6. Deep Reinforcement Learning

Reinforcement Learning Notes

Contents

1 Deep Reinforcement Learning

1.1 Introduction

Reinforcement learning has shown significant success in various domains; however, traditional RL methods face challenges when dealing with high-dimensional state spaces. Deep Reinforcement Learning (DRL) addresses this limitation by combining deep neural networks with reinforcement learning techniques, allowing agents to learn directly from raw sensory inputs.

The key advantage of deep reinforcement learning is its ability to automatically extract relevant features from raw input data, eliminating the need for manual feature engineering. This makes DRL particularly useful for tasks where the state space is large and complex, such as computer vision, natural language processing, and game playing.

1.2 Deep Q Learning

1.2.1 Neural Networks for Atari Games

Deep Q-Learning was first successfully applied to Atari games by Mnih et al. (2013/2015) at DeepMind. The key innovation was using convolutional neural networks to process raw pixel inputs from Atari games and learn control policies directly from this high-dimensional sensory input.

Input Processing The input to the neural network consists of the current and three previous game frames, stacked together to provide temporal information. This allows the network to infer motion and dynamics from the static frames.

Network Architecture The Deep Q-Network (DQN) uses:

- Convolutional layers to process the visual input
- Fully-connected layers for decision making
- Output layer with one node for each possible action, representing the estimated Q-value

Mathematical Formulation The neural network approximates the action-value function $Q(s, a)$ using a parameterized function $Q(s, a; \theta)$, where θ represents the network weights.

$$Q(s, a; \theta) \approx Q^*(s, a) \tag{1}$$

Notation Overview

- $Q(s, a; \theta)$: Approximated action-value function with parameters θ
- $Q^*(s, a)$: Optimal action-value function
- s: State (preprocessed game frames)
- a: Action (game controls)
- θ: Neural network parameters

1.2.2 Target Networks

A significant challenge in using neural networks for Q-learning is stability. The Q-learning update rule relies on the current estimate of the action-value function, which can lead to oscillations or divergence when function approximation is used.

The Problem The standard Q-learning update rule is:

$$\theta_{t+1} = \theta_t + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t) - Q(s_t, a_t; \theta_t) \right] \nabla_{\theta_t} Q(s_t, a_t; \theta_t) \tag{2}$$

Notation Overview

- θ_t: Network parameters at time t
- α: Learning rate
- r_{t+1}: Reward received after taking action a_t in state s_t
- γ: Discount factor
- $\max_a Q(s_{t+1}, a; \theta_t)$: Maximum Q-value for the next state
- $Q(s_t, a_t; \theta_t)$: Current Q-value estimate
- $\nabla_{\theta_t} Q(s_t, a_t; \theta_t)$: Gradient of the Q-value with respect to the parameters

The problem is that the target $r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t)$ depends on the same parameters θ_t that are being updated, creating a moving target.

The Solution: Target Network To stabilize learning, DQN uses a separate target network with parameters θ^- that are periodically updated to match the online network parameters θ:

$$\theta_{t+1} = \theta_t + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t^-) - Q(s_t, a_t; \theta_t) \right] \nabla_{\theta_t} Q(s_t, a_t; \theta_t) \tag{3}$$

Notation Overview

- θ_t^-: Target network parameters at time t
- $Q(s_{t+1}, a; \theta_t^-)$: Q-value estimated by the target network

The target network parameters θ^- are updated every C steps by copying the online network parameters θ, effectively fixing the target for C updates:

$$\theta_t^- = \theta_{t-C} \tag{4}$$

This approach reduces the correlation between the target and the current estimate, leading to more stable learning.

1.2.3 Experience Replay

Another challenge in deep reinforcement learning is the correlation between consecutive samples, which can lead to inefficient learning and instability.

The Problem In standard online reinforcement learning, experiences $(s_t, a_t, r_{t+1}, s_{t+1})$ are used immediately and then discarded. This leads to:

- Strong correlations between consecutive samples
- Inefficient use of data (experiences are used only once)
- Forgetting of rare but potentially important experiences

The Solution: Experience Replay Experience replay addresses these issues by storing experiences in a replay memory and sampling from this memory for training:

Algorithm 1 Deep Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        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 D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to θ
        Every C steps reset $\hat{Q} = Q$ (i.e., set $\theta^- = \theta$)
    end for
end for

Notation Overview

- D: Replay memory with capacity N
- ϕ: Preprocessing function (e.g., frame stacking, normalization)
- ϵ: Exploration rate for the ϵ-greedy policy
- y_j: Target value for the Q-function
- $\hat{Q}$: Target network for computing target values

Benefits of Experience Replay

- Breaks correlations between consecutive samples by random sampling
- More efficient use of data by reusing experiences
- Smooths the training distribution over many past behaviors
- Increases stability by reducing variance in updates

1.3 AlphaGo

AlphaGo, developed by Silver et al. at DeepMind, was the first computer program to defeat a world champion in the game of Go. It combines deep neural networks with Monte Carlo tree search (MCTS) to achieve superhuman performance.

1.3.1 Challenges in Go

Go presents several unique challenges that make it significantly harder for computers than games like chess:

High Branching Factor The game of Go has approximately 250 legal moves per position, compared to about 35 for chess. This creates a much larger search space.

Game Length Go games typically involve around 150 moves, further expanding the total number of possible game states.

Evaluation Difficulty Unlike chess, evaluating Go positions is extremely challenging. As Müller (2002) noted, "No simple yet reasonable evaluation function will ever be found for Go."

Mathematical Complexity The game tree complexity of Go is estimated to be around 10^{170}, making exhaustive search completely infeasible.

1.3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a key component of AlphaGo, used to explore the game tree efficiently.

Basic MCTS Algorithm The MCTS algorithm consists of four main steps:

1. **Selection:** Starting from the root node, traverse the tree using a tree policy until reaching a leaf node.
2. **Expansion:** Add one or more child nodes to the current leaf.
3. **Simulation:** From the new node, simulate a complete game using a rollout policy.
4. **Backpropagation:** Update the value estimates of all traversed nodes based on the simulation outcome.

UCT Selection Formula The Upper Confidence Bound applied to Trees (UCT) formula guides the selection phase:

$$a_t = \arg\max_a \left(Q(s_t, a) + c \cdot \frac{\sqrt{\ln N(s_t)}}{N(s_t, a)} \right) \tag{5}$$

Notation Overview

- a_t: Action selected at time t
- $Q(s_t, a)$: Estimated value of taking action a in state s_t
- c: Exploration constant
- $N(s_t)$: Number of times state s_t has been visited
- $N(s_t, a)$: Number of times action a has been taken in state s_t

1.3.3 Deep Neural Networks for Go

AlphaGo uses multiple deep neural networks to guide the MCTS process and evaluate positions.

Neural Network Types AlphaGo uses three different networks:

- **Policy Network:** Suggests promising moves for tree expansion
- **Value Network:** Evaluates board positions without simulation
- **Rollout Policy Network:** Guides fast rollout simulations

Policy Network The policy network $p_\sigma(a|s)$ is trained to predict expert human moves:

$$p_\sigma(a|s) \approx p(a|s) \tag{6}$$

Where $p(a|s)$ is the probability of an expert human player choosing action a in state s.

Notation Overview

- $p_\sigma(a|s)$: Policy network with parameters σ
- $p(a|s)$: Expert human policy
- s: Board position
- a: Move

The policy network is trained in two phases:

1. **Supervised Learning:** Training to predict expert moves from a database of human games
2. **Reinforcement Learning:** Further training by self-play and policy gradient methods

Value Network The value network $v_\theta(s)$ approximates the expected outcome from position s:

$$v_\theta(s) \approx v^\pi(s) = \mathbb{E}[z_t|s_t = s, a_{t...T} \sim \pi] \tag{7}$$

Notation Overview

- $v_\theta(s)$: Value network with parameters θ
- $v^\pi(s)$: True value function under policy π
- z_t: Final game outcome from time t (win=+1, loss=-1)
- π: Policy used for self-play games

The value network is trained on positions from self-play games using the reinforcement-learned policy network, with the game outcomes as targets.

Rollout Policy Network The rollout policy is a simpler, faster network used for MCTS simulations:

$$p_\psi(a|s) \approx p(a|s) \tag{8}$$

Notation Overview

- $p_\psi(a|s)$: Rollout policy network with parameters ψ
- ψ: Parameters of a linear network (much simpler than σ)

The rollout policy is designed for speed (about 1000 times faster than the policy network) at the cost of some accuracy.

1.3.4 AlphaGo's MCTS Algorithm

AlphaGo combines MCTS with the neural networks to select moves:

Algorithm 2 AlphaGo's MCTS Algorithm

function SEARCHTREE(position s, simulations n)
  for $i = 1$ to n **do**
    $s' \leftarrow s$ ▷ Copy the root position
    $path \leftarrow \emptyset$ ▷ Empty array of (state, action) pairs
    while s' has been visited before and is not terminal **do**
      $a \leftarrow \arg\max_a \left(Q(s', a) + u(s', a)\right)$ ▷ Select with exploration bonus
      Append (s', a) to $path$
      $s' \leftarrow$ next state after action a
    end while
    if s' is terminal **then**
      $v \leftarrow$ outcome of s' for the player to move in the root position
    else if s' has not been visited before **then**
      $P(s', \cdot) \leftarrow p_\sigma(\cdot|s')$ ▷ Use policy network for probabilities
      $v \leftarrow v_\theta(s')$ ▷ Use value network for evaluation
    else
      $a \sim p_\psi(\cdot|s')$ ▷ Sample from rollout policy
      Simulate a rollout from s' until terminal state s_T
      $v \leftarrow$ outcome of s_T for the player to move in the root position
    end if
    for each (s, a) in $path$ **do**
      $N(s, a) \leftarrow N(s, a) + 1$ ▷ Increment visit count
      $Q(s, a) \leftarrow Q(s, a) + \frac{v - Q(s,a)}{N(s,a)}$ ▷ Update value estimate
    end for
  end for
  return $\arg\max_a N(s, a)$ ▷ Return most visited action
end function

Notation Overview

- $u(s, a)$: Exploration bonus based on $P(s, a)$ and visit counts
- $P(s, a)$: Prior probability of selecting action a in state s from policy network
- $N(s, a)$: Visit count for state-action pair
- $Q(s, a)$: Action value estimate from search

1.4 AlphaGo Zero

AlphaGo Zero represents a significant advancement over the original AlphaGo by learning entirely from self-play, without using any human gameplay data.

1.4.1 Learning Without Human Knowledge

AlphaGo Zero starts from random play and learns entirely through self-play reinforcement learning. This approach offers several advantages:

- Avoids human biases and limitations
- Discovers novel strategies beyond human knowledge
- Simplifies the training pipeline
- Demonstrates the possibility of tabula rasa learning in complex domains

1.4.2 Network Architecture

AlphaGo Zero uses a single neural network instead of the separate policy and value networks in the original AlphaGo:

$$(p(a|s), v(s)) = f_\theta(s) \tag{9}$$

Notation Overview

- f_θ: Combined policy-value network with parameters θ
- $p(a|s)$: Policy output (probability distribution over moves)
- $v(s)$: Value output (predicted game outcome)
- s: Board position

This single network outputs both a move probability distribution and a position evaluation, with shared parameters between the policy and value components.

1.4.3 Self-Play Reinforcement Learning

The training process for AlphaGo Zero consists of three main components that operate in a continuous cycle:

Neural Network Training The network parameters θ are trained to minimize the loss:

$$l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2 \tag{10}$$

Notation Overview

- z: Game outcome (win=+1, loss=-1)
- v: Predicted value by the network
- π: Improved policy from MCTS
- p: Policy output from the network
- c: Regularization parameter
- $\|\theta\|^2$: L2 regularization term

Self-Play with MCTS Games are generated by self-play, with each move selected by MCTS guided by the current neural network:

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}} \tag{11}$$

Notation Overview

- $\pi(a|s)$: Search policy (MCTS output)
- $N(s,a)$: Visit count for action a in state s
- τ: Temperature parameter controlling exploration

Data Generation Each state s_t encountered during self-play is stored along with:

- The MCTS policy π_t
- The game outcome z_t

These data points (s_t, π_t, z_t) are used to train the neural network, completing the reinforcement learning loop.

1.4.4 AlphaGo Zero Algorithm

Algorithm 3 AlphaGo Zero Training Algorithm

Initialize neural network parameters θ randomly
Initialize replay buffer D with capacity N
for iteration $= 1, 2, ...$ **do**
  for episode $= 1, 2, ..., E$ **do**
    $s_1 \leftarrow$ initial board position
    for $t = 1$ until terminal state **do**
      $\pi_t \leftarrow \text{MCTS}(s_t, \theta)$ ▷ Run MCTS guided by neural network
      Sample action $a_t \sim \pi_t$ ▷ Select move based on MCTS policy
      $s_{t+1} \leftarrow$ next state after a_t
    end for
    $z \leftarrow$ final outcome of the game
    for each step t in the episode **do**
      Store (s_t, π_t, z_t) in D where $z_t = \pm z$ depending on player
    end for
  end for
  Sample mini-batch of (s, π, z) from D
  Update θ by gradient descent on loss $(z - v_\theta(s))^2 - \pi^T \log p_\theta(s) + c\|\theta\|^2$
end for

1.4.5 Comparison with Original AlphaGo

Feature	AlphaGo	AlphaGo Zero
Training data	Human expert games + self-play	Only self-play
Neural networks	Separate policy, value, and rollout networks	Single policy-value network
Input features	Handcrafted features + raw board position	Only raw board position
MCTS rollouts	Uses fast rollout policy	No rollouts, relies on value network
Training process	Multiple stages (SL then RL)	End-to-end reinforcement learning
Performance	Defeated world champion	Stronger than all previous versions

2 Summary

Deep Reinforcement Learning represents a powerful combination of deep learning and reinforcement learning techniques. The key innovations discussed in this chapter include:

- **Deep Q-Learning:** Combines Q-learning with deep neural networks, stabilized by target networks and experience replay to learn directly from high-dimensional sensory inputs.
- **AlphaGo:** A breakthrough system that combines deep neural networks with Monte Carlo Tree Search to master the game of Go, using separate policy and value networks trained on expert human games and self-play.
- **AlphaGo Zero:** A more advanced version that learns tabula rasa through pure self-play reinforcement learning, using a unified policy-value network and achieving superhuman performance without human knowledge.

These methods demonstrate the power of deep neural networks as function approximators in reinforcement learning, enabling solutions to previously intractable problems with high-dimensional state spaces. The success of these approaches has led to further advancements in robotics, game playing, and other domains requiring sequential decision making from complex sensory inputs.

3 Further Reading

- Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
- Silver, D., Huang, A., Maddison, C. J., et al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.
- Silver, D., Schrittwieser, J., Simonyan, K., et al. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354-359.
- Silver, D., Hubert, T., Schrittwieser, J., et al. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv preprint arXiv:1712.01815.