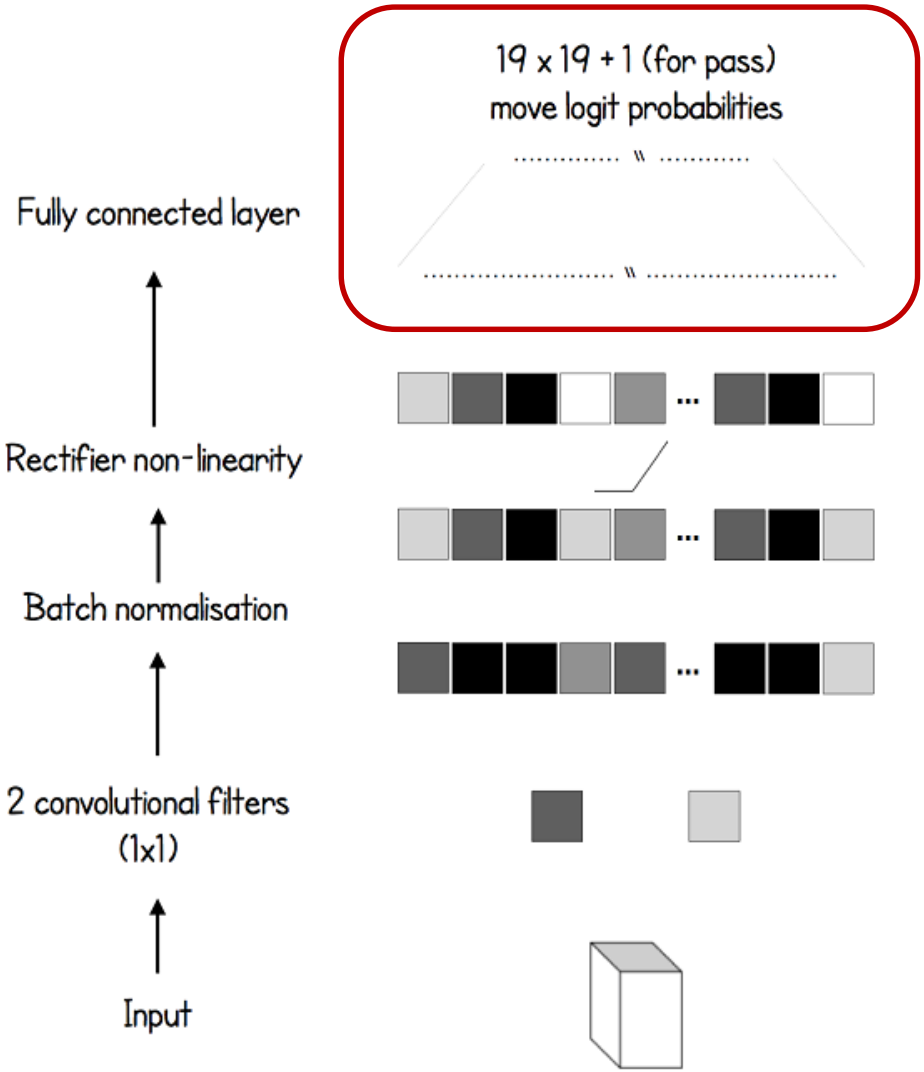
Bypass of layer. Allows much deeper nets

2x256 3x3 Convolution filters

Pooling and rectifier units

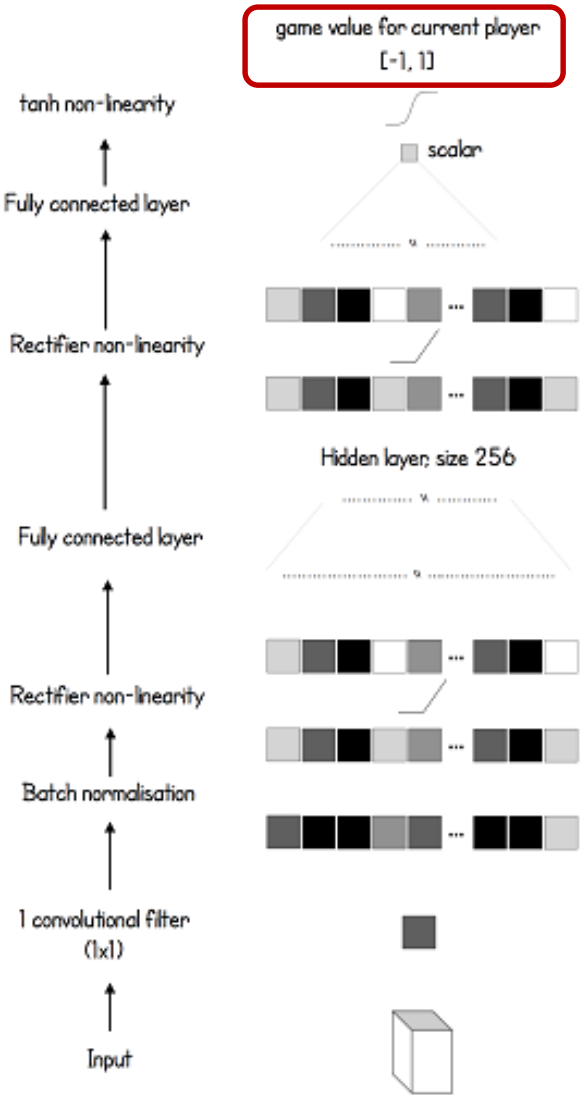
medium.com

Move Probability Output



medium.com

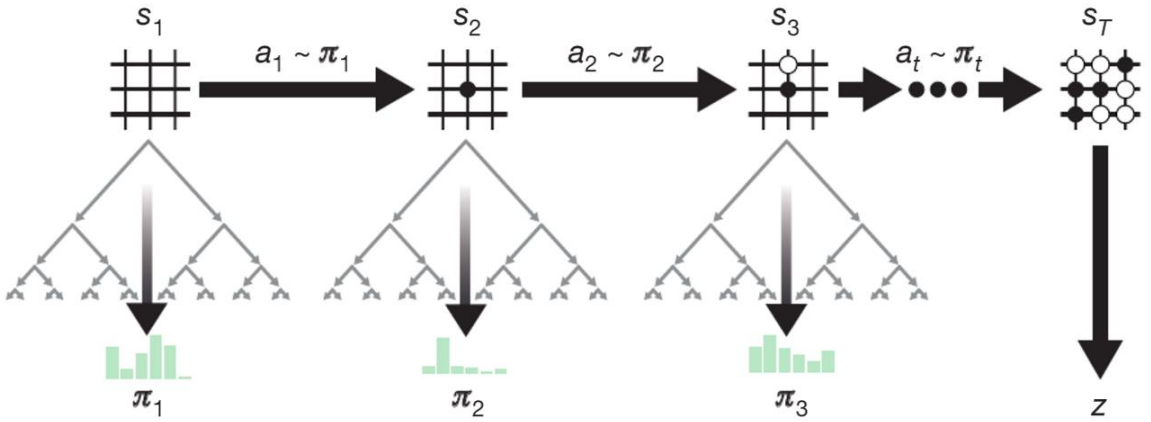
Board Value Output



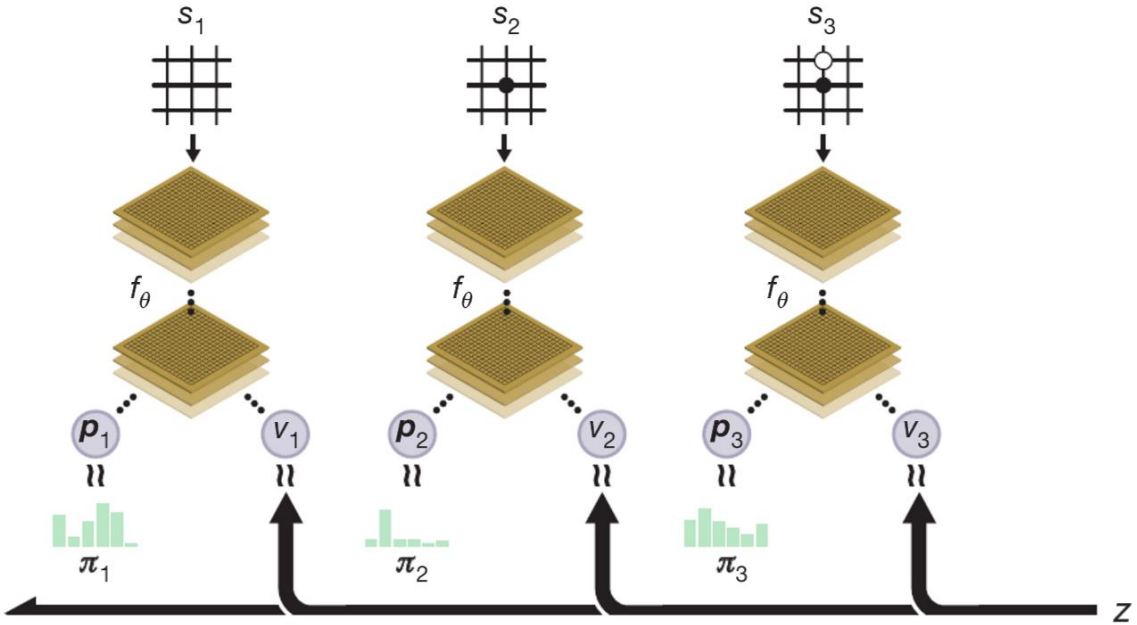
medium.com

Self-Play Reinforcement Learning

Self-play

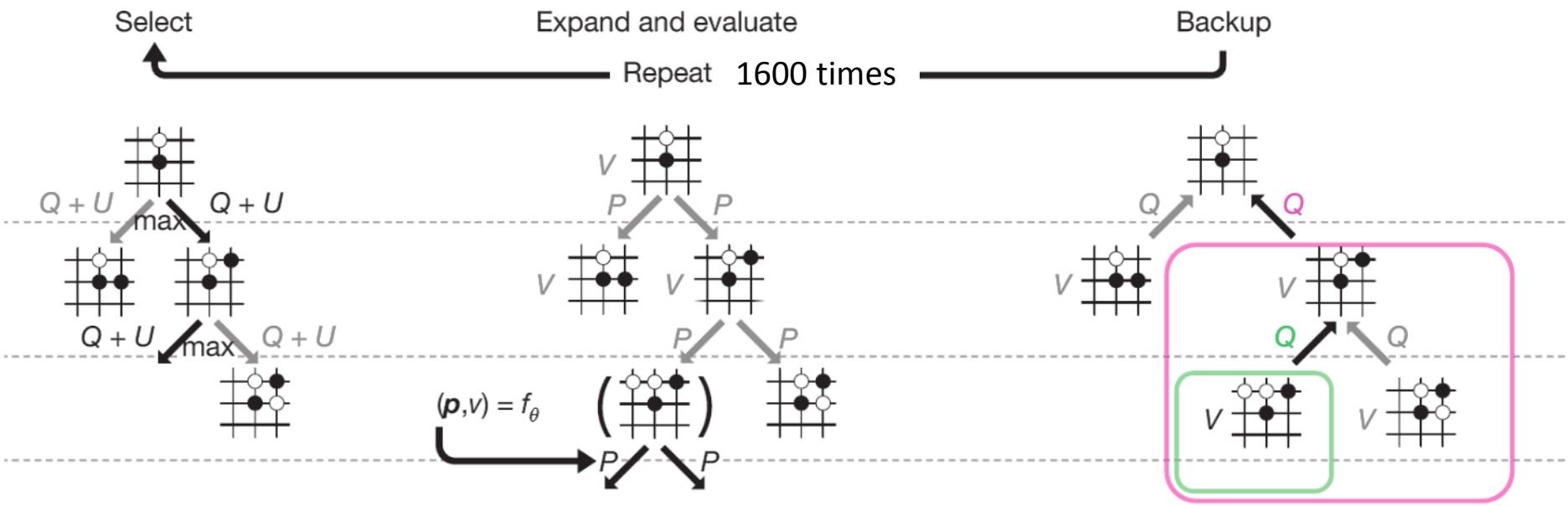


Neural network training



www.nature.com

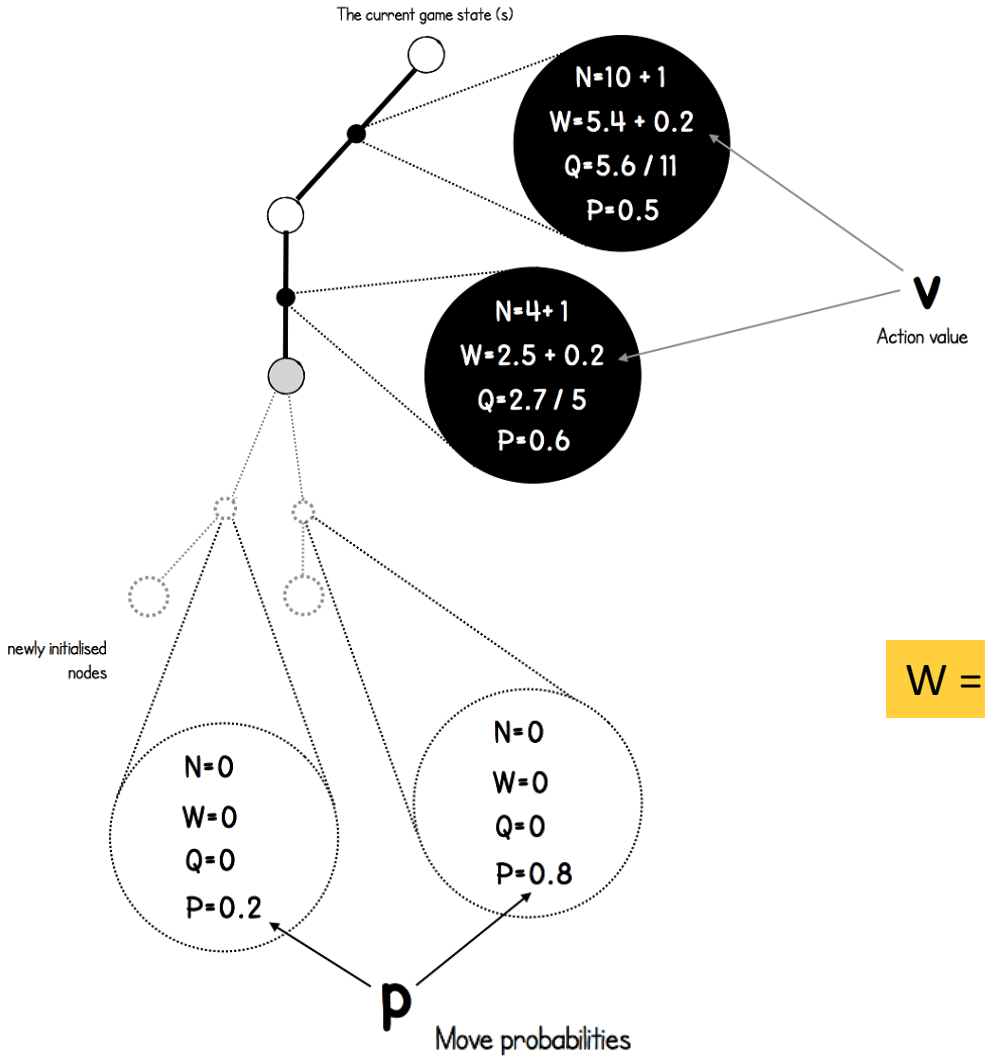
Monte Carlo Tree Search (MCTS)



$N(s,a)$	Number of times action a taken from state s
$P(s,a)$	NN probability of action a in state s
$Q(s,a)$	Mean value of next state
$U(s,a)$	$\propto P(s,a) \,/\, N(s,a)$

www.nature.com

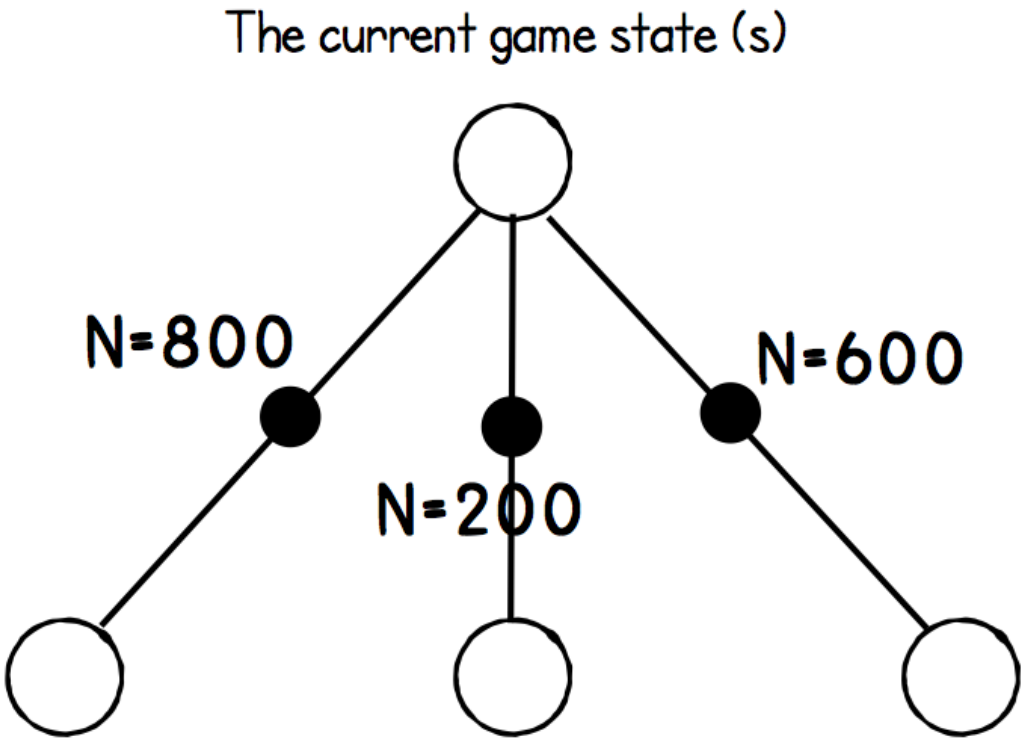
MCTS Update Example



W = Sum of V of the state

medium.com

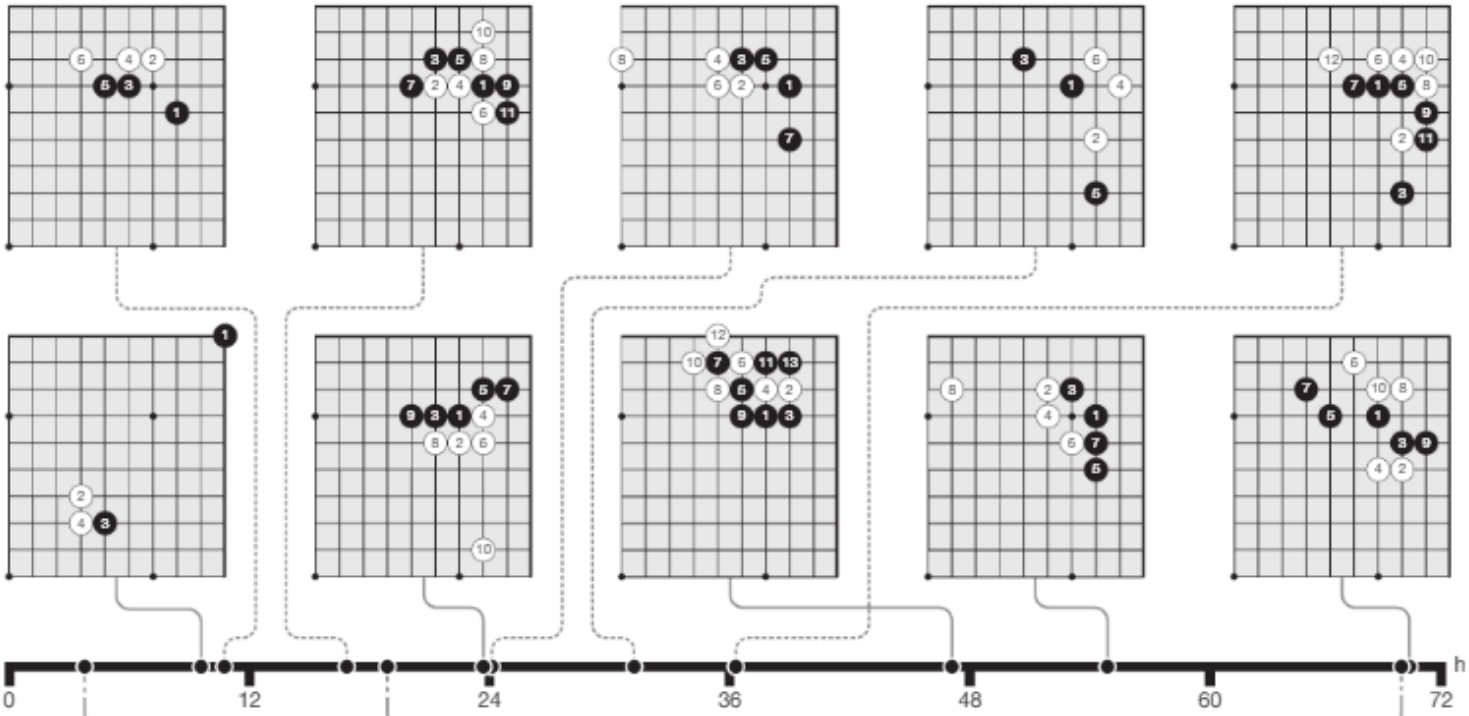
Resulting MCTS Policy (π)



π = (800/1600, 200/1600, 600/1600) = **(0.5, 0.125, 0.375)**

Empirical Analysis

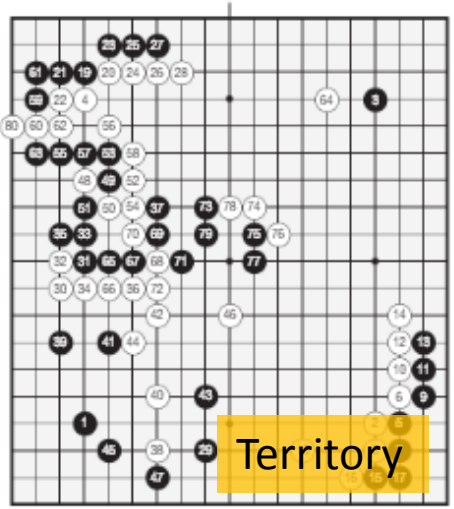
- NN initiated with random weights (tabula rasa)
- 4.9M games of self-play generated
- 1600 simulations of each MCTS (~ 0.4 sec)
- 72h on single machine with 4 TPUs



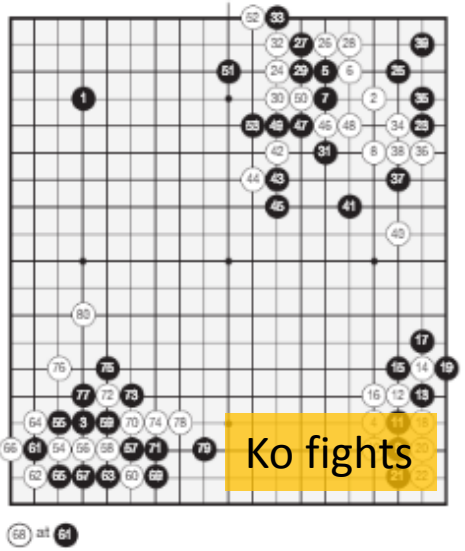
5 known Joseki discovered

Most frequent Joseki

Capturing



Territory

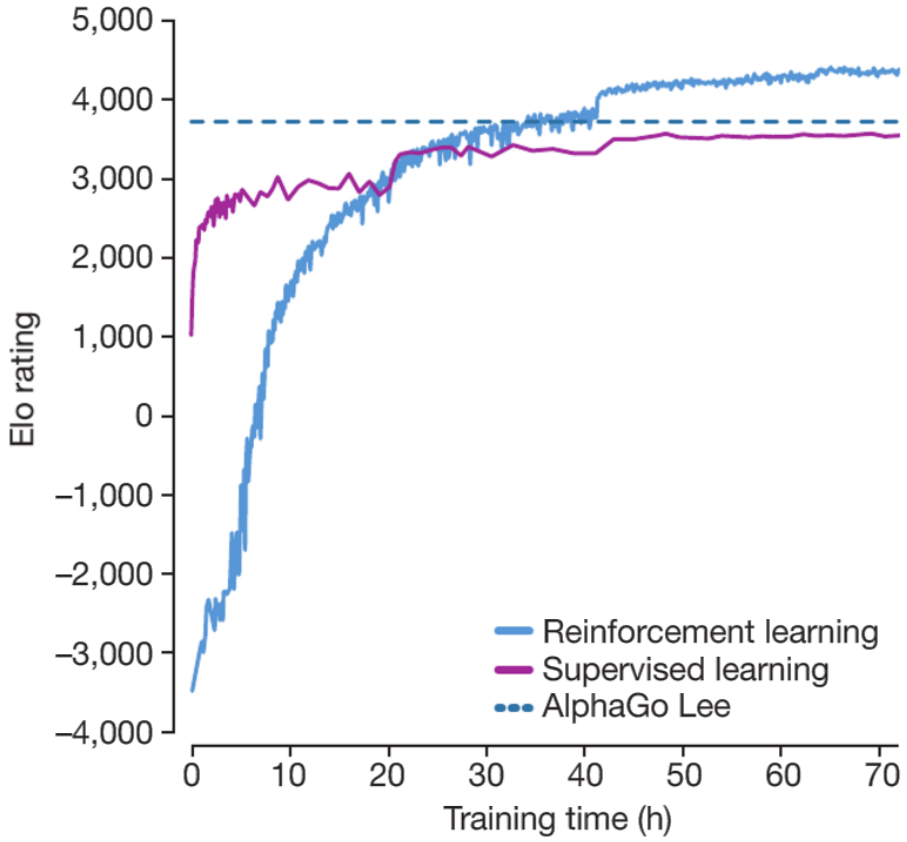


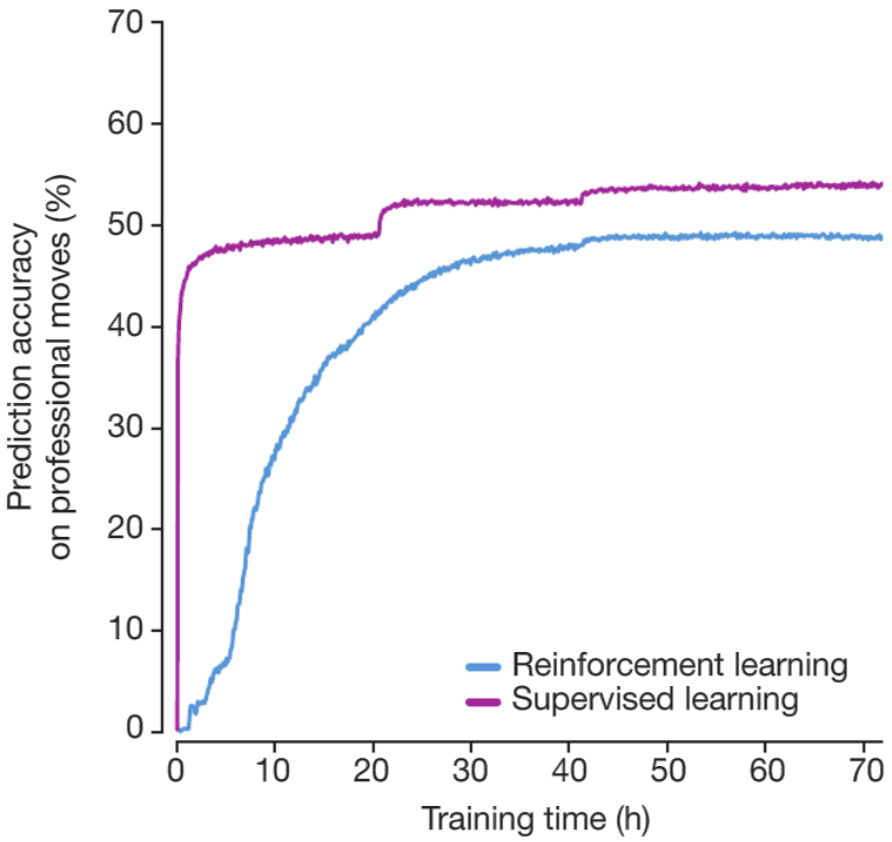
Ko fights

Lee Sedol "Divine moves" played by AlphaGo

www.nature.com

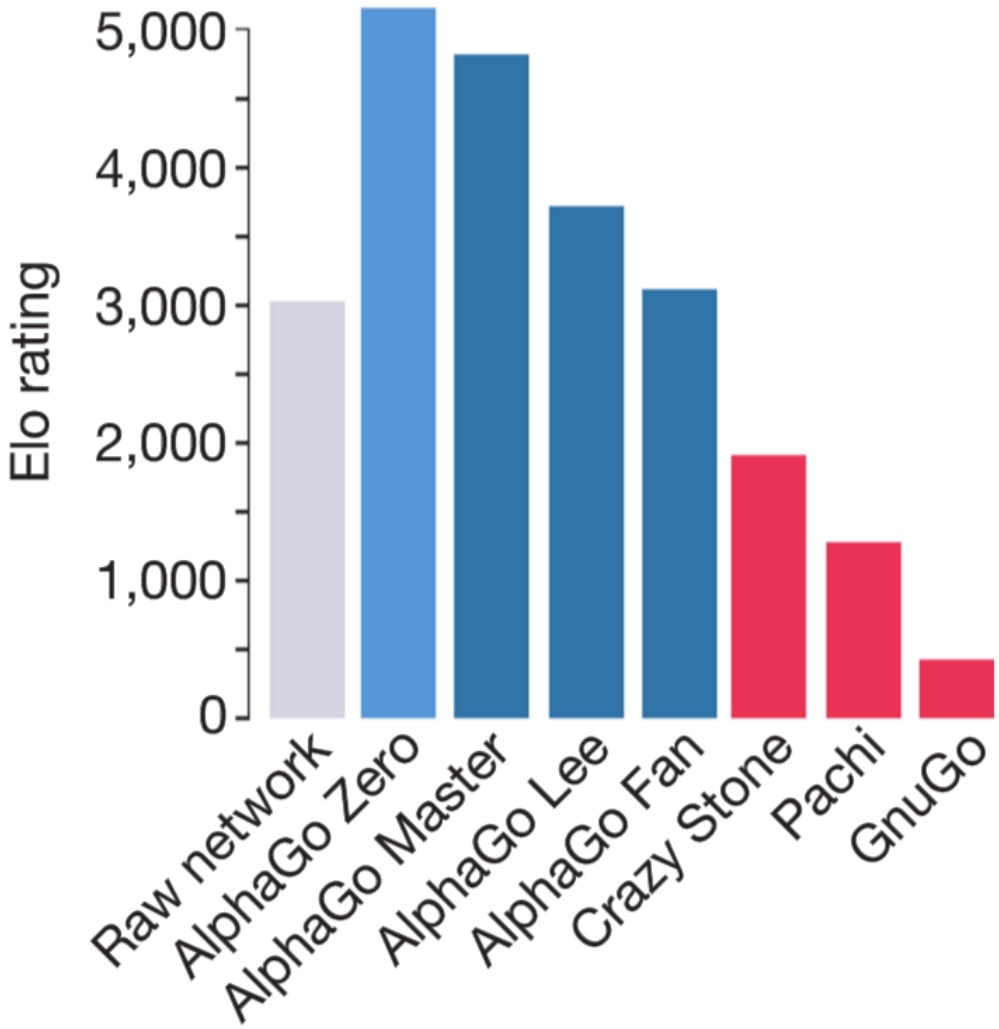
Sup-Human Level Achieved





www.nature.com

Related Work



www.nature.com

Conclusions

- **For zero-sum turn-taking board games: Alpha Zero > Minimax with Alpha Beta pruning**
- Opportunities in
  - Autonomous control
  - Combinatorial optimization
- Issues to consider
  - Size of state-space: go 10^{200}, containership 10^{20000}
  - "Game length": go 400, containership 20000
  - "Self-play" possible to get training data?