Games

Arthur J. Redfern
arthur.redfern@utdallas.edu

Apr 24, 2019
Apr 29, 2019

Greetings, Professor Falken.

Hello, Joshua.

A strange game. The only winning move is not to play. How about a nice game of chess?

– War Games (1983)

Figure from http://mentalfloss.com/article/545667/facts-about-wargames

Outline

- Motivation
- Reinforcement learning
- Decision making
  - Value based
  - Policy based
  - Model based
- Appendix
- References

Motivation

Why Play Games?

- Playing games requires making decisions
  - Vision, speech and language were about extracting information from data
  - But then what? What do you do with the information you extracted?
  - Frequently you want to make a decision based on it and take some action
  - So we'll look at how to make decisions in the context of games
  - Realize, though, that it's not a big jump to take the methods used for making decisions in games and apply them to controls, robotics, autonomous vehicles, ...
  - However, it's more fun to start with games so we'll start with games and a question

- How do you learn to make decisions?
  - Imitating someone who's good at making decisions
  - Passively observing the environment to learn what rewards different actions lead to
  - Interacting with the environment to learn what rewards different actions lead to

Reinforcement Learning

Agents And Environments

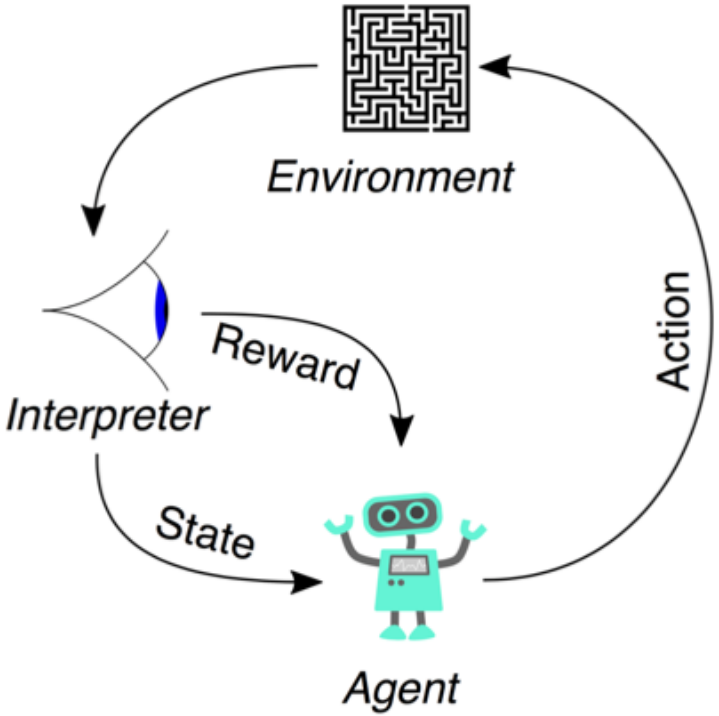


Figure from https://en.wikipedia.org/wiki/Reinforcement_learning

Markov Decision Processes

A mathematical description of the previous page

- S is the set of all valid states
  - s_t is the state at time t
  - The state is a complete description of the fully or partially observable environment
- A is the set of all valid actions
  - a_t is the action at time t
  - Actions are decisions and can be discrete or continuous, state independent or dependent
- R: S x A $\rightarrow \mathbb{R}$ is the reward function that maps states and actions to a reward
  - r_t is the reward at time t
  - In a stochastic setting R is a random variable with pmf $p_R(r_t \mid s_t, a_t)$ such that $r_t \sim R(s_t, a_t)$
- P: S x A $\rightarrow$ S is the state transition function that maps states and actions to new states
  - In a stochastic setting P is a random variable with pmf $p_P(s_{t+1} \mid s_t, a_t)$ such that $s_{t+1} \sim P(s_t, a_t)$
- μ: $\rightarrow$ S is the initial state function that returns the initial state
  - In a stochastic setting μ is a random variable with pmf $p_\mu(s_0)$ such that $s_0 \sim \mu$

Policies

- π: S $\rightarrow$ A is a policy function used by the agent to take actions
  - A policy tells an agent how to act: when in state s_t take action a_t
  - Deterministic policy $a_t = \pi(s_t)$
  - Stochastic policy $a_t \sim \pi(s_t)$ with pmf $p_\pi(a_t \mid s_t)$

- A function with parameters θ is frequently used for the policy
  - Stochastic policy $a_t \sim \pi_\theta(s_t)$ with pmf $p_{\pi\_\theta}(a_t \mid s_t)$
  - These slides will typically use xNNs for policy functions
    - Universal function approximators
    - Learning a policy requires training a network

Policies

- Categorical policies are used for discrete action spaces
  - Ex: xNN + softmax maps a state to a vector representing a pmf of actions
    - Elements in the vector represent the probability of the corresponding action
    - Actions are sampled based on the pmf
  - Log likelihoods are frequently needed
    - $\log p_\pi(a_t \mid s_t)$

- Diagonal Gaussian policies are (commonly) used for continuous action spaces
  - Ex: xNN maps a state to a vector representing mean actions
    - $\mu_\pi(s_t) = E[\pi(s_t)]$
    - Have a separate vector representing the std dev of actions $\sigma_\pi(s_t)$
    - $\sigma_\pi(s_t)$ is maybe fixed, maybe from a network
    - Let $z_t \sim N(0, I)$
    - Actions are sampled $a_t \sim \mu_\pi(s_t) + \sigma_\pi(s_t) \odot z_t$
  - Log likelihoods are frequently needed
    - $\log p_\pi(a_t \mid s_t) = -0.5(\Sigma_{i=0:k-1}((a_i - \mu_i)^2/\sigma_i^2 + 2 \log\sigma_i) + k \log 2\pi)$

Trajectories

- A trajectory τ is a sequence of states, actions and rewards
  - $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots)$
- The probability and generation of a trajectory given a policy π
  - $Pr(\tau \mid a \sim \pi) = p_\mu(s_0) \prod_{t=0:T-1} p_P(s_{t+1} \mid s_t, a_t)\, p_\pi(a_t \mid s_t)$
  - Initialize $s_0 \sim \mu$ and $t = 0$
  - Loop
    - $a_t \sim \pi(s_t)$
    - $r_t \sim R(s_t, a_t)$
    - $s_{t+1} \sim P(s_t, a_t)$
    - $t \leftarrow t + 1$
- Notation
  - $\tau \sim \pi$ will be used to denote that the trajectory is created as above unless otherwise specified (i.e., a part of the default is explicitly overwritten)

Total Rewards

- Finite horizon undiscounted total reward
  - $R_{t:T-1}(\tau) = \Sigma_{i=t:T-1}\, r_i$
- Infinite horizon discounted total reward
  - $R_{t:\infty}(\tau, \gamma) = \Sigma_{i=t:\infty}\, \gamma^{i-t}\, r_i\,,\ \gamma \in (0, 1)$
  - Allows convergence
  - Favors sooner rewards more than later rewards
- Finite horizon discounted total reward
  - $R_{t:T-1}(\tau, \gamma) = \Sigma_{i=t:T-1}\, \gamma^{i-t}\, r_i\,,\ \gamma \in (0, 1)$
  - Favors sooner rewards more than later rewards
  - Frequently used in practice
- Notation
  - $R_{t:}(\tau)$ will be used to generically indicate the total reward starting at time t and continuing to the end

The Goal Of Reinforcement Learning

- Choose a policy π that maximizes the expected total reward $R_{t:}(\tau)$
- Expected total reward
  - $J(\pi) = \int_{\tau} Pr(\tau \mid a \sim \pi)\, R_{t:}(\tau) = E[R_{t:}(\tau) \mid \tau \sim \pi]$
- The optimal policy is the policy that maximizes the expected total reward
  - $\pi^* = \arg\max_{\pi} J(\pi)$
  - The policy tells the agent how to act
  - So knowing the optimal policy allows the agent to maximize the expected total reward

Value Function

- The value function returns the expected total reward if
  - In state s_t and
  - Actions are chosen according to policy π for t onward
  - $V^{\pi}(s_t) = E[R_{t:}(\tau) \mid s_t, \tau \sim \pi \text{ for t onward}]$

- The optimal value is the maximum expected total reward
  - $V^*(s_t) = \max_{\pi} V^{\pi}(s_t) = \max_{a\_t} Q^*(s_t, a_t)$

- The value function tells you how good a state is for a given policy
  - These slides will typically use xNNs for value functions

Action Value Function (Q Function)

- The action value function (Q function) returns the expected total reward if
  - In state s_t and
  - Action a_t is taken and
  - Subsequent actions are chosen according to policy π for t + 1 onward
  - $Q^{\pi}(s_t, a_t) = E[R_{t:}(\tau) \mid s_t, a_t, \tau \sim \pi \text{ for t+1 onward}]$

- The optimal action value (optimal Q value) is the maximum expected total reward after taking action a_t in state s_t
  - $Q^*(s_t, a_t) = \max_{\pi} Q^{\pi}(s_t, a_t)$
  - $a_t^* = \arg\max_{a\_t} Q^*(s_t, a_t)$

- Knowing $Q^*(s_t, a_t)$ allows the selection of the optimal action a_t for a given state s_t
  - These slides will typically use xNNs for action value functions

Bellman Equation

- Value and action value functions satisfy the Bellman equation
  - This property will be exploited in a fixed point framework for iterative estimation
  - Note the max in the optimal value and action value functions, this will implicitly increase the value in that fixed point equation via looking ahead 1 action (later methods that integrate tree search will improve on this basic strategy)

- Values
  - $V^{\pi}(s_t) = E[R(s_t, a_t) + \gamma V^{\pi}(s_{t+1}) \mid s_t, \tau \sim \pi \text{ for t onward}]$
  - $V^{*}(s_t) = \max_{a\_t} E[R(s_t, a_t) + \gamma V^{*}(s_{t+1}) \mid s_t, \tau \sim \pi^{*} \text{ for t+1 onward}]$

- Action values
  - $Q^{\pi}(s_t, a_t) = E[R(s_t, a_t) + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) \mid s_t, a_t, \tau \sim \pi \text{ for t+1 onward}]$
  - $Q^{*}(s_t, a_t) = E[R(s_t, a_t) + \gamma \max_{a\_t+1} Q^{*}(s_{t+1}, a_{t+1}) \mid s_t, a_t, \tau \sim \pi^{*} \text{ for t+2 onward}]$

Advantage Function

- $A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$

- Interpretation
  - Used to specify how much better it is to take action a_t in s_t instead of $a_t \sim \pi(s_t)$
  - Assumes $a_i \sim \pi(s_i)$ for all $i \geq t + 1$
  - Positive advantage means that the chosen action was better than what was expected from the policy

Decision Making

Methods

- Model free methods
  - Value based
  - Policy based
  - Combined value and policy based
    - Will include these with the policy based methods
    - As this is how they're commonly presented
- Model methods
  - Tree search
    - Can still use value and policy functions
- Note
  - Only a few examples of the different methods are covered in these slides, many more variations exist

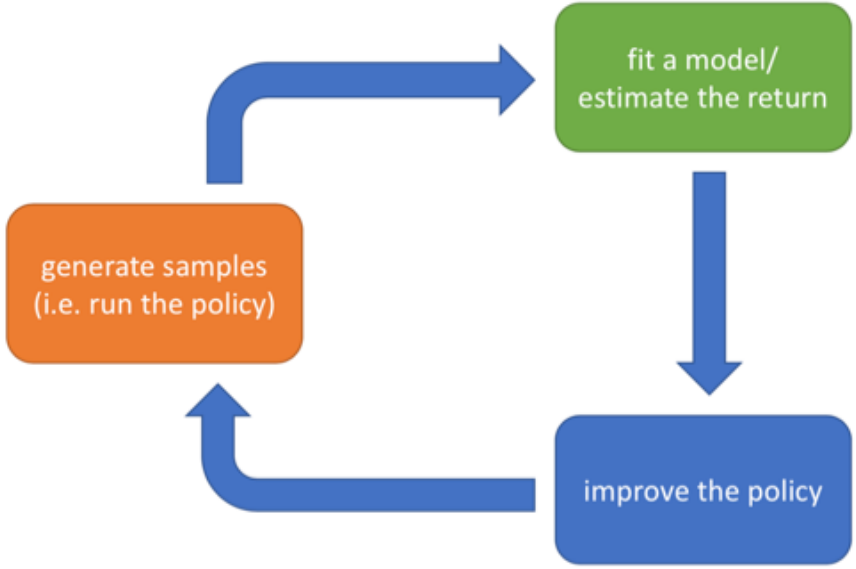
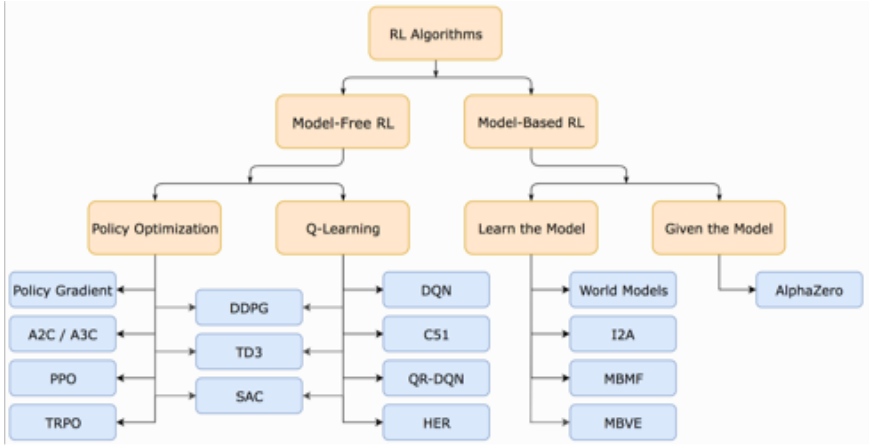


Figure from http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-4.pdf

Methods

- Model free methods
  - Value based
  - Policy based
  - Combined value and policy based
    - Will include these with the policy based methods
    - As this is how they're commonly presented
- Model methods
  - Tree search
    - Can still use value and policy functions
- Note
  - Only a few examples of the different methods are covered in these slides, many more variations exist



Note
- This list is not comprehensive
- We'll still only discuss a fraction of this list

Figure from https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Sample Efficiency

- A measure of how many samples (number of times that the policy needs to be run) to get a good policy
- Off policy methods
  - Methods that can improve a policy without generating new samples from the policy
- On policy methods
  - Every time the policy is changed new samples need to be generated to further improve the policy

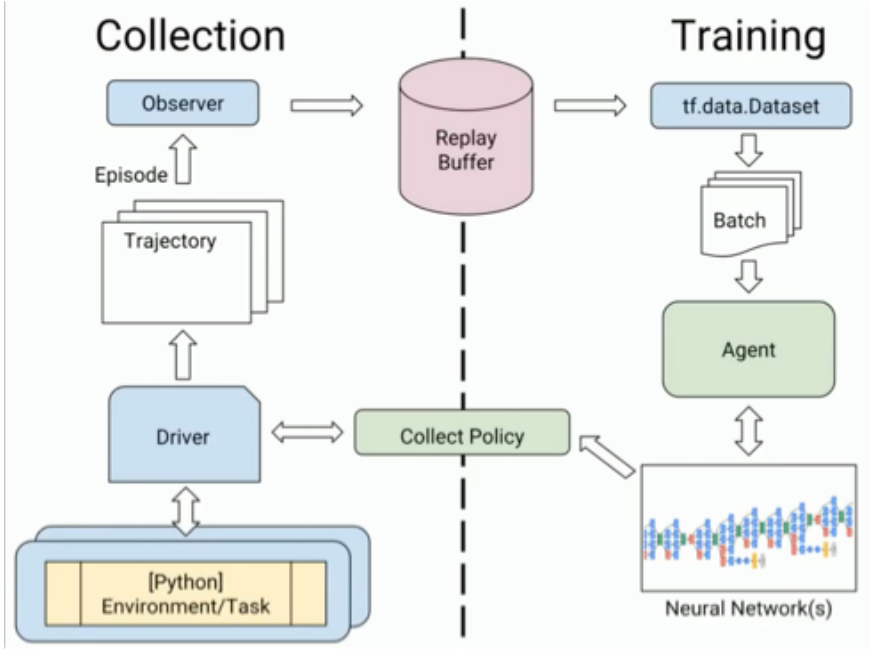
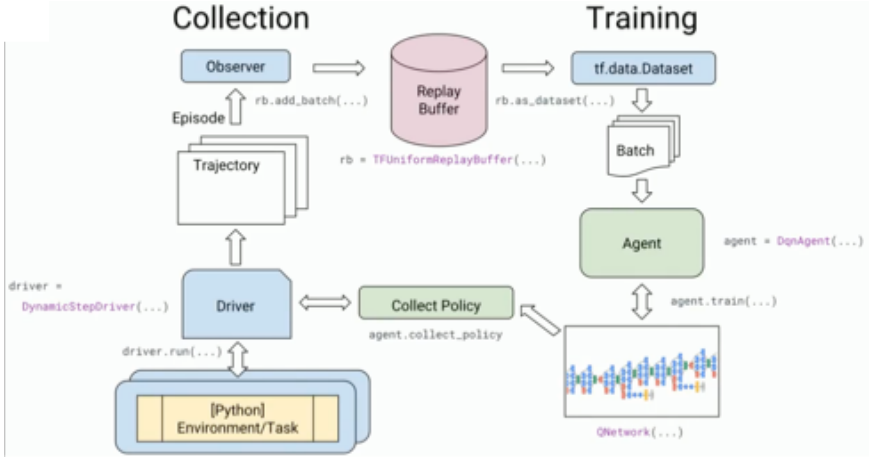


Figure from https://www.youtube.com/watch?v=-TTziY7EmUA

Sample Efficiency

- A measure of how many samples (number of times that the policy needs to be run) to get a good policy
- Off policy methods
  - Methods that can improve a policy without generating new samples from the policy
- On policy methods
  - Every time the policy is changed new samples need to be generated to further improve the policy



TensorFlow is adding a number of reinforcement learning related library components related to data collection and agent training, for additional information see:

- TF-Agents: a library for reinforcement learning in TensorFlow
  - https://github.com/tensorflow/agents
  - https://github.com/tensorflow/agents/tree/master/tf_agents/colabs
- Reinforcement learning in TensorFlow with TF-Agents (TF Dev Summit '19)
  - https://www.youtube.com/watch?v=-TTziY7EmUA

Figure from https://www.youtube.com/watch?v=-TTziY7EmUA

Examples

- Learning is slow and an agent makes lots of "mistakes”
- So it's ideal to learn in a simulated environment that's as close to the real environment as possible for things that matter
- This is where reinforcement learning has been most successful

Decision Making – Value Based

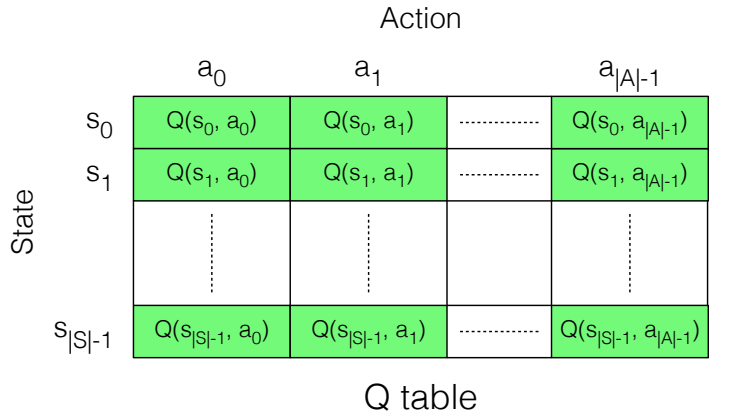
Strategy

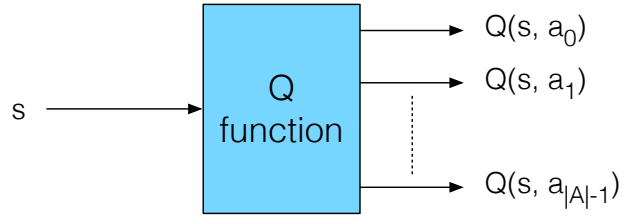
- Core idea
  - Approximate $Q^*(s_t, a_t)$ with $Q_\Theta(s_t, a_t)$
  - This gives the maximum expected reward for every state action pair
  - Select current action $a_t = \arg\max_{a\_t} Q_\Theta(s_t, a_t)$ resulting in the maximum expected reward

- Comments
  - This is indirectly optimizing what you want (the optimal action for a given state)
  - The Bellman equation is typically used in a fixed point setting for approximating $Q^*(s_t, a_t)$
  - This has failure modes → convergence to the optimal value is not guaranteed

Q Tables And Functions

- Q table
  - Use when there are a small number of states * actions
- Q function
  - Use when there are a large number of states * actions but a small number of actions (i.e., the state space is large but the action space is small)





Q Learning Via A Fixed Point Equation

- Q learning fixed point iteration
  - Initialize Q(s, a)
  - Loop
    - Choose an action a based on state s and Q(s, a)
      - Typically favor the action with the largest Q(s, a) value (exploit)
      - Sometimes take a random action (explore)
    - From the environment receive a reward r and new state s_{new}
    - Update entry Q(s, a) in the Q table
      - $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a\_new} Q(s_{new}, a_{new}) - Q(s, a))$
      - When Q(s, a) converges to the fixed point the update term on the RHS should be 0
- Comments
  - The Bellman equation allows us to find a valid Q function
  - The max allows us to increase the Q function towards Q* each iteration

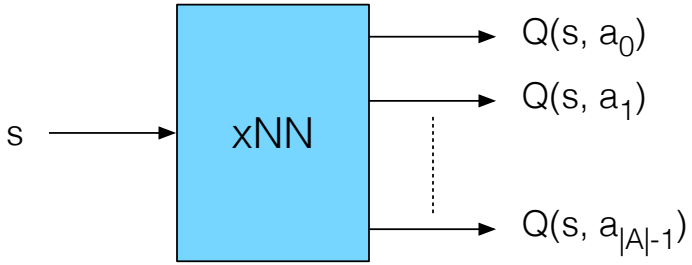
Exploitation Vs Exploration

- When learning you're effectively deciding between taking the action with the largest Q vs taking a random action

- The epsilon greedy approach to exploitation vs exploration during training
  - Let |A| be the number of possible actions
  - Assign the following probabilities to actions
    - Random actions each have probability $\epsilon / |A|$
    - The best action has probability $1 - \epsilon$
  - Use a value of ϵ close to 1 in the beginning (favor exploring)
  - As time goes by shrink ϵ to close to 0 at the end (favor exploiting)

Deep Q Network (DQN)

- Strategy
  - A xNN with parameters Θ is used to map from state s_t to $Q_\Theta(s_t, a_t)$, approximating $Q^*(s_t, a_t)$, and action a_t is selected corresponding to the maximum $Q_\Theta(s_t, a_t)$
  - Training data is collected via experience replay using an epsilon greedy approach to action selection and presented in randomized batches
  - A regression style loss is constructed via the Bellman equation
  - Parameters Θ are updated via gradient descent



- This is all standard xNN stuff
  - Just applied to a different type of problem with a few tricks for collecting data and computing losses

Data

- Experience replay
  - At each time step get experience (s_t, a_t, r_t, s_{t+1}) and save it to a buffer
  - Collect and save experience with any / all of the policies throughout the learning process; note the efficient collection and use of data
  - During learning select random batches (to break up correlations)
  - Modifications / improvements to Q learning modify the batch selection

- Prioritized experience replay
  - Select rare or important experiences more often (i.e., bias the batch selection)
  - Sample experience j ~ Pr(j) = $p_j^{\alpha}/\Sigma_i p_j^{\alpha}$
  - Update experience probability p_j = |target – output| (see subsequent slides) after computing error
    - Biases the sampling to experiences that have larger errors
  - Modify the gradient calculation to account for the biased sampling

Loss

- Error
  - $e_t = 0.5\,((r_t + \gamma\, Q_\Theta(s_{t+1}, \arg\max_{a\_t+1} Q_\Theta(s_{t+1}, a_{t+1}))) - Q_\Theta(s_t, a_t))^2$
    target output
  - Note the target – output form of the error
  - Modifications / improvements to Q learning shown in the next slides will change the target
- Gradient
  - $\partial e_t/\partial\Theta = ((r_t + \gamma\, Q_\Theta(s_{t+1}, \arg\max_{a\_t+1} Q_\Theta(s_{t+1}, a_{t+1}))) - Q_\Theta(s_t, a_t))\, \partial Q_\Theta(s_t, a_t)/\partial\Theta$
  - Error sensitivity with respect to parameters
  - The high level library can compute an error gradient of end to end differentiable Q_Θ functions using automatic differentiation with reverse mode accumulation
- Parameter update
  - $\Theta \leftarrow \Theta - \alpha\, \partial e_t/\partial\Theta$
  - Standard gradient descent (as always, other variants possible)

A Fixed Target For The Loss

- This is the loss used in the deep Q network paper

- Note that in the previous error both the target and output are updated every iteration
  - Target: $r_t + \gamma\, Q_\Theta(s_{t+1}, \arg\max_{a\_t+1} Q_\Theta(s_{t+1}, a_{t+1}))$
  - Output: $Q_\Theta(s_t, a_t)$

- It was observed that training stability improved if fixed targets are used
  - The output network was updated every iteration but
  - The target network was held fixed for T iterations then updated
  - Using Θ^- to indicated the parameters of the target network
  - Target: $r_t + \gamma\, Q_{\Theta^-}(s_{t+1}, \arg\max_{a\_t+1} Q_{\Theta^-}(s_{t+1}, a_{t+1}))$

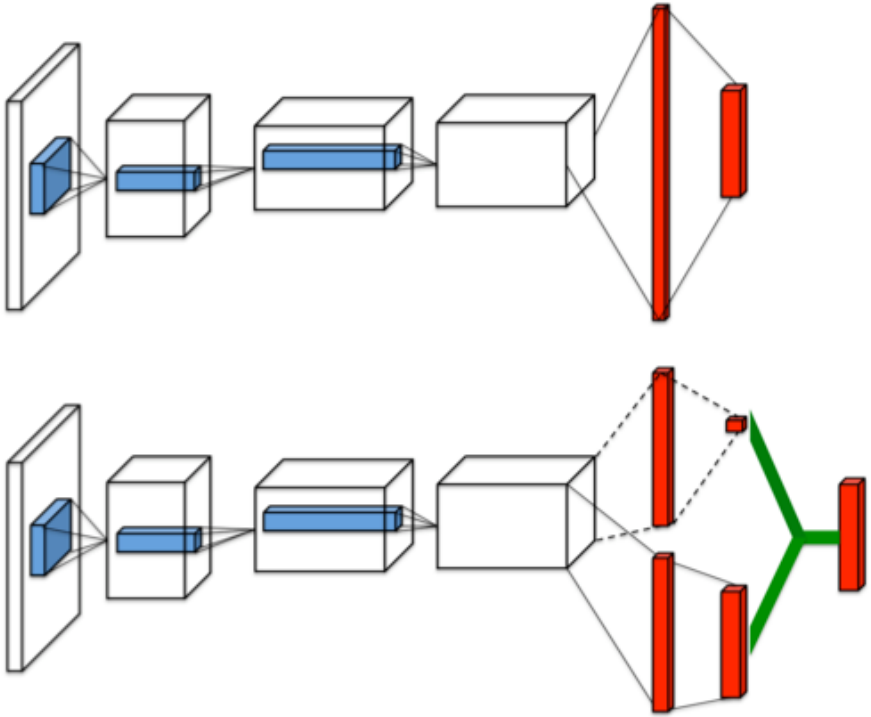
Double DQN Target For The Loss

- Another modification to the loss

- Fixed targets can overestimate the Q values when noisy
  - To prevent this can use 2 networks, 1 for the target and 1 for the output
  - Training is improved for some applications
  - In practice these can be combined into 1 network with different updates as below
  - Target: $r_t + \gamma Q_{\Theta^-}(s_{t+1}, \arg\max_{a\_t+1} Q_{\Theta}(s_{t+1}, a_{t+1}))$

Dueling DQN Network Modifications

- An improvement to the network structure for estimating Q values
  - Common tail and body
  - Then 2 heads for estimating V and A
    - 1 for estimating value V(s), a single value
    - 1 for estimating advantage A(s, a), a vector of length |A|
  - Followed by their combination to compute Q
    - $Q(s, a) = V(s) + A(s, a) - 1/|A| \Sigma_{a'} A(s, a')$
    - So this combination is really Q off by a scale factor but that's ok
  - This can improve stability



Example: Atari 2600 Games

- The following description and results are from the original Nature paper
  - This does not include all of the subsequent enhancements described on the previous pages
  - Regardless, it serves as a starting point for thinking about applying Q learning to Atari
- Environment
  - Atari 2600 simulator
- Input state s
  - Pre processing takes max over current and previous frame to catch flickering objects, extracts the luminance channel and rescales it to 84 x 84, then stacks the most recent 4 frames together
  - Result is 4 x 84 x 84 input that represents state
  - Note that the stack of 4 frames encodes motion
  - Inputs are generated once for every 4 frames to save on computation

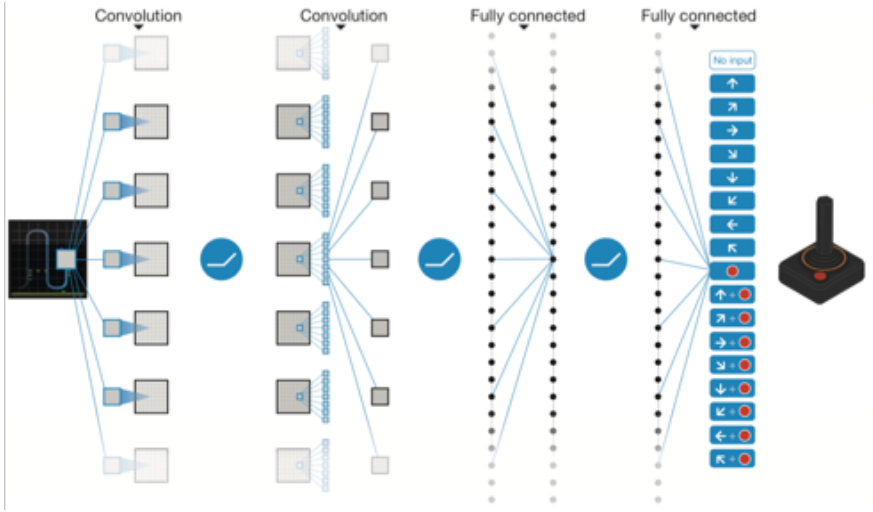


Figure from https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Example: Atari 2600 Games

- The network maps input state s to Q(s, a)
  - CNN layer with 32 x 4 x 8 x 8 / 4 filters and ReLU
  - CNN layer with 64 x 32 x 4 x 4 / 2 filters and ReLU
  - CNN layer with 64 x 64 x 3 x 3 / 1 filters and ReLU
  - Flatten
  - Fully connected layer with 512 outputs and ReLU
  - Fully connected layer with |A| outputs each corresponding to the Q value for a specific action
  - $4 \leq |A| \leq 18$ depending on the game
- Reward r
  - Score change, limited to {-1, 0, 1}
  - Discount factor $\gamma = 0.99$
- Loss
  - Fixed target loss

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter C from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ϵ in ϵ-greedy exploration.
final exploration	0.1	Final value of ϵ in ϵ-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ϵ is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

- Google DeepMind's deep Q-learning playing Atari Breakout
  - https://www.youtube.com/watch?v=V1eYniJ0Rnk

Figure from https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Example: Atari 2600 Games

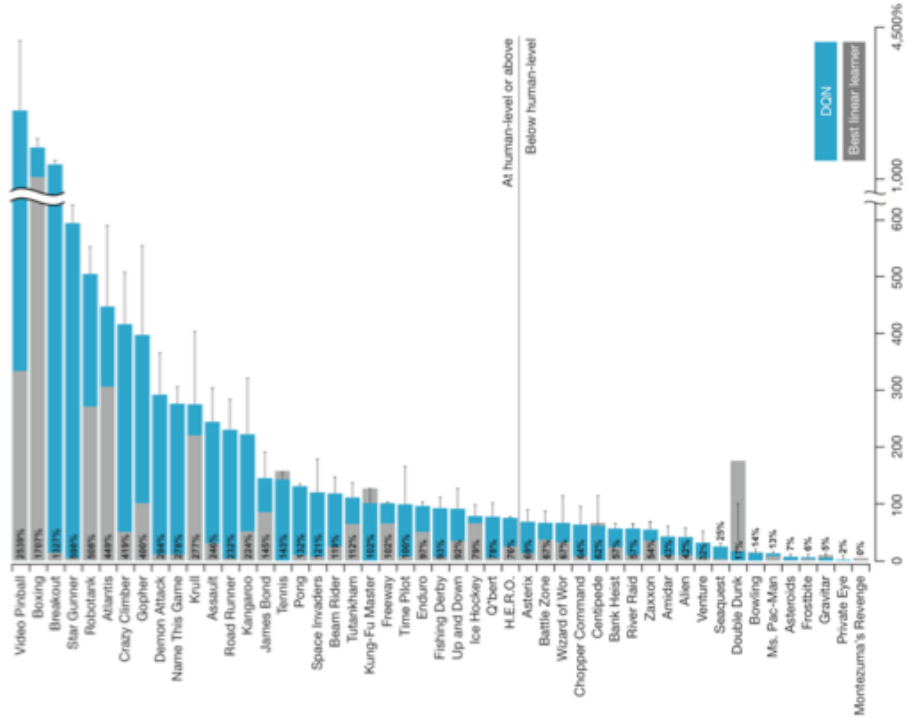


Figure from https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Decision Making – Policy Based

Strategy

- Core idea
  - The expected reward for following policy π is $J(\pi) = \int_\tau Pr(\tau \mid a \sim \pi)\, R_{0:}(\tau) = E[R_{0:}(\tau) \mid \tau \sim \pi]$
  - $\pi^* = \arg\max_\pi J(\pi)$ is the optimal policy that maximizes the expected reward
  - Approximate the optimal policy π^* with a xNN based policy π_Θ with parameters Θ
  - Update the policy π_Θ during training to increase the expected reward

- The basic REINFORCE algorithm
  - Run the policy and create samples (trajectories τ)
  - Estimate the policy gradient $\nabla_\Theta J(\pi_\Theta)$
  - Update the parameters Θ via gradient ascent or variant $\Theta_{k+1} = \Theta_k + \alpha \nabla_\Theta J(\pi_\Theta)$
  - Repeat

- Question: what is the policy gradient $\nabla_\Theta J(\pi_\Theta)$ and how do you estimate it?

Policy Gradients

- The policy gradient is the change in the expected reward $J(\pi_\Theta)$ for a small change in Θ evaluated at the current Θ
  - $\nabla_\Theta J(\pi_\Theta) = E[\nabla_\Theta \log p_P(\tau)\, R_{0:}(\tau) \mid \tau \sim \pi_\Theta]$
    $= E[(\Sigma_{t=0:T-1} \nabla_\Theta \log p_{\pi\_\Theta}(a_t \mid s_t))\, R_{0:}(\tau) \mid \tau \sim \pi_\Theta]$
    $= E[(\Sigma_{t=0:T-1} \nabla_\Theta \log p_{\pi\_\Theta}(a_t \mid s_t))\, (\Sigma_{i=0:T-1} R(s_i, a_i)) \mid \tau \sim \pi_\Theta]$
    $\approx (1/N)\, \Sigma_{n=0:N-1}((\Sigma_{t=0:T-1} \nabla_\Theta \log p_{\pi\_\Theta}(a_{n,t} \mid s_{n,t}))\, (\Sigma_{i=0:T-1} R(s_{n,i}, a_{n,i})))$
  - For the derivation of the above policy gradient see the appendix
  - Need to compute $\nabla_\Theta \log p_{\pi\_\Theta}(a_t \mid s_t)$

- $\nabla_\Theta \log p_{\pi\_\Theta}(a_t \mid s_t)$ computation strategy
  - For categorical policies the network output is an $|A| \times 1$ vector $p_{\pi\_\Theta}(a_t \mid s_t)$
    - Take the log of this, then apply auto diff
  - For diagonal Gaussian policies the network output is a mean action vector
    - Converted to a continuous action via the previously described procedure
    - Take the log of this, then apply auto diff

Policy Gradients

- A key issue is the variance in the estimation of $\nabla_\Theta J(\pi_\Theta)$
  - Subsequent slides will look at methods for reducing the variance
  - Most of the described strategies work by modifying the $R_{0:}(\tau)$ term
  - For the justification for these modifications see the EGLP lemma in the appendix

Reward To Go

- The $R_{0:}(\tau)$ term in the policy gradient weights all actions based on the total reward
  - But it doesn't make sense to reinforce future actions based on previous rewards
  - The reward to go modifies the objective to only include rewards from the present or future
  - This is done by replacing $R_{0:}(\tau)$ with $R_{t:}(\tau)$

- Reward to go policy gradient
  - $\nabla_\Theta J(\pi_\Theta) = E[\nabla_\Theta \log p_P(\tau)\, R_{t:}(\tau) \mid \tau \sim \pi_\Theta]$
    $= E[\Sigma_{t=0:T-1} \nabla_\Theta \log p_{\pi\_\Theta}(a_t \mid s_t)\, \Sigma_{i=t:T-1} R(s_i, a_i) \mid \tau \sim \pi_\Theta]$

Baseline Removal

- A further modification to the reward to go $R_{t:}(\tau)$ is the removal of a baseline
  - When the baseline is chosen as the value function the expected average return is removed from the reward to weight the policy updates based on the improvement relative to the average reward
  - This is appealing
  - The resulting algorithm has 2 parts, an actor (the policy) and a critic (the value function) and is referred to as an actor critic algorithm

- Policy gradient with value function baseline removal
  - $\nabla_\Theta J(\pi_\Theta) = E[\nabla_\Theta \log p_P(\tau) (R_{t:}(\tau) - V^{\pi\_\Theta}(s_t)) \mid \tau \sim \pi_\Theta]$
    $= E[\Sigma_{t=0:T-1} \nabla_\Theta \log p_{\pi\_\Theta}(a_t \mid s_t) (\Sigma_{i=t:T-1} R(s_i, a_i) - V^{\pi\_\Theta}(s_t)) \mid \tau \sim \pi_\Theta]$

- Removing the baseline value function requires an estimate of the value function
  - This is typically done using another xNN, $V_\phi(s_t)$, that approximates $V^{\pi\_\Theta}(s_t)$
  - The value function xNN is trained at the same time via the following objective
  - $\phi = \arg\min_\phi E[(V_\phi(s_t) - R_{t:}(\tau))^2 \mid \tau \sim \pi_\Theta]$

Advantage Function

- Replacing the reward term with the Q function results in the advantage function being used for the weighting term

- Policy gradient with advantage function
  - $\nabla_\Theta J(\pi_\Theta) = E[\nabla_\Theta \log p_P(\tau) (Q^{\pi\_\Theta}(s_t, a_t) - V^{\pi\_\Theta}(s_t)) \mid \tau \sim \pi_\Theta]$
    $= E[\Sigma_{t=0:T-1} \nabla_\Theta \log p_{\pi\_\Theta}(a_t \mid s_t) A^{\pi\_\Theta}(s_t, a_t) \mid \tau \sim \pi_\Theta]$
  - Note that $Q^{\pi\_\Theta}(s_t, a_t) = r_t + \gamma V^{\pi\_\Theta}(s_{t+1})$

- As before, $V^{\pi\_\Theta}(s_t)$ is estimated via another xNN, $V_\phi(s_t)$

Vanilla Policy Gradient Algorithm

- Initialize
  - Policy parameters Θ_0 and value function parameters ϕ_0
- Loop for k = 0 to ...
  - Collect a set of N trajectories $\{\tau_n\}_{n=0:N-1}$ by running the policy $p_{\pi\_\Theta}(a_t \mid s_t)$ in the environment
  - Compute the rewards to go $R_{t:}(\tau_n)$
  - Compute the advantage function from the individual rewards and estimated value function
    - $A^{\pi\_\Theta k}(s_{n,t}, a_{n,t}) = R(s_{n,t}, a_{n,t}) + \gamma V_{\phi k}(s_{t+1}) - V_{\phi k}(s_t)$
    - This is a 0 step estimate, multi step and generalized advantage estimates are also possible
  - Estimate the policy gradient
    - $\nabla_{\Theta k} J(\pi_{\Theta k}) = (1/N) \Sigma_{n=0:N-1} (\Sigma_{t=0:T-1} \nabla_{\Theta k} \log p_{\pi\_\Theta k}(a_{n,t} \mid s_{n,t}) A^{\pi\_\Theta k}(s_{n,t}, a_{n,t}))$
  - Update the policy parameters via stochastic gradient ascent or variant
    - $\Theta_{k+1} = \Theta_k + \alpha \nabla_{\Theta k} J(\pi_{\Theta k})$
  - Update the value function via gradient descent or variant based on the MSE
    - $\phi_{k+1} = \arg\min_{\phi} (1/N) \Sigma_{n=0:N-1} (V_{\phi}(s_t) - R_{t:}(\tau_n))^2$

Decision Making – Model Based

Strategy

- Basic idea
  - Learn a model
    - For a MDP this means learning the state transition function and reward function
    - Note that it may be given or need to be estimated
  - Use the model to simulate various outcomes
    - An alternative use of this is to use the model to generate data for a model free based method
  - Choose an action that results in the best outcome
    - Maybe directly from the simulations
    - Maybe indirectly via a policy trained from the simulations

- When action selection is done via a tree view of states and transitions
  - You need a method for deciding which nodes to assign a value to (what parts of the tree to explore)
  - You need a method for valuing those nodes
  - You need a method to select the next node based on the assigned values

How To Learn A Model

- A few possibilities
  - Supervised learning
  - Self play
  - The model is given

- Note
  - An inaccurate model can lead to sub optimal decisions

An Example Model Learning Algorithm

- Initialization
  - Run a data collection policy π to collect an initial dataset $\{(s, a, s')_t\}_{t=0:T-1}$
    - s' is used to indicate the next state
    - Time can be continuous so t+1 wasn't used
  - Learn a state transition model f(s, a) via minimizing $\Sigma_{t=0:T-1} ||f(s_t, a_t) - s_t'||^2$

- Loop indefinitely
  - Loop for N steps
    - Plan by using f(s, a) to look into the future and select an optimal action
    - Execute the action and observe the resulting state s'
      - This may or may not be the predicted next state based on the model accuracy
    - Append the new piece of data (s, a, s') to the dataset
  - Update the state transition model f(s, a) via minimizing $\Sigma ||f(s_t, a_t) - s_t'||^2$ using the updated dataset

Viewing Games As Trees

- Brushing aside a lot of important formalism of game theory (please consult a reference if interested) …

- You can view a game as a tree
  - Each node in the tree is a state in the game
    - The root node is initial state
    - A leaf node is a terminal state
    - Nodes (states) have values
  - A move is a transition from a node to a child node
    - The number of child nodes is the number of possible moves for the state

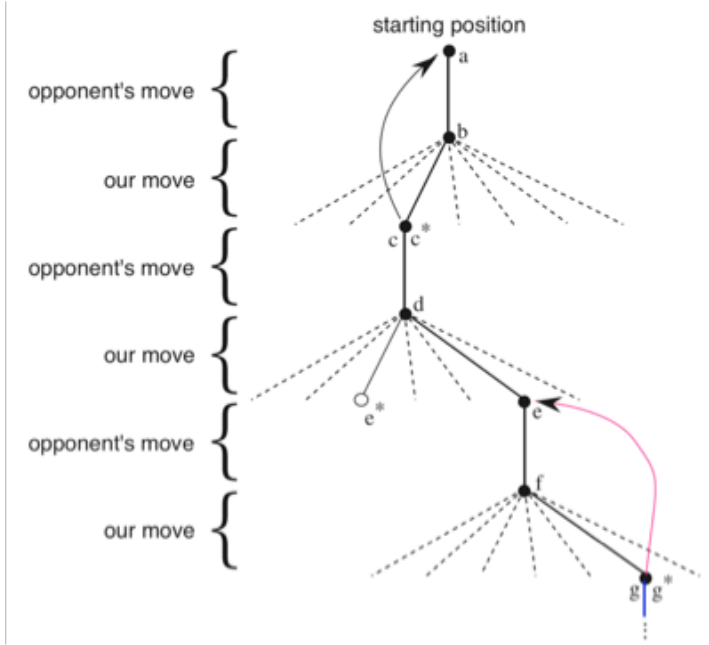


Figure from http://incompleteideas.net/book/bookdraft2017nov5.pdf

Policy From Value Assignment And Tree Search

- What is the optimal (move) child node to select?
  - To answer this question you need to have a value assigned to each child node of the root node
  - The optimal move would then be selecting the child node with the highest value
  - To assign a value to a child node of the root node you need to understand the sub tree under that child
- The problem is that the sub tree becomes too large to fully expand in a brute force manner
  - So you have to decide what parts of the tree you want to explore to help you decide what child node to select
  - As such, the tree search method is a key

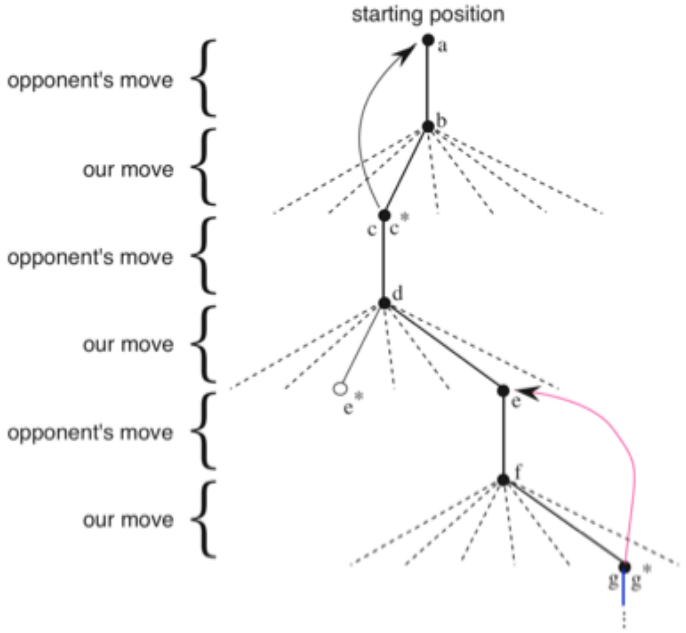


Figure from http://incompleteideas.net/book/bookdraft2017nov5.pdf

Fixed Depth Expansion + Value Estimation

- Consider a minimax strategy for selecting the next move (best child node) in a zero sum game
  - Assume your opponent plays optimally
  - You choose the strategy that maximizes your reward assuming that your opponent plays to minimize your reward (maximize their reward)

- To solve this you would like to expand the full game tree from root to all leaves
  - But since you can't, you instead fully expand the tree to a fixed depth corresponding to D moves
  - Then use Monte Carlo rollouts or other methods to estimate the value of all nodes at depth D, effectively treating them as leaves
  - Then apply minimax strategy from here

Alpha Beta Pruning To Reduce The Search

- Alpha beta pruning is a depth first method that decreases the number of nodes searched by the minimax algorithm via pruning the tree
  - Sub trees are pruned if a better move is available (a player would never choose that sub tree)
  - This implies that values for leaves are known (or estimated)
  - Alpha = the min score that the maximizing player is assured of
  - Beta = the max score that the minimizing player is assured of

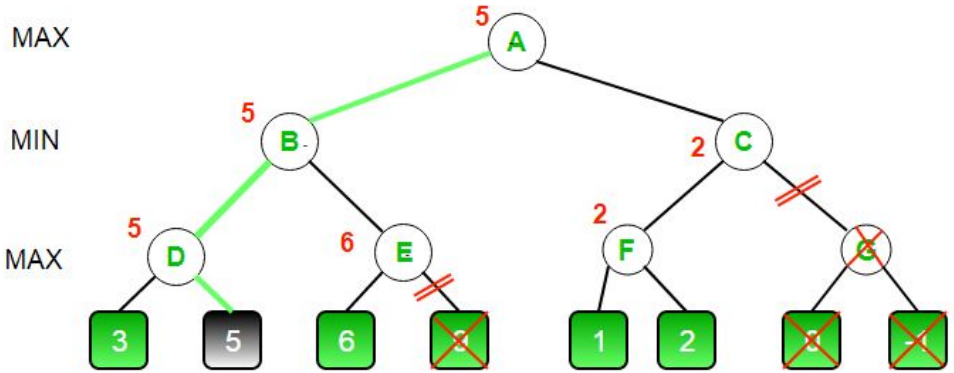


Figure from https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Monte Carlo Tree Search

- A different (not minimax) tree search strategy

- Basic idea
  - Use MCTS to assign to a subset of all possible nodes
    - A total simulation reward Q
    - A total number of visits N
  - Note
    - All possible nodes includes all possible paths from the root node to all termination states
    - The subset always includes all child nodes of the root node (all possible moves from the current state) and typically many more child nodes of those nodes
  - Select a move corresponding to a child node of the root node
    - Typically the most visited child
    - That corresponds to the most explored state
    - Noting that MCTS explores more promising states more of the time (will see soon)

Monte Carlo Tree Search

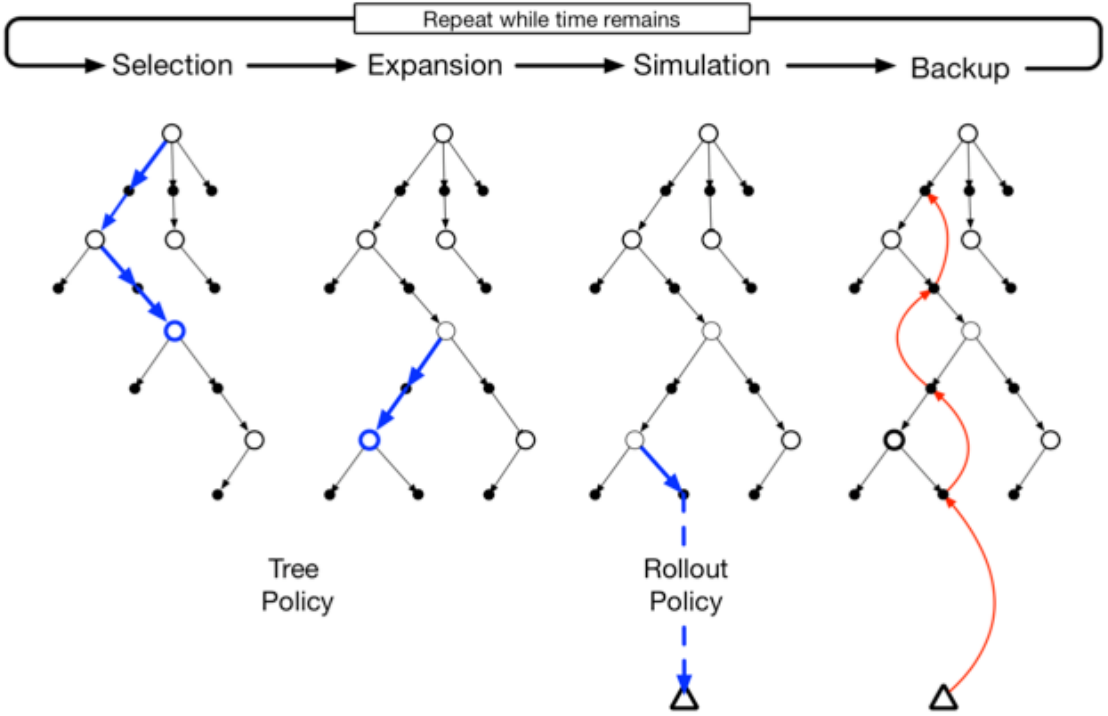


Figure from http://incompleteideas.net/book/bookdraft2017nov5.pdf

Monte Carlo Tree Search

- Definition: (non) fully expanded node
  - A node in which all of it's children have (not) been visited
- Step 1: Select an unvisited node
  - Start at the root
  - Search for a non fully expanded node (nota bene)
    - If the current node is fully expanded then choose it's child with the highest UCT and repeat the search from there
      - UCT(node, parent) = Q(node)/N(node) + c*sqrt(log(N(parent))/N(node))
      - 1st term favors exploitation
      - 2nd term favors exploration (prevents collapse to greedy)
      - c is a constant to balance between the exploitation and exploration
      - There are variants of this
    - If the current node is not fully expanded then stop (you found a non fully expanded node)
  - Select an unvisited child of the non fully expanded node

Monte Carlo Tree Search

- Step 2: Simulate the selected unvisited node to get a value
  - Need a policy to chose moves (can be random or better) or another method to estimate a value
  - Mark this node as visited
  - A lower node through which the simulation passes is still considered unvisited
  - If all children of the parent are visited then mark the parent as fully expanded
- Step 3: Back propagate (not calculus) the value from the selected node to the root and update encountered node statistics along the way
  - Q(node) = total simulation reward (e.g., add new value to current value of node)
  - N(node) = total number of visits (e.g., add 1 to the current value of the node)
- Repeat steps 1 – 3 until sufficient statistics are built up / a time limit is reached
  - Will have Q(node) and N(node) values for many nodes in the tree including all children of the root node (the current state)

Monte Carlo Tree Search

- Step 4: Select a move
  - Effectively, creating a policy from the tree
  - Effectively, choosing a child of the root
  - Typically the child with the highest number of visits
  - This is the most explored move
  - MCTS biases exploration towards more favorable moves
- Computation note
  - It's likely possible to reuse parts of the previously created tree for the next move

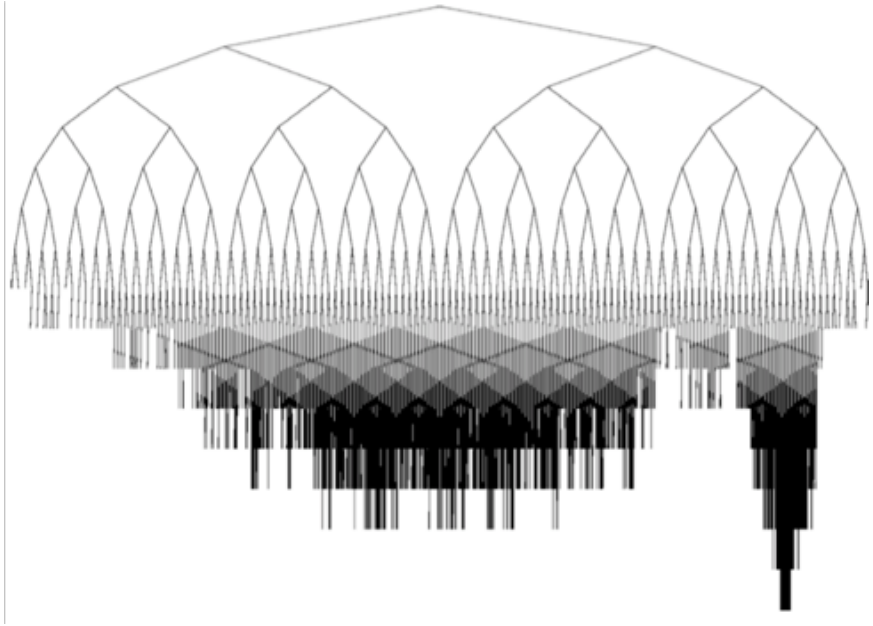


Figure from https://ieeexplore.ieee.org/document/6145622

Example: Go

- Problem
  - There's an optimal value for every board state that represents the outcome of the game assuming perfect play
  - Given ~ b = 250 possible moves per turn (breadth) and ~ d = 150 turns (depth) in a typical game an exhaustive search of b^d moves to determine the value of a state is infeasible
  - So you need a way to reduce the breadth and depth of the tree

- Strategy
  - To reduce the implicit breadth of the tree use a policy xNN to predict a pmf over the possible moves from an input board state and select the highest probability moves for exploration, effectively making the tree less wide
  - To reduce the implicit depth of the tree use a value xNN to predict the value (game outcome) from an input board state, effectively making the tree less deep
  - Combine the policy and value xNNs with MCTS for training and game play

Figure from https://www.wired.co.uk/article/alphago-deepmind-google-wins-lee-sedol

Example: Go

- There were many prior approaches to playing Go that achieved varying results but included important components of an overall effective strategy

- The Go slides will focus on 2 that achieved superhuman levels of play
  - AlphaGo (also known as AlphaGo Fan)
  - AlphaGo Lee
  - AlphaGo Master
  - AlphaGo Zero
  - Alpha Zero

- Descriptions are approximate and many details are omitted, please see the original papers for full information

Example: Go

AlphaGo

- AlphaGo summary
  - Training step 1: supervised learning of 2 policy networks to predict expert human moves
    - Output 1: The SL policy network – a 13 layer CNN that maps board states to expert move pmfs
    - Output 2: The fast rollout policy network – a less accurate but faster neural network that maps board states to expert move pmfs
  - Training step 2: improvement of the policy network via reinforcement learning
    - Output: The RL policy network – an improvement of the SL policy network trained via iterative self play and using a policy gradient for the update
  - Training step 3: estimation of a value network via supervised learning
    - Output: The value network – a 13 layer CNN that maps the board state to a win / loss prediction for the game
  - Game play: MCTS enhanced via policy and value networks to select actions
    - The RL policy network is used to bias the UCT algorithm for node selection towards exploring better moves, implicitly reducing the tree breadth
    - The value network and fast rollout policy networks are used to value board states, implicitly reducing the tree depth
    - Select a move based on the child node of the root node with the highest number of visits, this is the most explored move and MCTS biases exploration towards more favorable moves

Example: Go

AlphaGo

- Training step 1: supervised learning of 2 policy networks to predict expert human moves
  - The SL policy network
    - The input is an encoded representation of the board state using a number of feature maps; some of the feature maps include needed information, some of the feature maps include human selected heuristics
    - The network is a 13 layer CNN
    - The output is a 19x19 grid representing a pmf for each possible move
    - Training is done via stochastic gradient descent to map input states to human moves using the KGS Go Server database of millions of human positions and moves (state action pairs) for data
    - Achieves mid 50% accuracy in 3 ms per input
  - The fast rollout policy network
    - The input is a different encoded representation of the board state; as before, some of the encoding is needed information, some of the encoding is human selected heuristics
    - A smaller network is used
    - The output is a 19x19 grid representing a pmf for each possible move
    - Achieves mid 20% accuracy in 2μs per input

Example: Go

AlphaGo

- Training step 2: improvement of the policy network via reinforcement learning
  - The RL policy network
    - Same structure and initialized via the SL policy network
    - A pool of previous versions of the RL policy network is maintained
    - Self play against a randomly selected previous version to stabilize training is used
    - The reward function is +1 for winning and -1 for losing the game
    - A policy gradient algorithm with stochastic gradient ascent is used to update the weights to improve the policy

Example: Go

AlphaGo

- Training step 3: estimation of a value network via supervised learning
  - The value network
    - For input and labeled output data millions of board states were sampled from games, the game was played from the sampled state by the final RL policy network against itself and the +1 winning / -1 losing result was recorded (so the RL trained policy network was used to generate labeled data)
    - The network is a 13 layer CNN
    - The output is a single value representing the predicted value of the game given the input board state
    - Training is done via stochastic gradient descent minimizing a square error loss

Example: Go

AlphaGo

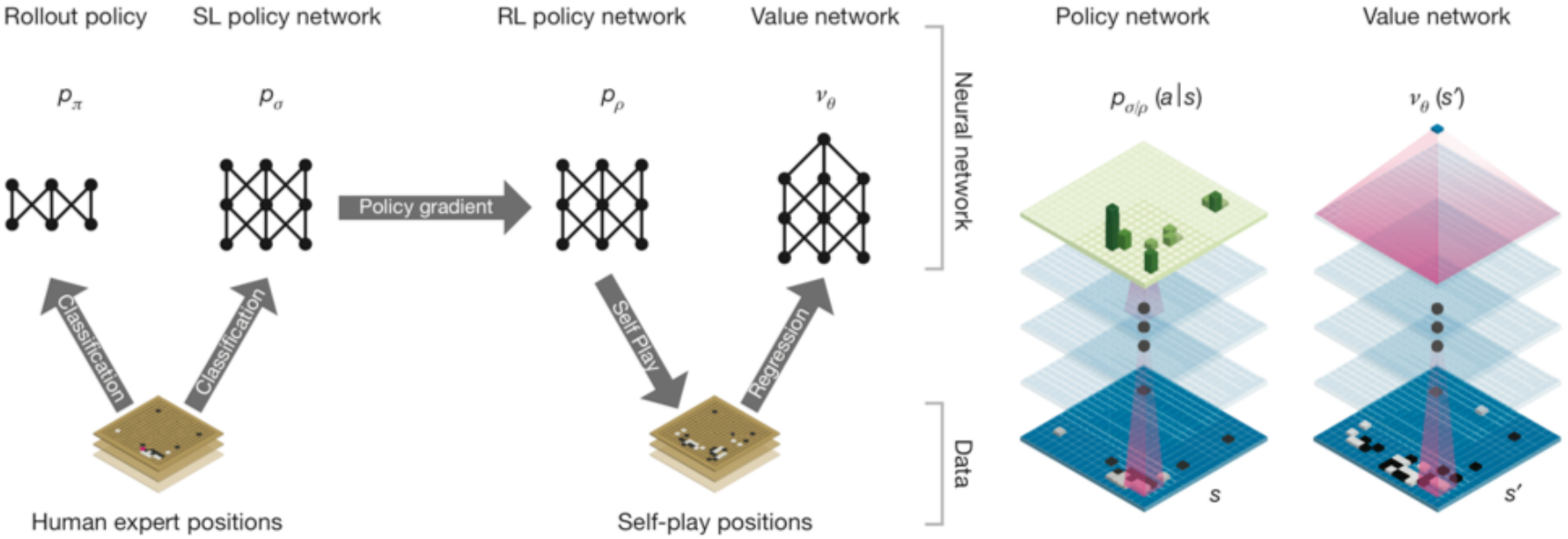


Figure from https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf

Example: Go

AlphaGo

- Game play: MCTS enhanced via policy and value networks to select actions
  - The following is an approximation / summary, please see the original paper for full details
  - Consider standard MCTS as a starting point
  - Use the RL policy network to modify the UCT used for node selection via biasing it towards stronger moves, effectively reducing the breadth of the tree
    - UCT(node, parent) = Q(node)/N(node) + c*Pr(parent, node)*sqrt(log(N(parent))/N(node))
    - Pr(parent, node) is the prior probability of the move from the RL policy network
  - Use the value network and fast rollout policy network to improve the node value assignment, effectively reducing the depth of the tree
    - The value at node S_L is a weighted sum of 2 parts
      - Part 1: z_L = result of the game using the fast rollout policy network
      - Part 2: $v_0(S_L)$ = output of value network
      - Sum: $V(S_L) = (1 - \alpha)*v_0(S_L) + \alpha*z_L$
  - Select a move based on the child node of the root node with the highest number of visits
    - This is the most explored move
    - MCTS biases exploration towards more favorable moves

Example: Go

AlphaGo

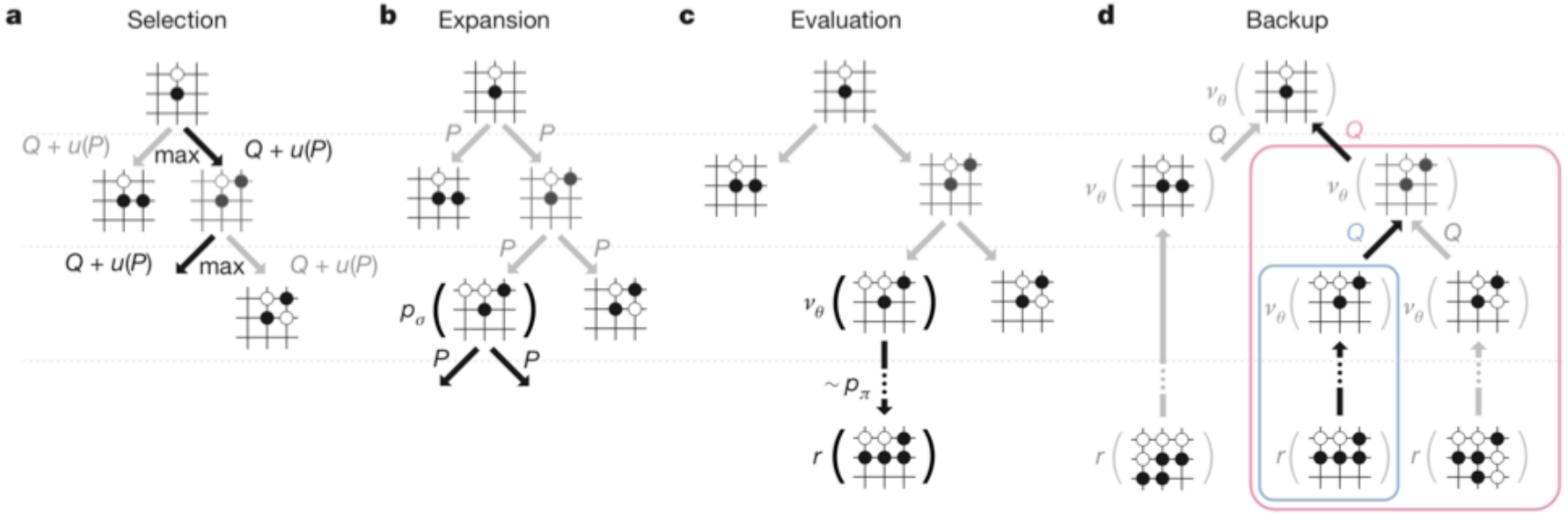


Figure from https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf

Example: Go

AlphaGo → AlphaGo Zero

- The results of AlphaGo were quite amazing
  - AlphaGo learned from humans to become pretty good
  - Improved itself via self play to become good
  - Then coupled it's good play with MCTS to become the best (better than the top human)

- But is it necessary to learn from humans to become super human in play? Or, is it possible for an algorithm to teach itself to play in a super human way?

Example: Go

AlphaGo Zero

- A goal of AlphaGo Zero
  - Remove human knowledge used in training AlphaGo
  - No human training games; only self play
  - No human heuristics in input feature maps; only white stones, black stones, history and turn info for input feature maps

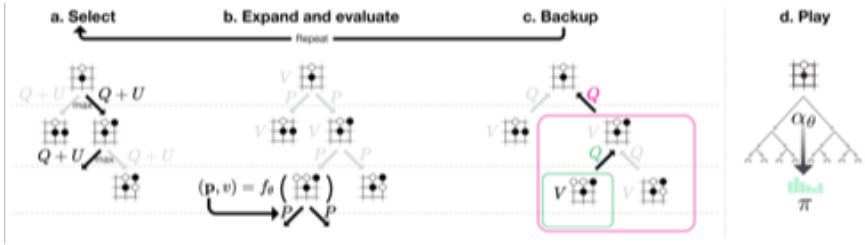
- Some additional key modifications relative to AlphaGo
  - A single network with policy and value heads
  - No Monte Carlo rollouts
  - MCTS based look ahead inside the training loop

Example: Go

AlphaGo Zero

- Consider a xNN that takes the board state as it's input and outputs a pmf of actions and a value of the present state: $(p_{\pi\_f}, V_f) = f_\Theta(s)$
  - f has parameters Θ and a ResNet V1 style structure with two 3x3 filters in the residual path, batch norm and ReLU
  - There were 2 versions of the shared portion of the network, 1 with 20 residual blocks and 1 with 40 residual blocks
  - The policy network has 2 additional layers: two CNN style 2D convolutions with 1x1 filters, batch norm, ReLU and a fully connected layer with $19^2 + 1 = 362$ outputs
  - The value network has 3 additional layers: 1 CNN style 2D convolution with 1x1 filters, batch norm, ReLU, a fully connected layer with 256 outputs and ReLU, a fully connected layer with 1 output and a tanh nonlinearity

- Training
  - Step 0: initialize the policy and value xNN f_Θ
  - Step 1: self play using MCTS enhanced with f_Θ
  - Step 2: xNN training to improve f_Θ
  - Repeat to Step 1
  - This can be thought of as a search based policy iteration



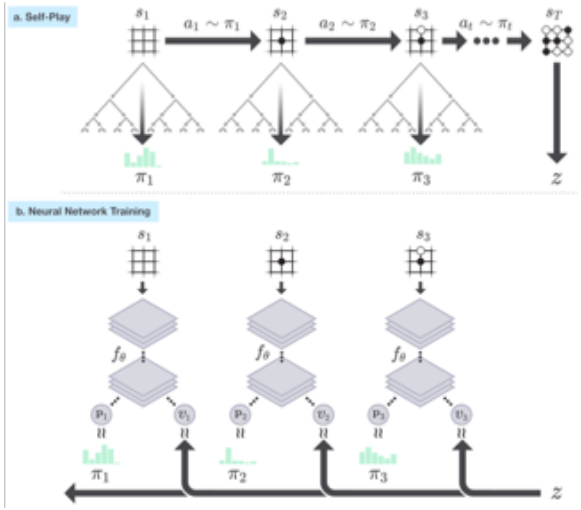
- Game play via MCTS enhanced with f_Θ

Figure from https://deepmind.com/blog/alphago-zero-learning-scratch/

Example: Go

AlphaGo Zero

- Step 1: self play using MCTS enhanced with f_Θ
  - AlphaGo Zero plays a game against itself
  - A pmf of moves p_{π_MCTS} is created via MCTS enhanced with the latest f_Θ
    - UCT is modified to include p_{π_f} in move selection for implicit breadth reduction
    - V_f is used for leaf scoring for implicit depth reduction
    - p_{π_MCTS} is proportional to visit count raised to a temperature parameter
  - The game is scored to determine the winner V_{MCTS}
  - Results are stored for all of the states and moves as (s_t, p_{π_MCTS}, V_{MCTS}) tuples
- Step 2: xNN training to improve f_Θ
  - For each board position s_t the xNN parameters Θ are updated to minimize a 3 part error term via SGD with momentum
    - Part 1: cross entropy between p_{π_f} and p_{π_MCTS}
    - Part 2: mean square error between V_f and V_{MCTS}
    - Part 3: scaled L_2 regularization term on the parameters Θ
  - MCTS is effectively creating an improved policy and value target to use to update the xNN weights

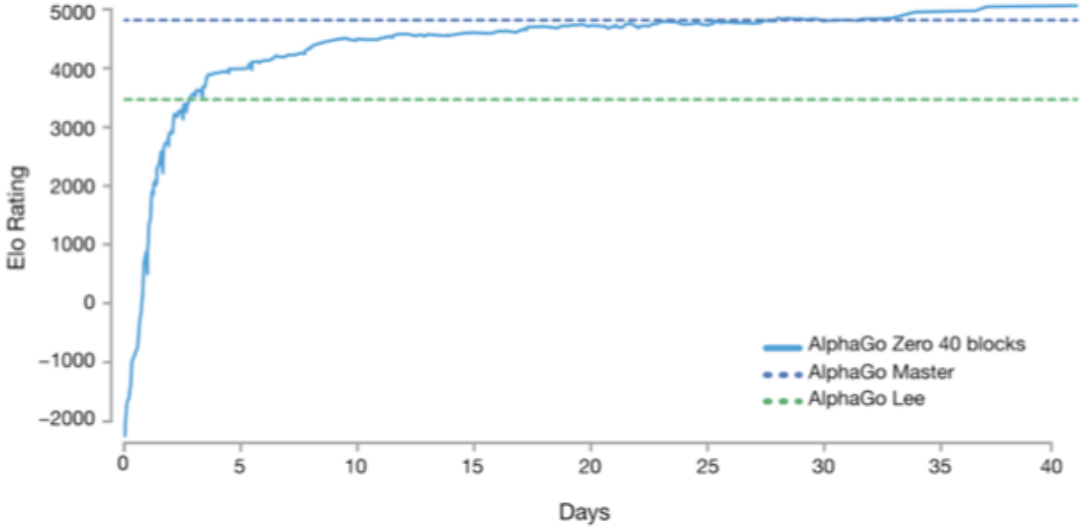


Initialize network parameters
Loop
  Step 1 self play
  Step 2 policy and value improvement

Figure from https://deepmind.com/blog/alphago-zero-learning-scratch/

Example: Go

AlphaGo Zero



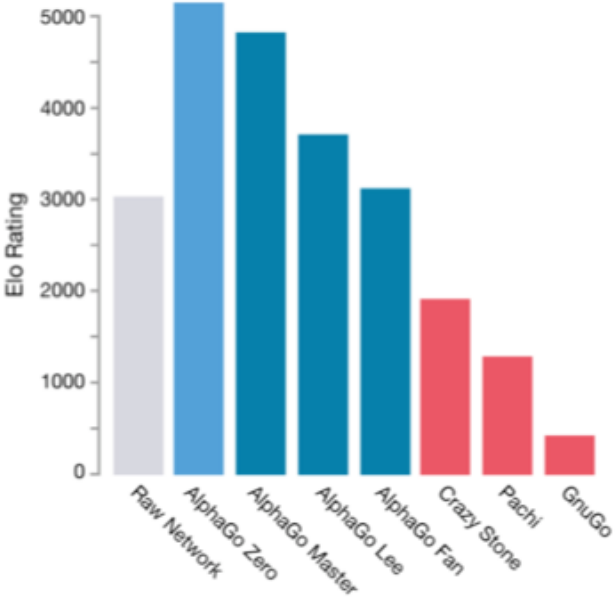


Figure from https://deepmind.com/blog/alphago-zero-learning-scratch/

Example: Chess

Alpha Zero

- AlphaGo Zero used a relatively general training strategy to learn a policy for a turn based 2 player zero sum game without human knowledge

- Alpha Zero makes a few additional modifications to the AlphaGo Zero training strategy then applies the same training strategy to multiple games and achieves (arguably) the best performance in each relative to all humans and other computer programs
  - Go
  - Chess
  - Shogi

- A summary of training strategy modifications relative to AlphaGo Zero
  - Value target based on expected outcome vs win / loss
  - No exploitation of rotation and reflection invariance
  - Continual updates of the single neural network vs maintaining a best player

Example: Chess

Alpha Zero

- An example of how MCTS focuses as a function of the number of simulations
- In general, Alpha Zero evaluates many fewer positions than programs like Stockfish that use Alpha Beta pruning but the positions that it evaluates are of much higher quality as a result of the policy and value network
- Sort of like a super human
  - High quality move evaluations like a person would do
  - Just lots more and deeper

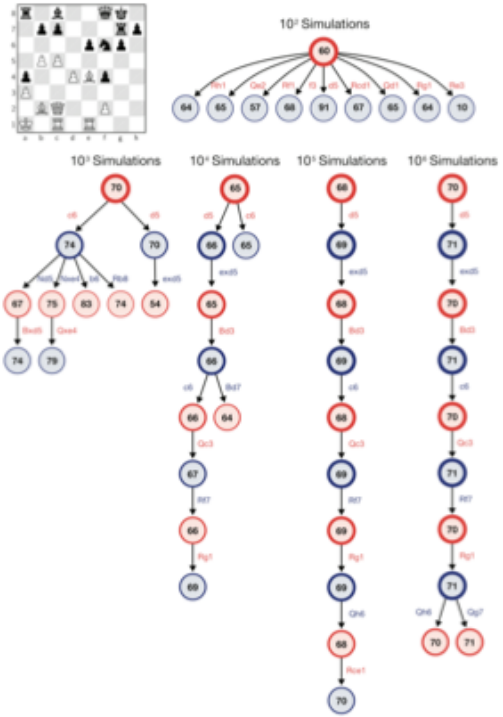


Figure from https://deepmind.com/research/publications/general-reinforcement-learning-algorithm-masters-chess-shogi-and-go-through-self-play/

Example: Chess

Alpha Zero

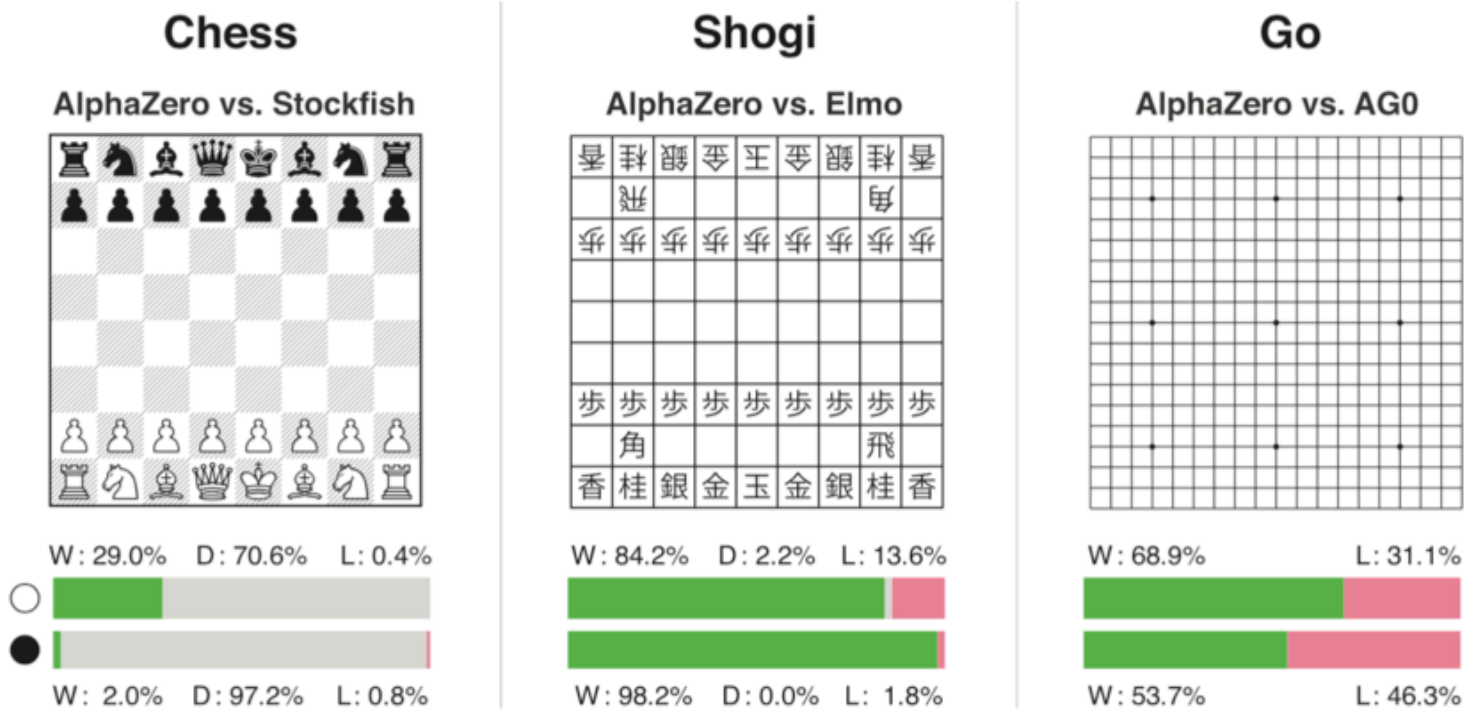


Figure from https://deepmind.com/research/publications/general-reinforcement-learning-algorithm-masters-chess-shogi-and-go-through-self-play/

Example: Chess

Alpha Zero

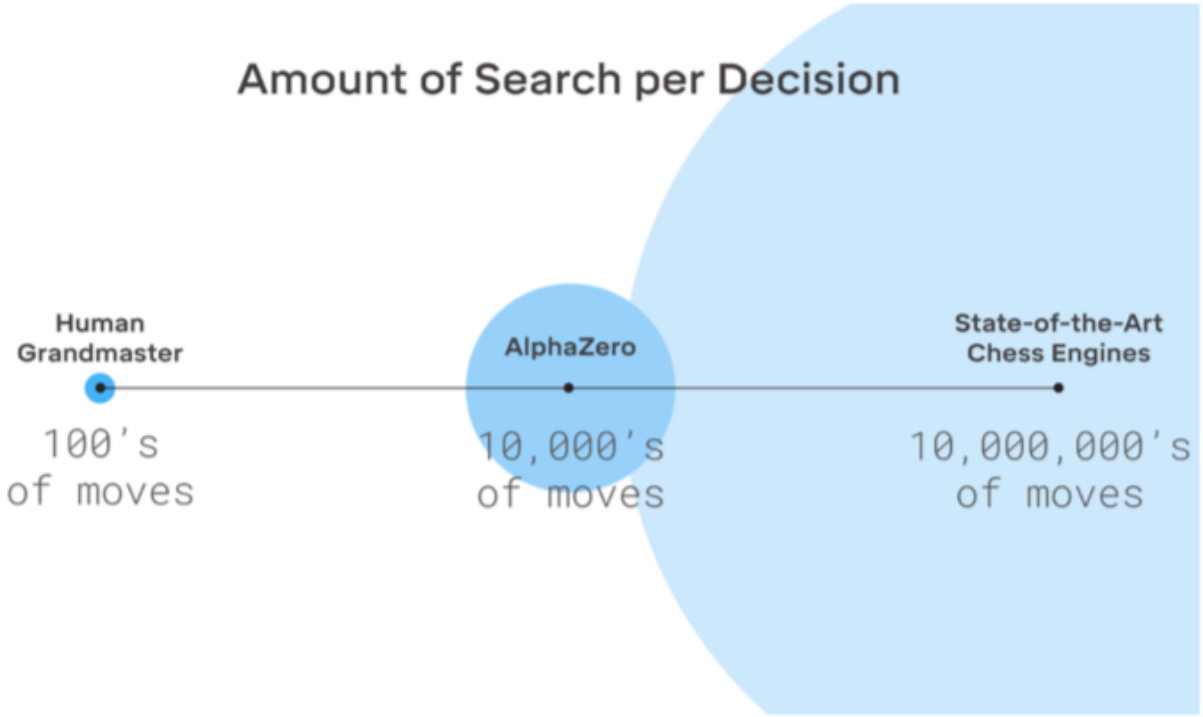


Figure from https://deeplearning.mit.edu

Example: Rubik's Cube

- Autodidactic iteration trains a value and policy network
  - Start from a solved cube and an initialized network that predictions values and move pmfs
  - Generate training samples via 90 degree moves from the set of 12 valid moves {F, F', B, B', L, L', R, R', U, U', D, D'} where the letter indicates the face and no prime / prime indicates the direction
  - For each of the training samples generate all of it's children via making all possible moves
  - For each child determine the value and use the maximum value and associated move to improve the network parameters

- MCTS used the trained value and policy networks to implicitly reduce breadth and depth

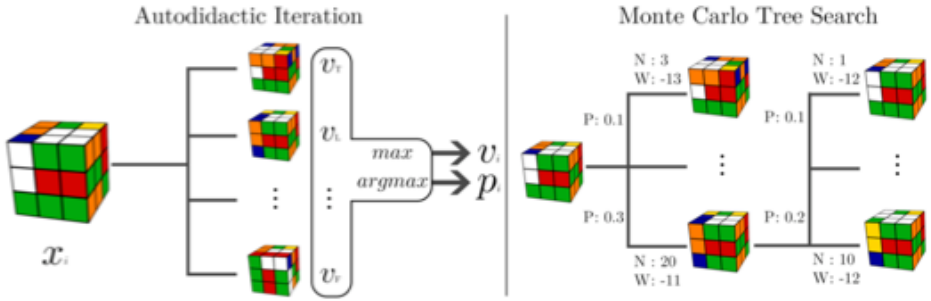
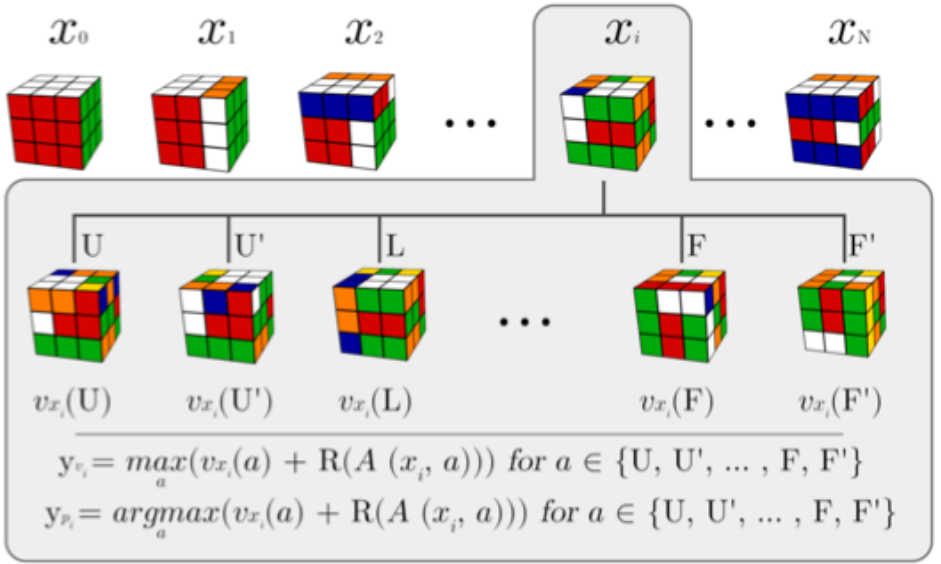


Figure from https://arxiv.org/abs/1805.07470

Example: Rubik's Cube

- Autodidactic iteration trains a value and policy network
  - Start from a solved cube and an initialized network that predictions values and move pmfs
  - Generate training samples via 90 degree moves from the set of 12 valid moves {F, F', B, B', L, L', R, R', U, U', D, D'} where the letter indicates the face and no prime / prime indicates the direction
  - For each of the training samples generate all of it's children via making all possible moves
  - For each child determine the value and use the maximum value and associated move to improve the network parameters



- MCTS used the trained value and policy networks to implicitly reduce breadth and depth

Figure from https://arxiv.org/abs/1805.07470

Appendix

Derivation Of Policy Gradient

- OpenAI spinning up part 3: policy optimization
  - https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

Derivation Of EGLP Lemma

- OpenAI spinning up part 3: policy optimization
  - https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

References

General

- Reinforcement learning: an introduction
  - http://incompleteideas.net/book/bookdraft2017nov5.pdf
- Spinning up in deep RL
  - https://spinningup.openai.com/en/latest/
- UC Berkeley CS 294-112 Deep reinforcement learning
  - http://rail.eecs.berkeley.edu/deeprlcourse/
- David Silver talks
  - http://www0.cs.ucl.ac.uk/staff/d.silver/web/Talks.html
- Deep reinforcement learning course
  - https://simoninithomas.github.io/Deep_reinforcement_learning_Course/
  - https://github.com/simoninithomas/Deep_reinforcement_learning_Course
- Policy gradient algorithms
  - https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html
- Deep reinforcement learning agents
  - https://github.com/awjuliani/DeepRL-Agents

Software

- Bullet Physics SDK
  - https://pypi.org/project/pybullet/
- Horizon: Facebook's open source applied reinforcement learning platform
  - https://arxiv.org/abs/1811.00260
- MuJoCo advanced physics simulation
  - http://www.mujoco.org
- OpenAI gym
  - https://github.com/openai/gym

Software

- Reinforcement learning in TensorFlow with TF-Agents (TF Dev Summit ‘19)
  - https://www.youtube.com/watch?v=-TTziY7EmUA
- TF-Agents: a library for reinforcement learning in TensorFlow
  - https://github.com/tensorflow/agents
  - https://github.com/tensorflow/agents/tree/master/tf_agents/colabs
- Dopamine
  - https://github.com/google/dopamine
- The DeepMind control suite and package
  - https://github.com/deepmind/dm_control
- Integration of TensorFlow and Unity 3D for a basketball game
  - https://medium.com/tensorflow/tf-jam-shooting-hoops-with-machine-learning-7a96e1236c32

Value Based Methods

- Human-level control through deep reinforcement learning
  - https://www.nature.com/articles/nature14236
- Deep reinforcement learning with double q-learning
  - https://arxiv.org/abs/1509.06461
- Prioritized experience replay
  - https://arxiv.org/abs/1511.05952
- Dueling network architectures for deep reinforcement learning
  - https://arxiv.org/abs/1511.06581
- Value iteration networks
  - https://arxiv.org/abs/1602.02867
- Rainbow: combining improvements in deep reinforcement learning
  - https://arxiv.org/abs/1710.02298

Value Based Methods

- Episodic memory deep Q-networks
  - https://arxiv.org/abs/1805.07603
- A theoretical analysis of deep Q-learning
  - https://arxiv.org/abs/1901.00137
- Google DeepMind's deep Q-learning playing Atari Breakout
  - https://www.youtube.com/watch?v=V1eYniJ0Rnk

Policy Based Methods

- Trust region policy optimization
  - https://arxiv.org/abs/1502.05477
- High-dimensional continuous control using generalized advantage estimation
  - https://arxiv.org/abs/1506.02438
- Learning continuous control policies by stochastic value gradients
  - https://arxiv.org/abs/1510.09142
- Asynchronous methods for deep reinforcement learning
  - https://arxiv.org/abs/1602.01783
- Proximal policy optimization algorithms
  - https://arxiv.org/abs/1707.06347
- Soft actor-critic algorithms and applications
  - https://arxiv.org/abs/1812.05905

Model Based Methods

- Thinking fast and slow with deep learning and tree search
  - https://arxiv.org/abs/1705.08439
- Efficient selectivity and backup operators in Monte-Carlo Tree search
  - https://hal.inria.fr/inria-00116992/document
- Monte Carlo Tree Search – beginners guide
  - https://int8.io/monte-carlo-tree-search-beginners-guide/
- A survey of Monte Carlo tree search methods
  - https://ieeexplore.ieee.org/document/6145622

AlphaGo

- Mastering the game of go with deep neural networks and tree search
  - https://www.nature.com/articles/nature16961
  - https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf
- AlphaGo: using machine learning to master the ancient game of go
  - https://blog.google/technology/ai/alphago-machine-learning-game-go/

AlphaGo Zero

- Mastering the game of go without human knowledge
  - https://www.nature.com/articles/nature242702017-silver.pdf
- AlphaGo Zero: learning from scratch
  - https://deepmind.com/blog/alphago-zero-learning-scratch/
- MiniGo: TensorFlow meets Andrew Jackson (TensorFlow meets)
  - https://www.youtube.com/watch?v=LRqlmjL3-n8
- Minigo: A minimalist go engine modeled after AlphaGo Zero, built on MuGo
  - https://github.com/tensorflow/minigo

Alpha Zero

- Mastering chess and shogi by self-play with a general reinforcement learning algorithm
  - https://arxiv.org/abs/1712.01815
- A general reinforcement learning algorithm that masters chess, shogi and go through self-play
  - https://deepmind.com/research/publications/general-reinforcement-learning-algorithm-masters-chess-shogi-and-go-through-self-play/
- AlphaZero: Shedding new light on the grand games of chess, shogi and go
  - https://deepmind.com/blog/alphazero-shedding-new-light-grand-games-chess-shogi-and-go/
- Deepmind AlphaZero - mastering games without human knowledge
  - https://www.youtube.com/watch?v=Wujy7OzvdJkELF
- OpenGo: an analysis and open reimplementation of AlphaZero
  - https://arxiv.org/abs/1902.04522
  - https://facebook.ai/developers/tools/elf-opengo
  - https://github.com/pytorch/ELF
- Leela chess zero
  - https://lczero.org

Alpha Zero

- AlphaZero: shedding new light on the grand games of chess, shogi and Go
  - https://www.youtube.com/watch?v=7L2sUGcOgh0
- agadmator's Chess Channel
  - https://www.youtube.com/user/AGADMATOR/videos
- Chess24
  - https://www.youtube.com/user/chess24media/videos
- AlphaZero's attacking chess
  - https://www.youtube.com/watch?v=nPexHaFL1uo

Rubik’s Cube

- Solving the Rubik's cube without human knowledge
  - https://arxiv.org/abs/1805.07470

DotA2

- OpenAI Five
  - https://blog.openai.com/openai-five/
- Emergent complexity via multi-agent competition
  - https://arxiv.org/abs/1710.03748
- The Dota 2 bot competition
  - https://ieeexplore.ieee.org/abstract/document/8356682
- Calculating optimal jungling routes in DOTA2 using neural networks and genetic algorithms
  - https://computing.derby.ac.uk/ojs/index.php/gb/article/view/14
- Skill-based differences in spatio-temporal team behaviour in defence of the ancients 2 (DotA 2)
  - https://ieeexplore.ieee.org/abstract/document/7048109