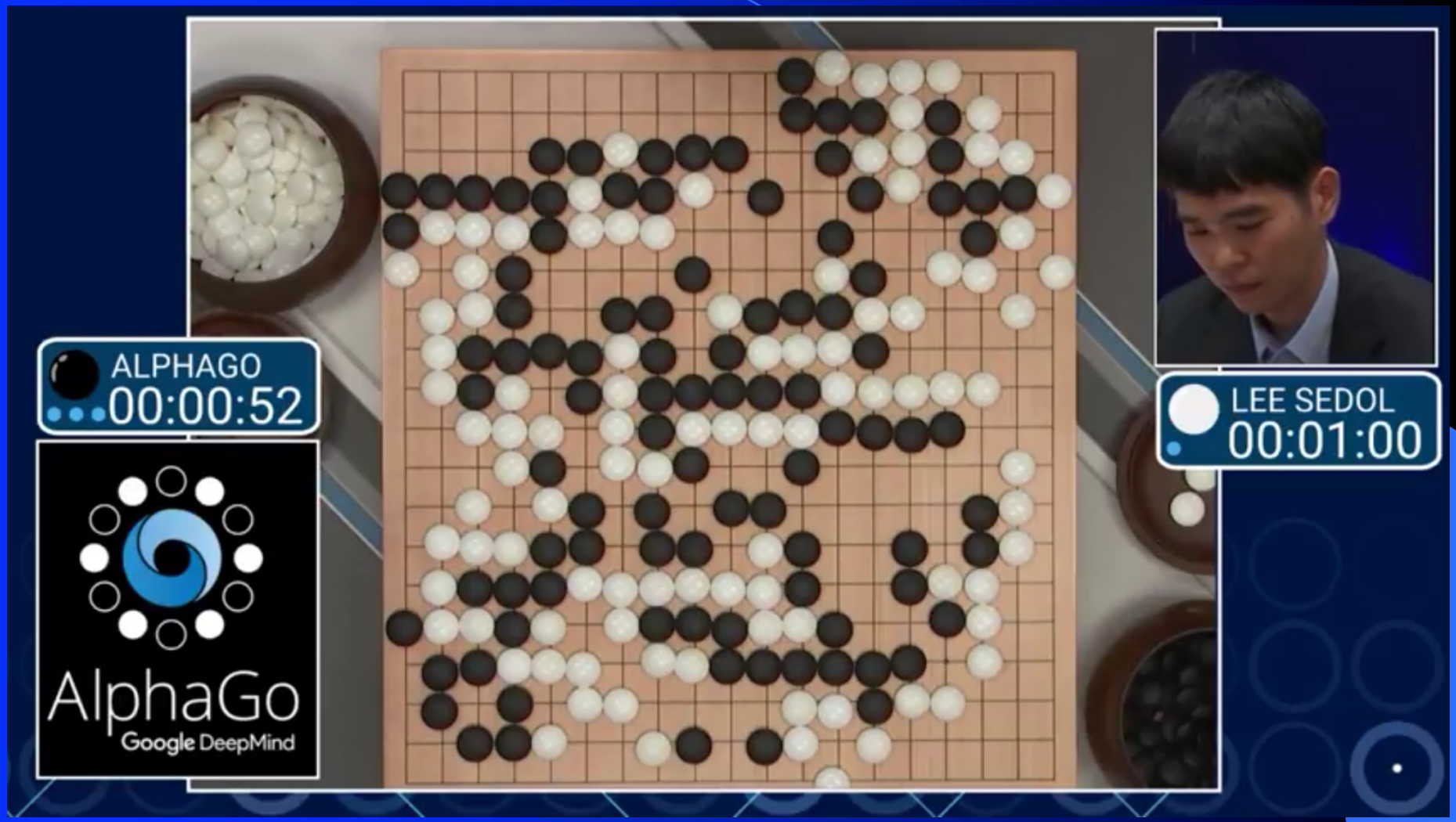
Reinforcement Learning (RL)

- 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) - 1992
- Deep RL (RL with deep neural networks) – Recently demonstrating tremendous potential
  - Especially nice for applications (e.g. games) where it is easy to generate data through simulation or self-play

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 training by 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, <u>and</u> not set up for <u>any</u> 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)
  - Trial and error approach – explore the action space and update action policy based on rewards

Learning a Policy

- Find optimal policy π: $S \rightarrow A$
- $a = \pi(s)$, 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 there 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.
  - Some reward states are absorbing states (e.g. end of game)
- 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

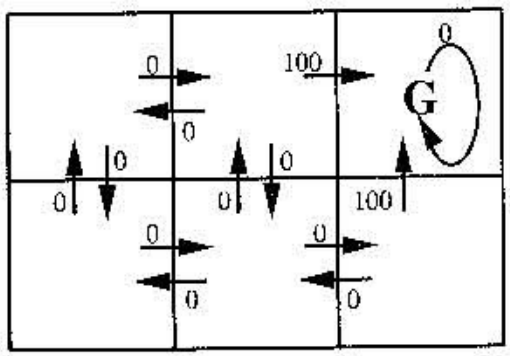
$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Q-Learning

- No model of the world required (is required in some types of RL)
- Just try an action and see what state you end up in and what reward you get. Update the policy based on these results.
- Rather than find the value function of a state, find the value function of an (s,a) pair and call it the Q-value
- Just need to try actions from a state and then incrementally update the policy based on the reward received
- $Q(s,a)$ = Sum of discounted reward for doing a from s and following the current policy thereafter
  - $Q^*(s,a)$ represents the *optimal* Q function giving an optimal policy $\pi^*(s)$

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

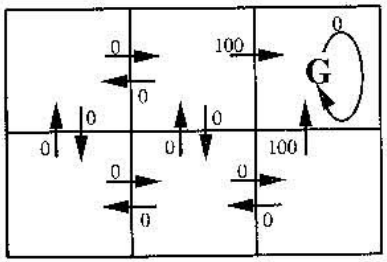
$$\pi^*(s) = \arg\max_{a} Q(s,a)$$



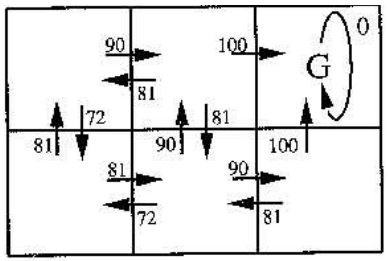
$r(s, a)$ (immediate reward) values

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

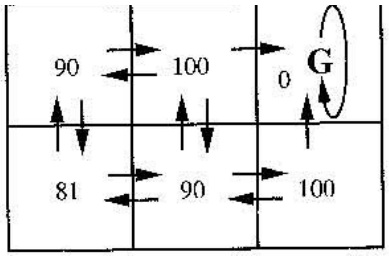
Assume all initial Q-values are 0 and discount factor is .9



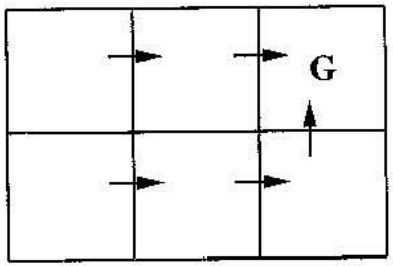
$r(s, a)$ (immediate reward) values



$Q(s, a)$ 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 optimal 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 changing very little) repeat:
  - Choose an arbitrary state *s*
  - Select any action *a* and execute (can use exploitation vs. exploration)
  - Update the Q-function table entry for (*s*, *a*)

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

- Monotonic Convergence – Once updated once, a Q-value can only increase
- We do not need to know the actual reward and state transition functions. Just sample them (Model-less).

Q-Learning Challenge Question

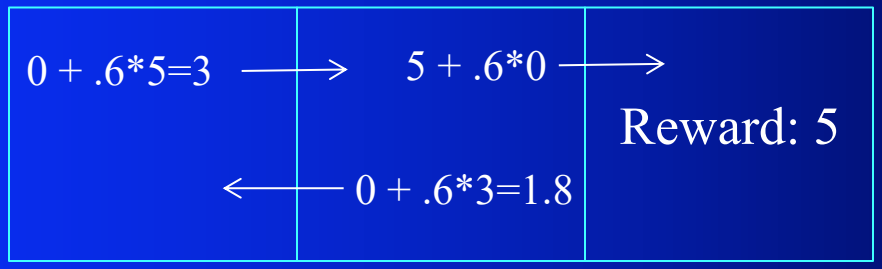
- Assume the deterministic 3 state world below (each cell is a state) where the immediate reward is 0 for entering all states, except the rightmost state, for which the reward is 5, and which is an absorbing state. The only actions are move right and move left (only one of which is available from the 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 and describe the optimal policy.

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

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 rightmost 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 and describe the optimal policy.

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



Optimal Policy: "Choose the Right!!"

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 leftmost 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.

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

Reward: 10			

Real Q-Learning

- System can't see reward states, etc. though we did for our exercise
- Assume a discount factor of .6, and all initial Q-values of 0, and the same reward as the challenge question but we don't see if beforehand
- How would our program learn it?

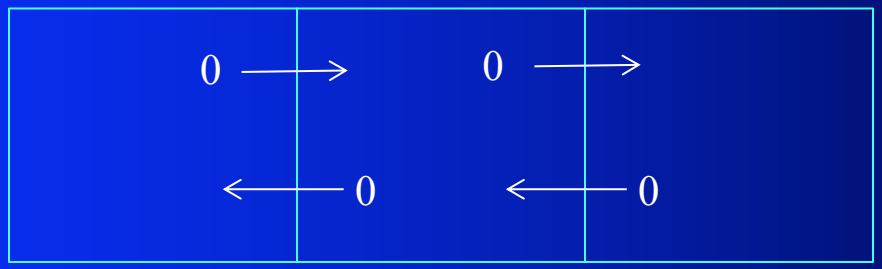
Until Convergence (Q-function not changing or changing very little)

Choose an arbitrary *s*

Select any action *a* and execute (exploitation vs. exploration)

Update the Q-function table entry

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



Real Q-Learning

- System can't see reward states, etc. though we did for our exercise – Model-less
- Assume a discount factor of .6, and all initial Q-values of 0
- How would our program learn it?

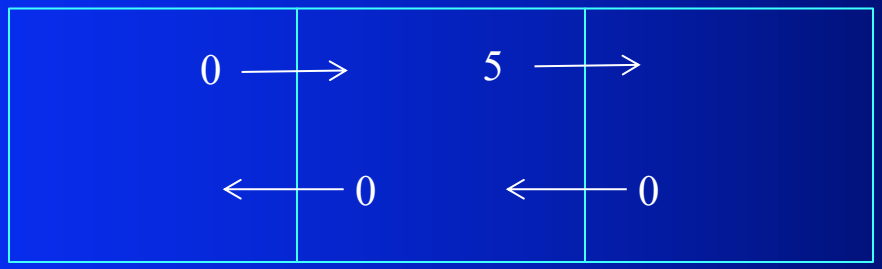
Until Convergence (Q-function not changing or changing very little)

Choose an arbitrary *s*

Select any action *a* and execute (exploitation vs. exploration)

Update the Q-function table entry

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



Real Q-Learning

- System can't see reward states, etc. though we did for our exercise – Model-less
- Assume a discount factor of .6, and all initial Q-values of 0
- How would our program learn it?

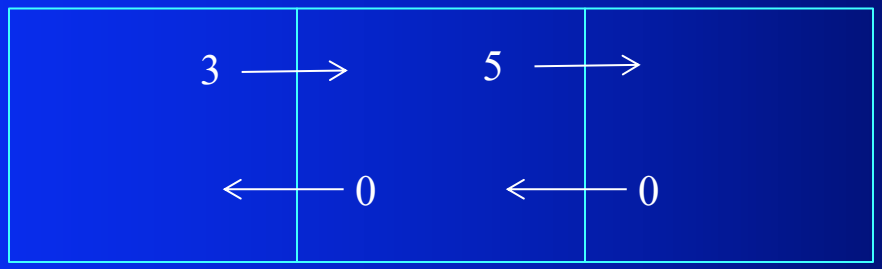
Until Convergence (Q-function not changing or changing very little)

Choose an arbitrary *s*

Select any action *a* and execute (exploitation vs. exploration)

Update the Q-function table entry

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



Real Q-Learning

- System can't see reward states, etc. though we did for our exercise – Model-less
- Assume a discount factor of .6, and all initial Q-values of 0
- How would our program learn it?

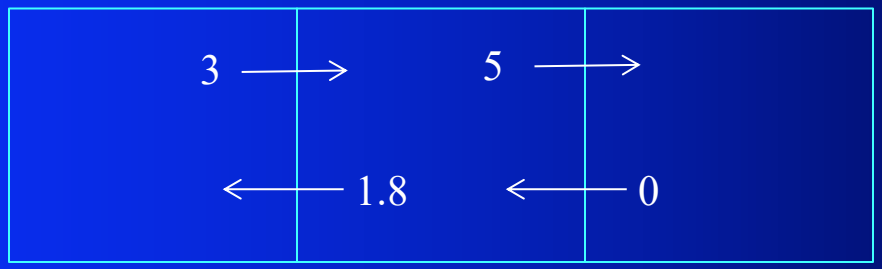
Until Convergence (Q-function not changing or changing very little)

Choose an arbitrary *s*

Select any action *a* and execute (exploitation vs. exploration)

Update the Q-function table entry

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



Non-Absorbing Reward States

- Would if the left state was non-absorbing with a negative reward of -1.

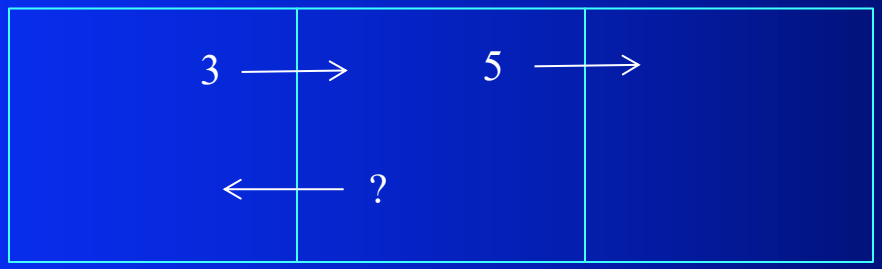
Until Convergence (Q-function not changing or changing very little)

Choose an arbitrary *s*

Select any action *a* and execute (exploitation vs. exploration)

Update the Q-function table entry

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



Non-Absorbing Reward States

- And would if the left state was non-absorbing with a negative reward of -1.

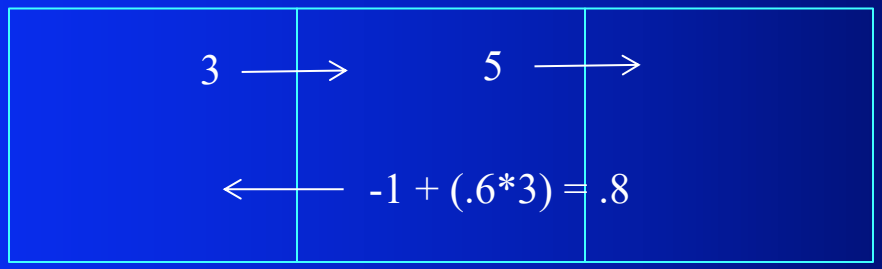
Until Convergence (Q-function not changing or changing very little)

Choose an arbitrary *s*

Select any action *a* and execute (exploitation vs. exploration)

Update the Q-function table entry

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



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

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

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

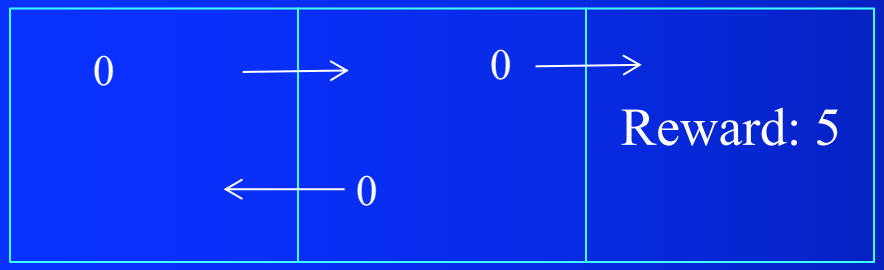
where α_n starts at 1 and decreases over time and n stands for the nth iteration. An example of adapting α_n is

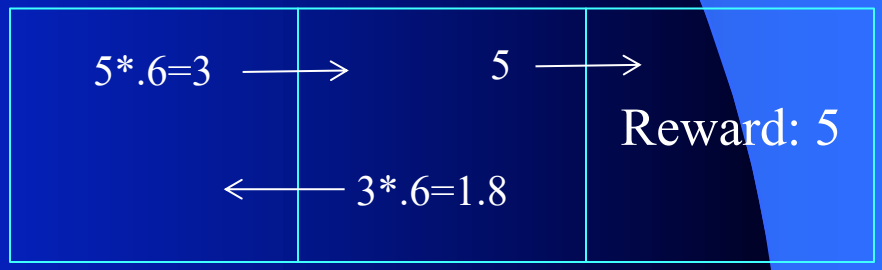
$$\alpha_n = \frac{1}{1 + \#ofvisits(s,a)}$$

- 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)

Episodic Updates

- Note that much efficiency could be gained if you worked back from the goal state (like you did on the challenge question). 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 we can do for Q-learning is rather than start with a new state each step, continue going from state to state until reaching a reward state (episode) Then, propagate the discounted Q-function update all the way back to the initial starting state, allowing multiple updates. This can speed up learning at a cost of memory. This approach is often used but it is not the "true" learning algorithm

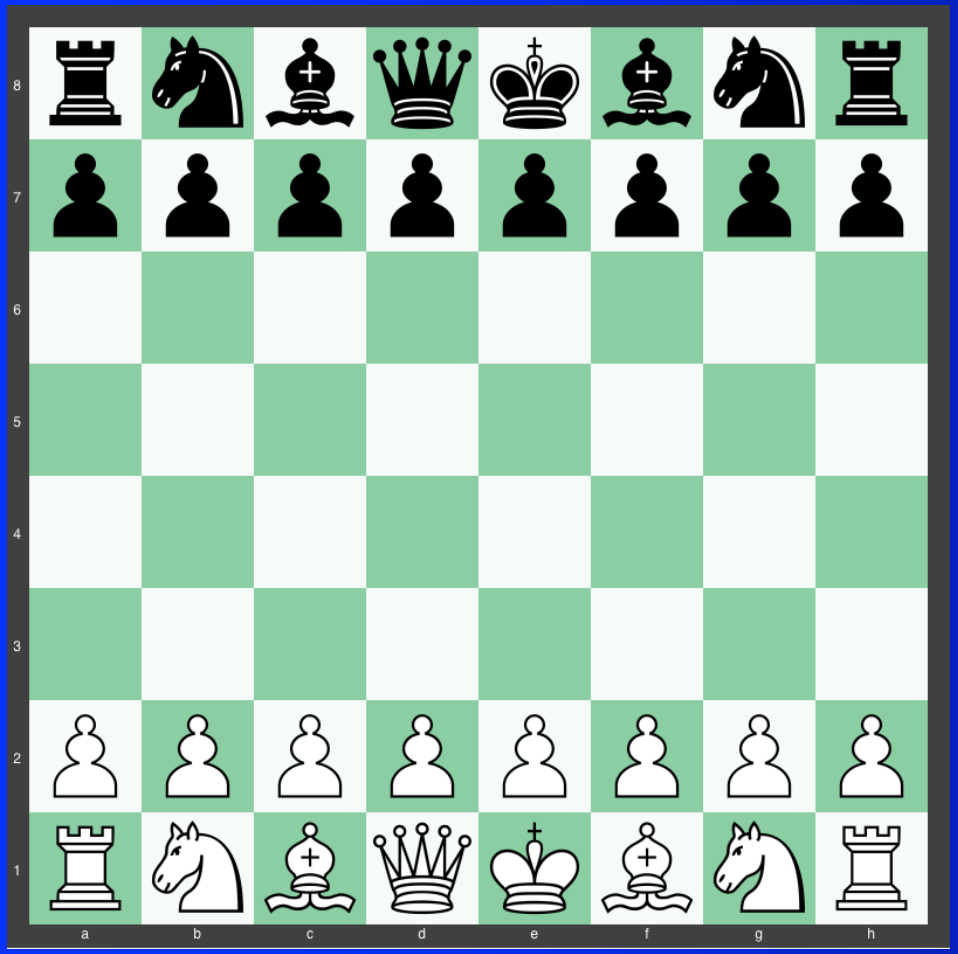


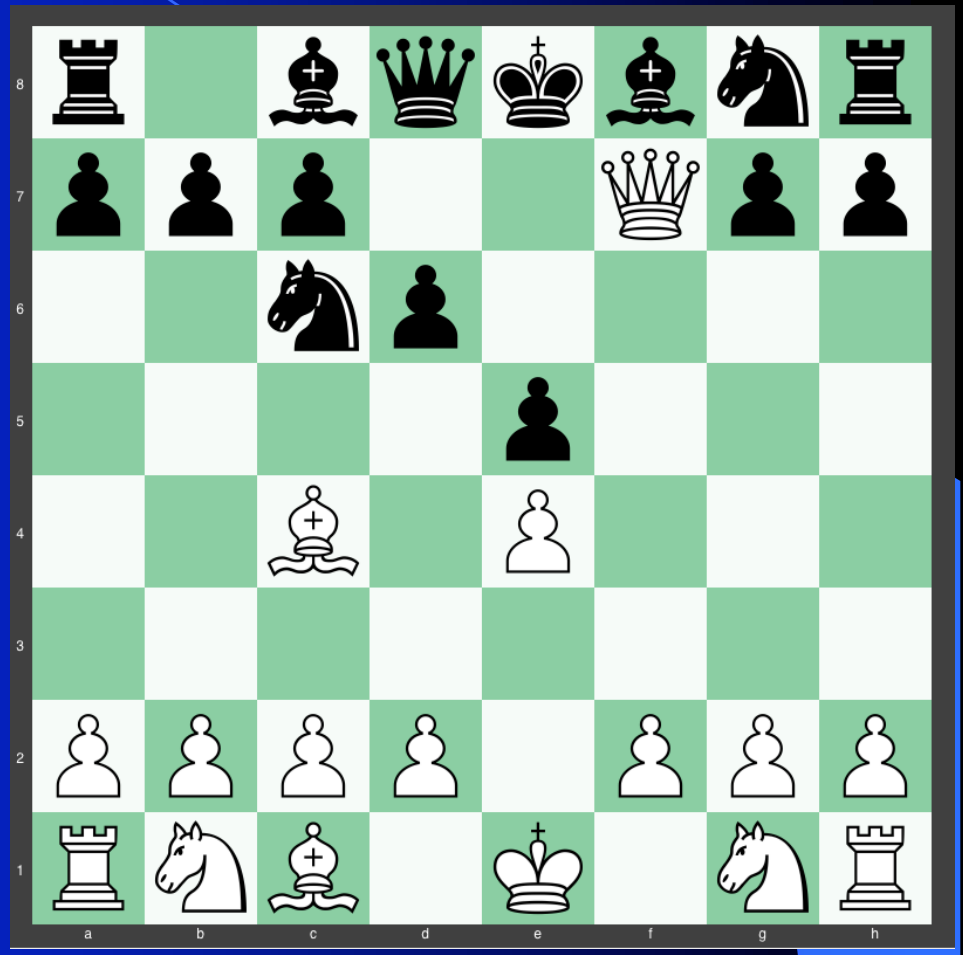


Example - Chess

- Assume reward of 0's except win (+10) and loss (-10)
- Set initial Q-function to all 0's
- Start from any legal state (typically normal start of game) and choose transitions until reaching an absorbing state (win or lose), Episodic
  - Starting in a random state more challenging in this case as many states aren't realistic, etc.
- Finally, after entering an absorbing state, the full chain of preceding state-action pair, gets updated (positive for win or negative for loss).
  - Earlier Q-values have higher discounts on their updates
  - Could do non-episodic, but much slower in this kind of task
- If other actions from a state 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





Replace Q-table with a Function Approximator

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

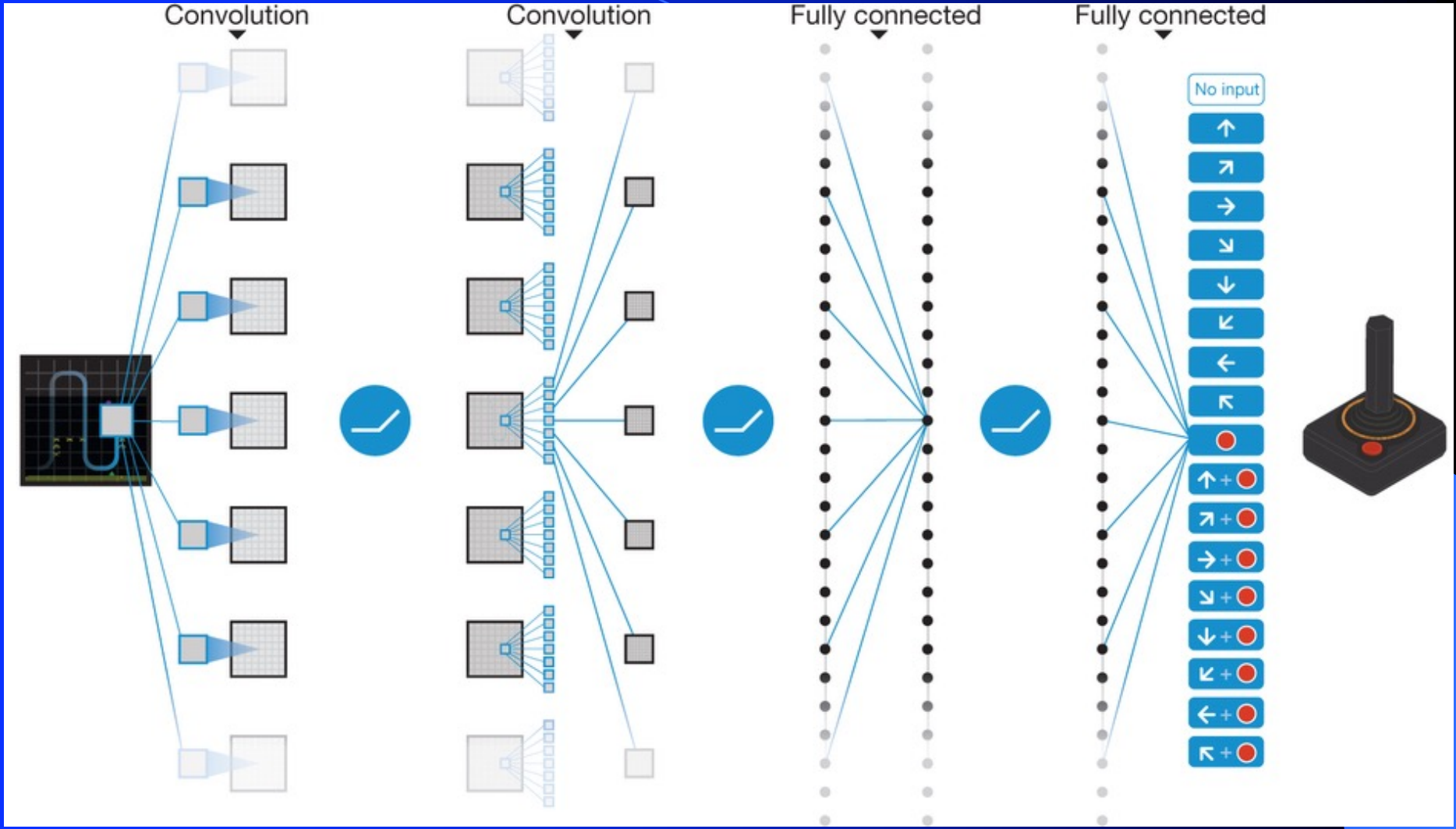
$$\hat{Q}(s,a) = r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$$

- 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
  - Avoids huge or infinite lookup tables (real values, etc.)
  - Allows generalization from all states, not just those seen during training
  - We are not training with the optimal Q-values (we don't know them)
  - Initial Q-values are random based on initial random weights
  - For each update the training error is the difference between the network's current Q-value output (generalization) and the updated Q-value expectation from our standard update equation above
    - If current Q-value is .4 and the updated Q-value is .7, then output error propagated back is .3
  - Converged when Q-values are no longer changing much

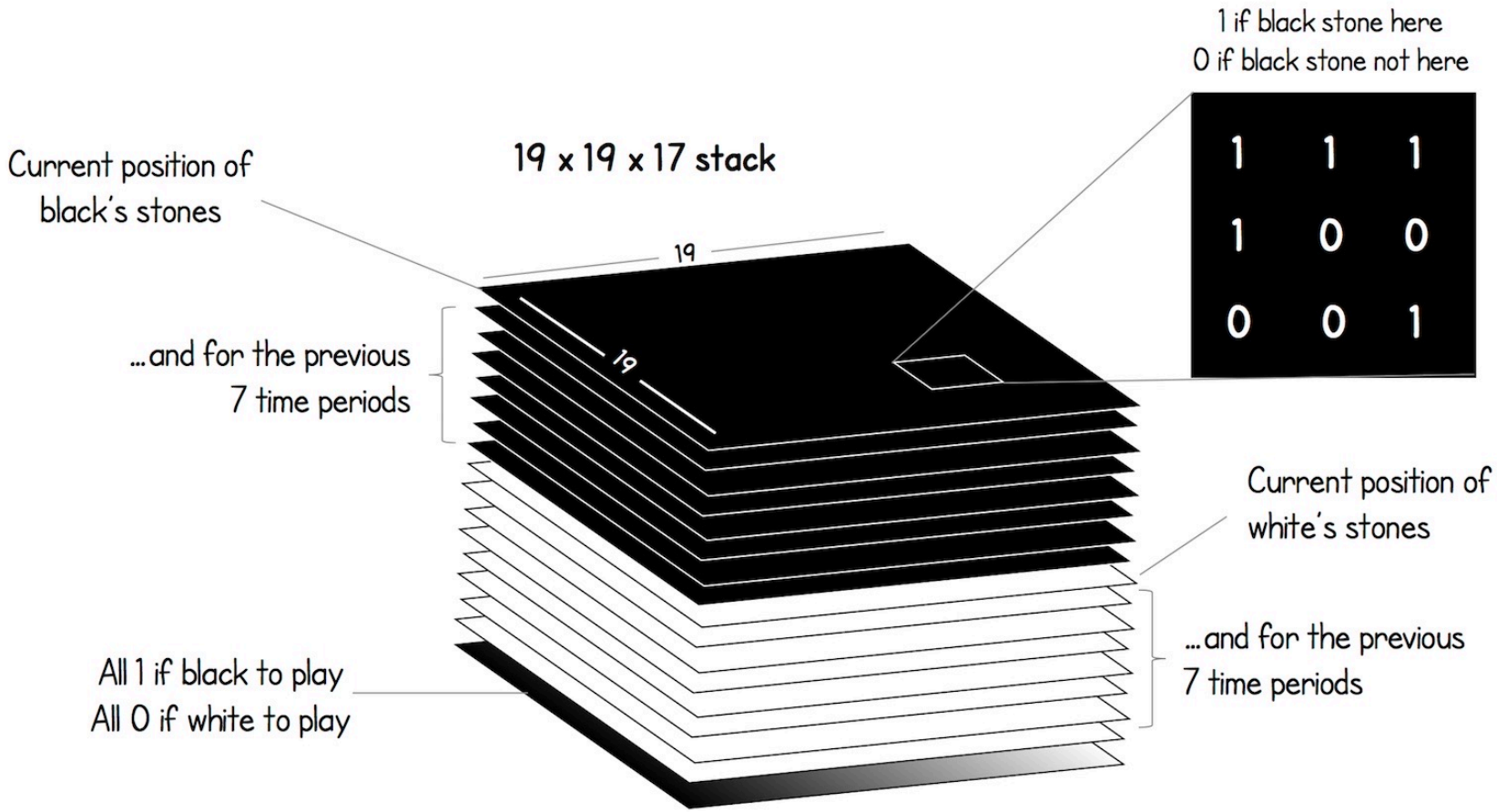
Deep Q-Learning Example

- 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

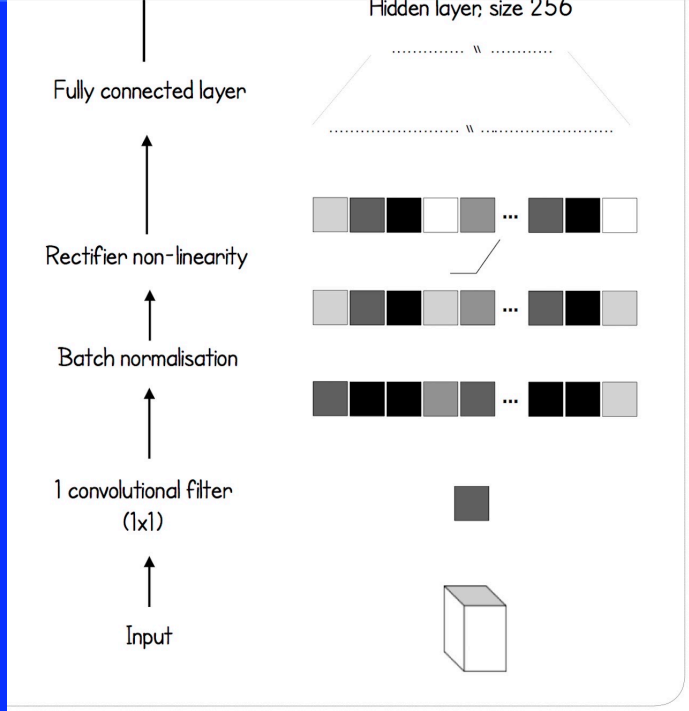
Deep Q Network – 49 Classic Atari Games

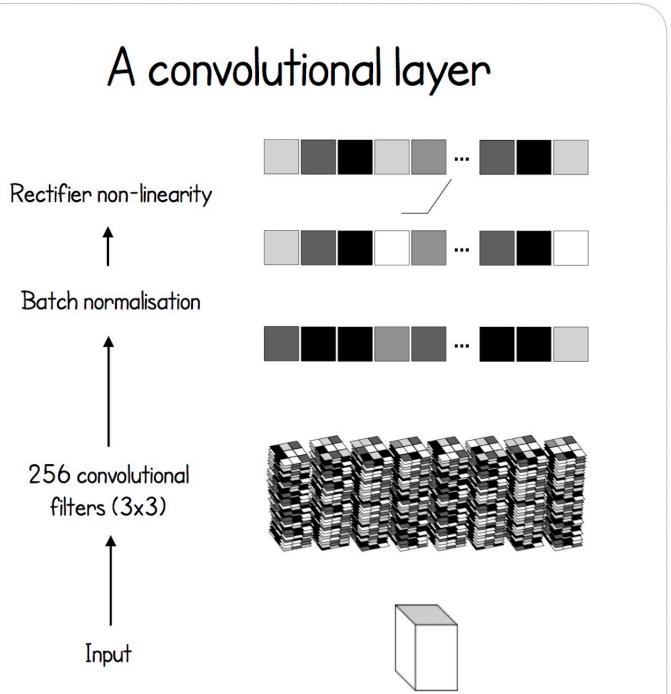


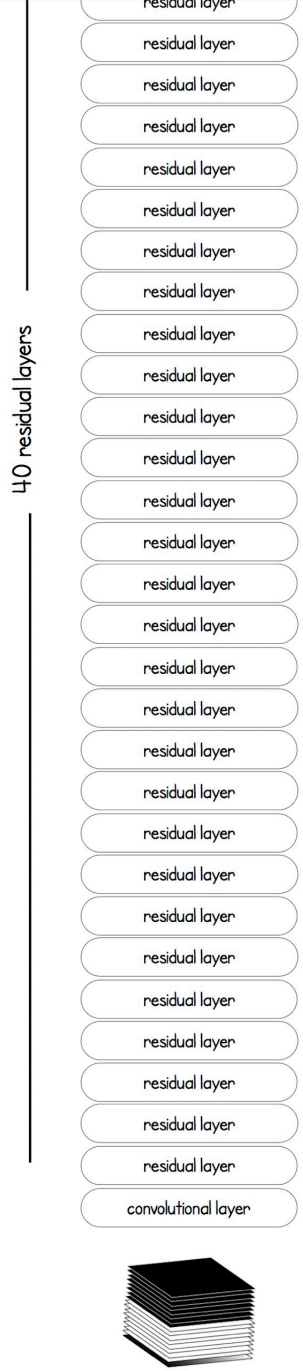
WHAT IS A 'GAME STATE'

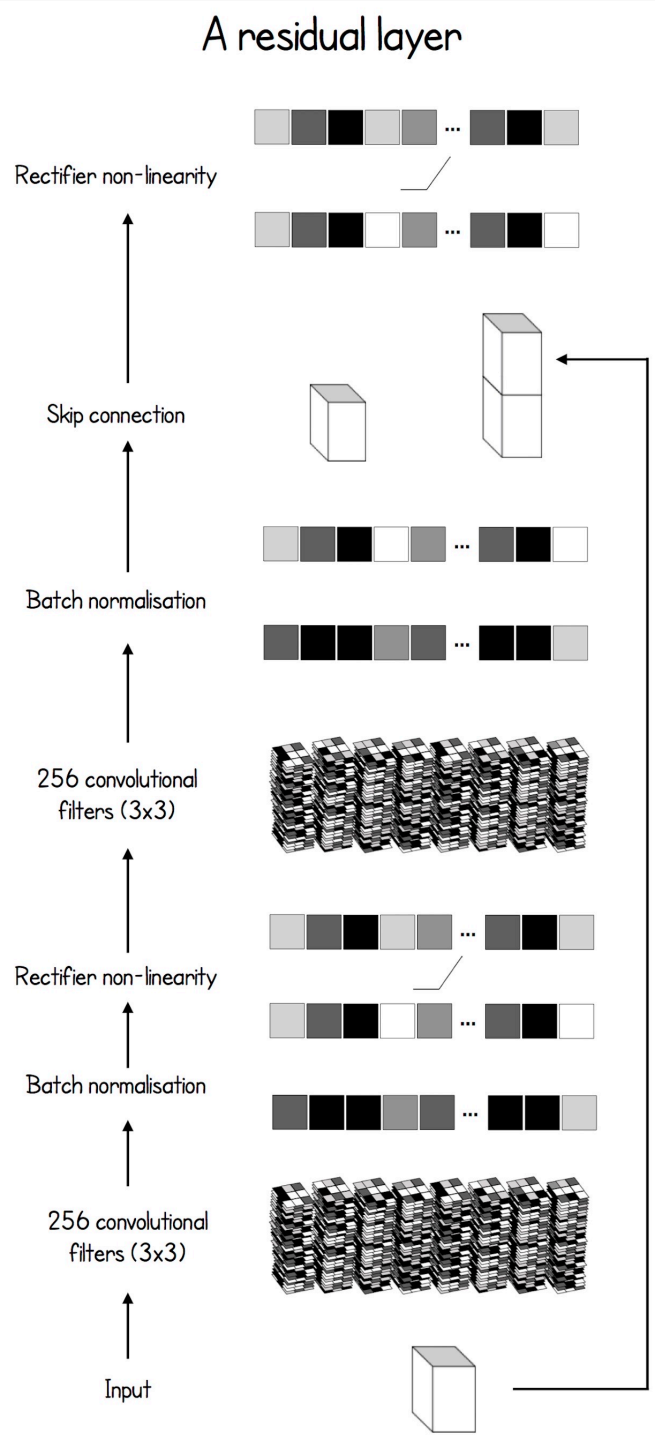


This stack is the input to the deep neural network


Hidden layer, size 256
Fully connected layer
Rectifier non-linearity
Batch normalisation
1 convolutional filter (1x1)
Input

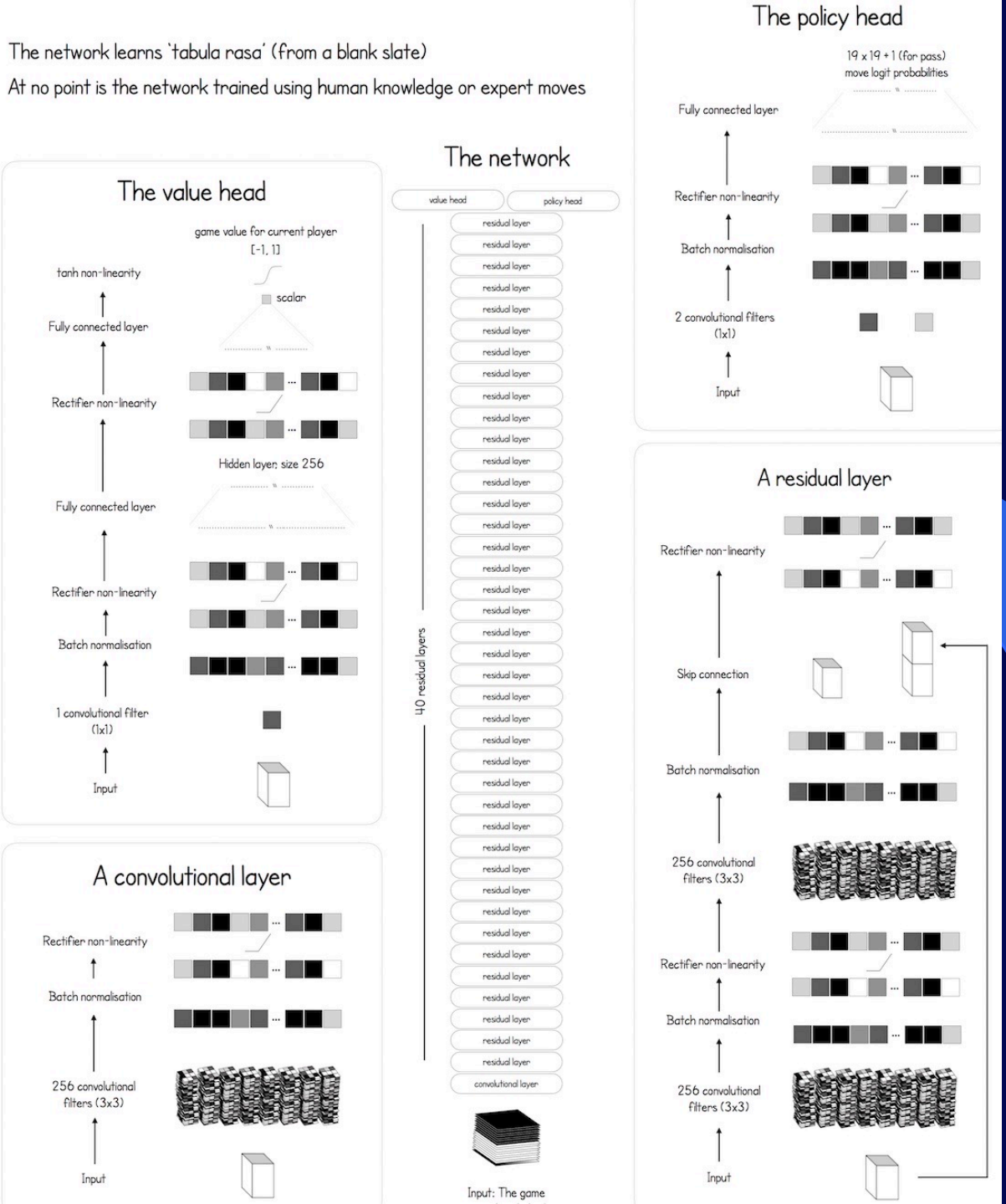

A convolutional layer
Rectifier non-linearity
Batch normalisation
256 convolutional filters (3x3)
Input


40 residual layers
residual layer
convolutional layer
Input: The game


A residual layer
Rectifier non-linearity
Skip connection
Batch normalisation
256 convolutional filters (3x3)
Rectifier non-linearity
Batch normalisation
256 convolutional filters (3x3)
Input

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 — π ACTUAL

After every 1,000 training loops, evaluate the network

EVALUATE NETWORK

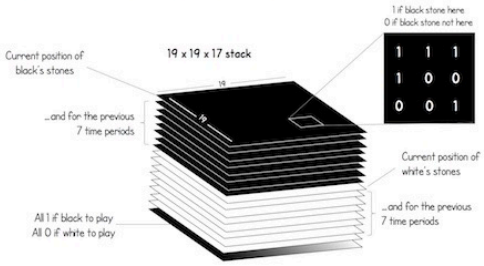
Test to see if the new network is stronger

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

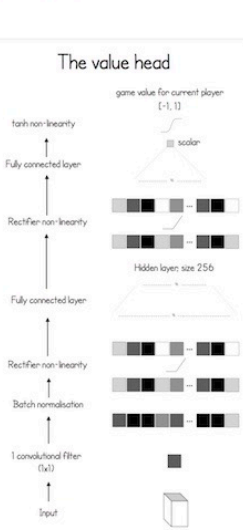
THE DEEP NEURAL NETWORK ARCHITECTURE

How AlphaGo Zero assesses new positions

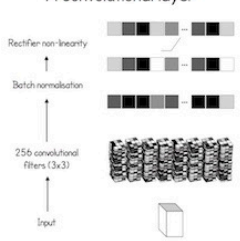
The network learns 'tabula rasa' (from a blank slate)

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

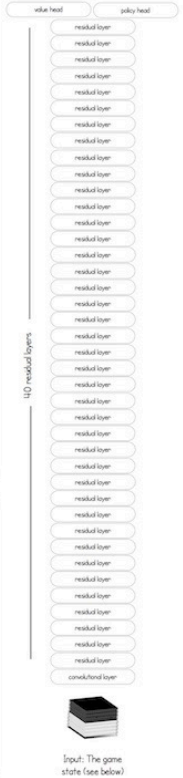
The value head



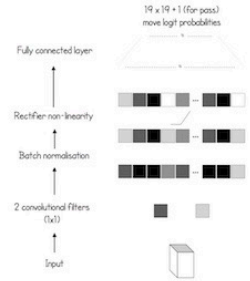
A convolutional layer



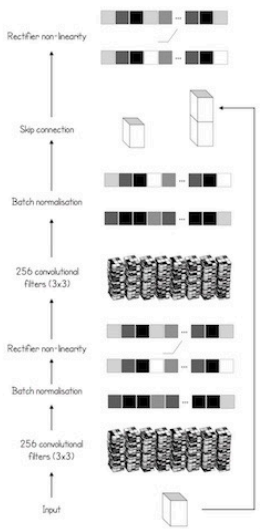
The network



The policy head

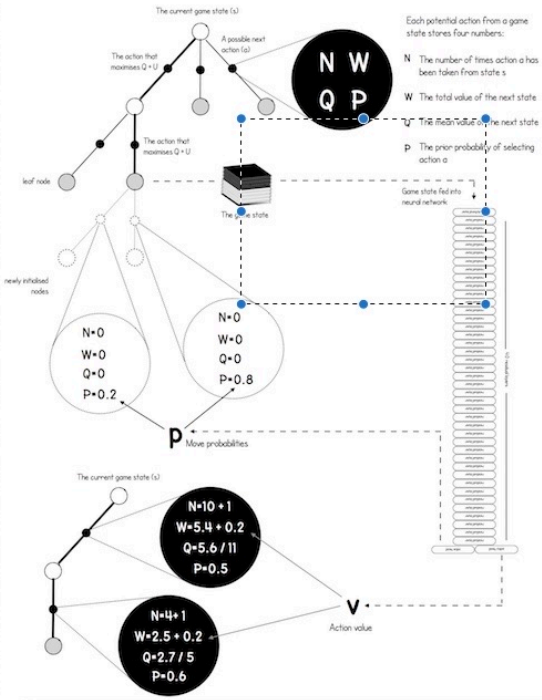


A residual layer



MONTE CARLO TREE SEARCH (MCTS)

How AlphaGo Zero chooses its next move



Each potential action from a game state stores four numbers:

- **N** The number of times action a has been taken from state s
- **W** The total value of the next state
- **Q** The mean value of the next state
- **P** The prior probability of selecting action a

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)

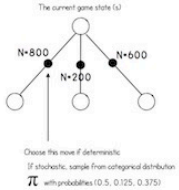
Choose the action from the current state with greatest **N**

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 (with vision input) 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, but…
  - Use simulators to create lots more data – Car just starts driving in simulated environments with rewards (positive and negative)
    - but need real good simulators!
  - Could learn Tabula Rasa if we want to do better than human

Reinforcement Learning Summary

- Learning can be slow even for small environments
  - Can be great for tasks where trial and error is reasonable or can be done through simulation
- 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 – more later
- Suitable for tasks which require state/action sequences
  - RL not used for choosing best pizza, but could be used to *discover* the steps to create the best or a better pizza
  - But need mechanism to efficiently explore (e.g. simulator, self-play, etc.)
- With RL we don't need labeled data. Just experiment and learn!