Machine learning: Part 6

- Deep reinforcement learning
- AlphaGo

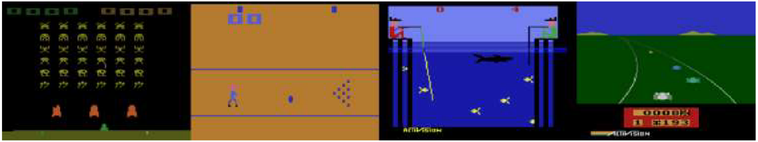
*Slides based on those of Pascal Poupart

Challenges for reinforcement learning

- To use RL successfully in situations approaching real-world complexity, agents are confronted with a difficult task:
- they must derive efficient representations of the environment from high-dimensional sensory inputs
- Humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems

Deep reinforcement learning

- Combine reinforcement learning with deep neural networks.
- Use a deep CNN to approximate the optimal action-value function $Q^*(s, a)$
- Tested on Atari 2600 games.



- Achieve a level comparable to that of a professional human games tester across a set of 49 games.

Quick recap

- $Q^*(s, a)$: the expected value of doing action a in state s, then following the optimal policy.
- $V^*(s)$: the expected value of following the optimal policy in state s.
- TD formula: $A_k = A_{k-1} + \alpha_k(v_k - A_{k-1})$
  - We have a sequence of values: $v_1, v_2, v_3, \ldots,$
  - To get the new estimate, the old estimate is updated by α_k times the TD error
- Q-learning: an experience $\langle s, a, r, s' \rangle$ gives a new estimate for the value of $Q^*(s, a)$: $r + \gamma \max_{a'} Q[s', a']$, so

$$Q[s, a] \leftarrow Q[s, a] + \alpha \left(r + \gamma \max_{a'} Q[s', a'] - Q[s, a] \right)$$

Recall: Q-learning

initialize $Q[S, A]$ arbitrarily
observe current state s
repeat forever:
    select and carry out an action a
    observe reward r and state s'
    $Q[s, a] \leftarrow Q[s, a] + \alpha \left(r + \gamma \max_{a'} Q[s', a'] - Q[s, a]\right)$
    $s \leftarrow s'$

Large state spaces

- Complexity of Q-learning depends on number of states and actions
- Go: 3^{361} states
- Atari: 210x160x3 dimensions (pixel values)

Q-networks

- Represent value function by Q-network with weights $\mathbf{w}$
  $Q(s, a, \mathbf{w}) \approx Q^*(s, a)$
- Q-learning:
  $Q[s, a] \leftarrow Q[s, a] + \alpha\,(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
- Treat $r + \gamma \max_{a'} Q[s', a']$ as a target
- Minimize squared error by gradient descent:
  $Loss(\mathbf{w}) = (r + \gamma \max_{a'} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}))^2$
- $\frac{\partial Loss(\mathbf{w})}{\partial w} = (r + \gamma \max_{a'} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}))\frac{\partial Q(s,a,\mathbf{w})}{\partial w}$

Gradient Q-learning

initialize weights $\mathbf{w}$ at random in [-1,1]
observe current state s
repeat forever:

select and carry out an action a
observe reward r and state s'
$\frac{\partial Loss(\mathbf{w})}{\partial w} = (r + \gamma \max_{a'} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w}))\frac{\partial Q(s,a,\mathbf{w})}{\partial w}$
$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial Loss(\mathbf{w})}{\partial w}$
$s \leftarrow s'$

Convergence of Linear Gradient Q-Learning

- Recall: Q-Learning converges to optimal Q-function under the following conditions:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

- Linear Gradient Q-Learning converges under the same conditions
  - $Q(s, a, \mathbf{w}) = \Sigma_i w_i s_i$, where $s = (x_1, \ldots, x_n)$

Divergence of non-linear Q-learning

- Even when the following conditions hold

$$\sum_{k=1}^{\infty} \alpha_k = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

  non-linear Q-learning may diverge

- Intuition: Adjusting $\mathbf{w}$ to increase Q at (s, a) might introduce errors at nearby state-action pairs.
- Mitigating divergence: Two tricks are often used
  - Experience replay
  - Use two networks: Q-network and Target network

Experience Replay

- Idea: store previous experiences (s, a, r, s') into a buffer and sample a mini-batch of previous experiences at each step to learn by Q-learning

s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
$\ldots$
$s_t, a_t, r_{t+1}, s_{t+1}$

$s_t, a_t, r_{t+1}, s_{t+1} \rightarrow$ buffer $\rightarrow s, a, r, s'$

- Advantages
  - Break correlations between successive updates (more stable learning)
  - Fewer interactions with environment needed to converge (greater data efficiency)

Target Network

- Idea: Use a separate target network that is updated only periodically

  For each experience $(\hat{s}, \hat{a}, \hat{r}, \hat{s}')$ in mini-batch

  $$\frac{\partial Loss(\mathbf{w})}{\partial w} = (\hat{r} + \gamma \max_{\hat{a}'} Q(\hat{s}', \hat{a}', \bar{\mathbf{w}}) - Q(\hat{s}, \hat{a}, \mathbf{w})) \frac{\partial Q(\hat{s}, \hat{a}, \mathbf{w})}{\partial w}$$

  $$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial Loss(\mathbf{w})}{\partial w}$$

- Advantage: mitigate divergence

Deep Q-network

- Google Deep Mind
- Deep Q-network: Gradient Q-learning with
  - Deep neural networks
  - Experience replay
  - Target network
- Breakthrough: human-level play in many Atari video games

Deep Q-network

initialize weights $\mathbf{w}$ at random in [-1,1]
observe current state s
repeat forever:

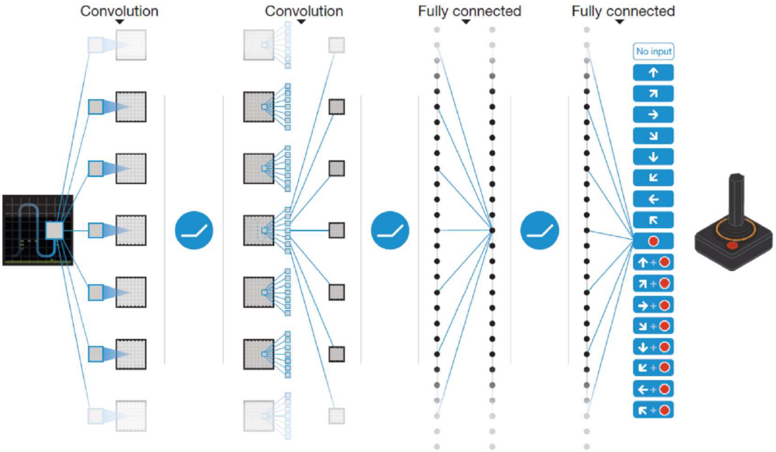
select and carry out an action a
observe reward r and state s'
Add (s, a, r, s') to experience buffer
Sample mini-batch of experiences from buffer
For each experience $(\hat{s}, \hat{a}, \hat{r}, \hat{s}')$ in mini-batch

$$\frac{\partial Loss(\mathbf{w})}{\partial w} = (\hat{r} + \gamma \max_{\hat{a}'} Q(\hat{s}', \hat{a}', \bar{\mathbf{w}}) - Q(\hat{s}, \hat{a}, \mathbf{w}))\frac{\partial Q(\hat{s}, \hat{a}, \mathbf{w})}{\partial w}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial Loss(\mathbf{w})}{\partial w}$$

$s \leftarrow s'$
Every c steps, $\bar{\mathbf{w}} \leftarrow \mathbf{w}$

Deep Q-Network for Atari

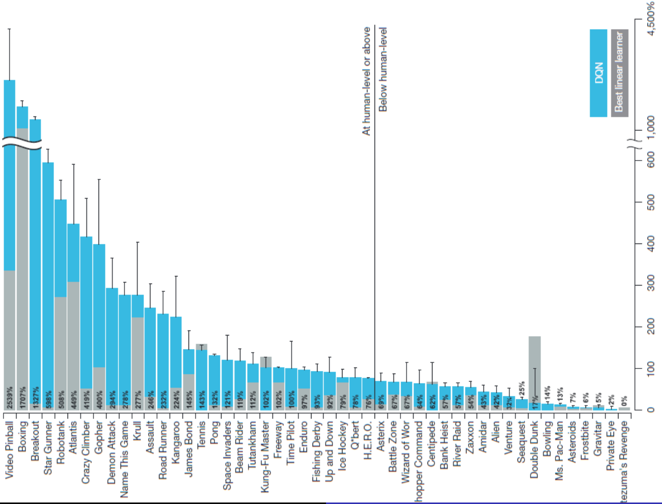


Deep Q-Network for Atari

- The input consists of an 84 x 84 x 4 image produced by the preprocessing map ϕ.
- Followed by three convolution layers and two fully connected layers.
- Each hidden layer is followed by a rectifier nonlinearity.
- There is a single output for each valid action.
- The number of valid actions varied between 4 and 18 on the games considered.

DQN versus Linear approx.

The normalized performance of DQN: (DQN score-random play score)/(human score-random play score).



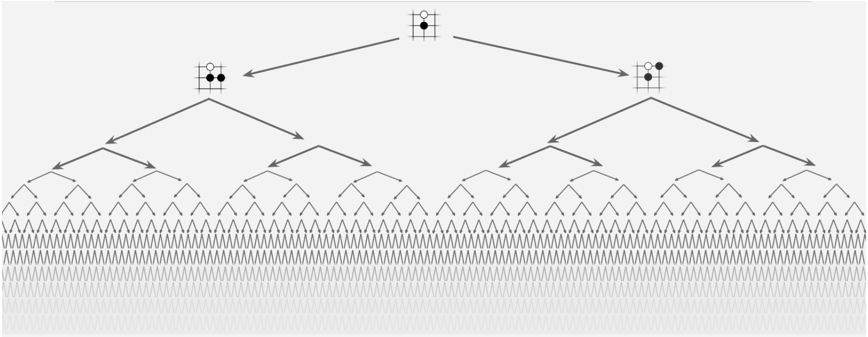
AlphaGo

- Go is hard for computers due to its enormous search space and difficulty of evaluating board positions and moves.
- A new approach to computer Go that uses ‘value networks’ to evaluate board positions and ‘policy networks’ to select moves.
- These DNNs are trained by a combination of supervised learning from human expert games, and reinforcement learning from games of self-play.
- A new search algorithm that combines Monte Carlo simulation with value and policy networks.

Perfect information games

- Have an optimal value function, $v^*(s)$, from every board position or state s, under perfect play by all players.
- These games may be solved by recursively computing the optimal value function in a search tree
  - containing $\approx b^d$ possible sequences of moves,
  - where b is the game's breadth (number of legal moves per position)
  - and d is its depth (game length).

Exhaustive search



Perfect information games

- In large games, exhaustive search is infeasible,
  - chess ($b \approx 35$, $d \approx 80$) and Go ($b \approx 250, d \approx 150$)
- but the effective search space can be reduced by two general principles

Reducing depth of search

- by position evaluation: truncating the search tree at state s by an approximate value function $v(s) \approx v^*(s)$.
- This approach has led to superhuman performance in chess, checkers and othello,
- but it was believed to be intractable in Go due to the complexity of the game.

Reducing breadth of search

- by sampling actions from a policy $p(a|s)$, a probability distribution over possible moves a in position s.
- *e.g.*, Monte Carlo rollouts search to maximum depth without branching at all, by sampling long sequences of actions for both players from a policy p.
- Averaging over such rollouts can provide an effective position evaluation, achieving superhuman performance in backgammon and Scrabble.

Monte Carlo tree search (MCTS)

- Uses Monte Carlo rollouts to estimate the value of each state in a search tree.
- As more simulations are executed, the search tree grows larger and the relevant values become more accurate.
- The policy used to select actions during search is also improved over time, by selecting children with higher values.
- This policy converges to optimal play, and the evaluations converge to the optimal value function.

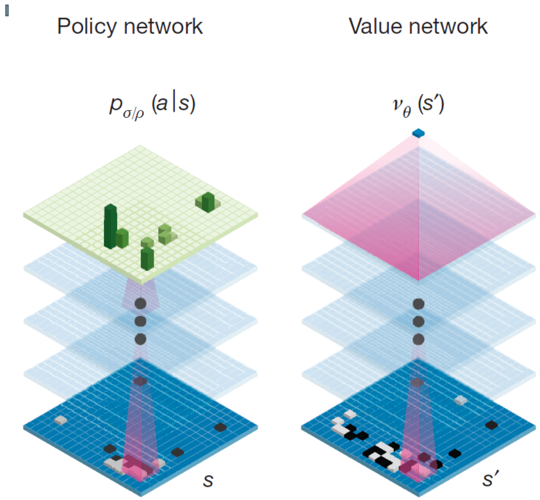
Enhancing MCTS

- MCTS can be enhanced by policies that are trained to predict human expert moves.
- These policies are used to narrow the search to a beam of high-probability actions, and to sample actions during rollouts.

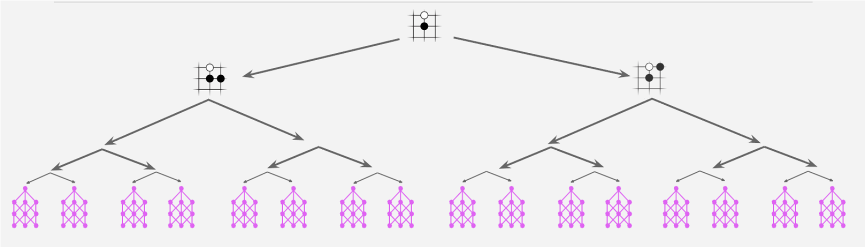
AlphaGo

- Pass in the board position as a 19 $\times$19 image and use convolutional layers to construct a representation of the position.
- Use these neural networks to reduce the effective depth and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network.

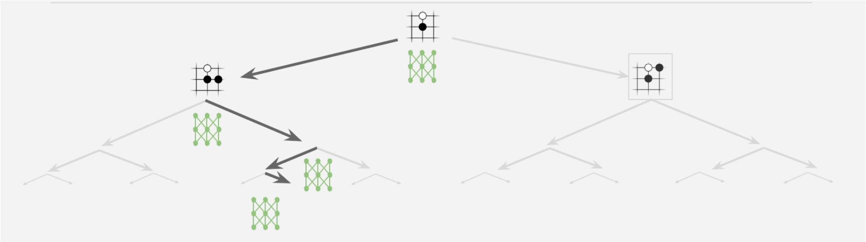
Policy and value networks



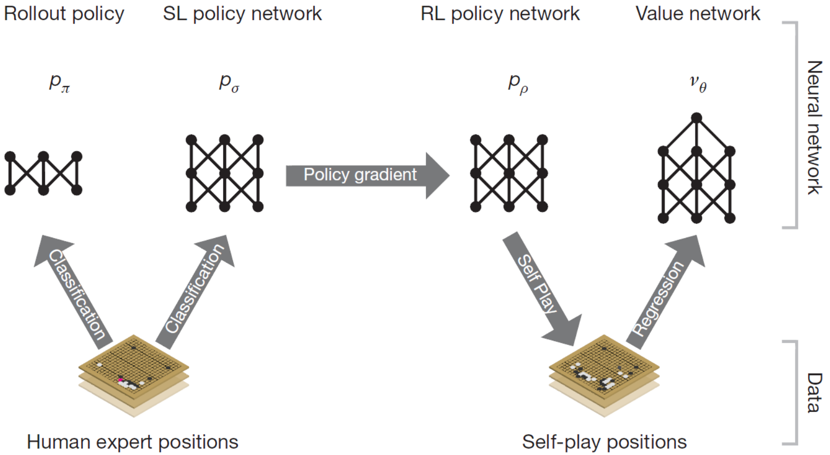
Reducing depth with value network



Reducing breadth with policy network



Neural network training pipeline



Neural network training pipeline

- A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions.
- A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning.
- A new data set is generated by playing games of self-play with the RL policy network.
- Finally, a value network v_θ is trained by regression to predict the expected outcome (*i.e.*, whether the current player wins) in positions from the self-play data set.

Supervised learning of policy networks

Policy network: 12 layer convolutional neural network

Training data: 30M positions from human expert games (KGS 5+ dan)

Training algorithm: maximise likelihood by stochastic gradient descent

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$$

Training time: 4 weeks on 50 GPUs using Google Cloud

Results: 57% accuracy on held out test data (state-of-the art was 44%)

randomly sampled state-action pairs (s, a)

Reinforcement learning of policy networks

Policy network: 12 layer convolutional neural network

Training data: games of self-play between policy network

Training algorithm: maximise wins z by policy gradient reinforcement learning

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma} z$$

Training time: 1 week on 50 GPUs using Google Cloud

Results: 80% vs supervised learning. Raw network ~3 amateur dan.

The outcome z is the terminal reward at the end of the game: $+1$ for winning and -1 for losing.

won more than 80% of games against the SL policy network

Reinforcement learning of value networks

Value network: 12 layer convolutional neural network

Training data: 30 million games of self-play

Training algorithm: minimise MSE by stochastic gradient descent

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$$

Training time: 1 week on 50 GPUs using Google Cloud

Results: First strong position evaluation function - previously thought impossible