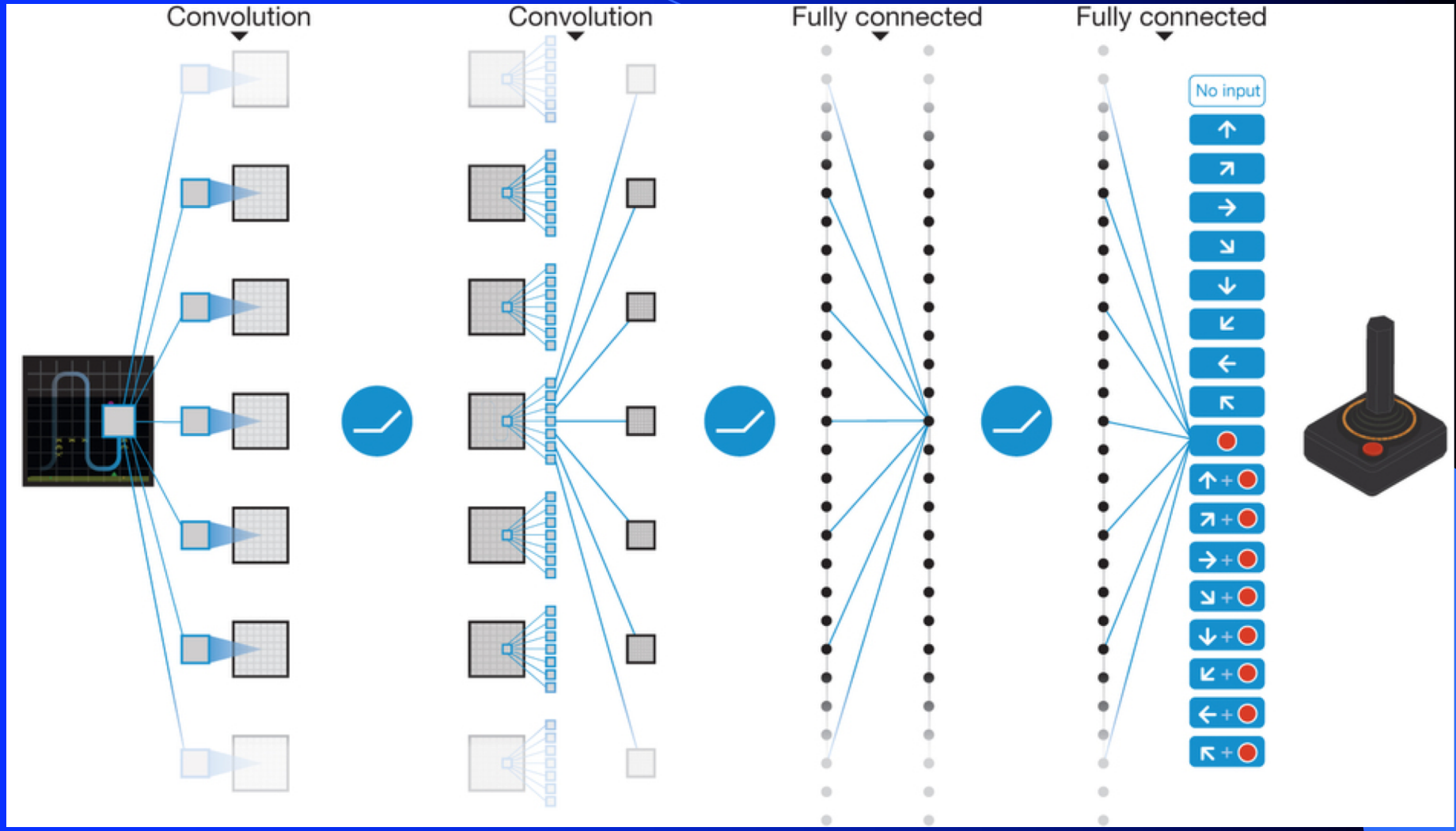
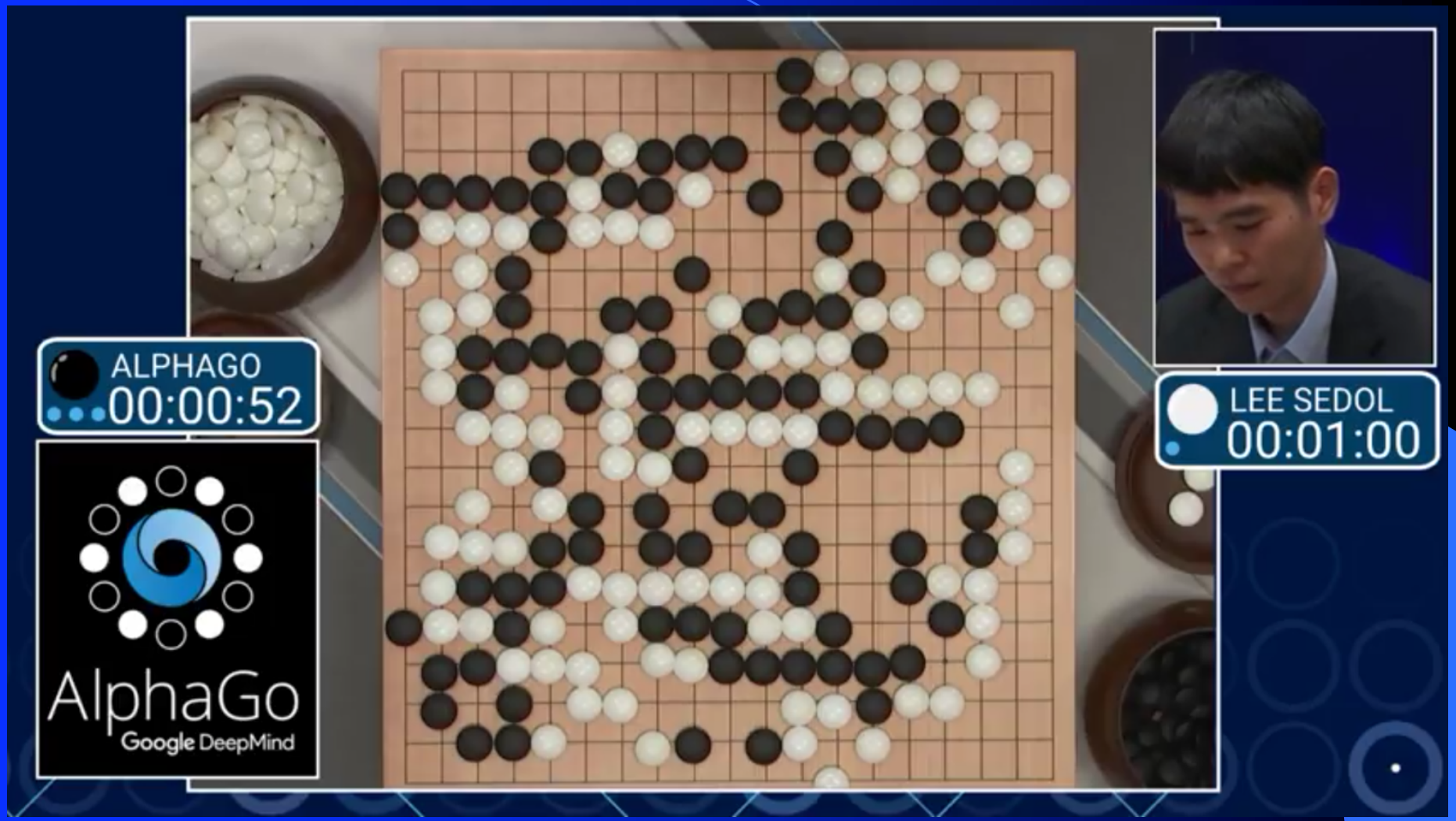
Reinforcement Learning

- Variation on Supervised Learning
- Exact target outputs are not given
- Some variation of reward is given either immediately or after some steps
  - Chess
  - Path Discovery
- RL systems learn a mapping from states to actions by trial-and-error interactions with a dynamic environment
- TD-Gammon (Neuro-Gammon)
- Deep RL (RL with deep neural networks) – Showing tremendous potential
  - Especially nice for games because easy to generate data through self-play

Deep Q Network – 49 Classic Atari Games



AlphaGo - Google DeepMind



Alpha Go

- Reinforcement Learning with Deep Net learning the value and policy functions
- Challenges world Champion Lee Se-dol in March 2016
  - AlphaGo Movie – Netflix, check it out, fascinating man/machine interaction!
- AlphaGo Master (improved with more training) then beat top masters on-line 60-0 in Jan 2017
- 2017 – Alpha Go Zero
  - Alpha Go started by learning from 1000's of expert games before learning more on its own, and with lots of expert knowledge
  - Alpha Go Zero starts from zero (Tabula Rasa), just gets rules of Go and starts playing itself to learn how to play – not patterned after human play – More creative
  - Beat AlphaGo Master 100 games to 0 (after 3 days of playing itself)

Alpha Zero

- Alpha Zero (late 2017)
- Generic architecture for any board game
  - Compared to AlphaGo (2016 - earlier world champion with extensive background knowledge) and AlphaGo Zero (2017)
- No input other than rules and self-play, and not set up for any specific game, except different board input
- With no domain knowledge and starting from random weights, beats worlds best players and computer programs (which were specifically tuned for their games over many years)
  - Go – after 8 hours training (44 million games) beats AlphaGo Zero (which had beat AlphaGo 100-0) – 1000's of TPU's for training
    - AlphaGo had taken many months of human directed training
  - Chess – after 4 hours training beats Stockfish8 28-0 (+72 draws)
    - Doesn't pattern itself after human play
  - Shogi (Japanese Chess) – after 2 hours training beats Elmo

RL Basics

- Agent (sensors and actions)
- Can sense state of Environment (position, etc.)
- Agent has a set of possible actions
- Actual rewards for actions from a state are usually delayed and do not give direct information about how best to arrive at the reward
- RL seeks to learn the optimal policy: which action should the agent take given a particular state to achieve the agents eventual goals (e.g. maximize reward)

Learning a Policy

- Find optimal policy π: S -> A
- $a = \pi(s)$, where a is an element of A, and s an element of S
- Which actions in a sequence leading to a goal should be rewarded, punished, etc. – Temporal Credit assignment problem
- Exploration vs. Exploitation – To what extent should we explore new unknown states (hoping for better opportunities) vs. taking the best possible action based on knowledge already gained
  - The restaurant problem
- Markovian? – Do we just base action decision on current state or is their some memory of past states – Basic RL assumes Markovian processes (action outcome is only a function of current state, state fully observable) – Does not directly handle partially observable states (i.e. states which are not unambiguously identified) – can still approximate

Rewards

- Assume a reward function $r(s,a)$ – Common approach is a positive reward for entering a goal state (win the game, get a resource, etc.), negative for entering a bad state (lose the game, lose resource, etc.), 0 for all other transitions.
- Could also make all reward transitions -1, except for 0 going into the goal state, which would lead to finding a minimal length path to a goal
- Discount factor γ: between 0 and 1, future rewards are discounted
- Value Function $V(s)$: The value of a state is the sum of the discounted rewards received when starting in that state and following a fixed policy until reaching a terminal state
- $V(s)$ also called the Discounted Cumulative Reward

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

4 possible actions: N, S, E, W

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

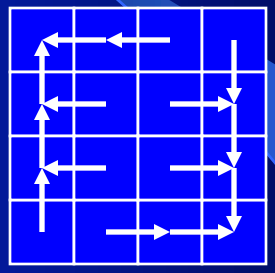
Reward Function

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0

V(s) with optimal policy and $\gamma = 1$

0	0	-1	-2
0	-1	-2	-1
-1	-2	-1	0
-2	-1	0	0

One Optimal Policy



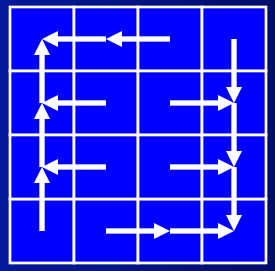
Reward Function

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	1

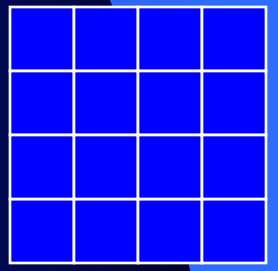
V(s) with optimal policy and $\gamma = .9$

0	1	.90	.81
1	.90	.81	.90
.90	.81	.90	1
.81	.90	1	0

One Optimal Policy



V(s) with random policy and $\gamma = 1$



4 possible actions: N, S, E, W

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Reward Function

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0

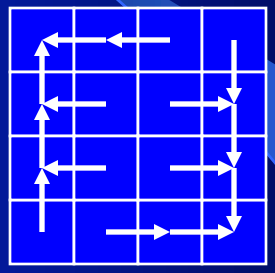
V(s) with random policy and $\gamma = 1$

0	-14	-20	-22
-14	-18	-22	-20
-20	-22	-18	-14
-22	-20	-14	0

V(s) with optimal policy and $\gamma = 1$

0	0	-1	-2
0	-1	-2	-1
-1	-2	-1	0
-2	-1	0	0

One Optimal Policy



Reward Function

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	1

V(s) with random policy and $\gamma = .9$

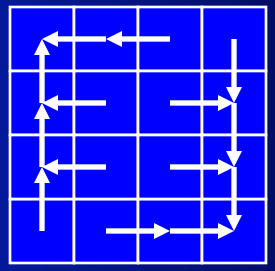
0	.25		
			0

$.25 = 1\gamma^{13}$

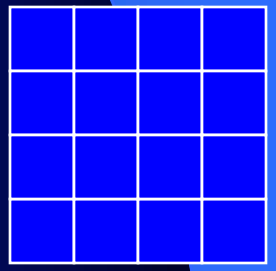
V(s) with optimal policy and $\gamma = .9$

0	1	.90	.81
1	.90	.81	.90
.90	.81	.90	1
.81	.90	1	0

One Optimal Policy



V(s) with random policy and $\gamma = 1$



Policy vs. Value Function

- Goal is to learn the optimal policy

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), (\forall s)$$

- $V^*(s)$ is the value function of the optimal policy. $V(s)$ is the value function of the current policy.
- $V(s)$ is fixed for the current policy and discount factor
- Typically start with a random policy – Effective learning happens when rewards from terminal states start to propagate back into the value functions of earlier states
- $V(s)$ could be represented with a lookup table and will be used to iteratively update the policy (and thus update $V(s)$ at the same time)
- For large or real valued state spaces, lookup table is too big, thus must approximate the current $V(s)$. Any adjustable function approximator (e.g. neural network) can be used.

Policy Iteration

Let π be an arbitrary initial policy

Repeat until π unchanged

For all states s

$$V^{\pi}(s) = \sum_{s'} P(s' \mid s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

For all states s

$$\pi'(s) = \arg\max_{a} \sum_{s'} P(s' \mid s, a) \cdot [R(s, a, s') + \gamma V^{\pi}(s')]$$

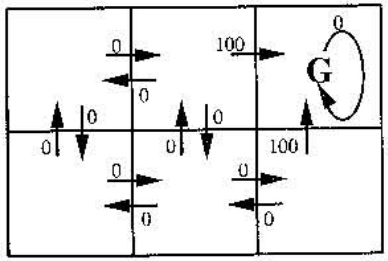
- In policy iteration the equations just calculate one state ahead rather than continue to an absorbing state
- To execute directly, must know the probabilities of state transition function and the exact reward function
- Also usually must be learned with a model doing a simulation of the environment. If not, how do you do the argmax which requires trying each possible action. In the real world, you can't have a robot try one action, backup, try again, etc. (e.g. environment may change because of the action, etc.)

Q-Learning

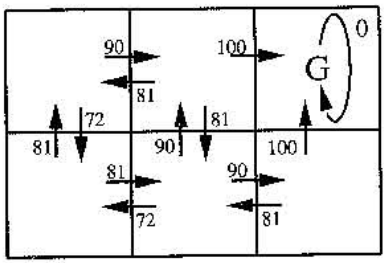
- No model of the world required – Just try one action and see what state you end up in and what reward you get. Update the policy based on these results. This can be done in the real world and is thus more widely applicable.
- Rather than find the value function of a state, find the value function of a (s,a) pair and call it the Q-value
- Only need to try actions from a state and then incrementally update the policy
- $Q(s,a)$ = Sum of discounted reward for doing a from s and following the optimal policy thereafter

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) = r(s,a) + \gamma \max_{a'} Q(s',a')$$

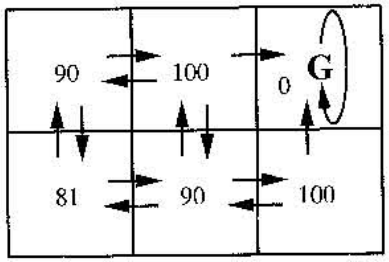
$$\pi^*(s) = \arg\max_a Q(s,a)$$



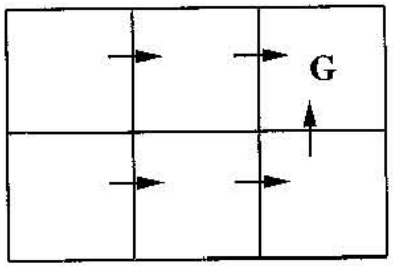
$r(s, a)$ (immediate reward) values



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

FIGURE 13.2
A simple deterministic world to illustrate the basic concepts of Q-learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s, a)$ gives reward 100 for actions entering the goal state **G**, and zero otherwise. Values of $V^*(s)$ and $Q(s, a)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal Q values, is also shown.

Learning Algorithm for Q function

• Create a table with a cell for every state and (s,a) pair with zero or random initial values for the hypothesis of the Q values which we represent by $\hat{Q}$
• Iteratively try different actions from different states and update the table based on the following learning rule (for deterministic environment)

$$\hat{Q}(s,a) = r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$$

• Note that this slowly adjusts the estimated Q-function towards the true Q-function. Iteratively applying this equation will in the limit converge to the actual Q-function if

- The system can be modeled by a deterministic Markov Decision Process – action outcome depends only on current state (not on how you got there)
- r is bounded ($r(s,a) < c$ for all transitions)
- Each (s,a) transition is visited infinitely many times

Learning Algorithm for Q function

Until Convergence (Q-function not changing or changing very little)

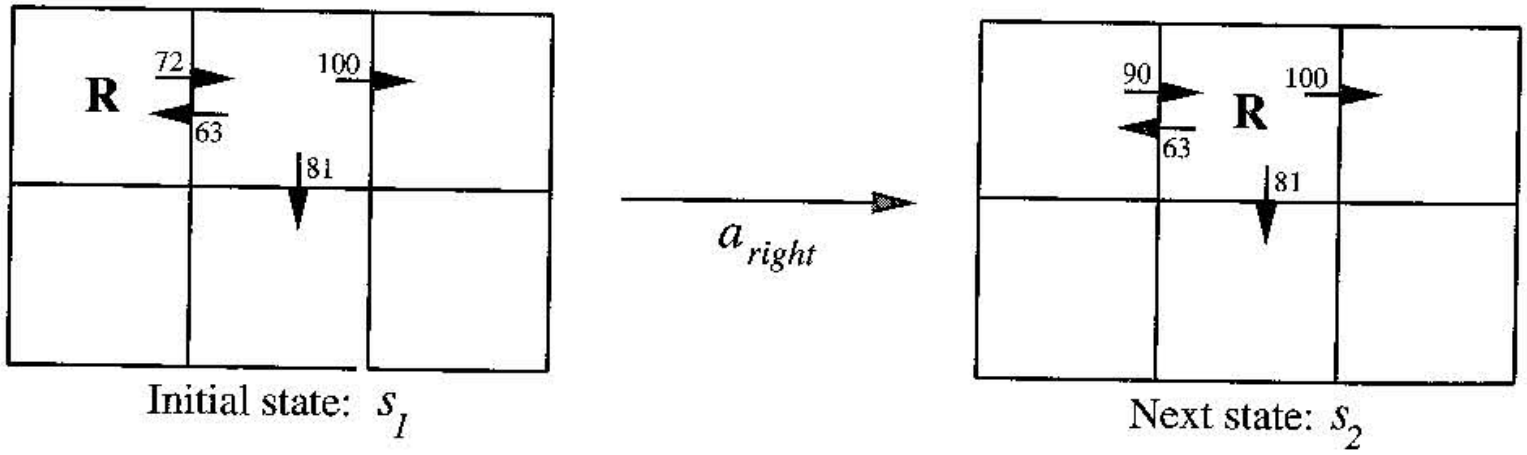
Start in an arbitrary s

Select an action a and execute (exploitation vs. exploration)

Update the Q-function table entry

$$\hat{Q}(s,a) = r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$$

- Could also continue (s -> s') until an absorbing state is reached (episode) at which point can start again at an arbitrary s.
- But sufficient to choose a new s at each iteration and just go one step
- Do not need to know the actual reward and state transition functions. Just sample them (Model-less).



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$

$$\leftarrow 0 + 0.9 \max\{63, 81, 100\}$$

$$\leftarrow 90$$

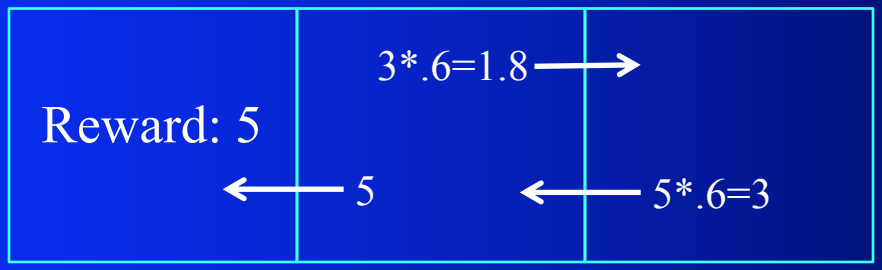
FIGURE 13.3
The update to $\hat{Q}$ after executing a single action. The diagram on the left shows the initial state s_1 of the robot (**R**) and several relevant $\hat{Q}$ values in its initial hypothesis. For example, the value $\hat{Q}(s_1, a_{right}) = 72$, where a_{right} refers to the action that moves **R** to its right. When the robot executes the action a_{right}, it receives immediate reward $r = 0$ and transitions to state s_2. It then updates its estimate $\hat{Q}(s_1, a_{right})$ based on its $\hat{Q}$ estimates for the new state s_2. Here $\gamma = 0.9$.

Q-Learning Challenge Question

- Assume the deterministic 3 state world below (each cell is a state) where the immediate reward is 0 for entering all states, except the leftmost state, for which the reward is 5, and which is an absorbing state. The only actions are move right and move left (only one of which is available from the border cells). Assume a discount factor of .6, and all initial Q-values of 0. Give the final optimal Q values for each action in each state.

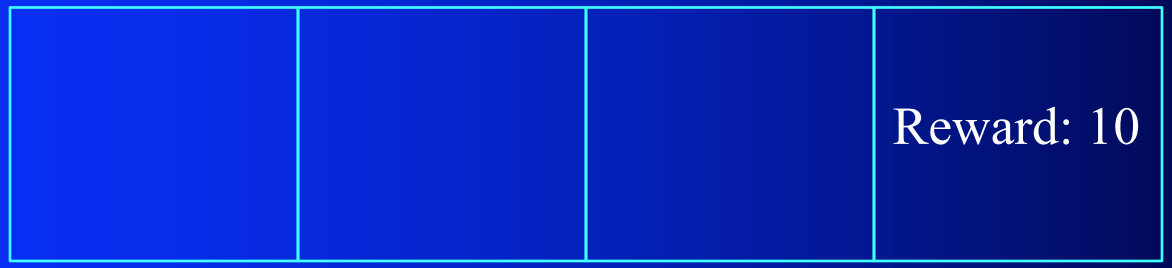
Q-Learning Challenge Question

- Assume the deterministic 3 state world below (each cell is a state) where the immediate reward is 0 for entering all states, except the leftmost state, for which the reward is 5, and which is an absorbing state. The only actions are move right and move left (only one of which is available from the border cells). Assume a discount factor of .6, and all initial Q-values of 0. Give the final optimal Q values for each action in each state.



Q-Learning Homework

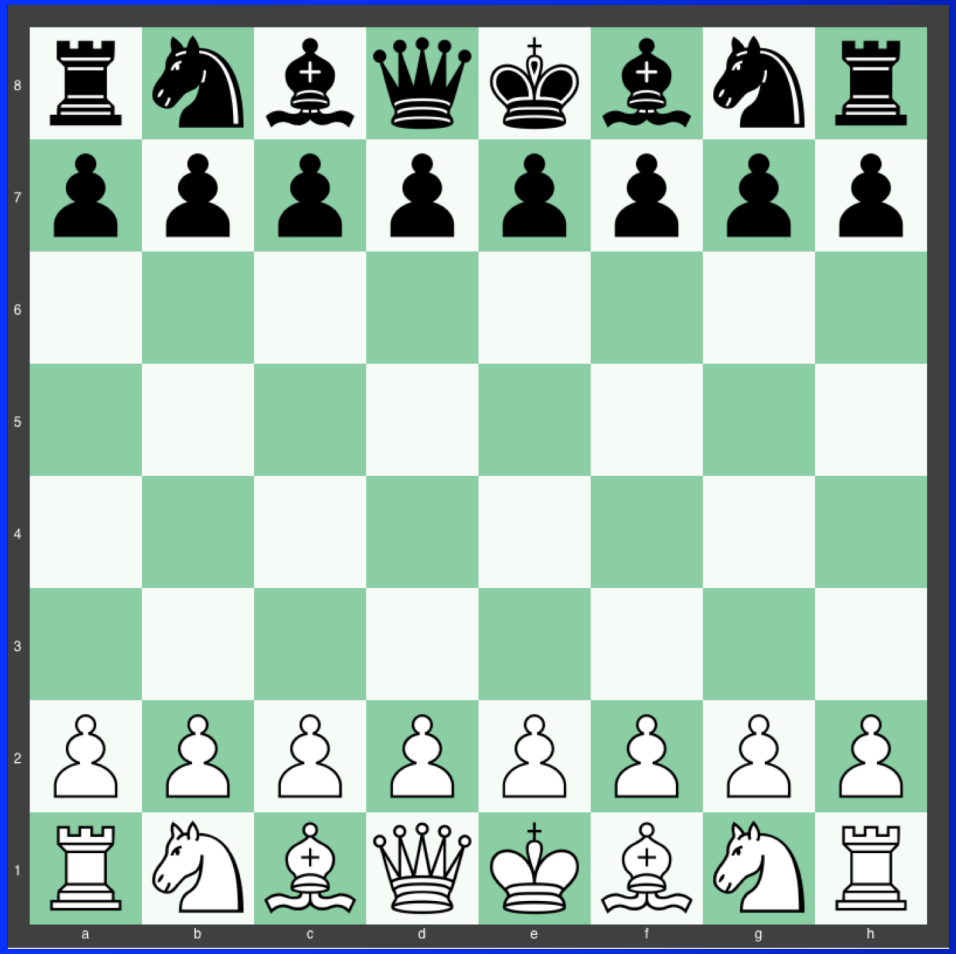
- Assume the deterministic 4 state world below (each cell is a state) where the immediate reward is 0 for entering all states, except the rightmost state, for which the reward is 10, and which is an absorbing state. The only actions are move right and move left (only one of which is available from the border cells). Assume a discount factor of .8, and all initial Q-values of 0. Give the final optimal Q values for each action in each state and describe an optimal policy.

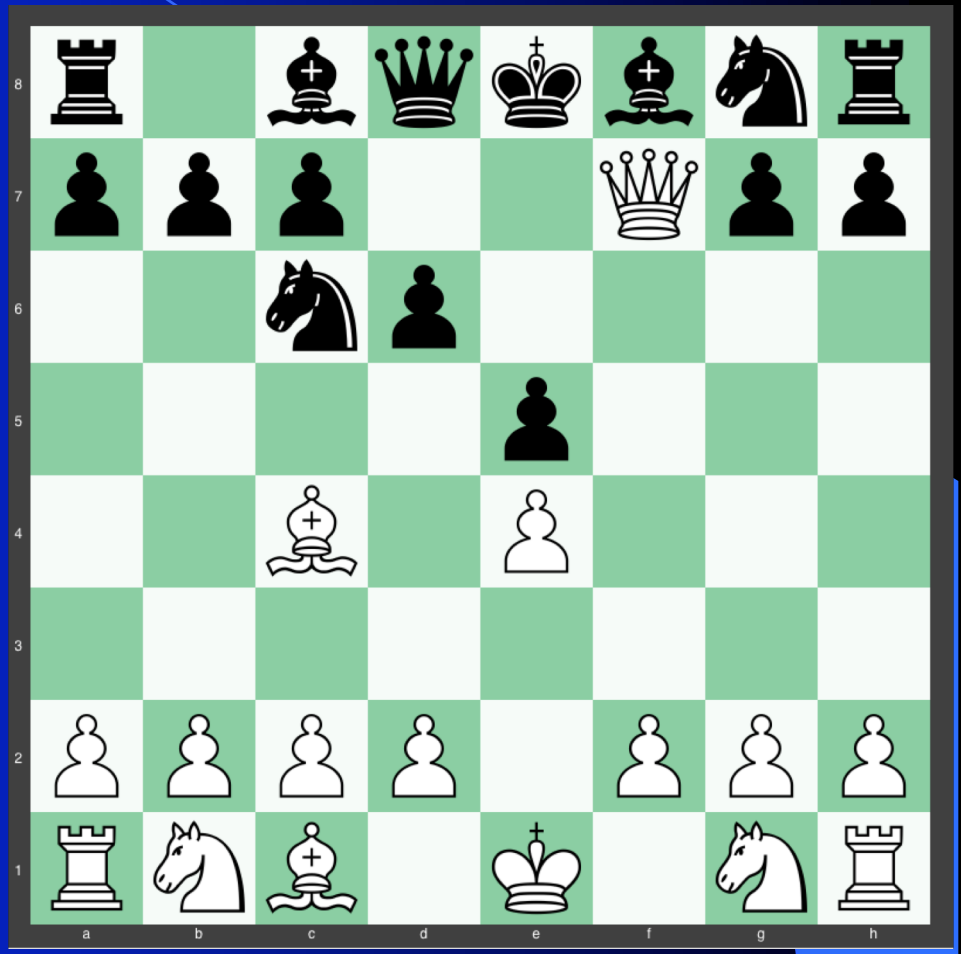


Example - Chess

- Assume reward of 0's except win (+10) and loss (-10)
- Set initial Q-function to all 0's
- Start from any initial state (could be normal start of game) and choose transitions until reaching an absorbing state (win or lose)
- During all the earlier transitions the update was applied but no change was made since rewards were all 0.
- Finally, after entering an absorbing state, $Q(s_{pre}, a_{pre})$, the preceding state-action pair, gets updated (positive for win or negative for loss).
- Next time around a state-action entering s_{pre} will be updated and this progressively propagates back with more iterations until all state-action pairs have the proper Q-function.
- If other actions from s_{pre} also lead to the same outcome (e.g. loss) then Q-learning will learn to avoid this state altogether (however, remember it is the max action out of the state that sets the actual Q-value)

Possible States for Chess





Exploration vs Exploitation

- Choosing action during learning (Exploitation vs. Exploration) – 2 Common approaches
- Softmax:

$$P(a_i \mid s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

  - Can increase k (constant >1) over time to move from exploration to exploitation
- ε-greedy: With probability ε randomly choose any action, else greedily take the action with the best current Q value.
  - Start ε at 1 and then decrease with time

Episode Updates

- Sequence of Update – Note that much efficiency could be gained if you worked back from the goal state, etc. However, with model free learning, we do not know where the goal states are, or what the transition function is, or what the reward function is. We just sample things and observe. If you do know these functions then you can simulate the environment and come up with more efficient ways to find the optimal policy with standard DP algorithms (e.g. policy iteration).
- One thing you can do for Q-learning is to store the path of an episode and then when an absorbing state is reached, propagate the discounted Q-function update all the way back to the initial starting state. This can speed up learning at a cost of memory.
- Monotonic Convergence

Q-Learning in Non-Deterministic Environments

- Both the transition function and reward functions could be non-deterministic

$$Q^*(s,a) = E[r(s,a) + \gamma \max_{a'} Q^*(s',a')]$$

- In this case the previous algorithm will not monotonically converge
- Though more iterations may be required, we simply replace the update function with

$$\hat{Q}_n(s,a) = (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(s',a')]$$

$$= \hat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(s',a') - \hat{Q}_{n-1}(s,a)]$$

where α_n starts at 1 and decreases over time and n stands for the n^{th} iteration. An example of α_n is

$$\alpha_n = \frac{1}{1 + \#ofvisits(s,a)}$$

- Large variations in the non-deterministic function are muted and an overall averaging effect is attained (like a small learning rate in neural network learning)

Replace Q-table with a Function Approximator

$$Q^*(s,a) \approx Q(s,a;\theta)$$

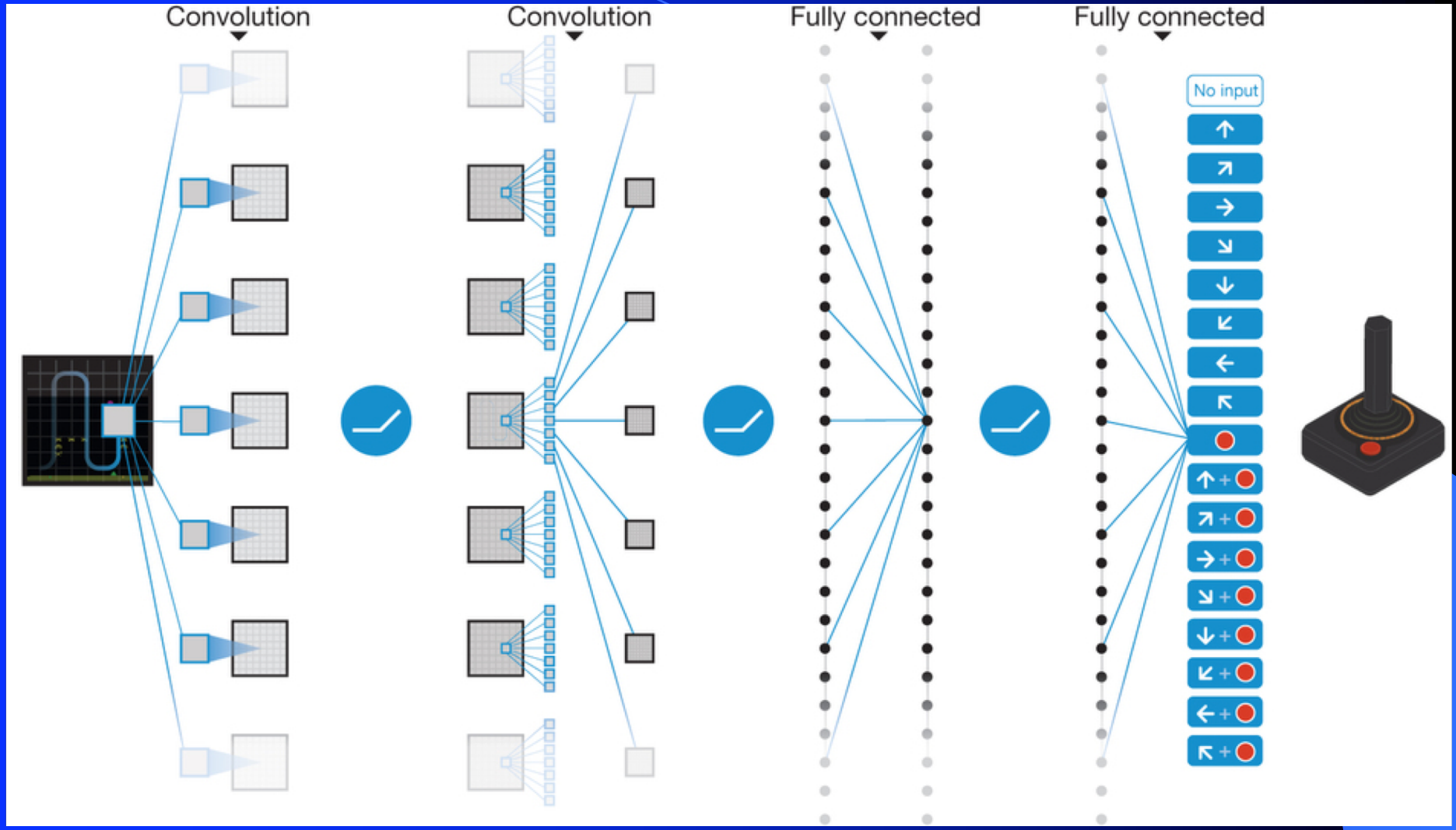
$$\hat{Q}(s,a) = r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$$

- Train a function approximator (e.g. ML model) to output approximate Q-values
  - Use an MLP/deep net in place of the lookup table, where it is trained with the inputs *s* and *a* with the current Q-value as output
    - Could alternatively have the input be *s* and the outputs be the Q-values for each (*s*, *a*) pair
  - Avoid huge or infinite lookup tables (real values, etc.)
  - Allows generalization from all states, not just those seen during training
  - Note that we are not training with the optimal Q-values (we don't know them). Thus we train with the current Q-values we have and those values keep updating over time. Thus later in learning, our output target is not the same for the same state *s* as it was initially.
  - Training error is the difference between the network's current Q-value output (generalization) and the current Q-value expectation

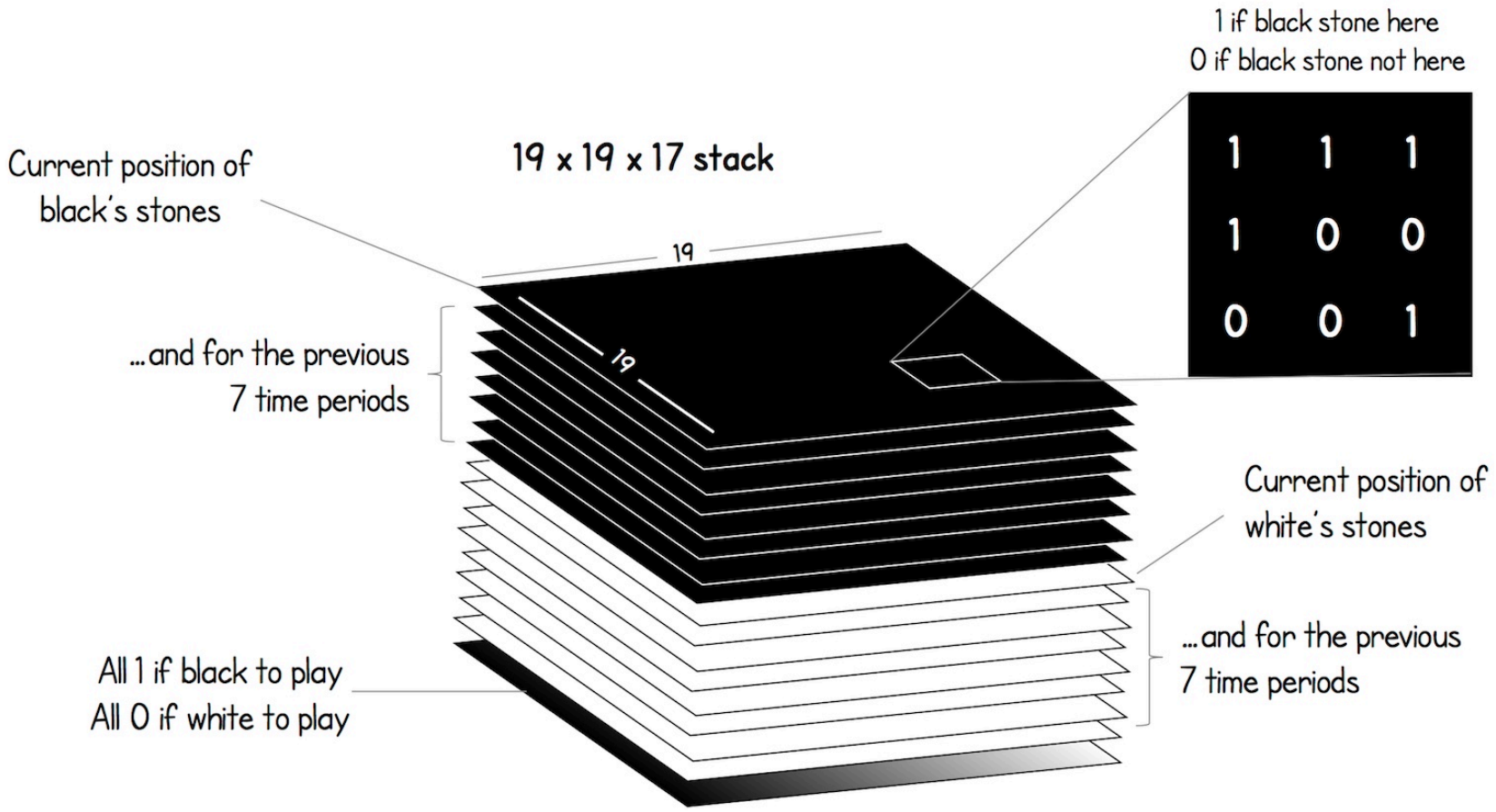
Deep Q-Learning Example

- Deep convolutional network trained to learn Q function
- To overcome Markov limitation (partially observed states) the function approximator can be given an input made up of *m* consecutive proceeding states (Atari and Alpha zero approach) or have memory (e.g. recurrent NN), etc.
  - Early Q learning used linear models or shallow neural networks
- Using deep networks as the approximator has been shown to lead to accurate stable learning
  - Learns all 49 classic Atari games with the only inputs being pixels from the screen and the score, at above standard human playing level with no tuning of hyperparameters.
  - Alpha-Zero

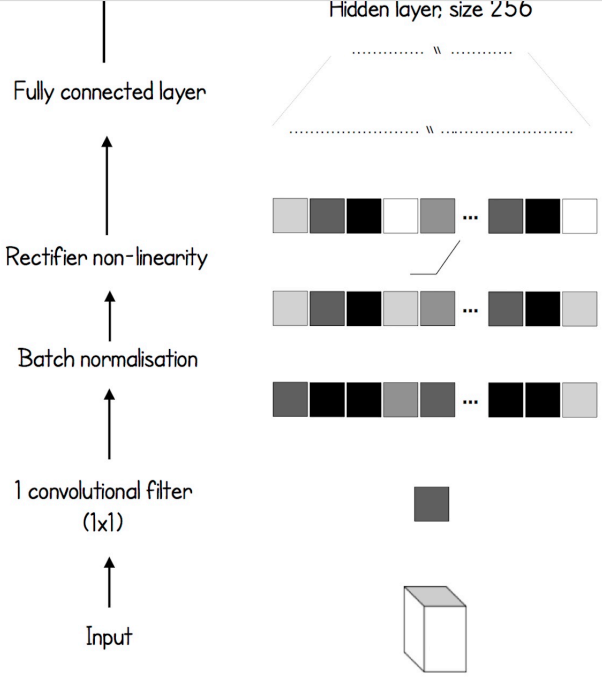
Deep Q Network – 49 Classic Atari Games



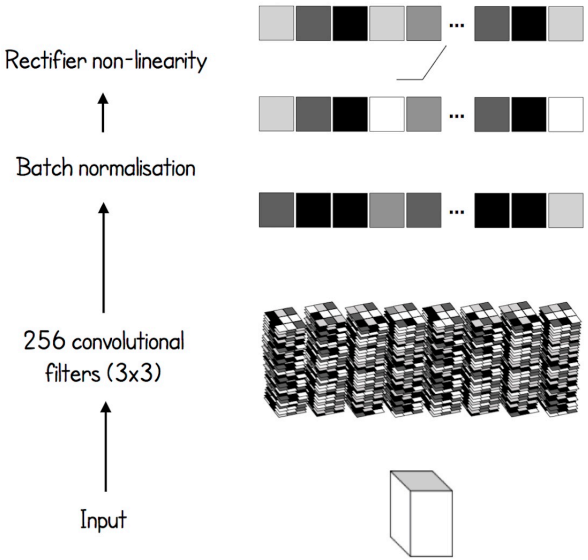
WHAT IS A 'GAME STATE'

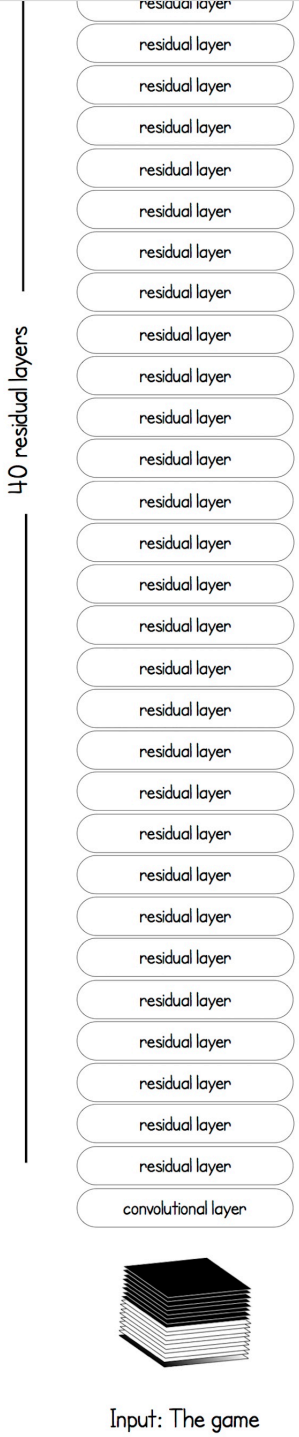


This stack is the input to the deep neural network

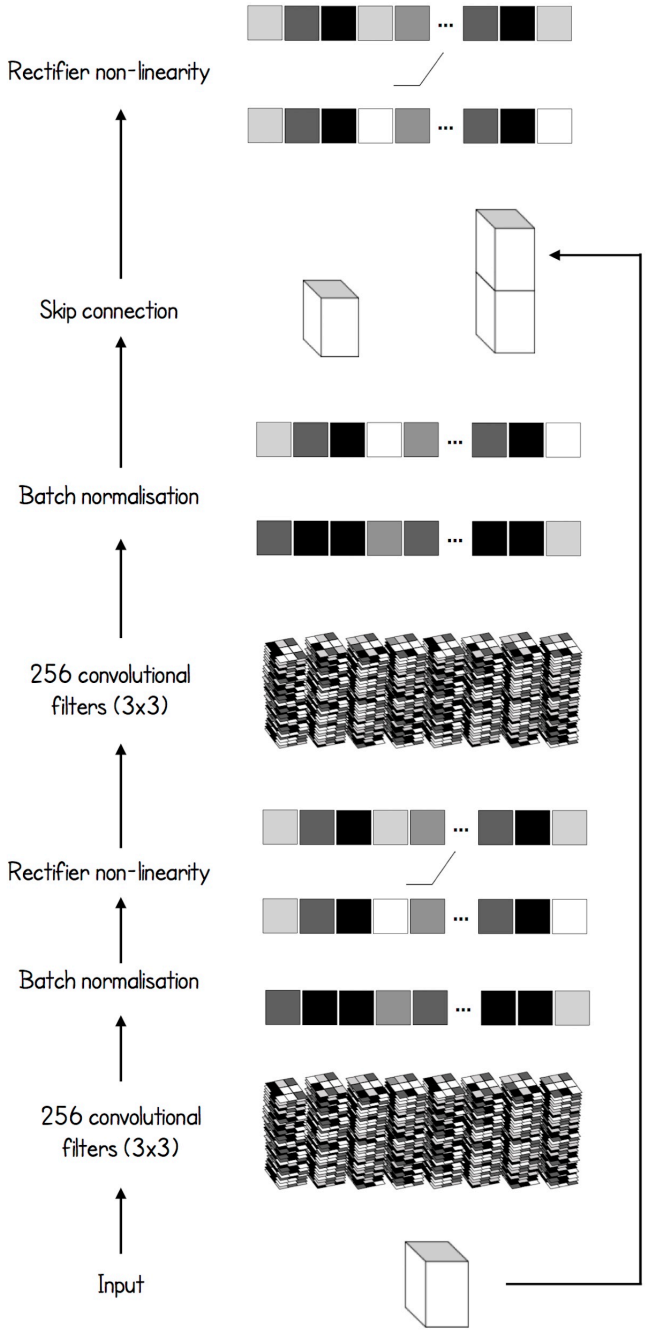


A convolutional layer



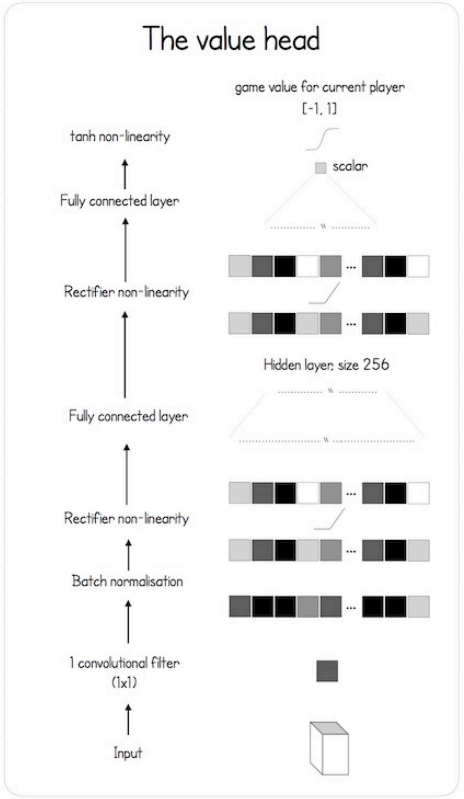


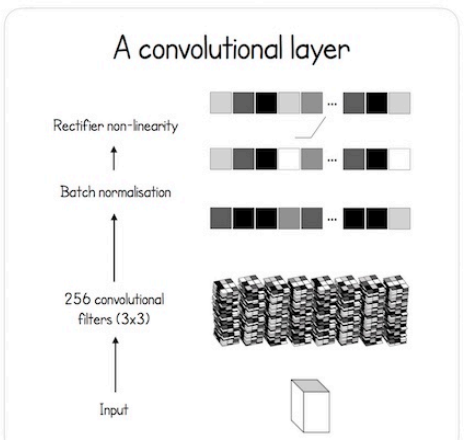
A residual layer

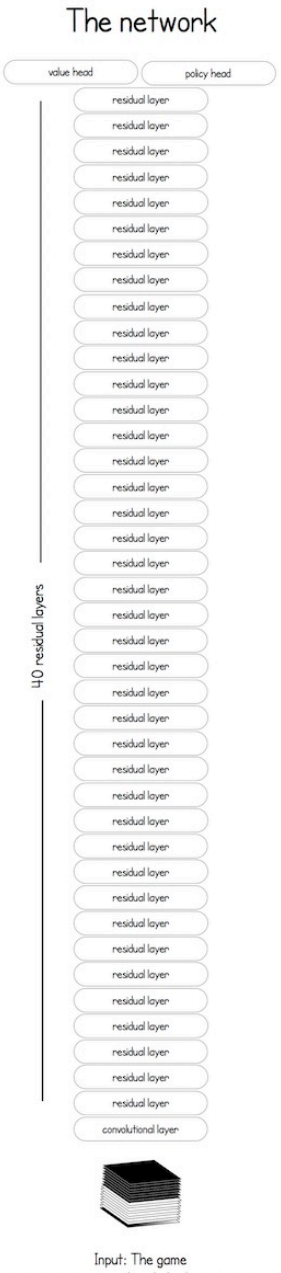


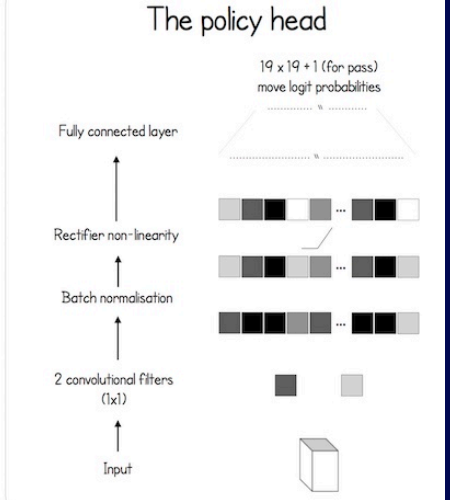
The network learns 'tabula rasa' (from a blank slate)

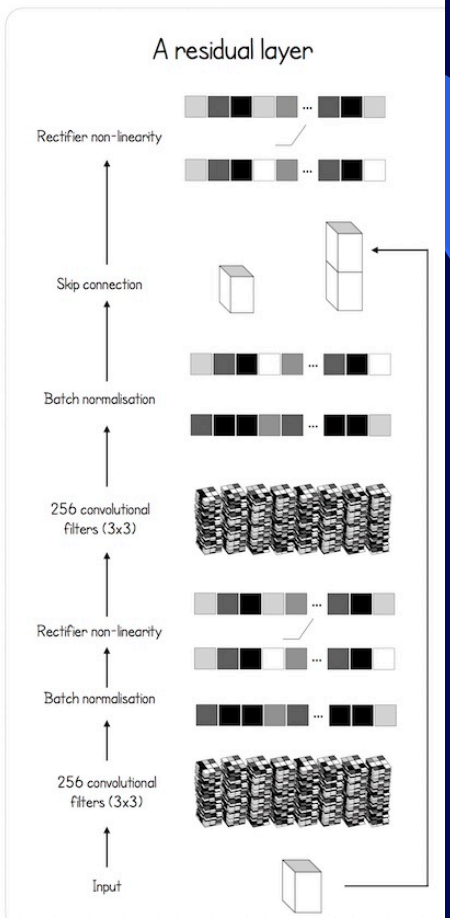
At no point is the network trained using human knowledge or expert moves











The training pipeline for AlphaGo Zero consists of three stages, executed in parallel

SELF PLAY

Create a 'training set'

The best current player plays 25,000 games against itself

See MCTS section to understand how AlphaGo Zero selects each move

At each move, the following information is stored

The game state
(see 'What is a Game State section')

The search probabilities
(from the MCTS)

The winner
(+1 if this player won, -1 if this player lost - added once the game has finished)

RETRAIN NETWORK

Optimise the network weights

A TRAINING LOOP

Sample a mini-batch of 2048 positions from the last 500,000 games

Retrain the current neural network on these positions
- The game states are the input (see 'Deep Neural Network Architecture')

Loss function
Compares predictions from the neural network with the search probabilities and actual winner

PREDICTIONS **p** **v** — Cross-entropy + Mean-squared error + Regularisation — ACTUAL 

After every 1,000 training loops, evaluate the network

EVALUATE NETWORK

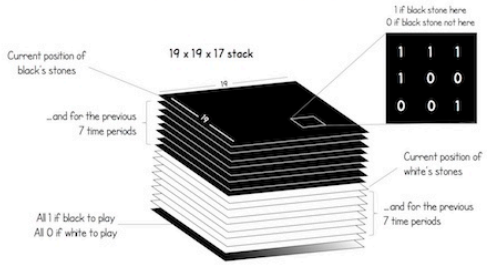
Test to see if the new network is stronger

Play 400 games between the latest neural network and the current best neural network

Both players use MCTS to select their moves, with their respective neural networks to evaluate leaf nodes

Latest player must win 55% of games to be declared the new best player

WHAT IS A 'GAME STATE'



This stack is the input to the deep neural network

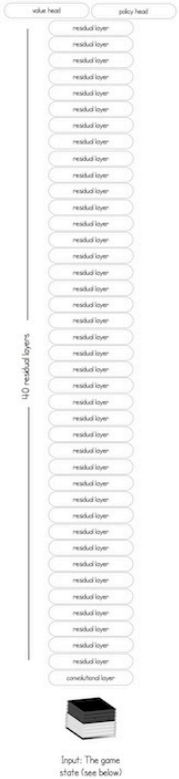
THE DEEP NEURAL NETWORK ARCHITECTURE

How AlphaGo Zero assesses new positions

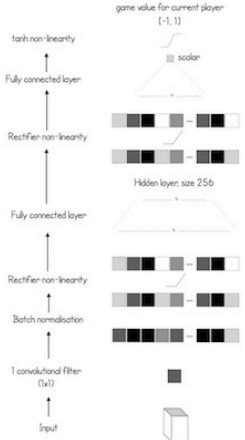
The network learns 'tabula rasa' (from a blank slate)

At no point is the network trained using human knowledge or expert moves

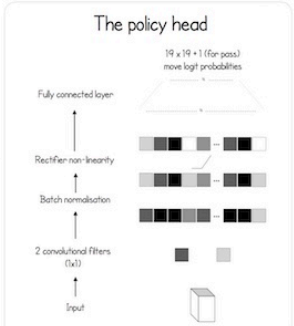
The network



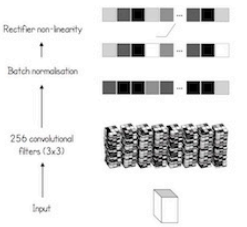
The value head



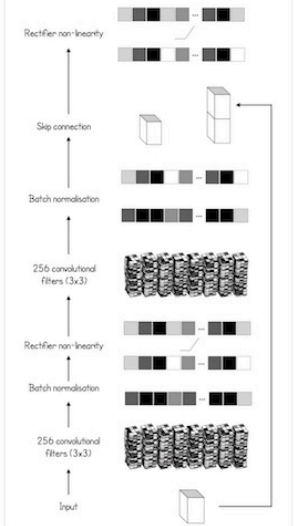
The policy head



A convolutional layer

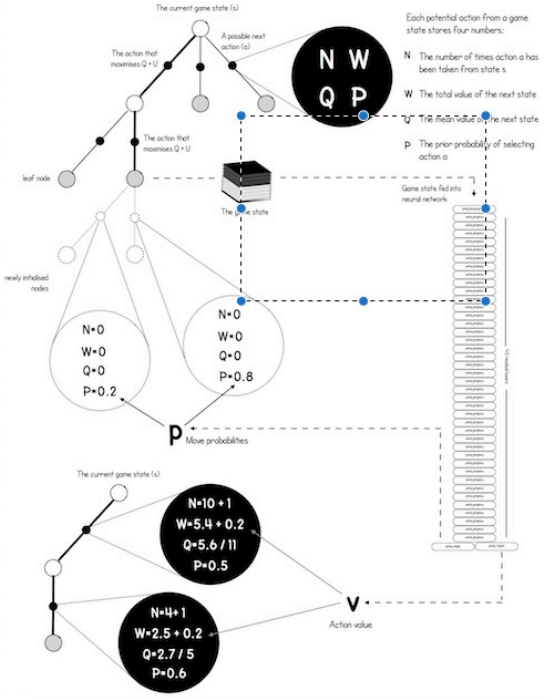


A residual layer



MONTE CARLO TREE SEARCH (MCTS)

How AlphaGo Zero chooses its next move



First, run the following simulation 1,600 times...

Start at the root node of the tree (the current game state)

1. Choose the action that maximises...

$$Q + U$$

The mean value of the next state

A function of P and N that increases if an action hasn't been explored much, relative to the other actions, or if the prior probability of the action is high

Early on in the simulation, U dominates (more exploration), but later, Q is more important (less exploration)

2. Continue until a leaf node is reached

The game state of the leaf node is passed into the neural network, which outputs predictions about two things:

p Move probabilities

v Value of the state (for the current player)

The move probabilities p are attached to the new feasible actions from the leaf node

3. Backup previous edges

Each edge that was traversed to get to the leaf node is updated as follows:

$N \rightarrow N + 1$

$W \rightarrow W + v$

$Q = W / N$

...then select a move

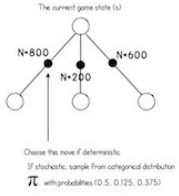
After 1,600 simulations, the move can either be chosen:

Deterministically (for competitive play)
Choose the action from the current state with greatest N

Stochastically (for exploratory play)
Choose the action from the current state from the distribution

$$\pi \sim N^{1/\tau}$$

where τ is a temperature parameter; controlling exploration



Other points

- The sub-tree from the chosen move is retained for calculating subsequent moves
- The rest of the tree is discarded

AlphaStar

- DeepMind considers "perfect information" board games solved
- Next step – Starcraft II - AlphaStar
  - Considered a next "Grand AI Challenge"
  - Complex, long-term strategy, stochastic, hidden info, real-time
  - Plays best Pros - AlphaStar limited to human speed in actions/clicks per minute – so just comparing strategy

Examples – What Learning Approach to Use

- Heart Attack Diagnosis?
- Checkers?
- Self Driving Car?

Examples – What Learning Approach to Use

- Heart Attack Diagnosis?
- Checkers?
- Self Driving Car?
  - Can do supervised with easy to record data of human drivers driving
  - Deep net to represent state and give output
  - But would if we want to learn to drive better than humans?
- RL with actions being steering wheel, brakes, gas, etc.
  - Could initialize training with human data
  - Could use simulators to create lots more data – but need real good simulators!
  - Could learn Tabula Rasa if we want to try to do better than human

Reinforcement Learning Summary

- Learning can be slow even for small environments
- Large and continuous spaces can be handled using a function approximator (e.g. MLP)
- Deep Q learning: States and policy represented by a deep neural network
- Suitable for tasks which require state/action sequences
  - RL not used for choosing best pizza, but could be used to *discover* the steps to make the best pizza
- With RL we don't need pre-labeled data. Just experiment and learn!