Q-Learning with Function Approximation

Reading

- Sutton, Richard S., and Barto, Andrew G. Reinforcement learning: An introduction. MIT press, 2018.
  - http://www.incompleteideas.net/book/the-book-2nd.html
  - Chapters 9.1-9.4
- David Silver lecture on Value Function Approximation
  - https://www.youtube.com/watch?v=UoPei5o4fps
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Overview

- Classic Q-learning only works for finite-state and finite-action MDPs
  - Can't be used for most real-world problems
  - Even if state-space is finite, it may be extremely large
    - Hard for Q-learning to even visit all states
    - E.g., all images
- In classic Q-learning, Q values are stored in a table
  - Here, we approximate the Q function with another function
    - E.g., line, decision tree, neural network
  - Essentially cast the problem as a regression problem
- Modern deep Q learning is an instantiation of this setting
  - Will talk about the first deep Q network (DQN)

Function Approximation

- The value function maps every state to a value
- Ideally, we want to approximate the value function, e.g., using least squares:

$$MSE = \frac{1}{|S|} \sum_{s \in S} \left(v(s) - \hat{v}(s) \right)^2$$

Function Approximation, cont'd

- What is the first challenge when minimizing squared error?

$$\sum_{s \in S} \left(v(s) - \hat{v}(s) \right)^2$$

- We don't have labels!
  - We don't know the true $v(s)$
  - We have no training data either!
- What is a naïve way of alleviating this challenge?
  - Collect returns G_t for each state, similar to MC
  - But G_t are not the actual values
    - Turns out that minimizing least squares over G_t is still unbiased

Linear Regression as a Function Approximator

- Suppose the approximator $\hat{v}$ is a linear function, i.e.,
$$\hat{v}(\boldsymbol{s}) = \boldsymbol{w}^T \boldsymbol{s}$$
  - where the state $\boldsymbol{s} \in \mathbb{R}^n$ can now be high-dimensional
    - E.g., position, velocity, etc.
- A simple way to train the value function would be to use linear regression with least squares
  - We collect data from multiple episodes
  - Collect all $(S_{t,i}, G_{t,i})$ pairs and treat it as training data
- What are some issues with this?
  - Waiting for returns is very slow, same as in the MC case
  - True value function may not be a linear function

Linear Regression as a Function Approximator

- What are some issues with this?
  - True value function may not be a linear function
    - Will address that with other functions (wink, wink)
  - Waiting for returns is very slow, same as in the MC case
    - We'll come up with an iterative solution, similar to TD learning

Gradient Descent

- In standard ML, we use SGD to minimize non-convex losses
- In RL, we can use SGD to iteratively update the weights of the approximation function
- Recall standard gradient descent (for least squares)
  - Suppose we receive a new pair $(\boldsymbol{x}, y)$

$$\boldsymbol{w}' = \boldsymbol{w} - \alpha\left[2\left(\boldsymbol{w}^T\boldsymbol{x} - y\right)\boldsymbol{x}\right]$$

- Of course, we don't have labeled data in RL
  - If we wait for the final return, could treat a point (S_t, G_t) as labeled data (rename to $(\boldsymbol{s}, g)$ just for simplicity)
  - Gradient descent is now

$$\boldsymbol{w}' = \boldsymbol{w} - \alpha\left[2\left(\boldsymbol{w}^T\boldsymbol{s} - g\right)\boldsymbol{s}\right]$$

  - Turns out this converges to the least squares optimum

Semi-Gradient Methods

$$\boldsymbol{w}' = \boldsymbol{w} - \alpha\left[2\left(\boldsymbol{w}^T\boldsymbol{s} - g\right)\boldsymbol{s}\right]$$

- If we don't wait for the final return, what can we do?
  - Think TD learning
  - Use the immediate reward
  - "Label" becomes the Bellman prediction

$$R_{t+1} + \gamma \boldsymbol{w}^T \boldsymbol{S}_{t+1}$$

- Now the update becomes

$$\boldsymbol{w}' = \boldsymbol{w} - \alpha\left[2\left(\boldsymbol{w}^T\boldsymbol{S}_t - R_{t+1} - \gamma\boldsymbol{w}^T\boldsymbol{S}_{t+1}\right)\boldsymbol{S}_t\right]$$

  - Called a semi-gradient because it's bootstrapped
    - i.e., we use out estimate of $\boldsymbol{w}$ to get the predicted return

Semi-Gradient Methods as Linear Systems

- Rewrite the semi-gradient

$$\begin{aligned}\boldsymbol{w}' &= \boldsymbol{w} - \alpha\left[2\left(\boldsymbol{w}^T\boldsymbol{S}_t - R_{t+1} - \gamma\boldsymbol{w}^T\boldsymbol{S}_{t+1}\right)\boldsymbol{S}_t\right] \\ &= \boldsymbol{w} - \alpha\boldsymbol{S}_t 2(\boldsymbol{S}_t - \gamma\boldsymbol{S}_{t+1})^T\boldsymbol{w} + \alpha 2R_{t+1}\boldsymbol{S}_t \\ &= \boldsymbol{A}\boldsymbol{w} + \alpha\boldsymbol{b}\end{aligned}$$

  - where $\boldsymbol{A} = \boldsymbol{I} - \alpha\boldsymbol{S}_t 2(\boldsymbol{S}_t - \gamma\boldsymbol{S}_{t+1})^T$, $\boldsymbol{b} = 2R_{t+1}\boldsymbol{S}_t$
- When does this system converge?
  - When all eigenvalues of $\boldsymbol{A}$ are in the unit circle
- Similarly, the conditional expectation is

$$\mathbb{E}[\boldsymbol{w}_{t+1}|\boldsymbol{w}_t] = \mathbb{E}[\boldsymbol{A}]\boldsymbol{w}_t + \alpha\mathbb{E}[\boldsymbol{b}]$$

- It can be shown that this converges (see book for proof)
  - Eigenvalues of $\mathbb{E}[\boldsymbol{A}]$ are in the unit circle
  - What does it converge to, however?

Semi-Gradient Linear Methods, cont'd

- The semi-gradient linear method converges to the "best" linear approximation of the value function
  - Where "best" is defined as the projection of the true value function to the set of linear functions
  - Don't have time to make this more formal
- Of course, the "best" linear approximation may not be good enough in many cases
  - Especially in rich settings such as images

Semi-Gradient Polynomial Methods

- How can we learn a polynomial approximation?
  - How does polynomial regression work?
  - Construct polynomial features and learn weights, e.g., $p(x_1, x_2) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$
- Essentially the same as linear regression
  - Need to construct features first
- To approximate a q value, need to stack states and actions
  - E.g., suppose you have n states and 1 action

$$\boldsymbol{S} = [s_1 \dots s_n \; a]$$

- Construct polynomial features, e.g., 2nd order:

$$\boldsymbol{f}(\boldsymbol{S}, a) = [1 \;\; s_1 \;\; \dots \;\; s_n \;\; a \;\; s_1^2 \;\; \dots \;\; s_n^2 \;\; a^2 \;\; s_1 s_2 \dots s_n a]$$

Semi-Gradient Polynomial Methods, cont'd

- To approximate a q value, need to stack states and actions
  - E.g., suppose you have n states and 1 action

$$\boldsymbol{S} = [s_1 \dots s_n \; a]$$

- Construct polynomial features, e.g., 2nd order:

$$\boldsymbol{f}(\boldsymbol{S}, a) = [1 \;\; s_1 \;\; \dots \;\; s_n \;\; a \;\; s_1^2 \;\; \dots \;\; s_n^2 \;\; a^2 \;\; s_1 s_2 \dots s_n a]$$

- Then,

$$\hat{q}(\boldsymbol{s}, a) = \boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{s}, a)$$

- The semi-gradient is now the same as before:

$$\boldsymbol{w}' = \boldsymbol{w} - \alpha \left[2 \left(\boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{S}_t, A_t) - R_{t+1} - \gamma \max_a \boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{S}_{t+1}, a) \right) \boldsymbol{f}(\boldsymbol{S}_t, A_t) \right]$$

  - Note this is still for the case of finite actions

Playing Atari with Deep Reinforcement Learning

Overview

- One of the first paper to apply RL to problems with raw image data
- Authors made use of several recent breakthroughs in ML and RL
  - CNNs with stochastic gradient descent, batch norm, etc.
  - Experience replay
  - New exploration mechanisms
  - Based also on standard Q-learning theory
- Achieved super-human performance on many Atari games that have image inputs
  - Input is 210×160 RGB video at 60Hz

Setup

- Environment is the Atari game engine
- Assumed to be a standard MDP: 5-tuple (S, A, P, R, η)
  - where S is the finite set of states (aka the state space)
  - The true game state is not observed – the observations are instead RGB images
    - Note that this means that the MDP is partially observable since the image does not capture things like velocity
    - A sequence of images should cover the full hidden state
      - Do you see any issues with the MDP assumption in this case?

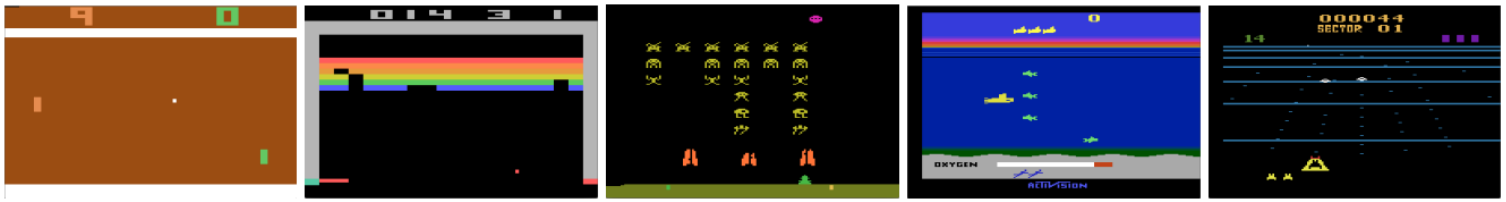


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

Setup, cont'd

- Environment is the Atari game engine
- Assumed to be a standard MDP: 5-tuple (S, A, P, R, η)
  - where A is the finite set of actions (aka the action space)
    - Here, A varies across games but is finite and typically small (< 10)
  - where P is the transition function
    - It is unknown but (mostly) deterministic in Atari games
      - Some environments have added non-determinism to prevent hardcoded policies
  - where η is the initial distribution
    - Some games have randomized initial positions for extra uncertainty
  - where $R: S \times A \times S \to \mathbb{R}$ is the reward function
    - Based on the engine's internal state, so unknown
    - It is deterministic in Atari games
    - Reward structure varies from game to game

MDP Assumption in Atari Environments

- Is the MDP assumption justified?
- Certainly, the environment is unobservable using a single image
  - Can address that by stacking multiple frames as “state”
- Games tend to change as you progress
  - E.g., enemies move faster
- If rewards are based on game scores, then returns from the same state may not be very meaningful
  - If score in Pong is 20:20, vs 0:0, what happens to the returns?

Q-learning learning

- Could use standard value iteration

$$Q_{i+1}(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q_i(S_{t+1}, a') \,|S_t = s, A_t = a]$$

  - As usual, we don't have good estimates of expected R and Q
  - Unstable if we use a single datapoint to estimate expectation

- Recall the Q-learning iteration

$$Q'(S_t, A_t) = Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

  - How do we proceed?

- Could use the semi-gradient method from before

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha \left[2\left(Q(S_t, A_t) - R_{t+1} - \gamma \max_a Q(S_{t+1}, a)\right) \nabla_{\boldsymbol{\theta}} Q(S_t, A_t)\right]$$

  - What issues do you see with this setup?
    - Q-learning can diverge with non-linear function approximators[1]

[1]Tsitsiklis, John N., and Benjamin Van Roy. "An analysis of temporal-difference learning with function approximation." *IEEE Transactions on Automatic Control*. 1997.

Q-learning learning, cont'd

- Recall the Q-learning iteration

$$Q'(S_t, A_t) = Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

- What is the limitation of the semi-gradient method
  - The semi-gradient method only looks at the latest data
  - How can we improve upon that?
  - What if we went back to older data as well?
- Could cast the problem as a "supervised" learning problem
  - Supervised learning is known to be a more stable learning setting

Q-learning learning, cont'd

- Recall the Q-learning iteration

$$Q'(S_t, A_t) = Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

  - The semi-gradient method only looks at the latest data
  - What if we went back to older data as well?
- Could cast the problem as a supervised learning problem
  - Change of notation: $Q \coloneqq Q_{\boldsymbol{\theta}_{i-1}}, Q' \coloneqq Q_{\boldsymbol{\theta}_i}$
  - For each historic tuple $(S_t, A_t, R_{t+1}, S_{t+1})$:
    - Inputs are S_t, A_t
    - (bootstrapped) Labels are $y = R_{t+1} + \gamma \max_a Q_{\boldsymbol{\theta}_{i-1}}(S_{t+1}, a)$
- Can use least-squares loss (or any other regression loss)

$$L(\boldsymbol{\theta}_i, S_t, A_t, y) = \left(Q_{\boldsymbol{\theta}_i}(S_t, A_t) - y \right)^2$$

Convergence of Q-learning with neural networks

- Convergence guarantees are out the window
  - If the algorithm does converge, unclear what the limit is
  - Could be a bad local optimum, as usual
- At the same time, just because some runs may diverge doesn't mean all runs diverge
  - Many techniques have been developed to improve the stability of RL since then
  - Will look at some in these slides

Experience Replay

- An old idea in the RL community[1]
- In standard Q-learning, each data-point is only used once and discarded
  - However, some past experiences are rare and may be costly to obtain (e.g., a crash)
  - Makes sense to train on past experience also
- On the other hand, past experience introduces a bias since the behavior policy may be significantly different from target
  - How can this be a problem?
    - May have too many suboptimal actions
    - "Training data" may be out of distribution
      - Bootstrapped Q-estimates may be bad

[1]Lin, Long-Ji. Reinforcement learning for robots using neural networks. Carnegie Mellon University, 1992.

Experience Replay, cont'd

- Store each experience as a tuple $(S_t, A_t, R_{t+1}, S_{t+1})$
- Can be used as training data in the Q-learning algorithm
- Typically, a buffer is used to store past experience, so that newer experiences gradually replace older ones
  - Also mitigates the bias of using past policies
- Many variants have been developed since the original paper
  - E.g., prioritized experience replay
- In the Atari games paper, they use a vanilla buffer and sample experiences at random
- Experience replay also removes data correlations
  - Semi-gradient method performs updates on correlated data

Training specifics

- As usual, we would like to do gradient descent over the entire dataset, but that's too expensive, so we use SGD
- We have now cast the problem as a supervised regression problem, so all standard hyperparameters need to be chosen
  - Mini-batch size, learning rate, NN architecture, etc.
  - Extra RL hyperparameter is the discount rate γ
    - Typically set to a large value, ≥ 0.9
- Algorithm is *model-free*
  - The underlying MDP is not known or learned
- Algorithm is *off-policy*
  - Training data is generated by a previous version of the policy
    - In essence, historic data is generated by a behavior policy

Data Preprocessing

- Raw images are $210 \times 160 \times 3$
  - Challenging both computationally and statistically
- Images are converted to grayscale, downscaled and cropped, for a final size of $84 \times 84 \times 1$
- 4 consecutive images are stacked together as the input to the NN
  - Effectively the MDP *state*; can capture dynamics such as velocity
  - Final input dimension is thus $84 \times 84 \times 4$

Model Architecture: Deep Q-Network (DQN)

- The intuitive thing to do is build a NN that has one output, i.e., the Q value of the input state-action pair
  - What is the drawback of this?

$$\pi(s) = \max_a Q(s, a)$$

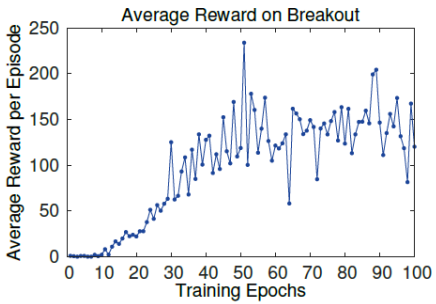
    - Need to compute $Q(s, a)$ for each action a
- The alternative is to have an output layer that has as many neurons as possible actions
  - Problem effectively becomes a classification task in which the action with the highest Q value is picked
- Used a CNN with the following layers
  1. 16 8×8 filters, stride = 4, ReLU
  2. 32 4×4 filters, stride = 2, ReLU
  3. Fully connected layer with 256 neurons, ReLU

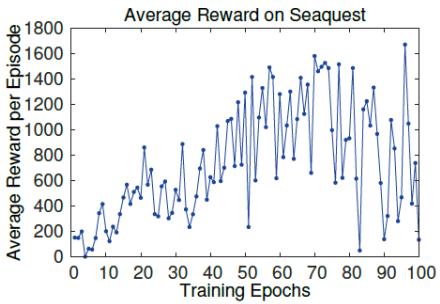
Experiments overview

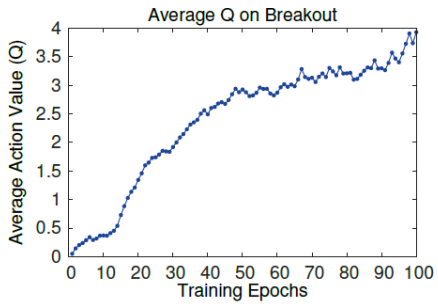
- Performed experiments on 7 Atari games
  - Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, Space Invaders
  - Atari games have become one of the most widely used benchmarks since then
- Used the same architecture and hyperparameters for all games
- Normalized all positive rewards to 1 and all negative rewards to -1
  - Scores vary too much in magnitude
  - Probably exist better ways of normalizing, in order to maintain the relative magnitude

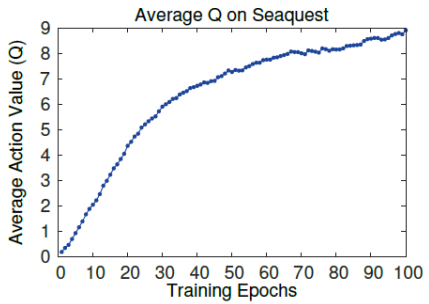
Training Stability

- To this day, stability remains a major challenge in RL
  - Learning quickly diverges even if it seems to have converged
- Rewards per episode vary considerably, though there is an overall trend
- Average Q values output by the NN increase consistently
  - Authors claim this is a good sign, though that is a questionable statement (why?)
    - could be overfitting, selecting wrong actions, maximization bias









Visualizing the Value Function

- One way to judge how good the learned policy is by looking at specific scenarios and looking at the value function output by the NN
- In the example below, we can see that the Q value is high when our sub is about to destroy an enemy sub
- And low when there are no immediate targets
  - Unclear if the relative difference should be that different

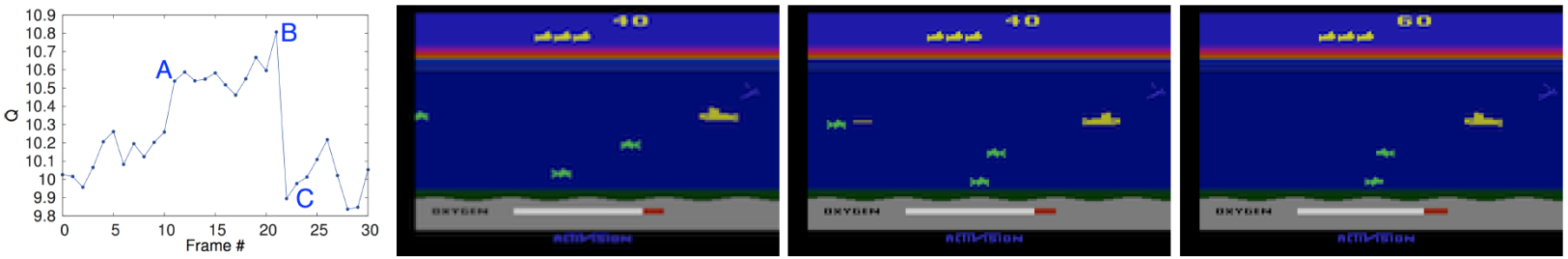


Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

Main Evaluation

- Compared DQN (in terms of average reward) with a number of methods using hand-crafted features from images
  - Used Q-learning-based methods on those images
  - Comparison is unfairly in favor of prior work since features use knowledge that objects have only one color, etc.
- Superhuman performance on some games!
  - Not so surprising anymore
  - Can nowadays achieve superhuman performance on most

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690

Conclusion

- Q-learning has now been applied to a number of hard control tasks, including challenging games such as Go, Starcraft, etc.
- The Atari games paper was one of the first to demonstrate the feasibility of RL in a challenging high-dimensional setting
- However, RL is far from mature
  - stability issues
  - exploration vs. exploitation
  - requires rewards (which makes it hard to use in a real-world setting)
  - robustness issues (!)
- Q-learning only works for discrete actions (more next)