NPFL099 Statistical Dialogue Systems
7. Dialogue Management (2)
Action Selection/Policy

Ondřej Dušek, Vojtěch Hudeček & Tomáš Nekvinda
http://ufal.cz/npfl099
15. 11. 2020
Action selection: Recap

- Action selection: deciding what to do (or say) next
  - based on dialogue state (i.e. uses tracking output)
  - follows a **policy** towards an end goal
- FSM, frames, rule-based
- **trained policies**: typically with RL
  - explore more different paths than supervised
  - plan ahead – optimize for the whole dialogue, not just 1 turn
- RL: MDP formalism – agent in an environment, **state-action-reward**
  - POMDP = MDP with continuous states
  - trained with user simulator

(from Milica Gašić’s slides)

(Sutton & Barto, 2018)
Reinforcement learning: Definition

• RL = finding a **policy that maximizes long-term reward**
  • unlike supervised learning, we don’t know if an action is good
  • immediate reward might be low while long-term reward high

\[ R_t = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \]

\[ \gamma \in [0, 1] = \text{discount factor} \]
(Immediate vs. future reward trade-off)

\[ \gamma < 1 : R_t \text{ is finite (if } r_t \text{ is finite)} \]
\[ \gamma = 0 : \text{greedy approach (ignore future rewards)} \]

• state transition is stochastic \(\rightarrow\) maximize **expected return**

\[ \mathbb{E}[R_t | \pi, s_0] \]
expected \(R_t\) if we start from state \(s_0\) and follow policy \(\pi\)
State-value Function

• Using return, we define the **value of a state** \( s \) under policy \( \pi \): \( V^\pi(s) \)
  
  • Expected return for starting in state \( s \) and following policy \( \pi \)
  
  • Return is recursive: \( R_t = r_{t+1} + \gamma \cdot R_{t+1} \)

• This gives us a recursive equation (Bellman Equation):

\[
V^\pi(s) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t r_{t+1} | \pi, s_0 = s \right] = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} p(s'|s, a)(r(s, a, s') + \gamma V^\pi(s'))
\]

• \( V^\pi(s) \) defines a **greedy policy**: 

\[
\pi(s, a) := \begin{cases} 
1 & \text{for } a = \arg \max_a \sum_{s' \in S} p(s'|s, a)(r(s, a, s') + \gamma V^\pi(s')) \\
0 & \text{otherwise}
\end{cases}
\]
**Action-value (Q-)Function**

- $Q^\pi(s, a)$ – return of taking action $a$ in state $s$, under policy $\pi$
  - Same principle as value $V^\pi(s)$, just **considers the current action, too**
  - Has its own version of the Bellman equation

$$
Q^\pi(s, a) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t r_{t+1} | \pi, s_0 = s, a_0 = a \right] = \sum_{s' \in S} p(s'|s, a) \left( r(s, a, s') + \gamma \sum_{a' \in A} Q^\pi(s', a') \pi(s', a') \right)
$$

- $Q^\pi(s, a)$ also defines a greedy policy:

$$
\pi(s, a) := \begin{cases} 
\frac{1}{\# \text{of } a'} & \text{for } a = \arg \max_a Q^\pi(s, a) \\
0 & \text{otherwise}
\end{cases}
$$

again, “actions that look best for the next step”

simpler: no need to enumerate $s'$, no need to know $p(s'|s, a)$ and $r(s, a, s')$

but $Q$ function itself tends to be more complex than $V$
Optimal Policy in terms of $V$ and $Q$

- **optimal policy** $\pi^*$ – one that maximizes expected return $\mathbb{E}[R_t | \pi]$
  - $V^\pi(s)$ expresses $\mathbb{E}[R_t | \pi]$ → use it to define $\pi^*$

- $\pi^*$ is a policy such that $V^{\pi^*}(s) \geq V^{\pi'}(s)$ $\forall \pi', \forall s \in S$
  - $\pi^*$ always exists in an MDP (need not be unique)
  - $\pi^*$ has the **optimal state-value function** $V^*(s) := \max_{\pi} V^{\pi}(s)$
  - $\pi^*$ also has the **optimal action-value function** $Q^*(s, a) := \max_{\pi} Q^{\pi}(s, a)$

- greedy policies with $V^*(s)$ and $Q^*(s, a)$ are optimal
  - we can search for either $\pi^*$, $V^*(s)$ or $Q^*(s, a)$ and get the same result
  - each has their advantages and disadvantages
RL Agents Taxonomy

• Quantity to optimize:
  • value function – critic
    • either $Q$ or $V$, typically $Q$ in practice
  • policy – actor
  • both – actor-critic

• Environment model:
  • model-based (assume known $p(s' | s, a), r(s, a, s)$)
    • nice but typically not satisfied in practice
  • model-free (don’t assume anything, sample)
    • this is the usual real-world case
    • this is where using $Q$ instead of $V$ comes handy
Reinforcement Learning Approaches

• How to optimize:
  • **dynamic programming** – find the exact solution from Bellman equation
    • iterative algorithms, refining estimates
    • expensive, assumes known environment → not practical for real-world use
  • **Monte Carlo learning** – learn from experience
    • sample, then update based on experience
  • **Temporal difference learning** – like MC but look ahead (bootstrap)
    • sample, refine estimates as you go

• Sampling & updates:
  • **on-policy** – improve the policy while you’re using it for decisions
    • can’t use that with batch learning (decision policy is changing constantly)
  • **off-policy** – decide according to a different policy

both used in practice
Deep Reinforcement Learning

• Exactly the same as “plain” RL
  • agent & environment, actions & rewards
• “deep” = part of the agent is handled by a NN
  • value function (typically $Q$)
  • policy
• function approximation approach
  • $Q$ values / policy are represented as a parameterized function $Q(s, a; \theta) / \pi(s; \theta)$
  • enumerating in a table would take up too much space, be too sparse
  • the parameters $\theta$ are optimized
• assuming huge state space
  • much fewer weights than possible states
  • update based on one state changes many states
• needs tricks to make it stable
Q-Learning

- temporal difference – update $Q$ as you go
- off-policy – directly estimates best $Q^*$
  - regardless of policy used for sampling
- choose learning rate $\alpha$, initialize $Q$ arbitrarily
- for each episode:
  - choose initial $s$
  - for each step:
    - choose $a$ from $s$ according to $\epsilon$-greedy policy based on $Q$
    - take action $a$, observe observe reward $r$ and state $s'$
    - $Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \left( r + \gamma \cdot \max_{a'} Q(s', a') \right)$
    - $s \leftarrow s'$

any policy that chooses all actions & states enough times will converge to $Q^*(s, a)$: we need to explore to converge

choose learning rate $\alpha$, initialize $Q$ arbitrarily

for each episode:

- choose initial $s$
- for each step:
  - choose $a$ from $s$ according to $\epsilon$-greedy policy based on $Q$
  - take action $a$, observe observe reward $r$ and state $s'$
  - $Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \left( r + \gamma \cdot \max_{a'} Q(s', a') \right)$
  - $s \leftarrow s'$

update uses best $a'$, regardless of current policy: $a'$ **is not necessarily taken in the actual episode**

https://towardsdatascience.com/td-in-reinforcement-learning-the-easy-way-f92ecfa9f3ce

Animated example for SARSA & Q-Learning: https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_td.html
Deep Q-Networks

• Q-learning, where $Q$ function is represented by a neural net

• “Usual” Q-learning doesn’t converge well with NNs:
  a) SGD is unstable
  b) correlated samples (data is sequential)
  c) TD updates aim at a moving target (using $Q$ in computing updates to $Q$)
  d) scale of rewards & $Q$ values unknown → numeric instability

• → DQN adds fixes:
  a) minibatches (updates by averaged $n$ samples, not just one)
  b) experience replay
  c) freezing target $Q$ function
  d) clipping rewards

(Mnih et al., 2013, 2015)
http://arxiv.org/abs/1312.5602
http://www.nature.com/articles/nature14236
DQN tricks  ~ making it more like supervised learning

- **Experience replay** – break correlated samples
  - run through some episodes (dialogues, games…)
  - store all tuples \((s, a, r', s')\) in a buffer
  - for training, don’t update based on most recent moves – use buffer
    - sample minibatches randomly from the buffer
  - overwrite buffer as you go, clear buffer once in a while
  - only possible for off-policy

  \[
  \text{loss} := \mathbb{E}_{(s,a,r',s') \in \text{buf}} \left[ (r' + \gamma \max_{a'} Q(s', a'; \overline{\theta}) - Q(s, a; \theta))^2 \right]
  \]

- **Target Q function freezing**
  - fix the version of Q function used in update targets
    - have a copy of your Q network that doesn’t get updated every time
  - once in a while, copy your current estimate over

  “generate your own ‘supervised’ training data”

  “have a fixed target, like in supervised learning”
DQN algorithm

• initialize $\theta$ randomly
• initialize replay memory $D$ (e.g. play for a while using current $Q(\theta)$)
• repeat over all episodes:
  • set initial state $s$
  • for all timesteps $t = 1 \ldots T$ in the episode:
    • select action $a_t$ from $\epsilon$-greedy policy based on $Q(\theta)$
    • take $a_t$, observe reward $r_{t+1}$ and new state $s_{t+1}$
    • store $(s_t, a_t, r_{t+1}, s_{t+1})$ in $D$
  • sample a batch $B$ of random $(s, a, r', s')$’s from $D$
  • update $\theta$ using loss $\mathbb{E}_{(s,a,r',s') \in B} \left[ (r' + \gamma \max_{a'} Q(s', a'; \overline{\theta}) - Q(s, a; \theta))^2 \right]$ (1 update)
• once every $\lambda$ steps (rarely):
  • $\overline{\theta} \leftarrow \theta$
  • “replay” a. k. a. training (1 update)

storing experience (1 step of Q-learning exploration)

update the frozen target function
DQN for Atari

• 4-layers:
  • 2x CNN
  • 2x fully connected with ReLU activations

• Another trick:
  • output values for all actions at once
    • ~ vector $Q(s)$ instead of $Q(s, a)$
    • $a$ is not fed as a parameter
    • faster computation

• Learns many games at human level
  • with the same network structure
  • no game-specific features

input: Atari 2600 screen, downsized to 84x84 (grayscale)
4 last frames

values for all actions (joystick moves)

(Mnih et al., 2015)

https://youtu.be/V1eYniJ0Rnk?t=18
DQN for Dialogue Systems

- DQN can drive dialogue action selection/policy
- **warm start** needed to make the training actually work:
  - **pretrain** the network using supervised learning
  - **replay buffer spiking** – initialize using simple rule-based policy
    - so there are at least a few successful dialogues
    - the RL agent has something to catch on

DQN – feed-forward, 1 hidden ReLU layer

rule-based simulator with agenda running on DA level

movie ticket booking: better than rule-based

replay memory initialized using a simple handcrafted policy

Policy Gradients

• Instead of value functions, train a **network to represent the policy**
  • allows better action sampling – according to actual stochastic policy
    • no need for $\epsilon$-greedy (which is partially random, suboptimal)
• To optimize, we need a **performance metric**: $J(\theta) = V_{\pi \theta}(s_0)$
  • expected return in starting state when following $\pi_{\theta}$
  • we want to directly optimize this using gradient ascent
• **Policy Gradient Theorem**:  
  • expresses $\nabla J(\theta)$ in terms of $\nabla \pi(a|s, \theta)$  
  \[
  \nabla J(\theta) \propto \sum_s \mu(s) \sum_a Q^\pi(s, a) \nabla \pi(a|s, \theta) = E_{\pi} \left[ \sum_a Q^\pi(s, a) \nabla \pi(a|s, \theta) \right]
  \]

  $\mu(s)$ is state probability under $\pi$ – this is the same as expected value $E_{\pi}$
REINFORCE: Monte Carlo Policy Gradients

- direct search for policy parameters by stochastic gradient ascent
  - looking to maximize performance $J(\theta) = V^{\pi \theta}(s_0)$
- choose learning rate $\alpha$, initialize $\theta$ arbitrarily
- loop forever:
  - generate an episode $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot | \cdot, \theta)$
  - for each $t = 0, 1 \ldots T$: $\theta \leftarrow \theta + \alpha \gamma^t R_t \nabla \ln \pi(a_t | s_t, \theta)$

returns $R_t = \sum_{i=t}^{T-1} \gamma^{i-t} r_{i+1}$

variant – **advantage** instead of returns:
  - discounting a **baseline** $b(s)$ (predicted by any model)
  - $A_t = R_t - b(s_t)$ instead of $R_t$
  - gives better performance

$V(s)$ is actually a good $b(s)$

this will guarantee the right state distribution/frequency $\mu(s)$

this is stochastic $\nabla J(\theta)$:
  - from policy gradient theorem
  - using single action sample $a_t$
  - expressing $Q^\pi$ as $R_t$ (under $E_\pi$)
  - using $\nabla \ln x = \frac{v_x}{x}$

(Sutton & Barto, 2018; p. 327f)
Policy Gradients (Advantage) Actor-Critic

- REINFORCE + $V$ approximation + TD estimates – better convergence
  - differentiable policy $\pi(a|s, \theta)$
  - differentiable state-value function parameterization $\hat{V}(s, w)$
  - two learning rates $\alpha^\theta$, $\alpha^w$

- loop forever:
  - set initial state $s$ for the episode
  - for each step $t$ of the episode:
    - sample action $a$ from $\pi(\cdot|s, \theta)$, take $a$ and observe reward $r$ and new state $s'$
    - compute advantage $A \leftarrow r + \gamma \hat{V}(s', w) - \hat{V}(s, w)$
    - update $\theta \leftarrow \theta + \alpha^\theta \gamma^t A \nabla \ln \pi(a|s, \theta)$, $w \leftarrow w + \alpha^w \cdot A \nabla \hat{V}(s, w)$
    - $s \leftarrow s'$

TD: update after each step

actor (policy update)

same as REINFORCE, except:
- we use $\hat{V}(s, w)$ as baseline
- $r$ is used instead of $R_t$ (TD instead of MC)

critic (value function update)

(Su et al., 2017)
http://arxiv.org/abs/1707.00130
ACER: Actor-Critic with Experience Replay

• off-policy actor-critic – using **experience replay** buffer
  • same approach as Q learning
  • since ER buffer has past experience with out-of-date policies (using “old” $\tilde{\theta}$), it’s considered off-policy (behaviour policy $\pi_{\tilde{\theta}} \neq$ target policy $\pi_{\theta}$)
    • sampling behaviour from $\pi_{\tilde{\theta}}$ is biased w. r. t. $\pi_{\theta}$
    • correcting the bias – **importance sampling**: multiply by importance weight $\rho_t = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\tilde{\theta}}(a_t|s_t)}$
  • all updates are summed over batches & importance-sampled
    • new objective/performance metric: $\hat{E}_t \left[ \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\tilde{\theta}}(a_t|s_t)} \hat{A}_t \right]$ using advantage instead of returns

Proximal Policy Optimization

- ACER is prone to very large updates, unstable
  - to avoid going “off a cliff”, it needs very low LR, trains slowly
  - change the objective to produce more stable updates
- Basically clipping the ACER objective
  - define \( r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\tilde{\theta}}(a_t|s_t)} \) – ratio to old params
  - starting from \( \hat{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\tilde{\theta}}(a_t|s_t)} \hat{A}_t \right] = \hat{E}_t[r_t(\theta)\hat{A}_t] \) (see ACER)
  - using \( \hat{E}_t\left[ \min(r_t(\theta)\hat{A}_t, \text{clip}[r_t(\theta)]_{1-\epsilon}\hat{A}_t) \right] \)

\[ r_t \]

original clipped to stay close to 1

minimum – lower bound on the unclipped objective

Rewards in RL

• Reward function is critical for successful learning
• Handcrafting is not ideal
  • domain knowledge typically needed to detect dialogue success
  • need simulated or paid users,
    can’t learn from users without knowing their task
  • paid users often fail to follow pre-set goals
• Having users provide feedback is costly & inconsistent
  • real users don’t have much incentive to be cooperative
• Learning/optimizing the rewards is desirable
Turn-level Quality Estimation

Interaction Quality

- turns annotated by experts (Likert 1-5)
- trained model (SVM/RNN)
  - very low-level features
  - mostly ASR-related
  - multi-class classification
- result is domain-independent
  - trained on a very small corpus (~200 dialogues)
  - same model applicable to different datasets
- can be used in a RL reward signal
  - works better than task success
  - can be blended with success

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRRecognitionStatus</td>
<td>ASR status: success, no match, no input</td>
</tr>
<tr>
<td>ASRConfidence</td>
<td>confidence of top ASR results</td>
</tr>
<tr>
<td>RePrompt?</td>
<td>is the system question the same as in the previous turn?</td>
</tr>
<tr>
<td>ActivityType</td>
<td>general type of system action: statement, question</td>
</tr>
<tr>
<td>Confirmation?</td>
<td>is system action confirm?</td>
</tr>
<tr>
<td>MeanASRConfidence</td>
<td>mean ASR confidence if ASR is success</td>
</tr>
<tr>
<td>#Exchanges</td>
<td>number of exchanges (turns)</td>
</tr>
<tr>
<td>#ASRSuccess</td>
<td>count of ASR status is success</td>
</tr>
<tr>
<td>%ASRSuccess</td>
<td>rate of ASR status is success</td>
</tr>
<tr>
<td>#ASRRejections</td>
<td>count of ASR status is reject</td>
</tr>
<tr>
<td>%ASRRejections</td>
<td>rate of ASR status is reject</td>
</tr>
<tr>
<td>{Mean} ASRConfidence</td>
<td>mean ASR confidence if ASR is success</td>
</tr>
<tr>
<td>{#} ASRSuccess</td>
<td>count of ASR is success</td>
</tr>
<tr>
<td>{#} ASRRejections</td>
<td>count of ASR status is reject</td>
</tr>
<tr>
<td>{#} RePrompts</td>
<td>count of times RePrompt? is true</td>
</tr>
<tr>
<td>{#} SystemQuestions</td>
<td>count of ActivityType is question</td>
</tr>
</tbody>
</table>

“reject” = ASR output doesn’t match in-domain LM

(Schmitt & Ultes, 2015; Ultes et al., 2017; Ultes, 2019; Ultes & Maier, 2021)
https://doi.org/10.1016/j.specom.2015.06.003
https://doi.org/10.21437/Interspeech.2017-1032
https://aclweb.org/anthology/W19-5902/
https://aclanthology.org/2021.sigdial-1.42
Turn-level adversarial rewards

- discriminator: policy vs. human-human
  - irrespective of success, can be done on turn level
- training process:
  - pretrain both \( \pi \) & \( f \) using supervised learning
  - sample dialogs using \( \pi \)
  - update \( f \) to distinguish sampled vs. human-human
  - update \( \pi \) via RL using rewards provided by \( f \)
- policy \( \pi \) & reward estimator \( f \) are feed-forward
  - ReLU, 1 hidden layer

Alternating supervised & RL

• we can do better than just supervised pretraining
• alternate regularly
  • start with supervised more frequently
    • alleviate sparse rewards, but don’t completely avoid exploring
  • later do more RL
    • but don’t forget what you learned by supervised learning
• options:
  • schedule supervised every $N$ updates
  • same + increase $N$ gradually
  • use supervised after RL does poorly (worse than baseline)
    • baseline = moving average over history + $\lambda \cdot$ std. error of the average
    • agent is less likely to be worse than baseline in later stages of learning

(Xiong et al., 2018)
Deep Dyna-Q: learning from humans & simulator

- humans are costly, simulators are inaccurate
- \( \Rightarrow \) learn from both, improve simulator as you go
  - direct RL = learn from users
  - world model learning = improve internal simulator
    - supervised, based on previous dialogues with users
  - planning = learn from simulator
- DQN, feed-forward policy
- simulator: feed-forward multi-task net
  - draw a goal uniformly at the start
  - predict actions, rewards, termination
  - use \( K \) simulated (“planning”) dialogues per 1 real
- discriminative DDQ: only use a simulated dialogue if it looks real (according to a discriminator)

(Peng et al., 2018) https://www.aclweb.org/anthology/P18-1203
(Su et al., 2018) https://www.aclweb.org/anthology/D18-1416
Hierarchical RL

- good for multiple subtasks
  - e.g. book a flight to London and a hotel for the same day, close to the airport
- top-level policy: select subtask $g_i$
- low-level policy: actions $a_j, g_i$ to complete subtask $g_i$
  - given initiation/termination conditions
    - keeps on track until terminal state is reached
  - shared by all subtasks (subtask=parameter)
  - internal critic (=prob. that subtask is solved)
- global state tracker
  - integrates information from subtasks

(Peng et al., 2017)  
http://aclweb.org/anthology/D17-1237
• spatial (slot-based) split instead of temporal
  • doesn’t need defined subtasks & sub-rewards
• belief state representation – features
  • master $\phi_m$, slot-independent $\phi_i$, per-slot $\phi_{sk}$
  • handcrafted (could be neural nets)
  • supports sharing parameters across domains

• two-step action selection:
  1) master action: “slot-dependent or not”?
     • master policy
  2) primitive action
     a) slot-independent policy
     b) slot-specific policies (with shared parameters, distinguished only by belief state)
       • chooses max. $Q$ for all slot-action pairs – involves choosing the slot
• everything is trained using the same global reward signal
Summary

• **RL** for action selection / dialogue policy
  • MDP / agent in an environment, taking actions, getting rewards
  • dynamic programming, **Monte Carlo**, **Temporal Difference**
  • optimizing **value function** $V/Q$ (**critic**), **policy** (**actor**), or both (**actor-critic**)
  • learning **on-policy** or **off-policy** (act by the policy you learn/not)

• **DQN** – representing & optimizing $Q$ function with a network
  • minibatches, target function freezing, experience replay

• **Policy gradients** – policy network & direct policy optimization
  • **REINFORCE** (MC policy gradients) + advantage
  • **Actor-critic** (REINFORCE + TD + $V$ estimates) + extensions (ACER, PPO)

• rewards can be learned/estimated (supervised/GAN-style)

• learning multiple tasks: hierarchical, feudal RL
Thanks

Contact us:  
https://ufaldsg.slack.com/  
{odusek,hudecek}@ufal.mff.cuni.cz  
Skype/Meet/Zoom (by agreement)

AIC short intro in 10 min  
(out of order, but will be on YouTube)

Get these slides here:  
http://ufal.cz/npfl099

Next Monday:  
Language Generation  
4th Assignment

References/Inspiration/Further:

- Milica Gašić’s slides (Cambridge University): http://mi.eng.cam.ac.uk/~mg436/teaching.html  
- Young et al. (2013): POMDP-Based Statistical Spoken Dialog Systems: A Review:  