Statistical Dialogue Systems

NPFL099 Statistické Dialogové systémy

6. Dialogue Policy

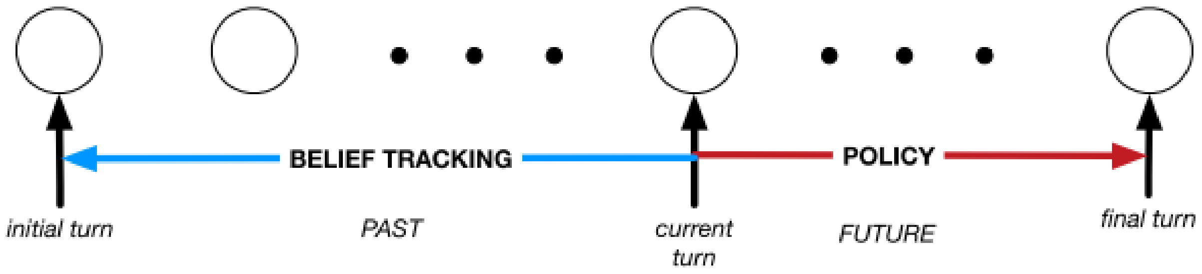
Ondřej Dušek & Vojtěch Hudeček

http://ufal.cz/npfl099

7. 11. 2019

Dialogue Management

- Two main components:
  - **State tracking** (last lecture)
  - **Action selection/Policy** (today)



(from Milica Gašić's slides)

- action selection – deciding what to do next
  - based on the current belief state – under uncertainty
  - following a **policy** (strategy) towards an end **goal** (e.g. book a flight)
  - controlling the coherence & flow of the dialogue
  - actions: linguistic & non-linguistic
- DM/policy should:
  - manage uncertainty from belief state
  - recognize & follow dialogue structure
  - plan actions ahead towards the goal

Did you say Indian or Italian?

follow convention, don't be repetitive

e.g. ask for all information you require

Action Selection Approaches

- Finite-state machines
  - simplest possible
  - dialogue state is machine state
- Frame-based (VoiceXML)
  - slot-filling + providing information – basic agenda
  - rule-based in essence
- Rule-based
  - any kind of rules (e.g. Python code)
- Statistical
  - typically using reinforcement learning

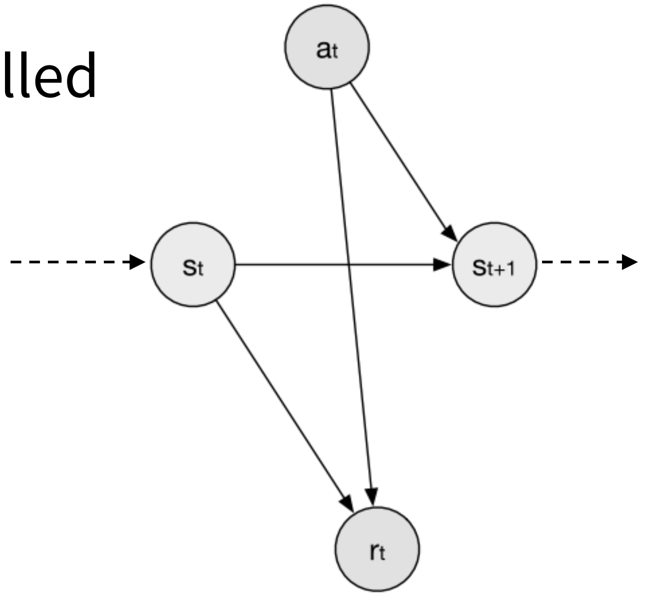
DM with supervised learning

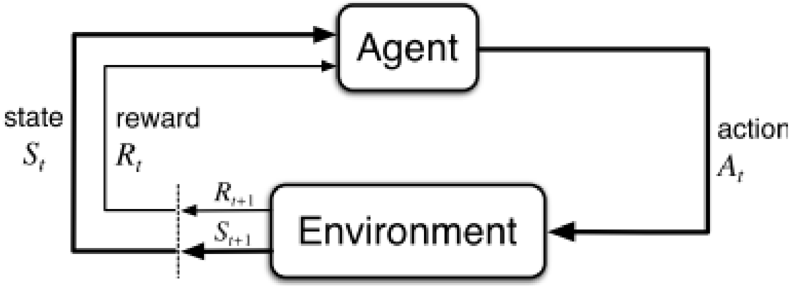
- **Action selection ~ classification** → use supervised learning?
  - set of possible actions is known
  - belief state should provide all necessary features
- Yes, but…
  - You'd **need** sufficiently large **human-human data** – hard to get
    - human-machine would just mimic the original system
  - Dialogue is ambiguous & complex
    - there's **no single correct next action**– multiple options may be equally good
    - but datasets will only have one next action
    - **some paths will be unexplored** in data, but you may encounter them
  - DSs won't behave the same as people
    - ASR errors, limited NLU, limited environment model/actions
    - DSs **should** behave differently – make the best of what they have

DM as a Markov Decision Process

- MDP = probabilistic control process
  - modelling situations that are partly random, partly controlled
  - **agent** in an **environment**:
    - has internal **state** $s_t \in \mathcal{S}$ (~ dialogue state)
    - takes **actions** $a_t \in \mathcal{A}$ (~ system dialogue acts)
    - actions chosen according to **policy** $\pi: \mathcal{S} \rightarrow \mathcal{A}$
    - gets **rewards** $r_t \in \mathbb{R}$ & state changes from the environment
  - rewards are typically handcrafted
    - very high positive for a successful dialogue (e.g. +40)
    - high negative for unsuccessful dialogue (-10)
    - small negative for every turn (-1, promote short dialogues)
  - Markov property – state defines everything
    - no other temporal dependency
  - policy may be **deterministic** or **stochastic**
    - stochastic: prob. dist. of actions, sampling

(from Milica Gašić's slides)

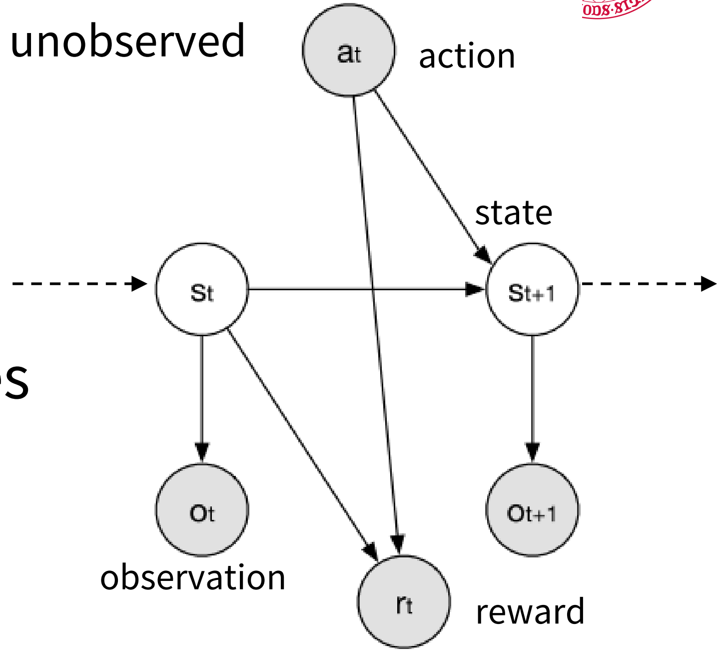




(Sutton & Barto, 2018)

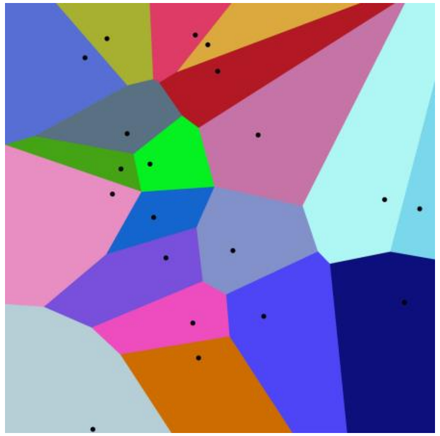
Partially-observable MDPs

grey = observed
white = unobserved



(from Milica Gašić's slides)

- POMDPs – **belief** states instead of dialogue states
  - true states ("what the user wants") are not observable
  - observations ("what the system hears") depend on states
  - belief – probability distribution over states
  - can be viewed as **MDPs with continuous-space states**
- All MDP algorithms work…
  - if we **quantize/discretize** the states
  - use grid points & nearest neighbour approaches
  - this might introduce errors / make computation complex
- Deep RL typically works out of the box
  - function approximation approach, allows continuous states


https://en.wikipedia.org/wiki/Voronoi_diagram

Reinforcement learning

- 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

alternative – **episodes**: only count to T when we encounter a terminal state
(e.g. 1 episode = 1 dialogue)

accumulated long-term reward

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

$\gamma \in [0,1]$ = **discount factor**
(immediate vs. future reward trade-off)

$\gamma < 1$: R_t is finite (if r_t is finite)
$\gamma = 0$: greedy approach (ignore future rewards)

- state transition is stochastic → maximize **expected return**

$$\mathbb{E}[R_t|\pi, s_0]$$

expected R_t if we start from state s_0 and follow policy π

State-value Function

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

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

prob. of choosing
a from s under π

transition
probs.

expected
immediate
reward

- $V^{\pi}(s)$ defines a **greedy policy**:

actions that look best for the next step

$$\pi(s,a) := \begin{cases} \frac{1}{\#\text{ of } a's} \text{ for } a = \arg\max_a \sum_{s' \in \mathcal{S}} p(s'|s,a)(r(s,a,s') + \gamma V^{\pi}(s')) \\ 0 \text{ otherwise} \end{cases}$$

Action-value (Q-)Function

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

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

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

again, "actions that look best for the next step"

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

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

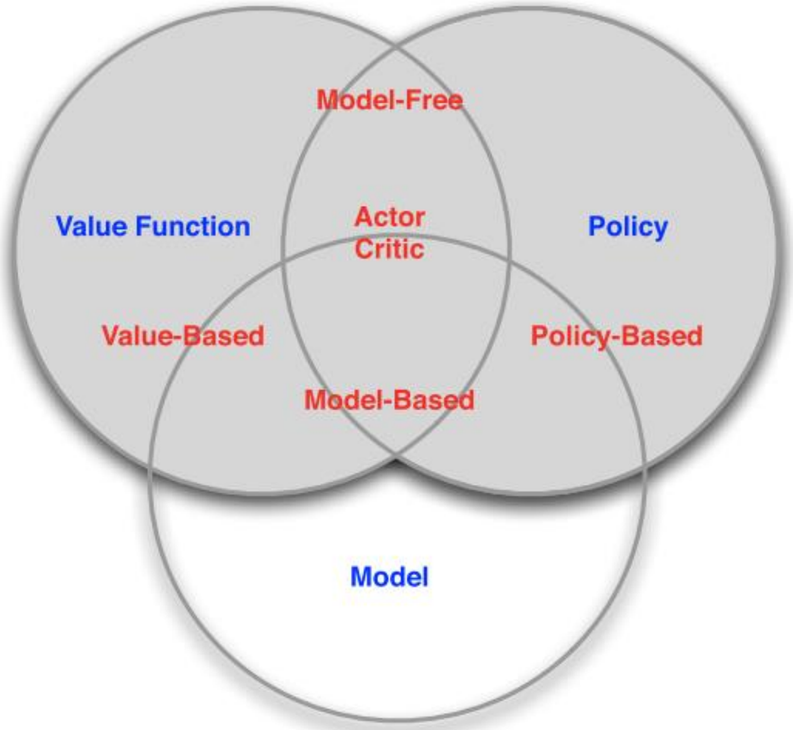
but Q function itself tends to be more complex than V

Optimal Policy in terms of V and Q

- **optimal policy** π^* – one that maximizes expected return $\mathbb{E}[R_t|\pi]$
  - $V^\pi(s)$ expresses $\mathbb{E}[R_t|\pi]$ → use it to define π^*
- π^* is a policy such that $V^{\pi^*}(s) \geq V^{\pi'}(s)\ \ \forall \pi', \forall s \in \mathcal{S}$
  - π^* always exists in an MDP (need not be unique)
  - π^* has the **optimal state-value function** $V^*(s) \coloneqq \max_\pi V^\pi(s)$
  - π^* also has the **optimal action-value function** $Q^*(s,a) \coloneqq \max_\pi Q^\pi(s,a)$
- greedy policies with $V^*(s)$ and $Q^*(s,a)$ are optimal
  - we can search for either π^*, $V^*(s)$ or $Q^*(s,a)$ and get the same result
  - each has their advantages and disadvantages

RL Agent Taxonomy

- Quantity to optimize:
  - value function – **critic**
  - policy – **actor**
  - both – **actor-critic**
- Environment model:
  - **model-based** (assume known $p(s'|s,a), r(s,a,s)$)
  - **model-free** (don't assume anything, sample)
    - this is where using Q instead of V comes handy



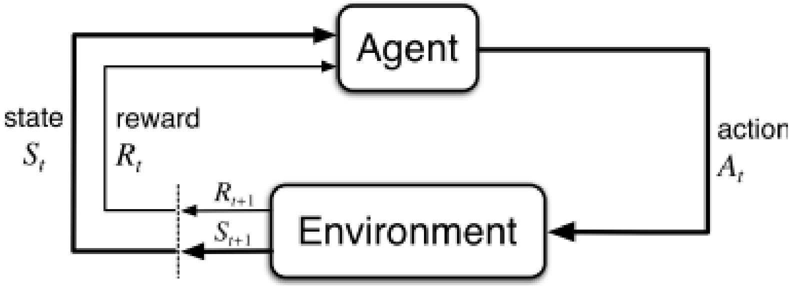
(from David Silver's slides)

RL Approaches

- How to optimize:
  - **dynamic programming** – find the exact solution from Bellman equation
    - iterative algorithms, refining estimates
    - expensive, assumes known environment
  - **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
  - **off-policy** – decide according to a different policy

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



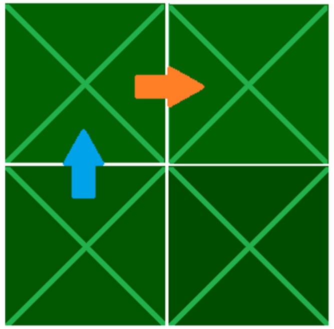
(Sutton & Barto, 2018)

- function approximation approach
  - Q values / policy are represented as a parameterized function $Q(s, a; \boldsymbol{\theta})$ / $\pi(s; \boldsymbol{\theta})$
  - enumerating in a table would take up too much space, be too sparse
  - the parameters θ 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

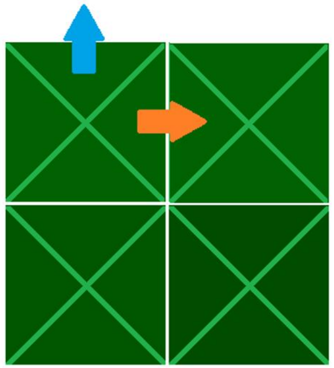
Q-Learning

- temporal difference – update Q as you go
- off-policy – directly estimates best Q^*
  - regardless of policy used for sampling

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



- choose learning rate α, initialize Q arbitrarily



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

$$a = \begin{cases} \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \\ \text{random action with probability } \epsilon \end{cases}$$

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

TD: moving estimates

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

(Mnih et al., 2013, 2015)
http://arxiv.org/abs/1312.5602
http://www.nature.com/articles/nature14236

- Q-learning, where Q function is represented by a neural net
- Causes of poor convergence in basic Q-learning 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
- Fixes in DQN:
  a) minibatches (updates by averaged n samples, not just one)
  b) experience replay
  c) freezing target Q function
  cool!
  d) clipping rewards

common NN tricks

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

"generate your own 'supervised' training data"

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

- **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

"have a fixed target, like in supervised learning"

DQN algorithm

- initialize $\boldsymbol{\theta}$ randomly
- initialize replay memory D (e.g. play for a while using current $Q(\boldsymbol{\theta})$)
- repeat over all episodes:
  - for episode, set initial state s
    - select action a from ϵ-greedy policy based on $Q(\boldsymbol{\theta})$
    - take a, observe reward r' and new state s'
    - store (s, a, r', s') in D
    - $s \leftarrow s'$

    storing experience

  often →
  - once every k steps:
    - sample a batch B of random (s, a, r', s')'s from D
    - update $\boldsymbol{\theta}$ using loss $\mathbb{E}_{(s,a,r',s')\in B}\left[\left(r' + \gamma \max_{a'} Q\left(s', a'; \overline{\boldsymbol{\theta}}\right) - Q(s, a; \boldsymbol{\theta})\right)^2\right]$

    "replay"
    a. k. a. training

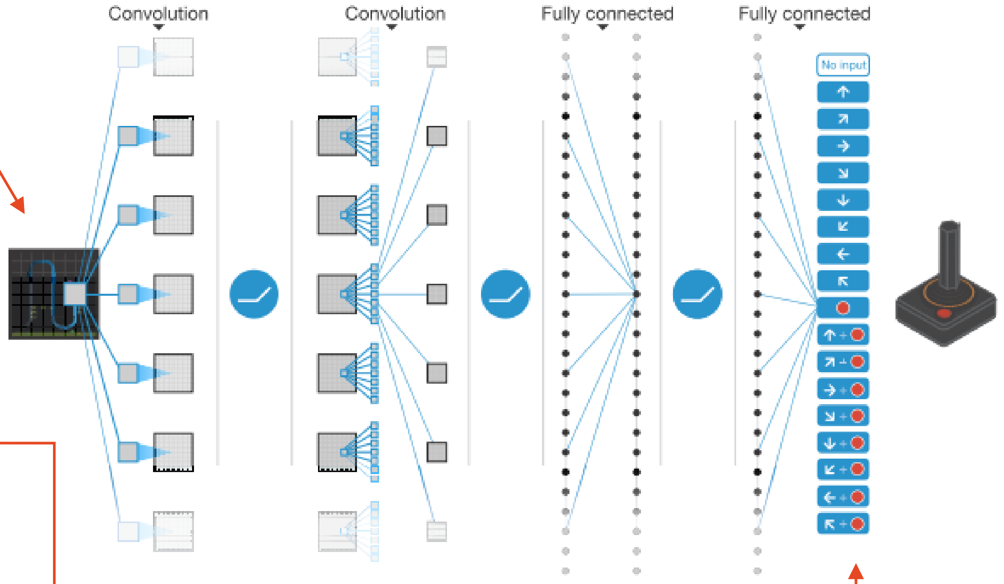
  rarely →
  - once every λ steps:
    - $\overline{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}$

DQN for Atari

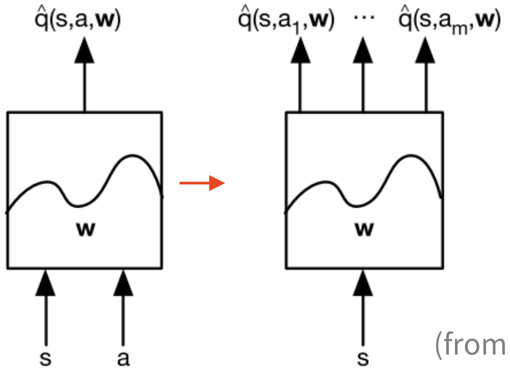
input: Atari 2600 screen,
downsized to 84x84 (grayscale)
4 last frames

(Mnih et al., 2015)

- 4-layers:
  - 2x CNN
  - 2x fully connected with ReLU activations
- Another trick:
  - output values for all actions at once
    - ~ vector $\boldsymbol{Q}(s)$ 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



values for all actions
(joystick moves)



https://youtu.be/V1eYniJ0Rnk?t=18

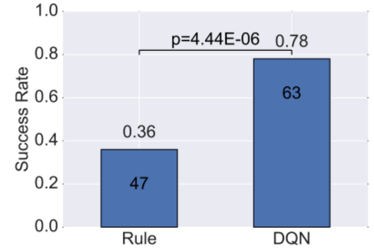
(from David Silver's slides)

DQN for Dialogue Systems

(Li et al., 2017)
https://arxiv.org/abs/1703.01008
https://github.com/MiuLab/TC-Bot

(Lipton et al., 2018)
https://arxiv.org/abs/1608.05081

- DQN can drive action selection
- **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

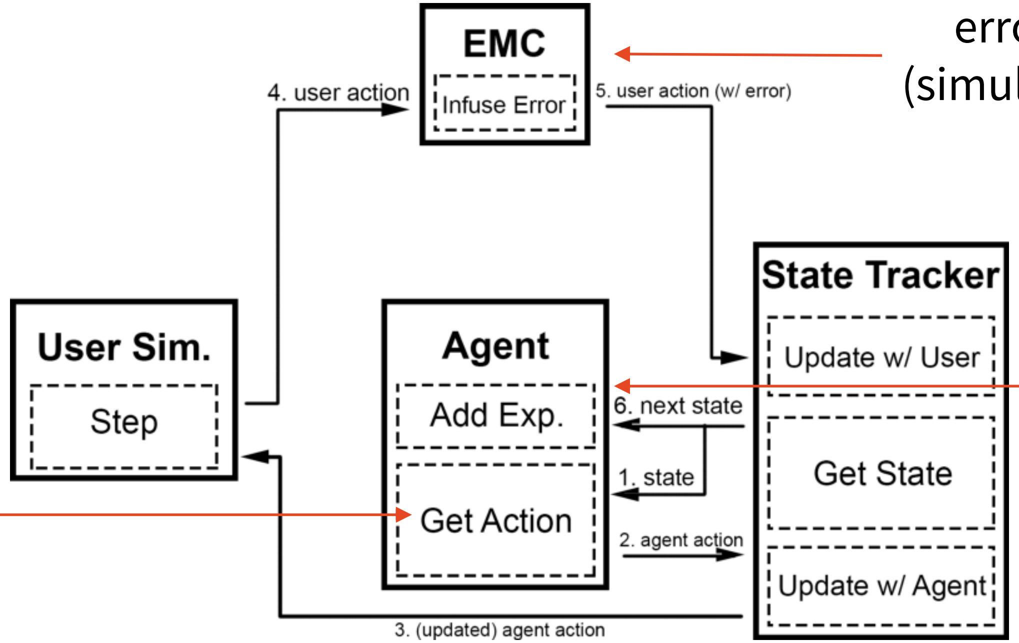


movie ticket booking:
better than rule-based

rule-based simulator
with agenda
running on DA level

error model controller
(simulating ASR/NLU noise)

DQN – feed-forward,
1 hidden ReLU layer



replay memory
initialized using a
simple handcrafted policy

https://towardsdatascience.com/training-a-goal-oriented-chatbot-with-deep-reinforcement-learning-part-i-introduction-and-dce3af21d383

BBQ – Bayes-by-Backprop Q-Networks

- better exploration than ϵ-greedy – explore uncertain regions
- **Bayes-by-Backprop** – prob. dist. over network weights
  - start from prior $p(\theta)$, learn posterior $p(\theta|D)$ for training data D
  - posterior approximated by Gaussians $q(\theta|w)$, each $\theta_i \sim \mathcal{N}(\mu_i, \sigma_i)$
    - now learning $w_i = \{(\mu_i, \rho_i)\}$ where $\sigma_i = \log(1 + \exp \rho_i)$, to keep σ_i positive
    - VAE-style: minimizing KL divergence between q and p, reparameterization trick
- using BB to represent DQN + posterior (Thompson) sampling
  - actions sampled acc. to posterior prob. they're optimal in current state
  - just sample θ_t from q, then choose $a_t = \arg\max_a Q(s_t, a; \theta_t)$
- no need to sample for the target network, just use $\overline{\mu}$
  - faster, actually more stable

BBQ performance

(Lipton et al., 2018)
https://arxiv.org/abs/1608.05081

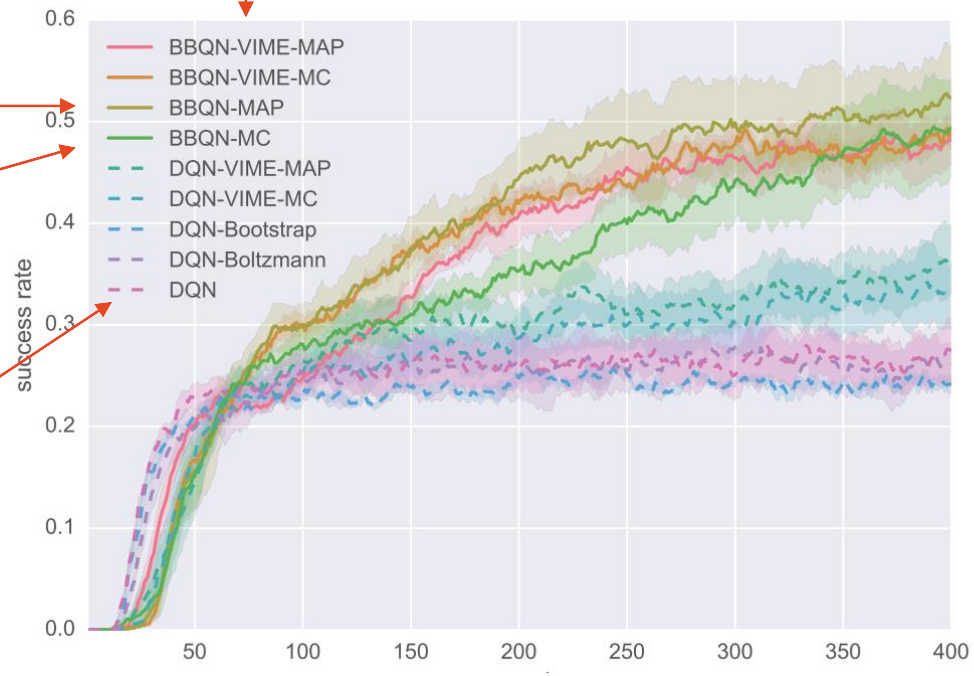
MLP with 2 hidden layers, ReLU, width=256
movie booking task
one-hot dialogue state representation (268 dim)
39 actions (basic *hello()*, *deny()*, *thanks()* etc. + inform/request for each slot)

(enhanced rewards with surprisal)

no sampling from
frozen network

sampling from
frozen network

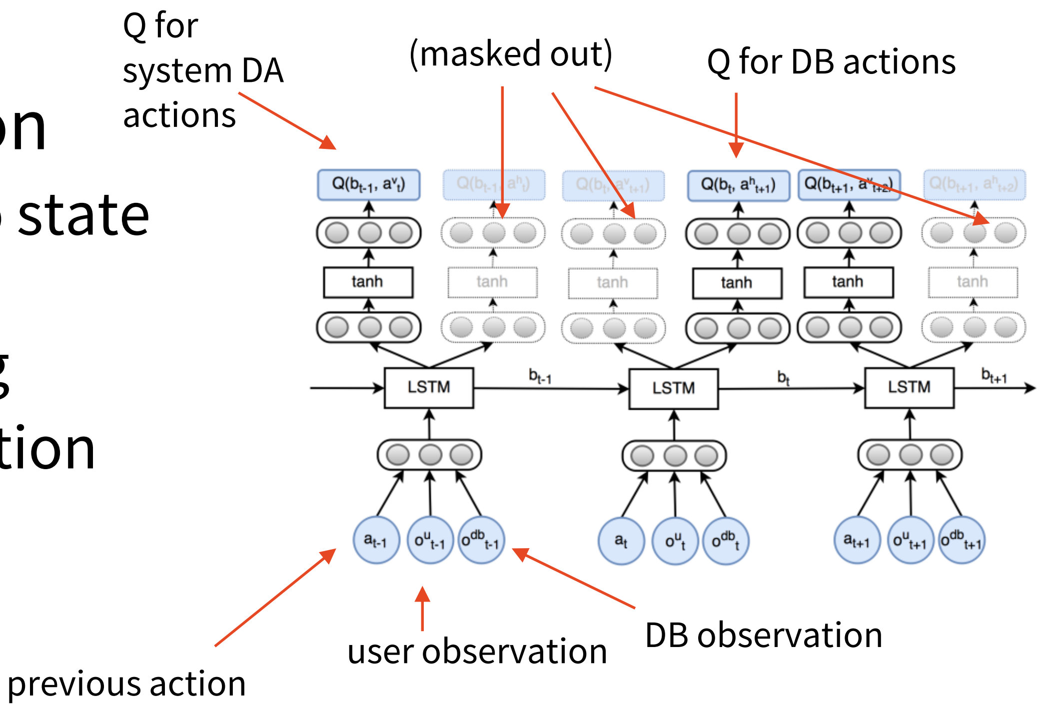
ϵ-greedy

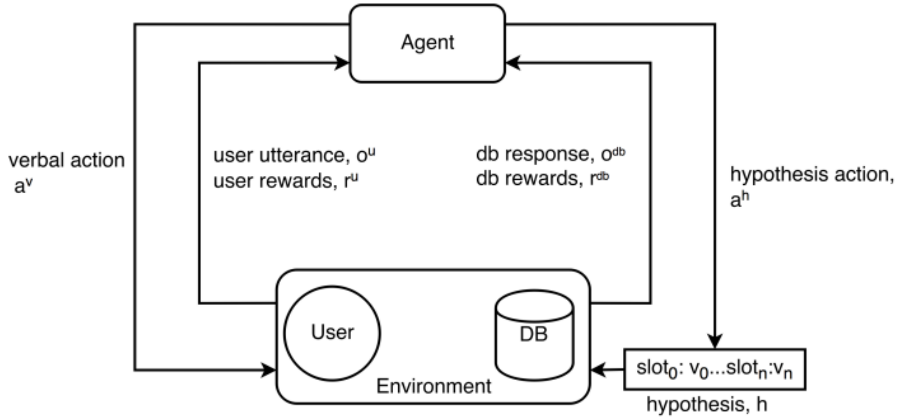


Recurrent Q-Networks

(Zhao & Eskenazi, 2016)
http://arxiv.org/abs/1606.02560

- Joint dialogue tracking & action selection
  - actions are either system DAs or updates to state (DB hypothesis)
  - forced to alternate action types by masking
  - rewards from DB for narrowing down selection
- Models the Q-network as a LSTM
  - or rather LSTM underlying multiple MLPs
    - LSTM maintains internal state representation
  - 1 MLP for system DAs
  - 1 MLP per slot (action=select value X)





Policy Gradients

- instead of value functions, train a network to represent the policy
- allows better sampling – acc. to actual stochastic policy
- performance metric: $J(\theta) = V^{\pi_\theta}(s_0)$
  - expected return in starting state when following π_θ
  - we want to directly optimize this using gradient ascent
- **Policy Gradient Theorem**:
  - expresses $\nabla J(\theta)$ in terms of $\nabla \pi(a|s,\theta)$

$$\nabla J(\theta) \propto \underbrace{\sum_s \mu(s)}_{\mu(s) \text{ is state probability under } \pi \text{ – this is the same as expected value } E_\pi} \sum_a Q^\pi(s,a) \nabla \pi(a|s,\theta) = E_\pi\left[\sum_a Q^\pi(s,a) \nabla \pi(a|s,\theta)\right]$$

REINFORCE: Monte Carlo Policy Gradients

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

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

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

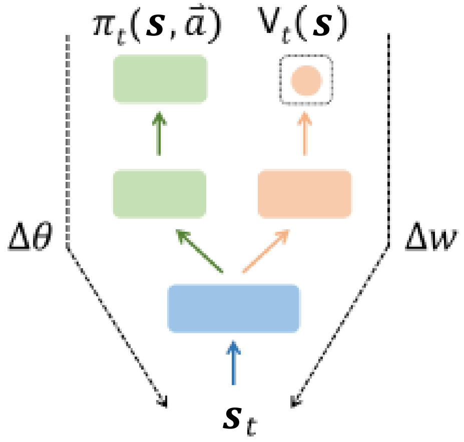
variant – **advantage** instead of returns: discounting a **baseline** $b(s)$ (predicted by any model) $A_t = R_t - b(s_t)$ instead of R_t gives better performance

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

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

Policy Gradients (Advantage) Actor-Critic

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



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

TD: update after each step

actor (policy update)

critic (value function update)

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

(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 π_θ)
    - sampling behaviour from $\pi_{\tilde{\theta}}$ is biased w. r. t. π_θ
    - 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
    - objective: $\hat{E}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\tilde{\theta}}(a_t|s_t)} \hat{A}_t]$

using advantage instead of returns

importance sampled

batch average

(Wang et al., 2017) http://arxiv.org/abs/1611.01224
(Su et al., 2017) http://arxiv.org/abs/1707.00130
(Weisz et al., 2018) http://arxiv.org/abs/1802.03753

(Wang et al., 2017) http://arxiv.org/abs/1611.01224
(Su et al., 2017) http://arxiv.org/abs/1707.00130
(Weisz et al., 2018) http://arxiv.org/abs/1802.03753

TRACER: Trust-Region ACER

- basic ACER may be unstable/slow to learn
  - prone to excessively large updates
    – need to set learning rates low
    - high learning rate = unstable, high variance
    - low learning rate = too slow
- limit on KL-divergence change b/t updated policy θ & average policy $\overline{\theta}$
  - $\overline{\theta}$ is a moving average of past policies: $\overline{\theta} \leftarrow \alpha\overline{\theta} + (1-\alpha)\theta$
  - modified policy gradient g is defined as: $\min_g \frac{1}{2}||\nabla\theta - g||_2^2$
    so that $\nabla KL[\pi_{\overline{\theta}}(s_t)||\pi_\theta(s_t)]^T g \leq \xi$
    - i.e. the closest you can get to the gradient,
      but don't increase KL between the average and new policy too much
    - quadratic programming, has closed solution

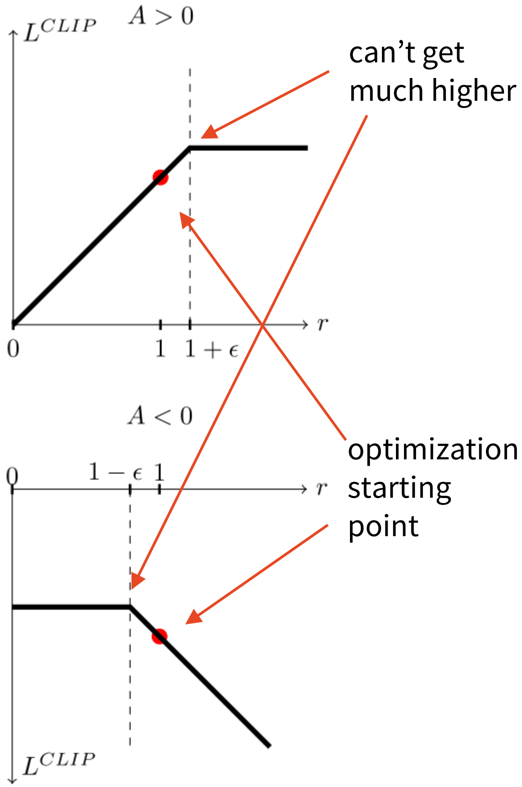
sum of squared differences (square of L2)

https://medium.com/@jonathan_hui/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9

Proximal Policy Optimization

- Changing the objective to be more like trust-region
  - without the need to adjust gradients & do the optimization
- Basically clipping the objective
  - define $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\tilde{\theta}}(a_t|s_t)}$ – ratio to old params
  - starting from $\hat{E}_t\left[\dfrac{\pi_\theta(a_t|s_t)}{\pi_{\tilde{\theta}}(a_t|s_t)}\hat{A}_t\right] = \hat{E}_t\left[r_t(\theta)\hat{A}_t\right]$ (see ACER)
  - using $\hat{E}_t\left[\min\left(\underbrace{r_t(\theta)\hat{A}_t}_{\text{original}}, \underbrace{\mathrm{clip}\left[r_t(\theta)\hat{A}_t\right]_{1-\epsilon}^{1+\epsilon}}_{\text{clipped to stay close to 1}}\right)\right]$

minimum – lower bound on the unclipped objective



Summary

- Action selection = deciding what to do next
  - following a **policy**
- Approaches:
  - FSM, Frames, Rule-based
  - Machine learning (RL better than supervised)
- **RL** – agent in an environment, taking actions, getting rewards
  - optimizing **value function** (V/Q) or **policy**
  - learning **on-policy** or **off-policy** (act by the policy you learn/not)
  - **DQN** – optimizing Q function with a network
    - batches, freezing, experience replay
  - **Policy gradients** – optimizing policy
  - **Actor-Critic** – optimizing policy & value function
    - ACER, PPO

Thanks

Contact us:

odusek@ufal.mff.cuni.cz
hudecek@ufal.mff.cuni.cz
(or on Slack)

Labs today
14:00 SW1
Projects updates

Get these slides here:

http://ufal.cz/npfl099

References/Inspiration/Further:

- Sutton & Barto (2018): Reinforcement Learning: An Introduction (2nd ed.): http://incompleteideas.net/book/the-book.html
- Nie et al. (2019): Neural approaches to conversational AI: https://arxiv.org/abs/1809.08267
- Filip Jurčíček's slides (Charles University): https://ufal.mff.cuni.cz/~jurcicek/NPFL099-SDS-2014LS/
- Milica Gašić's slides (Cambridge University): http://mi.eng.cam.ac.uk/~mg436/teaching.html
- Heidrich-Meisner et al. (2007): Reinforcement Learning in a Nutshell: https://christian-igel.github.io/paper/RLiaN.pdf
- Young et al. (2013): POMDP-Based Statistical Spoken Dialog Systems: A Review: http://cs.brown.edu/courses/csci2951-k/papers/young13.pdf