Deep Reinforcement Learning

Deep Learning: Bryan Pardo, Northwestern University, Fall 2020
Learning Types

• Supervised learning:
  • (Input, output) pairs of the function to be learned are given (e.g. image labeling)

• Unsupervised Learning:
  • No human labels provided (e.g. language modeling, image reconstruction)

• Reinforcement learning:
  • Reward or punishment for actions (winning or losing a game)
Reinforcement Learning

• Task
  • Learn how to behave to achieve a goal
  • Learn through experience from trial and error

• Examples
  • Game playing: The agent knows when it wins, but doesn’t know the appropriate action in each state along the way
  • Control: a traffic system can measure the delay of cars, but not know how to decrease it.
Basic RL Model

1. Observe state, $s_t$
2. Decide on an action, $a_t$
3. Perform action
4. Observe new state, $s_{t+1}$
5. Observe reward, $r_{t+1}$
6. Learn from experience
7. Repeat

Goal: Find a control policy that will maximize the observed rewards over the lifetime of the agent
A Cannonical Example: Gridworld

- States are grid cells
- 4 actions: N, S, E, W
- Reward for entering top right cell
- -0.01 for every other move
The Markov Property

• RL needs a set of states that are Markov
  • Everything you need to know to make a decision is included in the state
  • Not allowed to consult the past

• Rule-of-thumb
  • If you can calculate the reward function from the state without any additional information, you’re OK
We need some background

• Simple decision theory
• Markov Decision Processes
• Value functions
• Dynamic programming
Making Single Decisions

• Single decision to be made
  • Multiple discrete actions
  • Each action has an associated reward
• Goal is to maximize reward
  • Just pick the action with the largest reward
• State 0 has a value of 2
  • Reward from taking the best action
Markov Decision Processes

- We can generalize the previous example to multiple sequential decisions
  - Each decision affects subsequent decisions

- This is formally modeled by a Markov Decision Process (MDP)

Northwestern University, EECS 349, 2017
Markov Decision Processes

• Formally, a MDP is
  • A set of states, $S = \{s_1, s_2, \ldots, s_n\}$
  • A set of actions, $A = \{a_1, a_2, \ldots, a_m\}$
  • A reward function, $R: S \times A \times S \rightarrow \mathbb{R}$
  • A transition function, $T: S \times A \rightarrow S$
  • Sometimes $T: S \times A \rightarrow S$
  \[ P^a_{ij} = P(s_{t+1} = j | s_t = i, a_t = a) \]
  • A probability distribution over what the initial state will be: $\mu(s)$

• We want to learn a policy, $\pi: S \rightarrow A$
  • Maximize sum of rewards we see over our lifetime
Policies

- A policy $\pi(s)$ returns the action to take in state $s$.

- There are 3 policies for this MDP
  
  Policy 1: 0 $\rightarrow$ 1 $\rightarrow$ 3 $\rightarrow$ 5
  Policy 2: 0 $\rightarrow$ 1 $\rightarrow$ 4 $\rightarrow$ 5
  Policy 3: 0 $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 5
Comparing Policies

- Which policy is best?
- Order them by how much reward they see

Policy 1: \( 0 \rightarrow 1 \rightarrow 3 \rightarrow 5 = 1 + 1 + 1 = 3 \)
Policy 2: \( 0 \rightarrow 1 \rightarrow 4 \rightarrow 5 = 1 + 1 + 10 = 12 \)
Policy 3: \( 0 \rightarrow 2 \rightarrow 4 \rightarrow 5 = 2 - 1000 + 10 = -988 \)

![Diagram](image-url)
Value Functions

• For a given policy, we can associate a value with each state
  • How good is it to run policy $\pi$ from that state $s$?
  • This is the state value function, $V$

\[
\begin{align*}
V^1(s_0) &= 3 \\
V^2(s_0) &= 12 \\
V^3(s_0) &= -988 \\
V^1(s_1) &= 2 \\
V^2(s_1) &= 11 \\
V^1(s_3) &= 1 \\
V^3(s_2) &= -990 \\
V^2(s_4) &= 10 \\
V^3(s_4) &= 10
\end{align*}
\]

How do you tell which policy to follow from each state?

\[V^\pi(s) = R(s, \pi(s), s') + V^\pi(s')\]
Q Functions

- Define value without specifying the policy
  - It is the value of taking action A from state S and then performing optimally, thereafter

\[ Q(s, a) = R(s, a, s') + \max_{a'} Q(s', a') \]
Value Functions

• These can be extended to probabilistic actions
  (for when the results of an action are not certain, or when a policy is probabilistic)

\[ V^\pi(s) = \sum_{s'} P(s'| s, \pi(s))(R(s, \pi(s), s') + V^\pi(s')) \]

\[ Q(s, a) = \sum_{s'} P(s'| s, a)(R(s, a, s') + \max_{a'} Q(s', a')) \]
Getting the Policy

• If we have the value function, then finding the optimal policy, $\pi^*(s)$, is easy...just find the policy that maximized value

$$\pi^*(s) = \arg \max_a (R(s, a, s') + V^\pi(s'))$$

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

• Run whole policy, then receive a single reward

• Reward measures success of the whole policy

• If there are a small number of policies, we can exhaustively try them all

• This was not possible in most interesting problems, back in the day

• Nowdays, people do this a lot...but we’ll get to that in a bit.
Problems with Our Functions

• Consider this MDP
  • Number of steps is now unlimited because of loops
  • Value of states 1 and 2 is infinite for some policies
    \[ Q(1, A) = 1 + Q(1, A) \]
    \[ = 1 + 1 + Q(1, A) \]
    \[ = 1 + 1 + 1 + Q(1, A) \]
    \[ = \ldots \]
  • This is bad
    • All policies with a non-zero reward cycle have infinite value
Better Value Functions

- Introduce the *discount factor* $\gamma$, to get around the problem of infinite value
  - Three interpretations
    - Probability of living to see the next time step
    - Measure of the uncertainty inherent in the world
    - Makes the mathematics work out nicely

Assume $0 \leq \gamma \leq 1$

\[ V^\pi(s) = R(s, \pi(s), s') + \gamma V^\pi(s') \]

\[ Q(s, a) = R(s, a, s') + \gamma \max_{a'} Q(s', a') \]

\[ Q(s, a) = R(s, a, s') + \gamma \max_{a'} (R(s', a', s'') + \gamma \max_{a''} Q(s'', a'')) \]
Better Value Functions

- Optimal Policy:
  \( \pi(0) = B \)
  \( \pi(1) = A \)
  \( \pi(2) = A \)

\[ Q(0,A) = -1000 + \frac{\gamma}{1-\gamma} \]
\[ Q(0,B) = 1000 + \frac{\gamma}{1-\gamma} \]
\[ Q(1,A) = \frac{1}{1-\gamma} \]
\[ Q(1,B) = 0 \]
\[ Q(2,A) = \frac{1}{1-\gamma} \]
\[ Q(2,B) = 0 \]

Value now depends on the discount, \( \gamma \)
Dynamic Programming

• Given the complete MDP model, we can compute the optimal value function directly

\[
\begin{align*}
V(0) &= 1 + \gamma + 10\gamma^2 + 0\gamma^3 \\
V(1) &= 1 + 10\gamma + 0\gamma^2 \\
V(2) &= -1000 + 10\gamma + 0\gamma^2 \\
V(3) &= 1 + 0\gamma \\
V(4) &= 10 + 0\gamma \\
V(5) &= 0
\end{align*}
\]

[Bertsekas, 87, 95a, 95b]
Reinforcement Learning

• What happens if we don’t have the whole MDP?
  • We know the states and actions
  • We don’t have the system model (transition function) or reward function

• We’re only allowed to sample from the MDP
  • Can observe experiences \((s, a, r, s’)\)
  • Need to perform actions to generate new experiences

• This is Reinforcement Learning (RL)
  • Sometimes called Approximate Dynamic Programming (ADP)
Learning Value Functions

• We still want to learn a value function
  • We’re forced to approximate it iteratively
  • Based on direct experience of the world

• Four main algorithms
  • Certainty equivalence
  • TD $\lambda$ learning
  • Q-learning
  • SARSA
How are we going to do this?

• Reward whole policies?
  • That could be a pain

• What about incremental rewards?
  • Everything has a reward of 0 except for the goal

• Now what???
Exploration vs. Exploitation

• We want to pick good actions most of the time, but also do some exploration

• Exploring means we can learn better policies

• But, we want to balance known good actions with exploratory ones

• This is the exploration/exploitation problem
On-Policy vs. Off Policy

• On-policy algorithms
  • Final policy is influenced by the exploration policy
  • Generally, the exploration policy needs to be “close” to the final policy
  • Can get stuck in local maxima

• Off-policy algorithms
  • Final policy is independent of exploration policy
  • Can use arbitrary exploration policies
  • Will not get stuck in local maxima

Given enough experience
Picking Actions

$\varepsilon$-greedy
- Pick best (greedy) action with probability $1 - \varepsilon$
- Otherwise, pick a random action

Boltzmann (Soft-Max)
- Pick an action based on its Q-value

$$P(a|s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_{a'} e^{\frac{Q(s,a')}{\tau}}}$$

...where $\tau$ is the “temperature”
Temporal Difference Learning
TD(I)

- TD-learning estimates the value function directly
  - Don’t try to learn the underlying MDP

- Keep an estimate of $V^\pi(s)$ in a table
  - Update these estimates as we gather more experience
  - Estimates depend on exploration policy, $\pi$
  - TD is an on-policy method

[Sutton, 88]
What are we learning here?

Approach 1: Run the policy until you see the final outcome, then update your value $V(S)$. If $G$ is the final outcome...then

$$V(S_t) \leftarrow V(S_t) + \alpha \left[ G_t - V(S_t) \right]$$

Approach 2: Update as you go, using your existing estimate of state values as your proxy for the final reward

$$V(S_t) \leftarrow V(S_t) + \alpha \left[ R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$
The temporal difference error: $\delta_t$

This is the difference between our current estimate of this state’s value and our bootstrapped estimate of the value, now that we’ve taken an action according to a policy.

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$
TD(0)-Learning Algorithm

Tabular TD(0) for estimating $v_\pi$

Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in S^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    $A \leftarrow$ action given by $\pi$ for $S$
    Take action $A$, observe $R$, $S'$
    $V(S) \leftarrow V(S) + \alpha [R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until $S$ is terminal

$R = \text{reward}$
$\alpha = \text{learning rate}$
$\gamma = \text{discount factor}$

this formulation is from Sutton & Barto’s “Reinforcement Learning”
Driving example

- **(MC) Waiting till end**
- **(TD) Updating incrementally**

<table>
<thead>
<tr>
<th>State</th>
<th>Elapsed Time (minutes)</th>
<th>Predicted Time to Go</th>
<th>Predicted Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaving office, friday at 6</td>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>reach car, raining</td>
<td>5</td>
<td>35</td>
<td>40</td>
</tr>
<tr>
<td>exiting highway</td>
<td>20</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>2ndary road, behind truck</td>
<td>30</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>entering home street</td>
<td>40</td>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>arrive home</td>
<td>43</td>
<td>0</td>
<td>43</td>
</tr>
</tbody>
</table>
Why not wait till the end to do our update?
TD-Learning

• $V^\pi(s)$ is guaranteed to converge to $V^*(s)$
  • After an infinite number of experiences
  • If we decay the learning rate

\[ \alpha_t = \frac{c}{c + t} \]  will work, where $c$ is a constant and $t$ is the step index

• In practice, we often don’t need value convergence
  • Policy convergence generally happens sooner
What if we want to predict the action, not the state?

Replace our existing update as you go learning rule:

\[ V(S_t) \leftarrow V(S_t) + \alpha \left[ R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \]

With one that updates the Q table instead of the V table:

\[ Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right] \]
SARSA
SARSA

• SARSA iteratively approximates the state-action value function, $Q$
  • $L$, SARSA learns the policy and the value function simultaneously

• Keep an estimate of $Q(s, a)$ in a table
  • Update these estimates based on experiences
  • Estimates depend on the exploration policy
  • SARSA is an on-policy method
  • Policy is derived from current value estimates
SARSA: The algorithm

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+, a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  Loop for each step of episode:
    Take action $A$, observe $R, S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma Q(S', A') - Q(S, A) \right]$
    $S \leftarrow S'$; $A \leftarrow A'$
  until $S$ is terminal
Windy Grid World

- There is wind where there are arrows
- Wind moves you 1 step up at each turn
- This means there are some policies that will never terminate
- Therefore you must learn the policy on-the-fly
- You can’t just try all policies and wait till you see the final result
What if we want to learn the best action, regardless of current policy?

Replace our existing update to the Q table:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]$$

....with this:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right].$$
Q learning
Q-Learning

• Q-learning iteratively approximates the state-action value function, $Q$
  • We won’t estimate the MDP directly
  • Learns the value function and policy simultaneously

• Keep an estimate of $Q(s, a)$ in a table
  • Update these estimates as we gather more experience
  • Estimates do not depend on exploration policy
  • Q-learning is an off-policy method

[Watkins & Dayan, 92]
Q-Learning Algorithm

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+, a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  until $S$ is terminal

$0 \leq \alpha \leq 1$ is the learning rate & we should decay $\alpha$, just like in TD
Note: this formulation is from Sutton & Barto’s “Reinforcement Learning”
Breaking apart that update formula

\[ Q(s, a) \leftarrow Q(s, a) + \alpha (R + \gamma \max_a Q(s', a') - Q(s, a)) \]

This can be written another way...

\[ Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha (R + \gamma \max_a Q(s', a')) \]

Looked at this way, it is more obvious that \( \alpha \) controls whether we value past experience more or new experience more.
Q-learning

- Q-learning, learns the expected utility of taking a particular action $a$ in state $s$.
Convergence Guarantees

• The convergence guarantees for RL are “in the limit”
  • The word “infinite” crops up several times

• Don’t let this put you off
  • Value convergence is different than policy convergence
  • We’re more interested in policy convergence
  • If one action is significantly better than the others, policy convergence will happen relatively quickly
Rewards

• Rewards measure how well the policy is doing
  • Often correspond to events in the world
    • Current load on a machine
    • Reaching the coffee machine
    • Program crashing
  • Everything else gets a 0 reward

• Things work better if the rewards are incremental
  • For example, distance to goal at each step
  • These reward functions are often hard to design

These are sparse rewards

These are dense rewards

Northwestern University, EECS 349, 2017
Deep Q learning
Let’s talk state space & combinatorics

• The idea is to learn a probability distribution over the set of actions possible at each state
• We’ve assumed that there is a table of states and actions
• How big could such a table get?
Playing Video Games

We need to replace the table with....

• ...a parameterized function that can output the policy

• Historically, this could be any function

• These days it means...you guessed it...a deep net

• ...and this also means we need differentiable policy gradient
Going to a parameterized Q model

Take the standard Bellman equation for estimating the Q function:

\[ Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right] \]  \hspace{1cm} (1)

Take a loss function for a parameterized function \(\hat{y}_i\), where \(y_i\) is the target value and \(\theta_i\) are the parameters:

\[ L_i (\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} \left[ (y_i - Q(s, a; \theta_i))^2 \right], \]  \hspace{1cm} (2)

Take the gradient:

\[ \nabla_{\theta_i} L_i (\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]. \]  \hspace{1cm} (3)

Deep Q-learning

**Algorithm 1** Deep Q-learning with Experience Replay

- Initialize replay memory $\mathcal{D}$ to capacity $N$
- Initialize action-value function $Q$ with random weights
- for episode = 1, $M$ do
  - Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  - for $t = 1, T$ do
    - With probability $\epsilon$ select a random action $a_t$
    - otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    - Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    - Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    - Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    - Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    - Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    - Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  - end for
- end for

This was a breakthrough

<table>
<thead>
<tr>
<th></th>
<th>B. Rider</th>
<th>Breakout</th>
<th>Enduro</th>
<th>Pong</th>
<th>Q*bert</th>
<th>Seaquest</th>
<th>S. Invaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>354</td>
<td>1.2</td>
<td>0</td>
<td>−20.4</td>
<td>157</td>
<td>110</td>
<td>179</td>
</tr>
<tr>
<td>Contingency [4]</td>
<td>1743</td>
<td>6</td>
<td>159</td>
<td>−17</td>
<td>960</td>
<td>723</td>
<td>268</td>
</tr>
<tr>
<td>DQN</td>
<td>4092</td>
<td>168</td>
<td>470</td>
<td>20</td>
<td>1952</td>
<td>1705</td>
<td>581</td>
</tr>
<tr>
<td>Human</td>
<td>7456</td>
<td>31</td>
<td>368</td>
<td>−3</td>
<td>18900</td>
<td>28010</td>
<td>3690</td>
</tr>
<tr>
<td>HNeat Best [8]</td>
<td>3616</td>
<td>52</td>
<td>106</td>
<td>19</td>
<td>1800</td>
<td>920</td>
<td>1720</td>
</tr>
<tr>
<td>HNeat Pixel [8]</td>
<td>1332</td>
<td>4</td>
<td>91</td>
<td>−16</td>
<td>1325</td>
<td>800</td>
<td>1145</td>
</tr>
<tr>
<td>DQN Best</td>
<td>5184</td>
<td>225</td>
<td>661</td>
<td>21</td>
<td>4500</td>
<td>1740</td>
<td>1075</td>
</tr>
</tbody>
</table>

Table 1: The upper table compares average total reward for various learning methods by running an $\epsilon$-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an $\epsilon$-greedy policy with $\epsilon = 0.05$.

Let’s watch!

https://www.youtube.com/watch?v=V1eYniJ0Rnk
Alpha Go
The game of GO

- 19 by 19 board
- Each position is either empty, white, or black
- At each turn you can place a stone of your color on any empty position
- How big would the Q table be for this world?
AlphaGO

• 2015: First program to beat a professional Go master with no handicap

• 2017: Beat Ke Jie, the number one ranked player in the world at the time

The parts of Alpha Go

Train the policy network with supervised learning, based on prior human games

$\sigma$ are the network weights

$p_{\sigma}(a|s)$ is the distribution over the next actions output by the model.

$p_{\sigma}(a|s)$ is compared to true action $y$.

After training this predicted the true human move 57% of the time

They also trained a small fast “rollout” policy network to predict actions. 

\( p_\pi(a|s) \) is the distribution over the next actions output by fast model. 

After training this predicted the true human move 24% of the time.

...but it took 2 microseconds to predict the next move, which was useful for rollouts to play out games.
The parts of Alpha Go

Once they had a trained policy network, they engaged in self-play.

Make 2 copies of the network.

They play.

Apply reinforcement learning to the winner.

Thereafter, always play vs a previous version of the RL network.

Learn as you go...

The parts of Alpha Go

Then....they generated 30 million game states from self play and taught a Value network to estimate the value of every one of those states, based on the outcome of the game the state came from.

Monte Carlo Tree Search (MCTS)

Select the edge with maximum action-value $Q$, plus a bonus $u(P)$ that depends on a stored prior probability $P$ for that edge.

Monte Carlo Tree Search (MCTS)

- The leaf node may be expanded; the new node is processed once by the \textit{supervised} policy network $p_\sigma$ and the output probabilities are stored as prior probabilities $P$ for each action.

Monte Carlo Tree Search (MCTS)

- At the end of a simulation, the leaf node is evaluated in two ways:
  - using the value network $v_\theta$
  - running a rollout to the end of the game with the fast rollout policy $p_\pi$, then computing the winner with function $r$.

Monte Carlo Tree Search (MCTS)

- Action-values Q are updated to track the mean value of all evaluations $r()$ and $v()$ in the subtree below that action.

Do they really need to learn from humans?

• Alpha0 jettisoned all the human-trained supervised learning

• It learns exclusively from self-play

• How well does that work?

Alpha0 is a general game winner