Reinforcement Learning

UMaine COS 470/570 – Introduction to AI

Spring 2019

Created: 2019-04-23 Tue 13:56

1

Why reinforcement learning?

2.1

Why reinforcement learning?

- Supervised learning: need labeled examples
- Unsupervised learning: maybe learn structure, but…
- Often:
  - Do not have labeled examples
  - Have to do *something* – i.e., make some decision – before training is complete
  - But have *some* feedback about how agent is doing

2.2

Framing the problem

- *Reinforcement* of agent's actions via *rewards*
- Current state → choose action → new state + reward
  - Let $R(s)$ = reward for state s
  - Many states may have 0 reward:
$$s_0 \to a_1 \to s_1 \to a_2 \to \cdots a_n \to s_n$$
$$R(s_0) = R(s_1) = \cdots R(s_{n-1}) = 0$$
  - E.g., games
  - Instance of *credit assignment* problem
- Instance of *sequential decision problem*

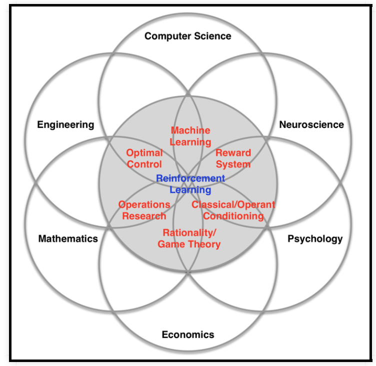
2.3

Reinforcement learning

- Rewards
- But no *a priori* knowledge of rewards, model (transition function)
- E.g.:
  - Given an unfamiliar board and pieces, alternate moves with opponent – only feedback is "you win" or "you lose"
  - Robot has to move around campus delivering mail, but doesn't know anything about campus, or delivering mail, or people, or… feedback: "good robot", "ouch!", falls over, etc.

2.4

Reinforcement learning



(From https://icml.cc/2016/tutorials/deep_rl_tutorial.pdf)

2.5

Learning approaches

- Learn utilities of states
  - Use to select action to maximize expected outcome utility
  - Needs model of environment, though to know s' resulting from taking action a in s
- Policy learning (reflex agent):
  - Directly learn $\pi(s)$: which action to take in s, bypassing $U(s)$
- *Q-learning*:
  - Learn an *action-utility function* Q
  - $Q(a, s)$ is the value (utility) of action a in state s
  - Model-less learning

2.6

Learning approaches

- *Passive learning:*
  - Policy is fixed
  - Task: learn $U(s)$ (or utility of state-action pairs)
  - Maybe learn model
- *Active learning:*
  - Has to learn what to do
  - May not even know what its actions do
  - Involves *exploration*

2.7

Passive reinforcement learning

3.1

Passive reinforcement learning

- Policy $\pi(s)$ is fixed
- Task: See how good policy is by learning:
$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$
- Doesn't know:
  - transition model $P(s' | s, a)$
  - reward function $R(s)$

3.2

Passive reinforcement learning

- Policy $\pi(s)$ is fixed
- Task: See how good policy is by learning:
$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$
- Doesn't know:
  - transition model $P(s' | s, a)$
  - reward function $R(s)$
- Approach:
  - Do series of *trials*
  - Each: start at start, follow policy to terminal state
  - Percepts ⇒ new state s', $R(s')$

3.2

Passive reinforcement learning

- Policy $\pi(s)$ is fixed
- Task: See how good policy is by learning:
$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$
- Doesn't know:
  - transition model $P(s' | s, a)$
  - reward function $R(s)$
- Approach:
  - Do series of *trials*
  - Each: start at start, follow policy to terminal state
  - Percepts ⇒ new state s', $R(s')$
- Stochastic transitions ⇒ different histories from same π

3.2

Direct estimation of $U^{\pi}(s)$

- Woodrow & Huff (1960 – adaptive control theory
- $U(s)$ = remaining reward = *reward-to-go*
- View: each trial ⇒ one sample of reward-to-go for each visited state
- Reduces reinforcement learning to supervised learning
- But although $R(s)$ and $R(s')$ are independent…
- …$U(s)$ and $U(s')$ are *not independent* – (cf. Bellman equation)
- Misses opportunities for learning – e.g.,
  - See s_1 for first time, it leads to known state s_2 that is known
  - Bellman: $U(s_2)$ tells us something about $U(s_1)$
  - Direct estimation: only $R(s1)$ matters
- Hypothesis space > needs to be

3.3

Adaptive dynamic programming

- First learn model of transition function $P(s'|s,a)$ from trials
- Now you have an MDP
- Solve it as per sequential decision process
- Could use Bayesian approaches to make this better (see R&N, 21.2.2)

3.4

Temporal difference learning

- Use the Bellman equations directly:
  $$U^{\pi}(s) = R(s) + \gamma \sum_{s}{}' (P(s'|s,\pi(s))U^{\pi}(s')$$
- General idea:
  - Start with no known $U(\cdot)$
  - Iterate:
    - Take step $\pi(s)$ to give s'
    - If s' is unknown state, use $R(s')$ as $U(s')$
    - Use $U(s')$ to adjust $U(s)$:
      $U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$

3.5

Temporal difference RL algorithm

function PASSIVE-TD-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    U, a table of utilities, initially empty
    N_s, a table of frequencies for states, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if s' is new **then** $U[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$
  if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$
  return a

3.6

Active reinforcement learning

Active reinforcement learning

Active reinforcement learning

- What if we not only don't know:
  - $P(s' \mid s, a)$
  - $R(s)$

  ...also don't know $\pi(s)$?

Active reinforcement learning

- What if we not only don't know:
  - $P(s' \mid s, a)$
  - $R(s)$

  ...also don't know $\pi(s)$?

- One approach: use passive learning, but for all possible actions
  - Use the adaptive dynamic programming agent, but for all $a \in A(s)$ at each state
  - This gives the transition model
  - Use value iteration or policy iteration $\Rightarrow U(s)$

Active reinforcement learning

- What if we not only don't know:
  - $P(s' | s, a)$
  - $R(s)$

  ...also don't know $\pi(s)$?
- One approach: use passive learning, but for all possible actions
  - Use the adaptive dynamic programming agent, but for all $a \in A(s)$ at each state
  - This gives the transition model
  - Use value iteration or policy iteration $\Rightarrow U(s)$
- Produces *greedy agent*:
  - Once good terminal state found, tends to keep using policy that found it
  - Seldom in practice converges to optimal policy π^*!

4.2

Greedy agent

- Why doesn't greedy agent converge?
- Only *exploits* known path – assumes model is good
- But model created based on learned π – leaves some states unexplored
- Actions leading to those states allow better learning of model
- Which allows better estimation of $U(s)$, π^*
- Have to balance exploitation with *exploration*

4.3

Incorporating exploration

- Using value iteration to get $U(s)$
- Now think of $U^+(s)$, the optimistic estimate of utility of s
- Design an exploration function $f(u, n)$ where:
  - u - expected utility of some new state s'
  - n - number of times action a (expected to lead to s' from s) has been tried in s
- New iteration function for (optimistic) utility:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s' | s, a) U^+(s'), N(s, a)\right)$$

  where $N(s, a) =$ number of times s has been tried in a

4.4

Q-learning

- Instead of learning utilities, learn $Q(s, a)$: utility of action a in s
- *Model-free*: doesn't have to know $U(s)$ at all
- Could do this:

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' | s, a) \max_{a'} Q(s', a')$$

  - A Bellman equation, but for \(\) pairs rather than s
  - Could use in adaptive dynamic programming as iteration method
  - But this isn't really model-free – need $P(s' | s, a)$
- Instead, use temporal difference method:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

4.5

Q-learning agent

function Q-LEARNING-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a, r \leftarrow s', \mathrm{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a']), r'$
  return a

SARSA

- *State-action-reward-state-action* (SARSA) - similar to Q-learning
  $$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a)$$
- Here, a' is action actually taken in s'
- Q-learning: uses *best* action from s'
- Still model-free, but have *some* policy that leads to choosing a'
- *Off-policy* vs *on-policy* algorithms
  - Off-policy algorithms pay no attention to any policy π – e.g., Q-learning
  - On-policy: actions with respect to some policy
- Off-policy more flexible...
- ...but if policy is constrained by others (e.g.), may be better to go with *realistic* actions taken rather than best possible

So...Q-learning or model-learning?

- R&N: “This is an issue at the foundations of artificial intelligence.”
- More generally: do we need models to behave intelligently, or not?
- Traditionally: model (most symbolic AI)
- Lately: model-free (e.g., neural networks)

Generalized RL

Generalized RL

Generalized RL

- So far:
  1. Learn $U(s)$
  2. Learn $Q(s, a)$

Generalized RL

- So far:
  1. Learn $U(s)$
  2. Learn $Q(s, a)$
- But what if state space is very large or infinite?

Generalized RL

- So far:
  1. Learn $U(s)$
  2. Learn $Q(s, a)$
- But what if state space is very large or infinite?
- Instead: Learn *function* approximating $U(s)$ or $Q(s, a)$ – $\widehat{U}(s)$ or $\widehat{Q}(s, a)$

Generalized RL

- So far:
  1. Learn $U(s)$
  2. Learn $Q(s, a)$
- But what if state space is very large or infinite?
- Instead: Learn *function* approximating $U(s)$ or $Q(s, a)$ – $\widehat{U}(s)$ or $\widehat{Q}(s, a)$
- E.g., approximate $U(s)$ by linear combination of features
  - Static eval for chess, etc.
  - $\widehat{U}(s) = \theta_1 f_1(s) + \cdots \theta_n f_n(s)$
  - Just learn θ_i values
  - For chess, $> 10^{40}$ states – now only learn n values, where $n \ll 10^{40}$

5.2

Generalized RL

- So far:
  1. Learn $U(s)$
  2. Learn $Q(s, a)$
- But what if state space is very large or infinite?
- Instead: Learn *function* approximating $U(s)$ or $Q(s, a)$ – $\widehat{U}(s)$ or $\widehat{Q}(s, a)$
- E.g., approximate $U(s)$ by linear combination of features
  - Static eval for chess, etc.
  - $\widehat{U}(s) = \theta_1 f_1(s) + \cdots \theta_n f_n(s)$
  - Just learn θ_i values
  - For chess, $> 10^{40}$ states – now only learn n values, where $n \ll 10^{40}$
- Not just save space: allows *generalization*

5.2

Generalized RL

- So far:
  1. Learn $U(s)$
  2. Learn $Q(s, a)$
- But what if state space is very large or infinite?
- Instead: Learn *function* approximating $U(s)$ or $Q(s, a)$ – $\widehat{U}(s)$ or $\widehat{Q}(s, a)$
- E.g., approximate $U(s)$ by linear combination of features
  - Static eval for chess, etc.
  - $\widehat{U}(s) = \theta_1 f_1(s) + \cdots \theta_n f_n(s)$
  - Just learn θ_i values
  - For chess, $> 10^{40}$ states – now only learn n values, where $n \ll 10^{40}$
- Not just save space: allows *generalization*
- On the other hand: maybe we choose wrong hypothesis space

5.2

Generalized RL

- So – how to approach?
- One way:
  - Choose utility approximator
  - Run a series of trials
  - Find best fit of feature weights to data (min. squared error)
  - ⇒ Supervised learning

5.3

Generalized RL

- Better to use *online* algorithm for RL
  - Estimate $\widehat{U}(s)$ (random to start)
  - Run trial
  - Adjust $\widehat{U(s)} accordingly
- How to adjust?
  - Compute gradient with respect to each parameter
  - Move parameter down gradient
  - Sound familiar?

5.4

Generalized RL: Delta rule

5.5

Generalized RL: Delta rule

- *Widrow-Hoff rule* (*delta rule*)

5.5

Generalized RL: Delta rule

- *Widrow-Hoff rule* (*delta rule*)
- For trial j, observed utility $u_j(s)$, and parameters θ, let error:
  \begin{eqnarray*} E_j(s) &=& (\widehat{U}$_\theta$(s) - u_j(s))^2/2
  ∇E_{θ_i} &= & ∂ $E_j/\partial\theta_i$
  θ_i &←& θ_i - α\frac{\partial E_j(s)}{\partial \theta_i}
  &←& θ_i + α(u_j(s) - \widehat{U}$_\theta$(s)) \frac{∂ \widehat{U}$_\theta$(s)}{∂ θ_i}

5.5

Generalized RL: Delta rule

- *Widrow-Hoff rule* (*delta rule*)
- For trial j, observed utility $u_j(s)$, and parameters θ, let error:
  \begin{eqnarray*} E_j(s) &=& (\widehat{U}$_\theta$(s) - u_j(s))^2/2
  ∇ E_{θ_i} &= & ∂ E_j/∂θ_i
  θ_i &←& θ_i - α\frac{\partial E_j(s)}{\partial \theta_i}
  &←& θ_i + α(u_j(s) - \widehat{U}$_\theta$(s)) \frac{∂ \widehat{U}$_\theta$(s)}{∂ θ_i}
- **The $\boldsymbol{\theta}$ parameters can also be the weights in a neural network!**

5.5

Deep reinforcement learning

6.1

Deep reinforcement learning

- In RL, we can learn:
  - $U(s)$
  - $Q(s, a)$
  - $\pi(s)$
- In generalized RL: learn parameters θ of functions approximating U, Q, π
- Inputs: percepts
- Outputs: actions
- Have to know form of function (hypothesis space)
- Deep learning: excels in learning nonlinear functions mapping inputs to outputs
- Maybe combine RL and DL ⇒ *deep reinforcement learning*

6.2

Model learning

- Need to learn $U(s)$, $P(s'|s, a)$
- Either/both can be learned by DL
- DL is responsible for understanding what the state *s* is given percepts
- E.g., for $U(s)$:
  - Weights θ
  - Many trials
  - Each trial:

$$\theta \leftarrow \underset{\theta}{\text{argmin}} \frac{1}{2} \sum_i ||U_\theta^\pi(s_i) - y_i||^2$$

  - Compute y the target value via Monte Carlo method
    - Using policy, go from s to end to find utility
    - Average multiple trials
- Or use Bellman equation, with temporal difference
  - Now, however: don't store $U(s)$ – adjust the NN's weights

6.3

Deep Q-learning

- Can we use deep learning to do model-free Q-learning?
- *Deep Q-network* (DQN):
  - Function approximating $Q(s, a)$: $Q(s, a; \theta)$
  - Here, θ are the parameters: the weights of NN
- Problem: Can't just treat as supervised learning problem
  - Q-learning isn't stable w/ DL
  - Q-learning balances exploitation with exploration
  - ⇒ Input space, actions – changing as we explore more
  - As these change, target value for Q changes
  - So net's input space, output space changing rapidly as explore

(Some material from here, also Mnih *et al.*, 2013)

6.4

DQN

- Train by minimizing *sequence* of loss functions:
$$L_i(\theta_i) = E\left[(y_i - Q(s, a; \theta_i))^2\right]$$
where y_i is the target:
$$y_i = E\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a\right]$$
- s here is a *sequence* of states here (so not Markovian?)
- Expected value for L based on probability function over sequences of states
- Target determined from emulator/world + *previous* θ
- Optimize loss function $L_i(\theta_i)$ with parameters from previous iteration θ_{i-1} held fixed
- Target depends on weights – not like supervised learning
- Gradient
$$\nabla_{\theta_i} L_i(\theta_i) = E\left[(r + \gamma Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \nabla_{\theta_i} Q(s, a; \theta_i)\right]$$
- Use stochastic gradient descent

6.5

DQN

- As implemented by Mnih *et al.* (2013) at DeepMind
- Use past experience, past weights to slow down changes in input, output space
- Allows gradual learning of Q
- *Experience replay*:
  - Keep last million or so <s, a, r> in *replay buffer*
  - Train using batches from here
- *Target network:*
  - Use *two* networks
  - Update one constantly
  - Other (target net): synchronize with other occasionally
  - Target network provides Q values instead of using the rapidly-changing one
  - So: Q from old weights trains new weights, then new becomes old occasionally

6.6

DQN algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

(From Mnih *et al.*, 2013)

6.7

DQN results

- DeepMind's early work:Atari games
- Played most better than any other RL program, some better than humans
- Input: raw frames (201 × 160 pixels, 128 colors)
- Output: actions
- Pre-processing: convert to grayscale, downsample + crop to rough game area
- Convolutional neural network
  - First layer: 16 8 × 8 filters, stride 4, ReLu
  - Second layer: 32 4 × 4 filters, stride 2, ReLu
  - Last hidden layer: fully-connected, 256 ReLu units
  - Output: Fully connected linear layer, single output per valid action

Example: ConvNetJS

From Karpathy @ Stanford's Deep Learning in Your Browser site

Double DQN

- Q-learning problem: can be overly optimistic on value of Q due to approximation error
- Update function for Q-learning
  $$\theta_{t+1} = \theta_t + \alpha(Y_t - Q(s_t, a_t; \theta_t))\nabla_{\theta_t} Q(s_t, a_t; \theta_t)$$
  where:
  $$Y_t \equiv R(s_{t+1}) + \gamma \max_a Q(s_{t+1}, a; \theta_t)$$
- For DQN:
  $$Y_t \equiv R(s_{t+1}) + \gamma \max_a Q(s_{t+1}, a, \theta_t^-)$$
- $\max_a$ portion: target weights select and evaluate best action *it* would take
- May not be action that online net selects ⇒ possible overestimate

(From van Hasselt *et al.* (2016): Deep Reinforcement Learning with Double Q-Learning, *AAAI-16*.)

Double DQN

- Best if "best action" is one online net would choose...
- ...but estimated target is per target net
- ⇒ Double DQN target:
  $$Y_t \equiv R(s_{t+1} + \gamma Q(s_{t+1}, \operatorname*{argmax}_a Q(s_{t+1}, a; \theta_t); \theta_t^-)$$
- Much better learning due to fewer overestimates

Dueling DQN

- Sometimes:
  - No action is necessary in a state; or
  - It doesn't matter much which action is done; or
  - One action is better than another in a range of states.
- $Q(s, a)$ conflates assessing states and assessing values (as would $U(S)$, then picking action)
- What if split $Q(s, a) = V(s) + A(s, a)$
  - Value of state s $V(s)$ is basically $U(s)$
  - Advantage of action a in state s $A(s, a)$ is state-dependent action worth

(From Wang et al., Dueling network architectures for deep reinforcement learning, 2016)

6 . 12

Dueling DQN

- Learn $V(S)$ and $A(s, a)$ separately, then recombine to give $Q(s, a)$:

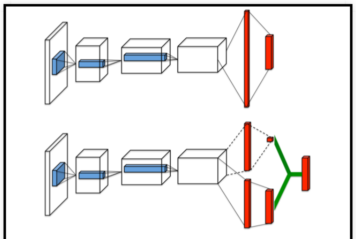


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

6 . 13

Dueling DQN

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \right). \quad (9)$$

6 . 14

Dueling DQN advantage

- Learn values of states when actions don't matter
- Don't worry about choosing an action when it doesn't matter

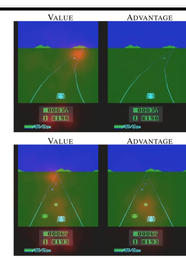


Figure 2. See, attend and drive: Value and advantage saliency maps (red-tinted overlay) on the Atari game Enduro, for a trained dueling architecture. The value stream learns to pay attention to the road. The advantage stream learns to pay attention only when there are cars immediately in front, so as to avoid collisions.

(Source: here.)