Deep Q-Learning, AlphaGo and AlphaZero

Lyle Ungar

with a couple slides from Eric Eaton

DQN
AlphaGo
AlphaZero, MuZero
Remember Q-Learning

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

\[ Q(s', \mu(s')) = \max_{a'} Q(s', a') \]

Converges when this is zero
Aside: off vs. on policy

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

Converges when this is zero

**Q-learning** - evaluate next state **off policy** (greedy)
\[ Q(s', \mu(s')) = \max_{a'} Q(s', a') \]

**SARSA** - evaluate next state **on policy** (\(\varepsilon\)-greedy)
\[ Q(s', \mu(s')) = Q(s', \pi(s')) = \max_{a'} Q(s', a') \text{ with prob } 1-\varepsilon \]
\[ \text{random}_{a'} Q(s', a') \text{ with prob } \varepsilon \]
Deep Q-Learning (DQN)

Inspired by

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Represent $Q(s, a)$ by a neural net

Estimate using gradient descent with loss function:

$$(R + \gamma \max_{a'} Q(s', a') - Q(s, a))^2$$

The policy, $\pi(a)$, is then given by maximizing the predicted Q-value
Separate Q- and Target Networks

**Issue:** Instability (e.g., rapid changes) in the Q-function can cause it to diverge

**Idea:** use two networks to provide stability

- The Q-network is updated regularly
- The target network is an older version of the Q-network, updated occasionally

\[
\left( R(s, a, s') + \gamma \max_{a'} Q(s', a') \right) - Q(s, a)^2
\]

- computed via target network
- computed via Q-network
Deep Q-Learning (DQN) Algorithm

Initialize replay memory $D$
Initialize Q-function weights $\theta$
for episode $= 1 \ldots M$, do
  Initialize state $s_t$
  for $t = 1 \ldots T$, do
    $a_t \leftarrow \begin{cases} 
    \text{random action} & \text{with probability } \epsilon \\
    \max_a Q^* (s_t, a; \theta) & \text{with probability } 1 - \epsilon
    \end{cases}$
    Execute action $a_t$, yielding reward $r_t$ and state $s_{t+1}$
    Store $\langle s_t, a_t, r_t, s_{t+1} \rangle$ in $D$
    $s_t \leftarrow s_{t+1}$
    Sample random minibatch of transitions $\{\langle s_j, a_j, r_j, s_{j+1} \rangle\}_{j=1}^{N}$ from $D$
    $y_j \leftarrow \begin{cases} 
    r_j & \text{for terminal state } s_{j+1} \\
    r_j + \gamma \max_{a'} Q (s_{j+1}, a'; \theta) & \text{for non-terminal state } s_{j+1}
    \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
  end for
end for

DQN on Atari Games

Image Sources:
https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756
https://deepmind.com/blog/going-beyond-average-reinforcement-learning/
AlphaGo

https://medium.com/@jonathan_hui/alphago-how-it-works-technically-26ddcc085319 2016
1. Train a CNN to predict (supervised learning) moves of human experts

2. Use as starting point for policy gradient (self-play against older self)

3. Train value network with examples from policy network self-play

4. Use Monte Carlo tree search to explore possible possible games

Learn policy

- $a = f(s)$
- $a =$ where to play (19*19)
- $s =$ description of board
<table>
<thead>
<tr>
<th>Feature</th>
<th># of planes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stone colour</td>
<td>3</td>
<td>Player stone / opponent stone / empty</td>
</tr>
<tr>
<td>Ones</td>
<td>1</td>
<td>A constant plane filled with 1</td>
</tr>
<tr>
<td>Turns since</td>
<td>8</td>
<td>How many turns since a move was played</td>
</tr>
<tr>
<td>Liberties</td>
<td>8</td>
<td>Number of liberties (empty adjacent points)</td>
</tr>
<tr>
<td>Capture size</td>
<td>8</td>
<td>How many opponent stones would be captured</td>
</tr>
<tr>
<td>Self-atari size</td>
<td>8</td>
<td>How many of own stones would be captured</td>
</tr>
<tr>
<td>Liberties after move</td>
<td>8</td>
<td>Number of liberties after this move is played</td>
</tr>
<tr>
<td>Ladder capture</td>
<td>1</td>
<td>Whether a move at this point is a successful ladder capture</td>
</tr>
<tr>
<td>Ladder escape</td>
<td>1</td>
<td>Whether a move at this point is a successful ladder escape</td>
</tr>
<tr>
<td>Sensibleness</td>
<td>1</td>
<td>Whether a move is legal and does not fill its own eyes</td>
</tr>
<tr>
<td>Zeros</td>
<td>1</td>
<td>A constant plane filled with 0</td>
</tr>
<tr>
<td>Player color</td>
<td>1</td>
<td>Whether current player is black</td>
</tr>
</tbody>
</table>
AlphaGo – lots of hacks

- **Bootstrap**
  - Initialize with policy learned from human play

- **Self-play**

- **Speed matters**
  - Rollout network (fast, less accurate game play)
  - Monte Carlo search

- **It still needed fast computers**
  - > 100 GPU weeks
AlphaZero

- Self-play with a single, continually updated neural net
  - No annotated features - just the raw board position
- Uses Monte Carlo Tree Search
  - Using $Q(s,a)$
- Does policy iteration
  - Learns $V(s)$ and $P(s)$
- Beat AlphaGo (100-0) after just 72 hours of training
  - On 5,000 TPUs

Monte Carlo Tree Search (MCTS)

- In each state $s_{root}$, select a move, $a_t \sim \pi_t$ either proportionally (exploration) or greedily (exploitation)

- Pick a move $a_t$ with
  - low visit count (not previously frequently explored)
  - high move probability (under the policy)
  - high value (averaged over the leaf states of MC plays that selected $a$ from $s$) according to the current neural net

- The MCTS returns an estimate $z$ of $\nu(s_{root})$ and a probability distribution over moves, $\pi = p(a|s_{root})$
AlphaZero loss function

NNet: \((p, v) = f_\theta(s)\)

- Minimizes the error between the value function \(v(s)\) and the actual game outcome \(z\)
- Maximizes the similarity of the policy vector \(p(s)\) to the MCTS probabilities \(\pi(s)\).
- \(L_2\) regularize the weights \(\theta\)

\[
l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2.
\]
AlphaZero

Believe it or not, we now have all elements required to train our unsupervised game playing agent! Learning through self-play is essentially a policy iteration algorithm— we play games and compute Q-values using our current policy (the neural network in this case), and then update our policy using the computed statistics.

Here is the complete training algorithm. We initialise our neural network with random weights, thus starting with a random policy and value network. In each iteration of our algorithm, we play a number of games of self-play. In each turn of a game, we perform a fixed number of MCTS simulations starting from the current state \( s_t \). We pick a move by sampling from the improved policy \( \pi_t \). This gives us a training example \( (s_t, \pi_t, \_\_\_\_) \). The reward \( \_\_\_\_ \) is filled in at the end of the game: +1 if the current player eventually wins the game, else -1. The search tree is preserved during a game.

At the end of the iteration, the neural network is trained with the obtained training examples. The old and the new networks are pit against each other. If the new network wins more than a set threshold fraction of games (55% in the DeepMind paper), the network is updated to the new network. Otherwise, we conduct another iteration to augment the training examples.

And that’s it! Somewhat magically, the network improves almost every iteration and learns to play the game better. The high-level code for the complete training algorithm is provided below.

https://web.stanford.edu/~surag/posts/alphazero.html
MuZero

◆ Slight generalization of AlphaZero
  ● Also uses Monte Carlo Tree Search (MCTC), self-play
◆ Learns Go, Chess, Atari ...
◆ Learns 3 CNNs
  ● Representation model: observation history $\rightarrow$ state$_t$
  ● Dynamics model: state$_t$ * action$_t$ $\rightarrow$ state$_{t+1}$
  ● Policy: state$_t$ $\rightarrow$ action$_t$
StarCraft

- StarCraft-playing AI model consists of 18 agents, each trained with 16 Google v3 TPUs for 14 days.
- Thus, at current prices ($8.00 / TPU hour), the company spent $774,000 on this model.
Applied RL Summary

- Superhuman game playing using DQL
  - But it is fragile

- Why is DeepMind losing $500 million/year?