“Real World” Reinforcement Learning

CS4246/CS5446

AI Planning and Decision Making

Please join:

pollev.com/anarayan

This lecture will be recorded!

Topics

- Function Approximation (23.4.1 - 23.4.3)
  - Approximating direct utility estimation
  - Approximating temporal difference learning
  - Deep Reinforcement learning
- Policy Search (23.5)
  - REINFORCE
  - Actor-Critic
  - Correlated Sampling

Recall: Reinforcement Learning (RL)

- Based on:

  A Markov decision process (MDP) $M \triangleq (S, A, T, R)$ consisting of:
  - A set S of states
  - A set A of actions
  - Missing transition function T?
  - Missing reward function R?
- Learning (prediction):
  - Assume policy
  - Solution is an (optimal) utility (value) function: $U: S \rightarrow \Re$ or $V: S \rightarrow \Re$
- Planning (control)
  - Assume utility function or Q-function (action-utility function)
  - Solution is an (optimal) policy: $\pi\colon\ S \rightarrow A$

- Categorization of methods
- Monte Carlo (MC) Learning
  - aka Direct utility estimates
- Adaptive dynamic programming (ADP)
- Temporal difference (TD) learning:
  - Q-learning and SARSA

Quiz

Quiz answer

Scaling

- Number of states grow exponentially with no. of state variables (features)
- Tabular representation (for utility function and Q-function) scales to tens of thousands of states
    - e.g., Number of states in Backgammon & Chess are of the order of 10^{20} & 10^{40}
  - Still considered relatively small as compared to real-world problems!
  - Cannot visit all the states infinitely often
- Approaches to scaling up:
  - Function approximation – approximating utility (value) functions
  - Policy search – systematic search for good policies

Function Approximation

Linear:

- Approximate Monte Carlo Learning
- Approximate temporal difference (TD) learning

Non-linear:

- Deep neural networks

Function Approximation

- Function approximation constructs compact representation of true utility (value) function and Q-function
  - Example: Represent evaluation function for chess as a linear function of features (or basis functions)

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

  - Function approximation uses the n no. of θ parameters to represent a function over a very large number of states
  - RL agent learns θ that best approximate the evaluation (utility) function

Function Approximation

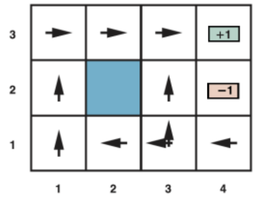
- Function approximation allows generalization from small number of states observed in training data to entire state space
  - Example: Backgammon agent learned to play as well as the best human players by observing only $\approx 10^{12}$ states out of 10^{20} states

- Caveat:
  - If n (no. of parameters) is too small, may fail to achieve good approximation

Linear Function Approximation

Approximate Monte Carlo Learning
Approximate temporal difference (TD) learning

Example: Navigation in Grid World



Source: RN Figure 17.2

- Use:

Compact representation + generalization

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

  - If $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}(1,1) = 0.8$

- Given a collection of trials:
  - Obtain a set of sample values of $\hat{U}_\theta(x, y)$
  - Find the best fit, in the sense of minimizing the squared error, using standard linear regression

Approximating Monte Carlo Learning

- For MC learning we get a set of training samples

$$\big((x_1, y_1), u_1\big), \big((x_2, y_2), u_2\big), \dots, \big((x_n, y_n), u_n\big)$$

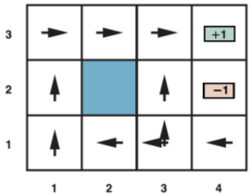
Where u_j is the measured utility of the j^{th} example - observed total reward from state s onward in the j^{th} trial

- This is a supervised learning problem
  - Standard linear regression problem with squared error and linear function
  - Minimize squared-error (loss) function – when partial derivatives wrt to coefficients of linear function are zero

Quiz

Quiz answer

Example: Navigation in Grid World



- Use:

$$\widehat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

  - If $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\widehat{U}(1,1) = 0.8$

- After a single trial:
  - Suppose we run a trial and the total reward obtained starting at $(1,1)$ is 0.4. This suggests that $\widehat{U}(1,1)$ currently 0.8, is too large and must be reduced.
  - How?

Online Learning

ComboBox[illegible]
onward in the jth trial

- To minimize the squared error using online learning
  - Update the parameters after each trial.
- For the j^{th} example, take a step in the direction of the gradient of error function:

Half the squared difference of
predicted total and actual total

$$\boxed{\mathcal{E}_j(s)} = \frac{\left(\hat{U}_\theta(s) - \boxed{u_j(s)}\right)^2}{2}.$$

Widrow-Hoff Rule Or
Delta rule
for online least squares

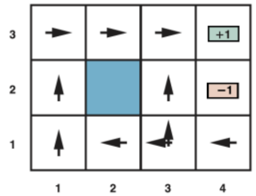
- For parameter θ_i:

$$\theta_i \leftarrow \theta_i - \alpha \boxed{\frac{\partial \mathcal{E}_j(s)}{\partial \theta_i}} = \theta_i + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

ComboBox[illegible]

- Notes:
  - Changing the parameters θ_i in response to an observed transition between two states also changes the values of $\hat{U}_\theta$ for every other state! $\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$
  - Function approximation allows a reinforcement learner to generalize from its experiences.

Example: Navigation in Grid World



- Use:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

  - If $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}(1,1) = 0.8$
- After a single trial:
  - Suppose we run a trial and the total reward obtained starting at $(1,1)$ is 0.4.
  - Apply delta rule for online least squares to the example where $\hat{U}_\theta(x, y)$ is 0.8 and $u_j(1,1)$ is 0.4.
  - Parameters θ_0, θ_1, and θ_2 are all decreased by 0.4α, which reduces the error for $(1,1)$.
- Applying Delta Rule for linear function approximator:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial \mathcal{E}_j(s)}{\partial \theta_i} = \theta_i + \alpha \left(u_j(s) - \hat{U}_\theta(s) \right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

$$\theta_0 \leftarrow \theta_0 + \alpha \left(u_j(s) - \hat{U}_\theta(s) \right)$$

$$\theta_1 \leftarrow \theta_1 + \alpha \left(u_j(s) - \hat{U}_\theta(s) \right) x$$

$$\theta_2 \leftarrow \theta_2 + \alpha \left(u_j(s) - \hat{U}_\theta(s) \right) y$$

Approximating Temporal Difference Learning

- For TD learning the same idea of online learning can be applied
    - Adjust the parameters to reduce the temporal difference between successive states
  - For utilities:

$$\theta_i \leftarrow \theta_i + \alpha\left[\boxed{R(s,a,s') + \gamma \hat{U}_\theta(s')} - \hat{U}_\theta(s)\right]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

  - For Q-learning:

TD target

Update parameter to reduce temporal difference

$$\theta_i \leftarrow \theta_i + \alpha\left[\boxed{R(s,a,s') + \gamma \max_a \hat{Q}_\theta(s',a')} - \hat{Q}_\theta(s,a)\right]\frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

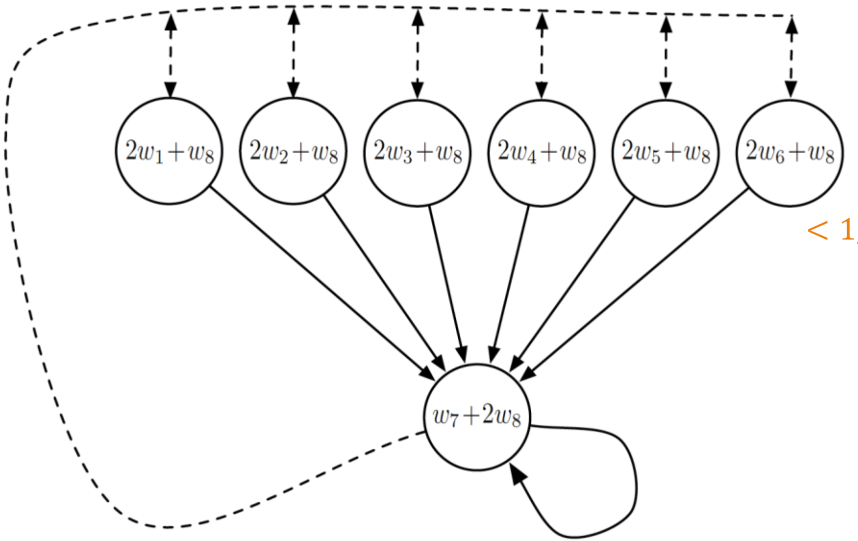
- Notes:
  - Also called semi-gradient as the target is not a true value, depends on θ
  - For passive TD learning, update rule converges for linear function when using on-policy

The Deadly Triad

- Challenges for active learning and non-linear functions:
  - Instability and divergence may arise when following 3 elements are combined:
- Function approximation:
  - Required when state space is large, e.g., using linear function approximation or deep neural nets
- Bootstrapping:
  - Using existing estimates as targets, e.g., in TD, rather than complete returns like in MC methods
- Off-policy training:
  - Training on transitions other than those produced by the target policy

Example: Baird's Counterexample

- Off-policy TD with linear function approximation diverges



$r(s,a,s') = 0$

$\widehat{U}_\theta(s)$=0

$< 1,1,1,1,1,1,10,1 >^T$

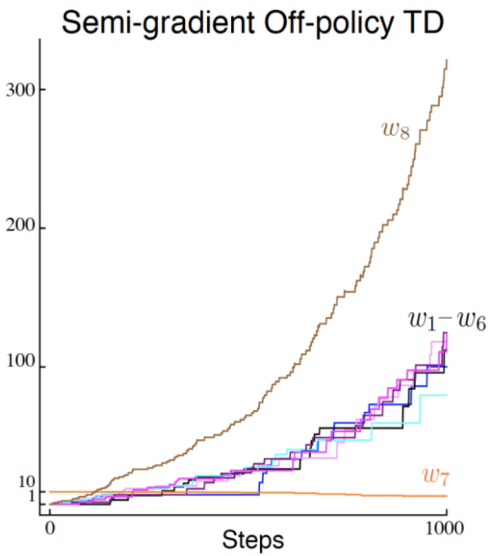
$\pi(\text{solid}|\cdot) = 1$

$b(\text{dashed}|\cdot) = 6/7$

$b(\text{solid}|\cdot) = 1/7$

$\gamma = 0.99$

Source: SB Figure 11.1



Source: SB Figure 11.2

Catastrophic Forgetting

- Problems with over-training
  - Forgotten about the “dangerous zones” of the learning regions
- Potential solution: Experience replay
  - Retain “relevant” training examples or trajectories from entire learning process
  - Replay those trajectories to ensure utility or value function is still accurate for parts of state space it no longer visits

Non-Linear Functional Approximation

Deep reinforcement learning

Deep Reinforcement Learning

- Deep neural network in function approximation
  - Discovers useful features by itself
  - “Transparent” to show selected features if last network layer is linear

$$\widehat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

  - Parameters are all weights in all the layers of the network
  - Gradients required the same for supervised learning, computed by back propagation

Deep Reinforcement Learning

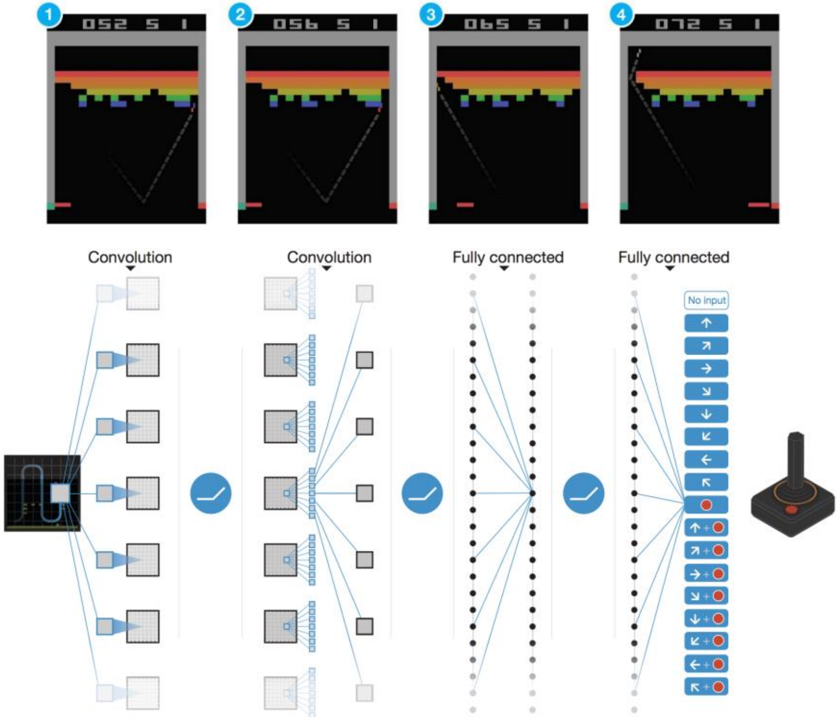
- **Learning the parameters:**
  - For utilities:

$$\theta_i \leftarrow \theta_i + \alpha\left[R(s,a,s') + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)\right]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

  - For Q-learning:

$$\theta_i \leftarrow \theta_i + \alpha\left[R(s,a,s') + \gamma \max_{a} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)\right]\frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

Deep Q-learning for Atari Games[1]



Video: https://youtu.be/TmPfTpjtdgg

[1]Mnih, V., et al., *Human-level control through deep reinforcement learning.* Nature, 2015. **518**(7540): p. 529-533.

Deep Q-Network (DQN)

- Uses deep neural networks with Q-learning to play 49 Atari games
  - Online Q-learning with non-linear function approximators is unstable and may diverge
- DQN uses experience replay with fixed Q-targets:
  - Take action a_t using ϵ-greedy policy
  - Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in a large buffer D of most recent transitions
  - Sample a random mini-batch (s, a, r, s') from D
  - Set targets to $r + \gamma \max_{a'} Q(s', a', \theta^-)$
  - Do gradient step on the minibatch squared loss w.r.t θ – Optimize MSE btw Q-network and Q-learning targets:

$$\mathcal{L}_i(\theta_i) = E_{s,a,r,s' \sim D_i}\left[\left(\boxed{r(s,a,s') + \gamma \max_{a'} Q(s', a'; \theta_i^-)} - Q(s,a;\theta_i)\right)^2\right]$$

  - Set θ^- to θ every C steps
- Experience replay and fixed target
  - Help reduce instability by making input less correlated

Deep Q-Network (DQN)

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 $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Source: Mnih et al., Nature 2015

- Optimise MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

Source: Silver, D. Lecture 6 Notes on RL, 2015.

← Take action a_t using ϵ-greedy policy

← Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in a large buffer D

← Sample a random mini-batch (s, a, r, s') from D

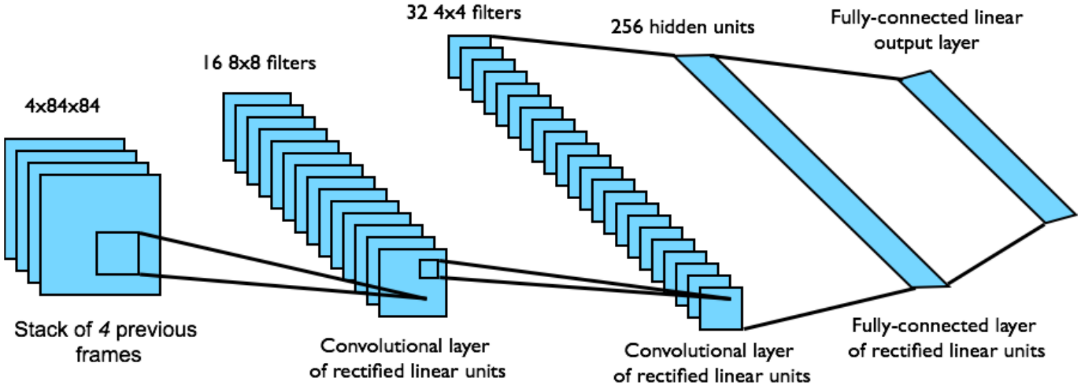
← Set targets to $r + \gamma \max_{a'} Q(s', a', \theta^-)$

← Do gradient step on minibatch squared loss w.r.t θ

← Set θ^- to θ every C steps

DQN in Atari

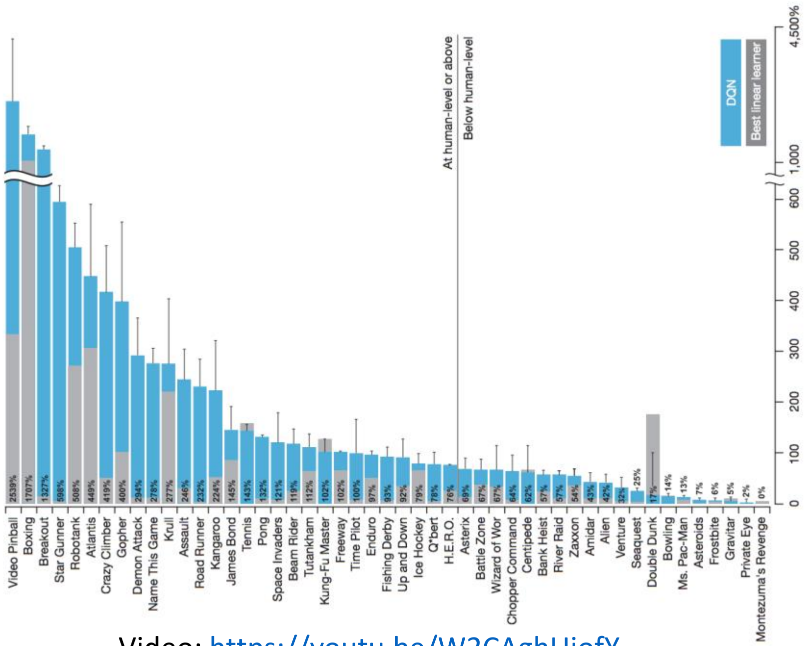
- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick positions
- Reward is change in score for that step



Network architecture and hyperparameters fixed across all games

Source: Silver, D. Lecture 6 Notes on RL, 2015, p 36.

Deep Q-learning for Atari Games[1]



Video: https://youtu.be/W2CAghUiofY

[1]Mnih, V., et al., *Human-level control through deep reinforcement learning.* Nature, 2015. **518**(7540): p. 529-533.

AlphaGo[2]

Video: https://youtu.be/WXuK6gekU1Y

[2]Silver, D., et al., *Mastering the game of Go with deep neural networks and tree search.* Nature, 2016. **529**: p. 484+.

AlphaGo[2]

- Go Game
  - Space size of about 10^{170} with branching factor that starts at 361
  - Difficult to define good evaluation function
  - Need function approximation to represent value and policy functions
- AlphaGo[2] used deep reinforcement learning to beat best human players
  - A Q-function with no look-ahead suffices for Atari games
  - Go requires substantial lookahead.
  - AlphaGo learned both a value function and a Q-function that guided its search by predicting which moves are worth exploring.
  - Q-function implemented as a convolutional neural network – accurate enough by itself to beat most amateur human players without search.

[2] Silver, D., et al., *Mastering the game of Go with deep neural networks and tree search.* Nature, 2016. **529**: p. 484+.

Deep Reinforcement Learning: Reality

"

Despite its impressive successes, deep RL still faces significant obstacles: it is often difficult to get good performance and the trained system may behave very unpredictably if the environment differs even a little from the training data.

Deep RL is rarely applied in commercial settings. It is, nonetheless, a very active area of research.

"

Russell, Stuart; Norvig, Peter. Artificial Intelligence (Pearson Series in Artificial Intelligence) (p. 858). Pearson Education. 4th Edition.