Neural Networks
Lecture Notes
Reinforcement learning

Learning Goals

After studying the lecture material you should:

1. understand the difference between critic, actor and actor-critic methods
2. understand what is meant by deep reinforcement learning
3. understand the update rule in Q learning
4. understand how a deep Q network implements Q learning
5. know what is meant by the REINFORCE algorithm
6. understand why solving Go is an important problem and how AlphaGo achieves this
7. understand what Monte Carlo tree search does
8. know the distinction between policy networks and value networks

Notes

Reinforcement learning

Reinforcement learning (RL) is a general-purpose framework for artificial intelligence. It can be used by an agent with the capacity to act. Each action influences the agent's future state and success is measured by a scalar reward signal. RL in a nutshell means to select actions that maximise future reward. This is the essence of an intelligent agent. Deep reinforcement learning is a special case which uses Deep Neural Networks (DNNs) as a component in RL.

Reinforcement learning relies on the interplay between an agent and its environment. At each step t the agent:

- Receives state s_t
- Receives scalar reward r_t
- Executes action a_t

whereas the environment:

- Receives action a_t
- Emits state s_t
- Emits scalar reward r_t

Definitions

We now define a number of important concepts. A *policy*

$$\pi(s, a)$$

specificies the probability of selecting an action a given a state s.

The *return* is the total reward accumulated in an episode:

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1}$$

where γ is a discount factor that discounts future rewards.

Define the *expected return*:

$$\rho^\pi = \mathbb{E}[R \mid \pi]$$

The goal is find the optimal policy π^* which maximizes the expected return.

The value function $V(s)$ (aka state-value function) is the expected total reward when starting in state s:

$$V^\pi(s) = \mathbb{E}_\pi[R \mid s]$$

That is, it measures the quality of being in a certain state.

The *action-value function* $Q(s; a)$ is the expected total reward by taking action a in state s under policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[R \mid s, a]$$

It tells us how good action a is in state s.

Approaches

We can identify three approaches to reinforcement learning:

- Critic only: Estimate the optimal action-value function $Q(s; a)$. Can be used to determine the optimal action at each time step
- Actor only: Search directly for the optimal policy π^*. This is the policy achieving maximum future reward
- Actor-critic: Uses both an actor and a critic to find the optimal policy.

Note that RL can be model-based (i.e. we have a model of the environment) or model-free (we only learn from rewards). RL can be implemented using neural networks.

Critic-based RL

We will here focus on critic-based RL. The action-value function

$$Q : S \times A \to R$$

can be learned using a procedure called Q-learning:

- Before learning has started, Q returns an (arbitrary) fixed value
- Each time the agent selects an action, and observes a reward and a new state that may depend on both the previous state and the selected action, Q is updated.
- The core of the algorithm is a simple value iteration update.
- It assumes the old value and makes a correction based on the new information.

The update equation is given by:

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \eta \left(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right)$$

where r_{t+1} is the reward observed after performing a_t in s_t, and where η is the learning rate.

In essence, the action-value function is one huge lookup table. This becomes impossible to maintain for large (real-world) inputs (most states are never seen). The solution is to represent the action-value function by a function parameterized by $\boldsymbol{\theta}$:

$$Q(s, a; \theta) \approx Q^{\pi}(s, a) \, .$$

Good parameters may allow us to generalize to states we never encountered.

We can choose this parameterized function to be a deep neural network. Define the objective function by the mean-squared TD error:

$$\mathcal{L}(\theta) = \mathbb{E} \left[\left(r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta) \right)^2 \right] \, .$$

This leads to the following Q-learning gradient:

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \mathbb{E} \left[\left(r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta) \right) \frac{\partial Q(s_t, a_t; \theta)}{\partial \theta} \right]$$

As always, we optimise the objective function end-to-end by stochastic gradient descent.

Deep Q networks

Naive Q-learning oscillates or diverges with neural nets, e.g. for the following reasons:

1. Data is sequential
   - Successive samples are correlated (i.e. non-iid - not **i**ndependent and not **i**dentically **d**istributed)
2. Policy changes rapidly with slight changes to Q-values
   - Policy may oscillate
   - Distribution of data can swing from one extreme to another
3. Scale of rewards and Q-values is unknown
   - Naive Q-learning gradients can be largely unstable when backpropagated

However Deep Q networks (DQN) [MKS$^+$15] provide a stable solution to deep value-based RL:

1. Use experience replay
   - Break correlations in data, bring us back to iid setting
   - Learn from all past policies
2. Freeze target Q-network
   - Avoid oscillations
   - Break correlations between Q-network and target
3. Clip rewards or normalize network adaptively to sensible range
   - Robust gradients

Policy-based RL

Instead of parameterizing the value function or action-value function, we can parameterize the policy and directly optimize the parameters:

$$\pi_{\boldsymbol{\theta}}(s, a) \approx \pi(s, a)$$

Optimization requires a policy objective function J which expresses the quality of a chosen policy. This allows gradient ascent steps of the form:

$$\Delta\boldsymbol{\theta} = \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

The *policy gradient theorem* tells us that

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a) Q^{\pi_{\boldsymbol{\theta}}}(s, a)]$$

where $\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a)$ is known as the score function.

The REINFORCE algorithm uses the return at time t as an approximation of the action-value function to obtain:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_t, a_t) R_t]$$

In practice the gradients are computed automatically via backprop:

1. Run neural network for a number of time steps
2. Compute R_t post hoc for each time step
3. Add $-\log \pi_{\boldsymbol{\theta}}(s_t, a_t) R_t$ to the loss
4. Perform backpropagation (through time) to get the policy gradients
5. Update parameters and repeat

Actor-critic RL

Actor-critic uses a critic to approximate the action value function in actor-based methods:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a) f_{\mathbf{w}}(s, a)]$$

See [LIR].

AlphaGo

We now discuss how AlphaGo ([SHM$^+$16]) was able to one of the world's best Go players, Lee Sedol. In general, to solve games, we need to perform tree search, selecting the best action via a minimax algorithm. In games such as chess and Go, this becomes impossible due to the huge search space. AlphaGo uses Monte-Carlo tree search (MCTS) and deep convolutional networks to determine the optimal action. The convolutional networks are conceptually similar to the evaluation function in Deep Blue, except that they are learned and not designed. The tree search procedure can be regarded as a brute-force approach, whereas the convolutional networks provide a level of intuition to the game-play. MCTS works as follows:

- Run many game simulations starting at the current game state and stopping when the game is won by one of the two players.
- At first, the simulations are completely random
- At each simulation, some values are stored, such as how often each node has been visited, and how often this has led to a win.

- These numbers guide the later simulations in selecting actions
- The more simulations are executed, the more accurate these numbers become at selecting winning moves.
- It can be shown that as the number of simulations grows, MCTS converges to optimal play.

Different convolutional networks are trained, of two different kinds:

- *Policy networks*
  - Supervised learning (SL) 13-layer policy network $p_\sigma(a \mid s)$
  - A fast shallow rollout policy network $p_\pi(a \mid s)$
  - A reinforcement learning (RL) policy network $p_\rho(a \mid s)$
- *Value network* $v_\theta(s')$

A policy network provides guidance regarding which action to choose, given the current state of the game.

- Input to the policy network is the whole game board
- The output is a probability value for each possible legal move (i.e. the output of the network is as large as the board).
- Actions (moves) with higher probability values correspond to actions that have a higher chance of leading to a win.

The SL policy network was trained on 30 million positions from games played by human exports, available at the KGS Go server. An accuracy on a withheld test-set of 57% was achieved. Stochastic gradient descent was used to maximize the likelihood of the human move a selected in state s:

$$\Delta_\sigma \propto \frac{\partial \log p_\sigma(a \mid s)}{\partial \sigma}$$

The fast rollout policy network was trained on these data as well. Its accuracy is much lower (24.2%), but is much faster (2 microseconds instead of 3 milliseconds: 1500 times faster). Further improvement is obtained using the RL policy network. The RL policy network parameters were initialized to the SL network parameters. Games are played between the current RL policy network and randomly selected previous iterations of the RL policy network. Define reward function

$$r(s_t) = \begin{cases} 0 & \text{if } t < T \\ +1 & \text{if } t = T \text{ and winner} \\ -1 & \text{if } t = T \text{ and loser} \end{cases}$$

where T is the time step at which the game ended. The RL policy network weights are updated at time step t using SGD in the direction that maximizes the expected outcome:

$$\Delta_\rho \propto \frac{\partial \log p_\rho(a_t \mid s_t)}{\partial \rho} z_t$$

where $z_t = r(s_T)$. I.e. reward games that led to a win and punish games that led to a loss.

A value network provides an estimate of the value of the current state of the game: what is the probability to ultimately win the game, given the current state?

- Input to the value network is the whole game board
- Output of the value network is a single number, representing the probability of a win.

The value network was used to estimate a value function that predicts the outcome from position s using games played that were played using the best RL policy p_ρ (i.e.assuming that this policy was followed by both players). The value function estimated using the RL policy is assumed to approximate the optimal value function under perfect play. The network was trained by regressing on state-outcome pairs (s, z):

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

I.e. try to predict for each state whether it will result in a win or a lose.

AlphaGo combines the policy and value networks in an MCTS algorithm that selects actions by lookahead search. The tree is traversed by simulation (that is, descending the tree in complete games without backup), starting from the root state. Each edge (s, a) of the search tree stores an action value $Q(s, a)$, visit count $N(s, a)$, and prior probability $P(s, a)$. At each time step t of each simulation, an action a_t is selected from state s_t:

$$a_t = \arg\max_a (Q(s_t, a) + u(s_t, a))$$

so as to maximize action value plus a bonus

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)}$$

that is proportional to the prior probability but decays with repeated visits to encourage exploration.

When the traversal reaches a leaf node s_L at step L, the leaf node may be expanded. The leaf position s_L is processed just once by the SL policy network p_σ. The output probabilities are stored as prior probabilities P for each legal action a: $P(s, a) = p_\sigma(a \mid s)$. The leaf node is evaluated in two very different ways:

- by the value network $v_\theta(s_L)$
- by the outcome z_L of a random rollout played out until terminal step T using the fast rollout policy p_π

Evaluations are combined, using a mixing parameter λ, into a leaf evaluation:

$$V(s_L) = (1 - \lambda) v_\theta(s_L) + \lambda z_L \, .$$

At the end of simulation, the action values and visit counts of all traversed edges are updated. Each edge accumulates the visit count and mean evaluation of all simulations passing through that edge:

$$\begin{aligned} N(s, a) &= \sum_{i=1}^{n} 1(s, a, i) \\ Q(s, a) &= \frac{1}{N(s, a)} \sum_{i=1}^{n} 1(s, a, i) V(s_L^i) \end{aligned}$$

where s_L^i is the leaf node from the ith simulation, and $1(s, a, i)$ indicates whether an edge (s, a) was traversed during the ith simulation.

Basically, the policy network reduces tree width and value network reduces tree depth. Once the search is complete, the algorithm chooses the most visited move from the root position.

Required reading material

The textbook by Sutton and Barto is the definitive text on reinforcement learning.
See: https://webdocs.cs.ualberta.ca/~sutton/book/the-book.html

References

[LIR] M Lauer, C Igel, and M Riedmiller. Reinforcement Learning in a Nutshell.

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[SHM$^+$16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, and Koray Kavukcuoglu. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7585):484–489, 2016.