Reinforcement Learning Demystified for Non-RL People

杨健程 YANG Jiancheng

Jan, 2018

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

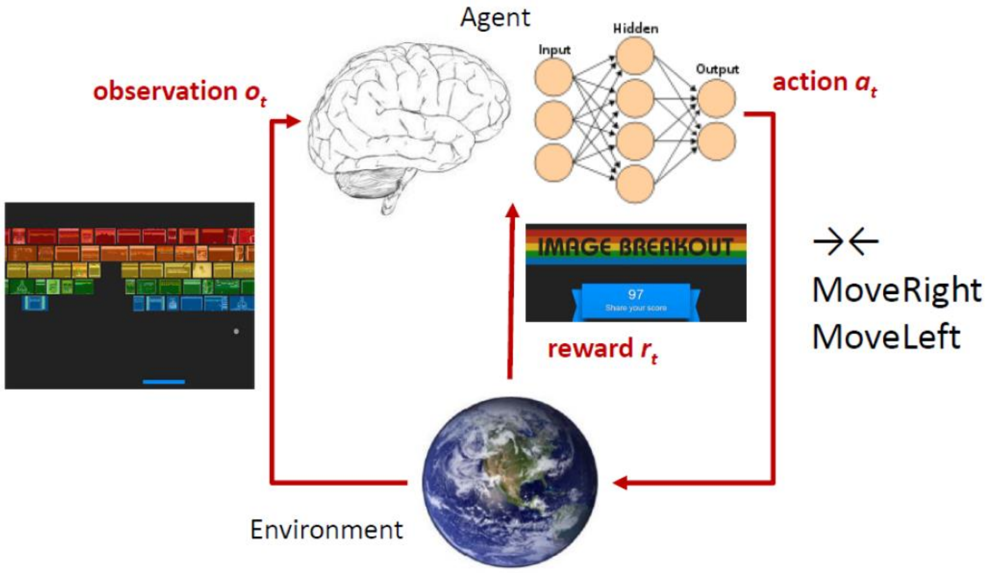
Agenda

- Introduction
- Algorithms
    - Deep Q Learning
    - REINFOCE
    - Actor-Critic
    - Imitation Learning
- Applications
    - SeqGAN
    - Hard Attention

Agenda

- Introduction
- Algorithms
  - Deep Q Learning
  - REINFOCE
  - Actor-Critic
  - Imitation Learning
- Applications
  - SeqGAN
  - Hard Attention

Elements of RL



An MDP is defined by:

- Set of states S
- Set of actions A
- Transition function $P(s' \mid s, a)$
- Reward function $R(s, a, s')$
- Start state s_0
- Discount factor γ
- Horizon H

Introduction

Markov Decision Process

- Experience (trajectory) is described as

$$\tau = (o_0, a_1, r_1, o_1, \dots, a_t, r_t, o_t)$$

- State is a summary of experience

$$s_t = f(o_0, a_1, r_1, o_1, \dots, a_t, r_t, o_t)$$

- Fully observed environment

$$s_t = o_t$$

- Goal: $\max_\pi \mathbb{E}\left(\sum_{t=0}^{H} \gamma^t r_t\right)$

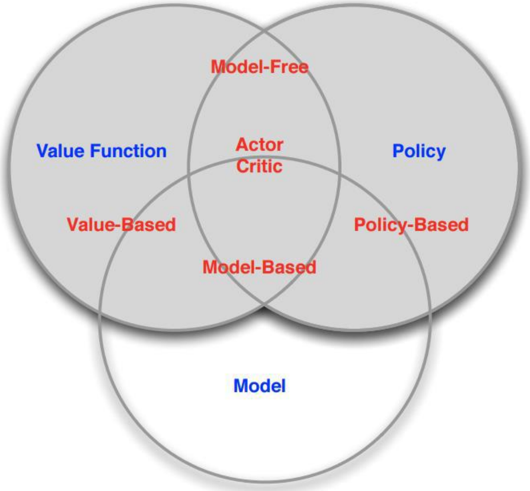
An MDP is defined by:

- Set of states S
- Set of actions A
- Transition function $P(s' \mid s, a)$
- Reward function $R(s, a, s')$
- Start state s_0
- Discount factor γ
- Horizon H

Introduction

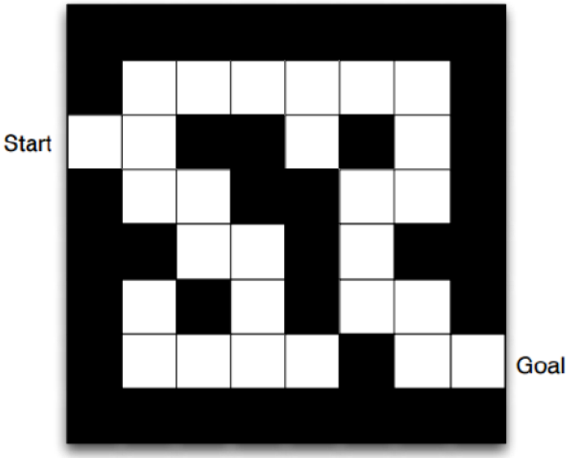
Approaches

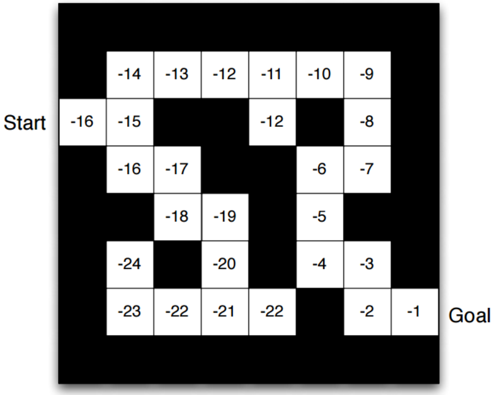
- Value-based RL
  - Estimate the optimal value function $Q^*(s,a)$
  - This is the maximum value achievable under any policy
- Policy-based RL
  - Search directly for the optimal policy π^*
  - This is the policy achieving maximum future reward
- Model-based RL
  - Build a model of the environment
  - Plan (e.g. by lookahead) using model
- All of the three can be represented by neural nets



Introduction

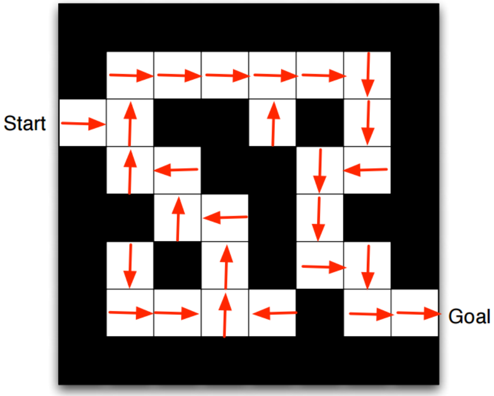
Maze Example





	-14	-13	-12	-11	-10	-9	
-16	-15			-12		-8	
	-16	-17			-6	-7	
		-18	-19		-5		
	-24		-20		-4	-3	
	-23	-22	-21	-22		-2	-1

Value based



Policy based

Agenda

- Introduction
- Algorithms
  - Deep Q Learning
  - REINFOCE
  - Actor-Critic
  - Imitation Learning
- Applications
  - SeqGAN
  - Hard Attention

Algorithms: Deep Q Learning

Value Function

- A value function is a prediction of future reward

- Q-value function

$$Q^{\pi}(a|s) = \mathbb{E}\left(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots |s,a\right)$$

- Bellman equation

$$Q^{\pi}(a|s) = \mathbb{E}_{s',a'}(r + \gamma Q^{\pi}(a'|s')|s,a)$$

- Optimal Q^*

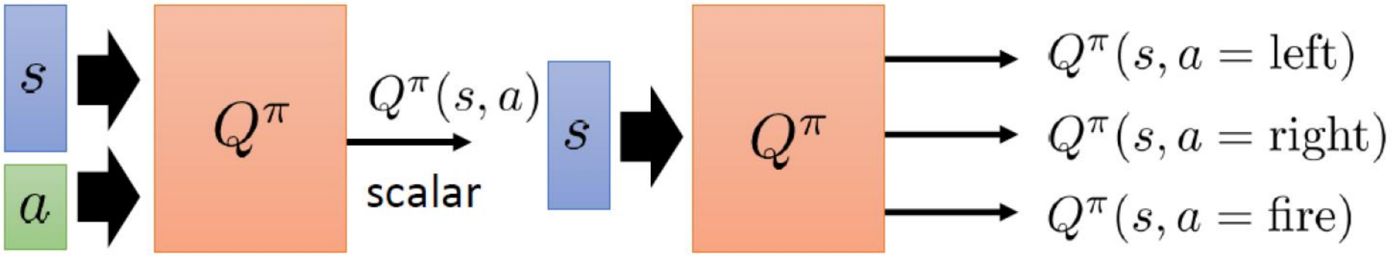
$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = Q^{\pi^*}(s,a)$$

$$\pi^*(s) = argmax_a Q^*(s,a)$$

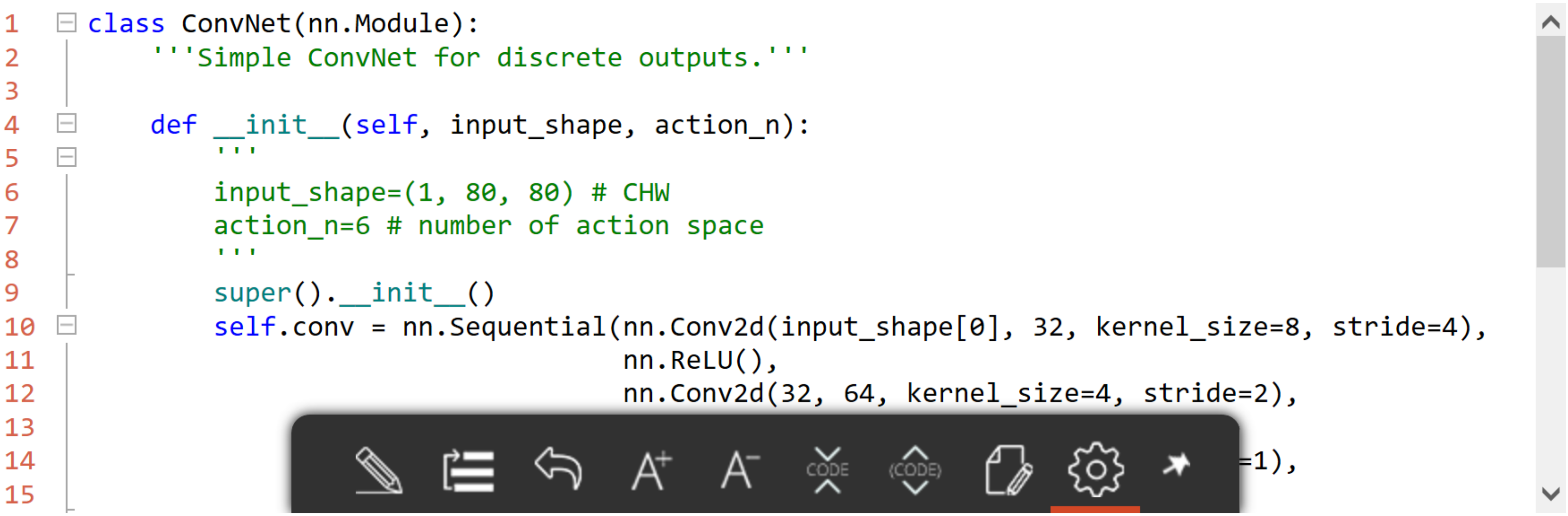
$$Q^*(s,a) = r_{t+1} + \gamma \max_{a_{t+1}} r_{t+2} + \gamma^2 \max_{a_{t+2}} r_{t+3} + \cdots = r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$$

Algorithms: Deep Q Learning

Q Network


s
a
Q^π
$Q^\pi(s,a)$
scalar
s
Q^π
$Q^\pi(s, a = \text{left})$
$Q^\pi(s, a = \text{right})$
$Q^\pi(s, a = \text{fire})$

for discrete action only



```
class ConvNet(nn.Module):
    '''Simple ConvNet for discrete outputs.'''

    def __init__(self, input_shape, action_n):
        '''
        input_shape=(1, 80, 80) # CHW
        action_n=6 # number of action space
        '''
        super().__init__()
        self.conv = nn.Sequential(nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
                                  nn.ReLU(),
                                  nn.Conv2d(32, 64, kernel_size=4, stride=2),

                                                                         =1),

```

Algorithms: Deep Q Learning

Q Learning

- Optimal Q-values should obey Bellman equation

$$Q^*(s,a) = \mathbb{E}_{s'}\left(r + \gamma \max_{a'} Q^*(s',a') | s,a\right)$$

- Use it as target!
- Optimize the MSE loss to find Q^*

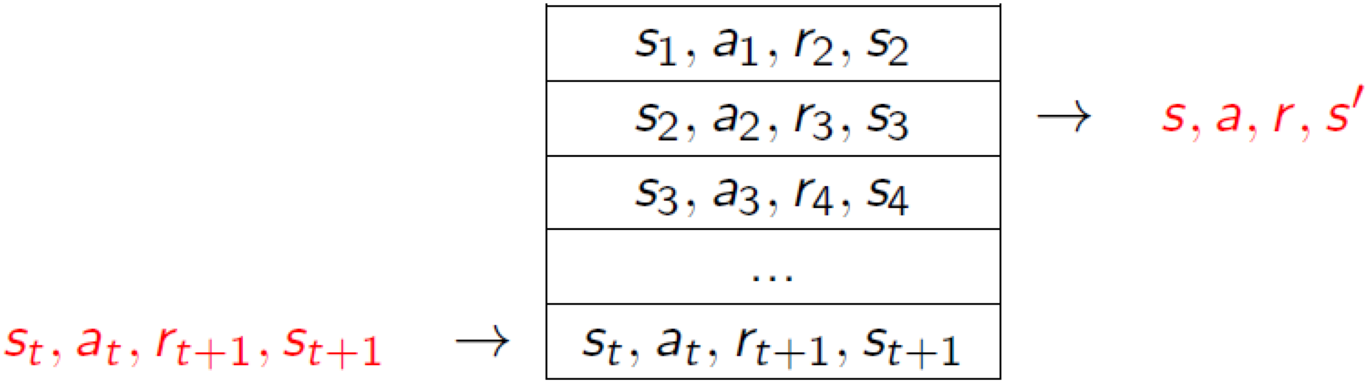
$$loss = \left(r + \gamma \max_{a'} Q(s',a',w) - Q(s,a,w)\right)^2$$

- Converge issues
    - Correlation between samples
    - Non-stationary target

Algorithms: Deep Q Learning

Experience Replay

- To remove correlations, build data-set from agent‘s own experience
- Sample from replay and update



s_1, a_1, r_2, s_2

s_2, a_2, r_3, s_3

s_3, a_3, r_4, s_4

$\ldots$

$s_t, a_t, r_{t+1}, s_{t+1}$

$s_t, a_t, r_{t+1}, s_{t+1} \rightarrow$

$\rightarrow s, a, r, s'$

Algorithms: Deep Q Learning

Fix Target Q Network

- To deal with non-stationarity, fix target Q network for a while

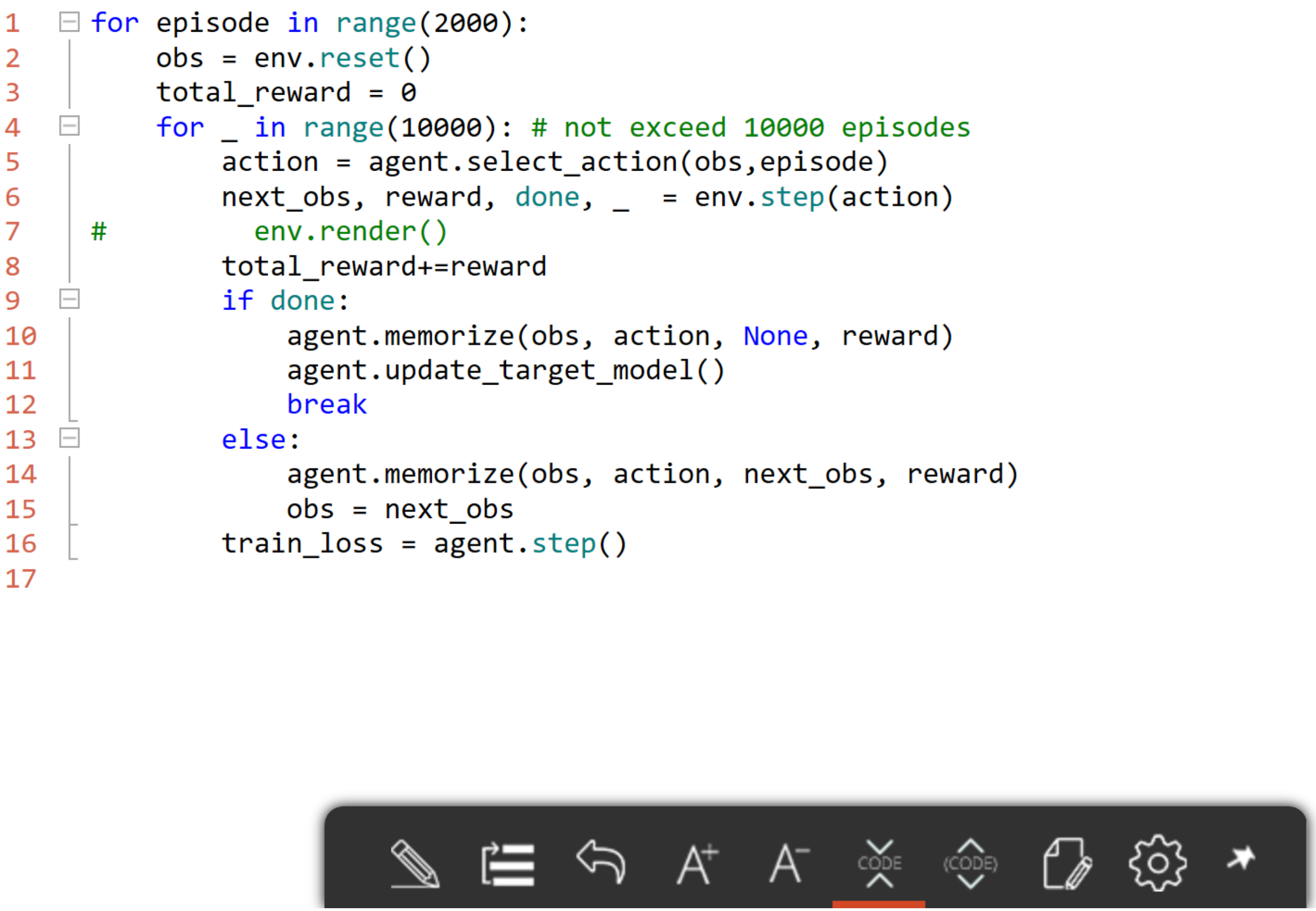
$$loss = \left(r + \gamma \max_{a'} Q(s', a', w^{-}) - Q(s, a, w)\right)^2$$

- Double DQN
  - Current Q-network w is used to select actions
  - Older target Q-network w^{-} is used to evaluate actions

$$loss = \left(r + \gamma Q\left(s', argmax_{a'} Q(s', a', w), w^{-}\right) - Q(s, a, w)\right)^2$$

Algorithms: Deep Q Learning

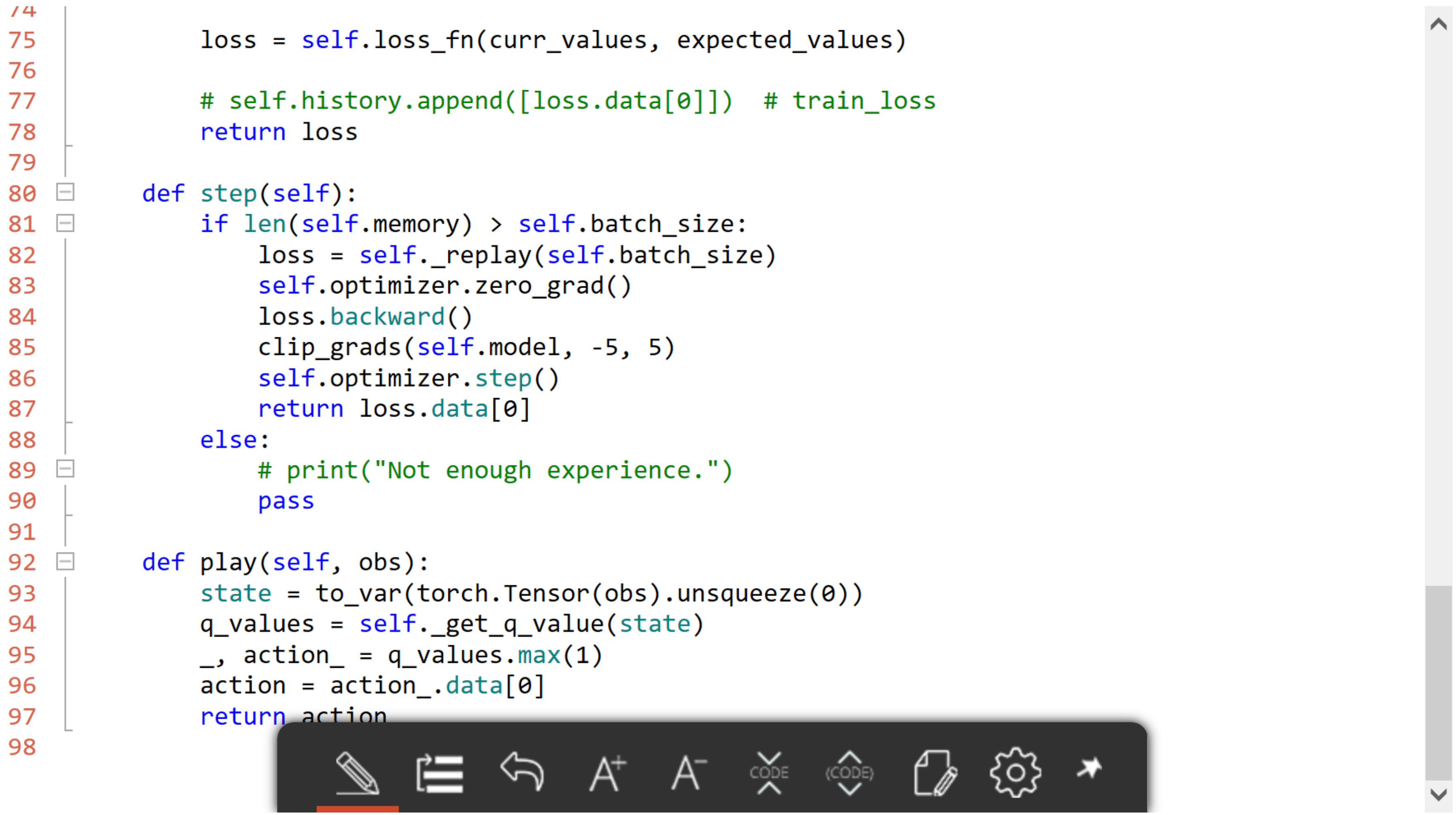
Double DQN code



```python
for episode in range(2000):
    obs = env.reset()
    total_reward = 0
    for _ in range(10000): # not exceed 10000 episodes
        action = agent.select_action(obs,episode)
        next_obs, reward, done, _  = env.step(action)
#         env.render()
        total_reward+=reward
        if done:
            agent.memorize(obs, action, None, reward)
            agent.update_target_model()
            break
        else:
            agent.memorize(obs, action, next_obs, reward)
            obs = next_obs
        train_loss = agent.step()
```

Algorithms: Deep Q Learning

Double DQN code



```
        loss = self.loss_fn(curr_values, expected_values)

        # self.history.append([loss.data[0]])  # train_loss
        return loss

    def step(self):
        if len(self.memory) > self.batch_size:
            loss = self._replay(self.batch_size)
            self.optimizer.zero_grad()
            loss.backward()
            clip_grads(self.model, -5, 5)
            self.optimizer.step()
            return loss.data[0]
        else:
            # print("Not enough experience.")
            pass

    def play(self, obs):
        state = to_var(torch.Tensor(obs).unsqueeze(0))
        q_values = self._get_q_value(state)
        _, action_ = q_values.max(1)
        action = action_.data[0]
        return action
```

Algorithms: REINFOCE

Policy Network

- Use network π with parameter θ to represent policy

$$a = \pi_\theta(s)$$

- Total reward

$$R(\theta) = \mathbb{E}_\tau\left(r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots |\pi_\theta\right)$$

- Maximize the parameter θ to get more reward **with gradient ascent** ?

$$\theta^* = argmax_\theta(R(\theta))$$

- The reward depends on π_θ by **sampling**, differentiable?

- How to compute $\frac{\partial R(\theta)}{\partial \theta}$?

Algorithms: REINFOCE

Policy Gradient

- Use Monte Carlo estimate!

$$R(\theta) = \mathbb{E}_{\tau}(R(\tau)|\pi_{\theta}) = \sum_{\tau} R(\tau)P_{\theta}(\tau) \cong \frac{1}{n}\sum_{\tau}^{n} R(\tau)$$

$$\frac{\partial R(\theta)}{\partial\theta} = \sum_{\tau} R(\tau)\frac{\partial P_{\theta}(\tau)}{\partial\theta} = \sum_{\tau} R(\tau)P_{\theta}(\tau)\frac{1}{P_{\theta}(\tau)}\frac{\partial P_{\theta}(\tau)}{\partial\theta} = \sum_{\tau} R(\tau)P_{\theta}(\tau)\frac{\partial logP_{\theta}(\tau)}{\partial\theta}$$

$$\cong \frac{1}{n}\sum_{\tau}^{n} R(\tau)\frac{\partial logP_{\theta}(\tau)}{\partial\theta} = \frac{1}{n}\sum_{\tau}^{n} R(\tau)\sum_{t=1}^{H_{\tau}}\frac{\partial logP_{\theta}(a_t|s_t)}{\partial\theta}$$

- Intuition
  - $R(\tau) > 0$, try to increase $P_{\theta}(\tau)$
  - $R(\tau) < 0$, try to decrease $P_{\theta}(\tau)$

Algorithms: REINFOCE

Temporal Structure

$$R_t(\tau) = \sum_{t}^{H_\tau} \gamma^{t-1} r_t$$

Don’t depend on a_t

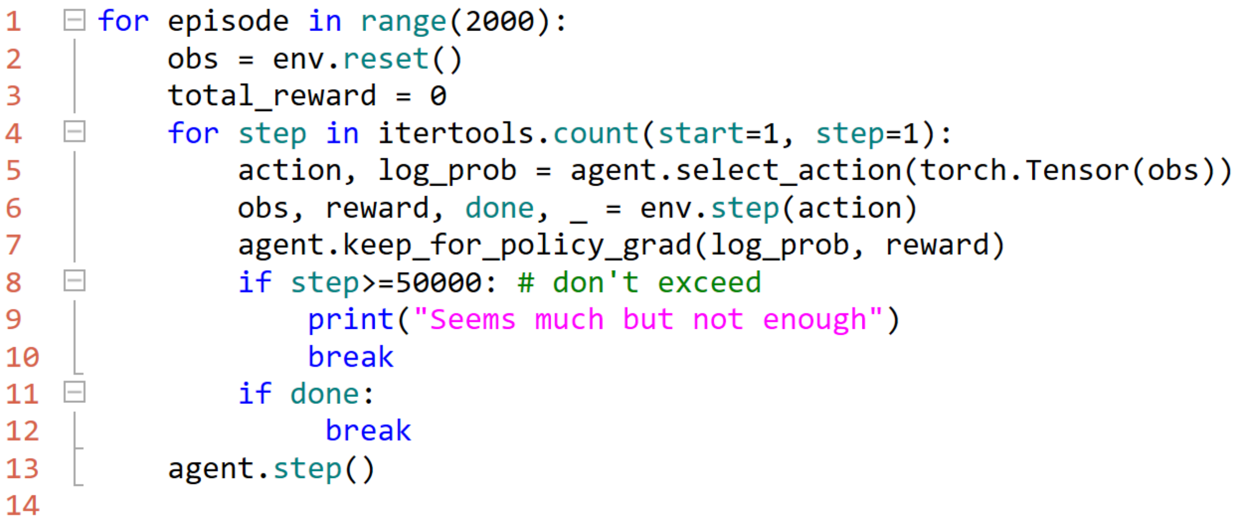
$$\frac{\partial R(\theta)}{\partial \theta} = \frac{1}{n}\sum_{\tau}^{n}\left(\boxed{r_0 + \cdots + \gamma^{t-2} r_{t-1}} + \sum_{i=t}^{H_\tau} \gamma^{i-1} r_i\right)\sum_{t=0}^{H_\tau}\frac{\partial logP_\theta(a_t|s_t)}{\partial \theta}$$

$$= \frac{1}{n}\sum_{\tau}^{n}\sum_{t=0}^{H_\tau}\frac{\partial logP_\theta(a_t|s_t)}{\partial \theta}\boxed{\sum_{i=t}^{H_\tau} \gamma^{i-1} r_i}$$

Advantage

Algorithms: REINFOCE

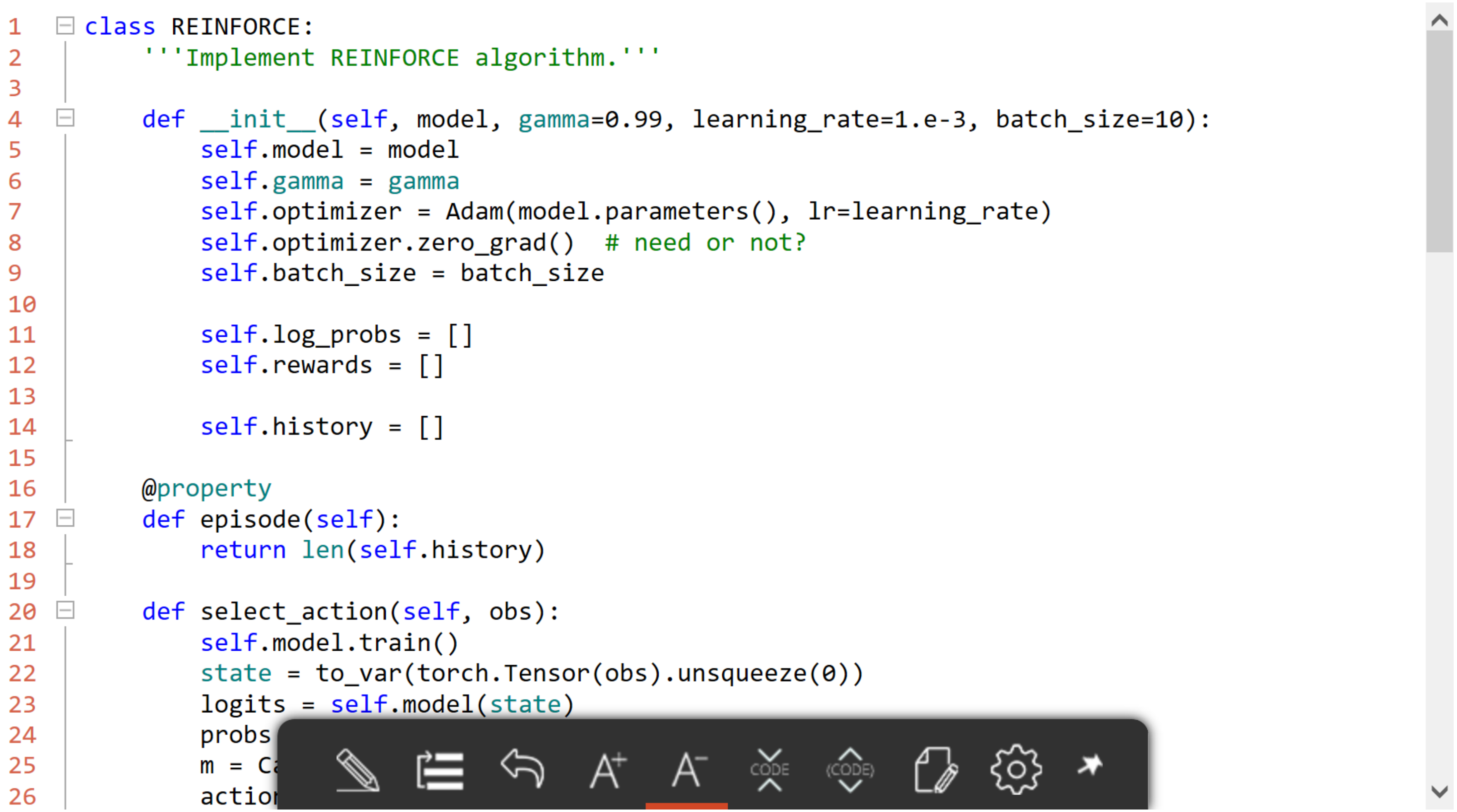
REINFOCE code



```python
for episode in range(2000):
    obs = env.reset()
    total_reward = 0
    for step in itertools.count(start=1, step=1):
        action, log_prob = agent.select_action(torch.Tensor(obs))
        obs, reward, done, _ = env.step(action)
        agent.keep_for_policy_grad(log_prob, reward)
        if step>=50000: # don't exceed
            print("Seems much but not enough")
            break
        if done:
             break
    agent.step()

```

Algorithms: REINFOCE

REINFOCE code



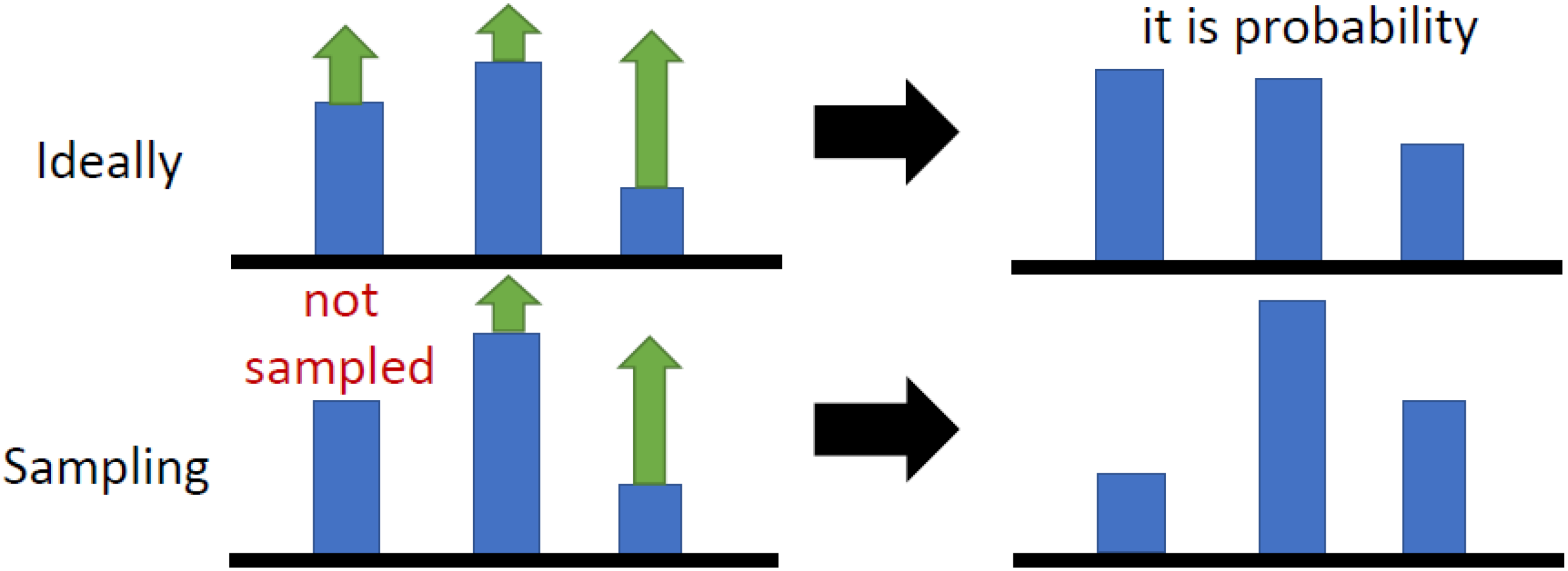
```python
class REINFORCE:
    '''Implement REINFORCE algorithm.'''

    def __init__(self, model, gamma=0.99, learning_rate=1.e-3, batch_size=10):
        self.model = model
        self.gamma = gamma
        self.optimizer = Adam(model.parameters(), lr=learning_rate)
        self.optimizer.zero_grad()  # need or not?
        self.batch_size = batch_size

        self.log_probs = []
        self.rewards = []

        self.history = []

    @property
    def episode(self):
        return len(self.history)

    def select_action(self, obs):
        self.model.train()
        state = to_var(torch.Tensor(obs).unsqueeze(0))
        logits = self.model(state)
        probs
        m = C
        actio
```

Algorithms: Actor-Critic

REINFORCE with Baseline

Still unbiased estimate!
[Williams 1992]

$$\frac{\partial R(\theta)}{\partial \theta} = \frac{1}{n}\sum_{\tau}^{n}\sum_{t=0}^{H_\tau}\frac{\partial logP_\theta(a_t|s_t)}{\partial \theta}(\sum_{i=t}^{H_\tau}\gamma^{i-1}r_i \boxed{- b)}$$

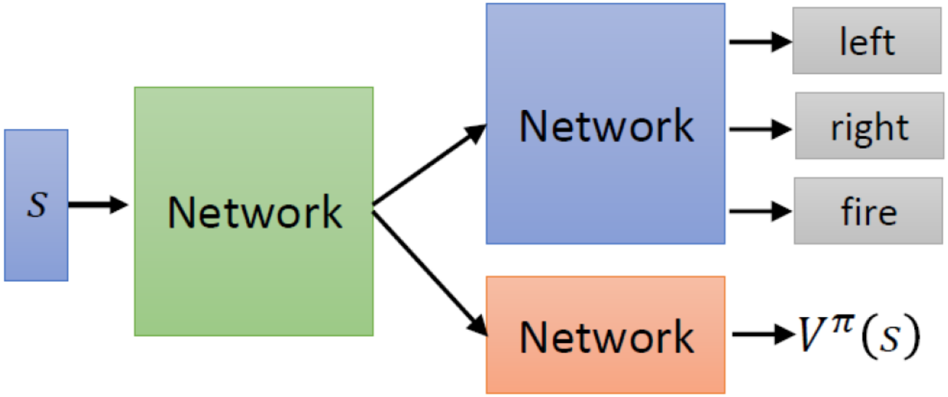


Algorithms: Actor-Critic

Advantage Actor Critic (A2C)

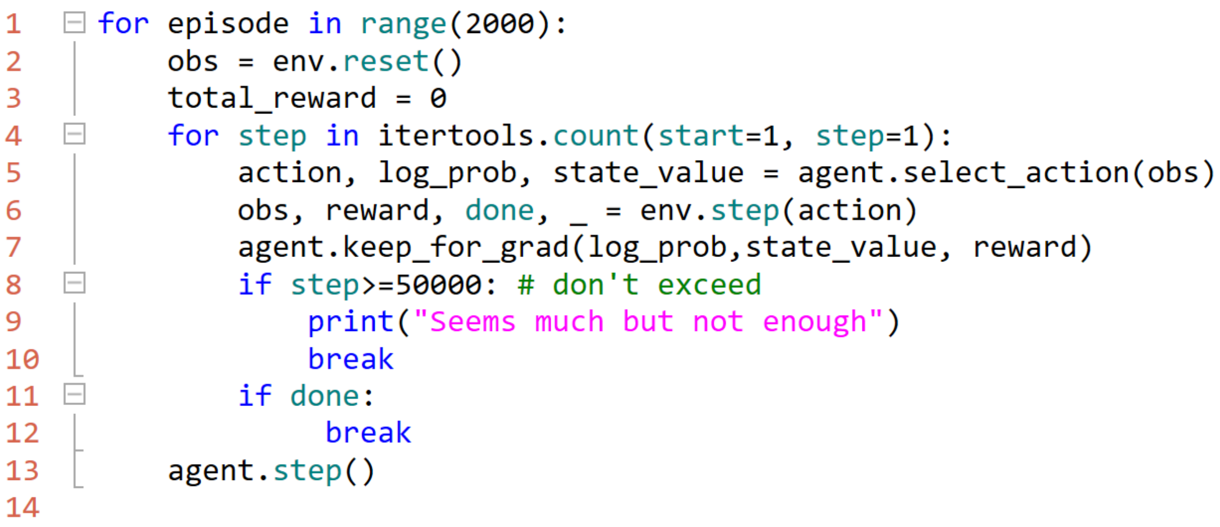
$$\frac{\partial R(\theta)}{\partial\theta} = \frac{1}{n}\sum_{\tau}^{n}\sum_{t=0}^{H_\tau}\frac{\partial logP_\theta(a_t|s_t)}{\partial\theta}(\sum_{i=t}^{H_\tau}\gamma^{i-1}r_i - V(s_t))$$

$$V(s_t)\ try\ to\ fit\ \sum_{i=t}^{H_\tau}\gamma^{i-1}r_i$$



Algorithms: Actor-Critic

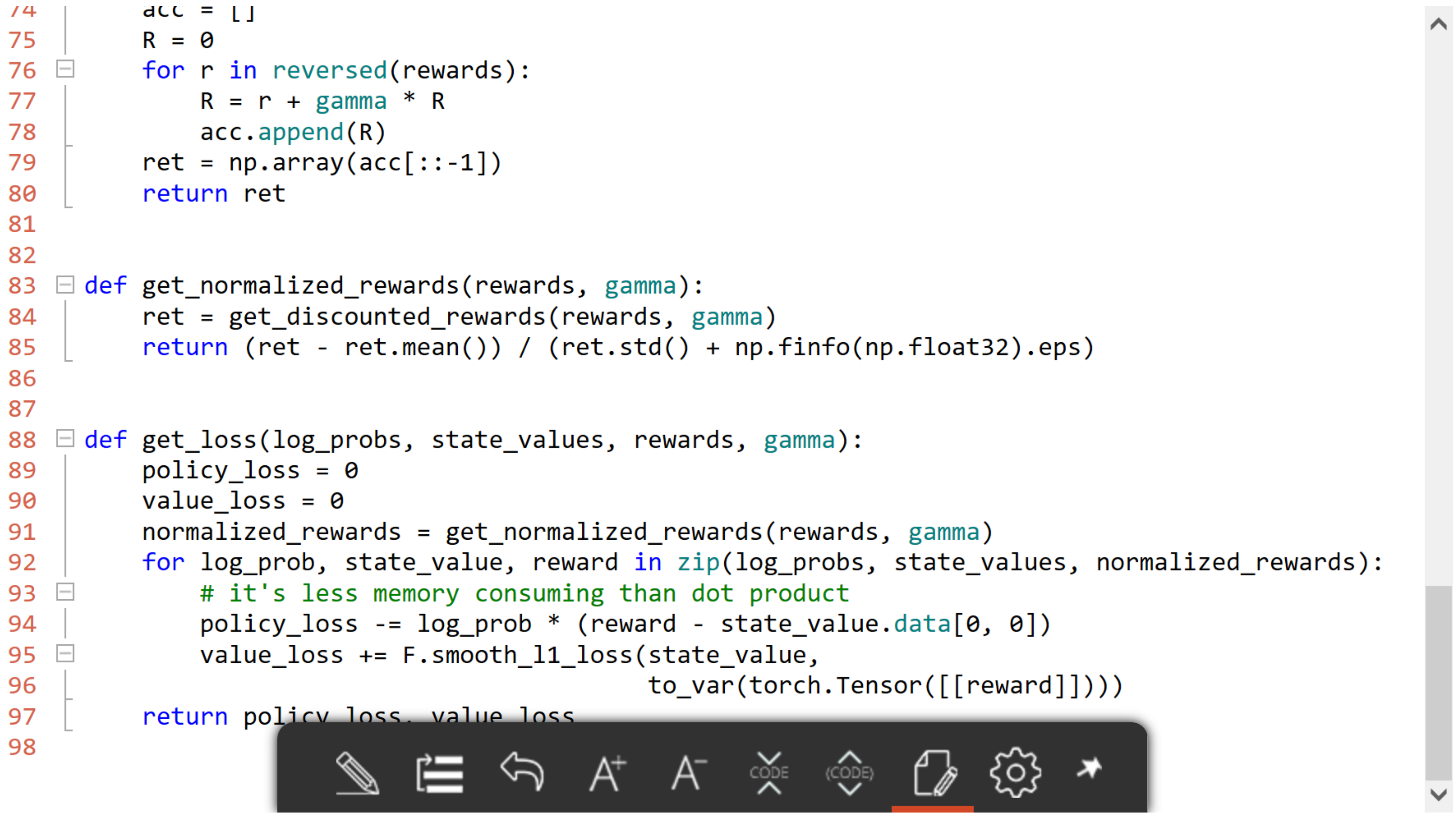
A2C code



```
for episode in range(2000):
    obs = env.reset()
    total_reward = 0
    for step in itertools.count(start=1, step=1):
        action, log_prob, state_value = agent.select_action(obs)
        obs, reward, done, _ = env.step(action)
        agent.keep_for_grad(log_prob,state_value, reward)
        if step>=50000: # don't exceed
            print("Seems much but not enough")
            break
        if done:
             break
    agent.step()

```

Algorithms: Actor-Critic

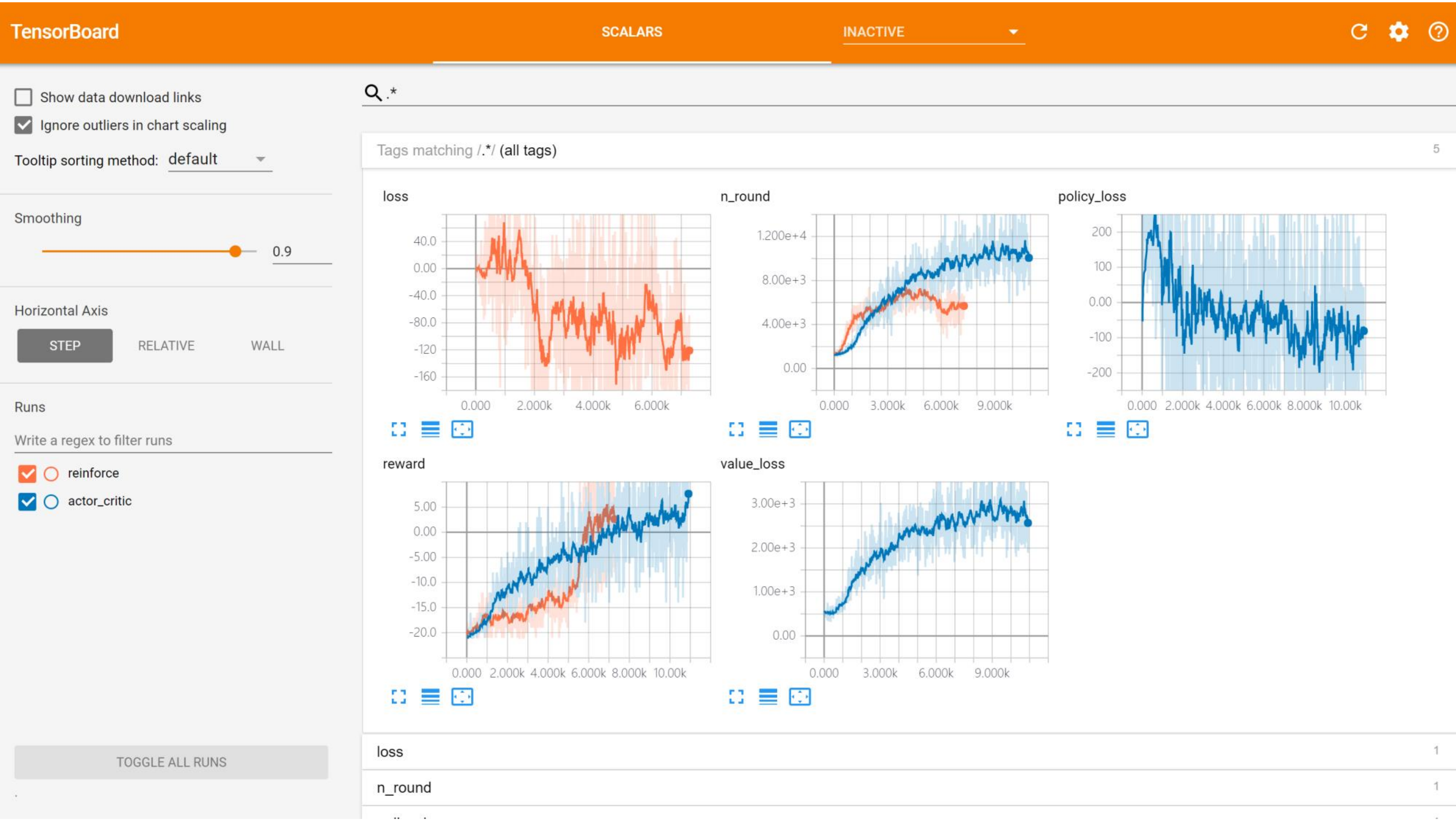
A2C code



```
    acc = []
    R = 0
    for r in reversed(rewards):
        R = r + gamma * R
        acc.append(R)
    ret = np.array(acc[::-1])
    return ret


def get_normalized_rewards(rewards, gamma):
    ret = get_discounted_rewards(rewards, gamma)
    return (ret - ret.mean()) / (ret.std() + np.finfo(np.float32).eps)


def get_loss(log_probs, state_values, rewards, gamma):
    policy_loss = 0
    value_loss = 0
    normalized_rewards = get_normalized_rewards(rewards, gamma)
    for log_prob, state_value, reward in zip(log_probs, state_values, normalized_rewards):
        # it's less memory consuming than dot product
        policy_loss -= log_prob * (reward - state_value.data[0, 0])
        value_loss += F.smooth_l1_loss(state_value,
                                       to_var(torch.Tensor([[reward]])))
    return policy_loss, value_loss
```

Algorithms: Actor-Critic

A2C and REINFOCE on PONG



TensorBoard SCALARS INACTIVE

☐ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.9

Horizontal Axis: STEP RELATIVE WALL

Runs

Write a regex to filter runs

☑ ○ reinforce

☑ ○ actor_critic

TOGGLE ALL RUNS

.*

Tags matching /.*/ (all tags) 5

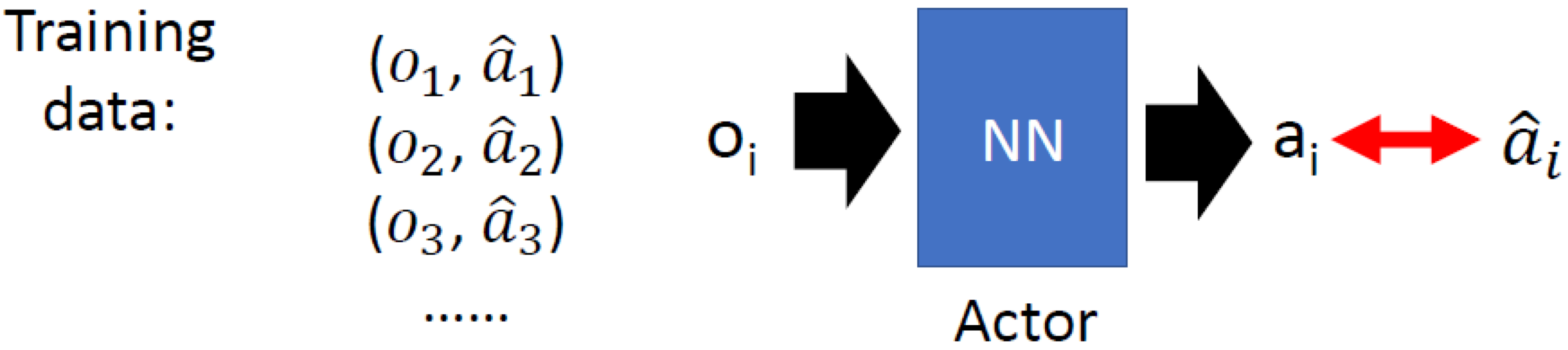
loss 1

n_round 1

Algorithms: Imitation Learning

Behavior Cloning

- Just like supervised learning

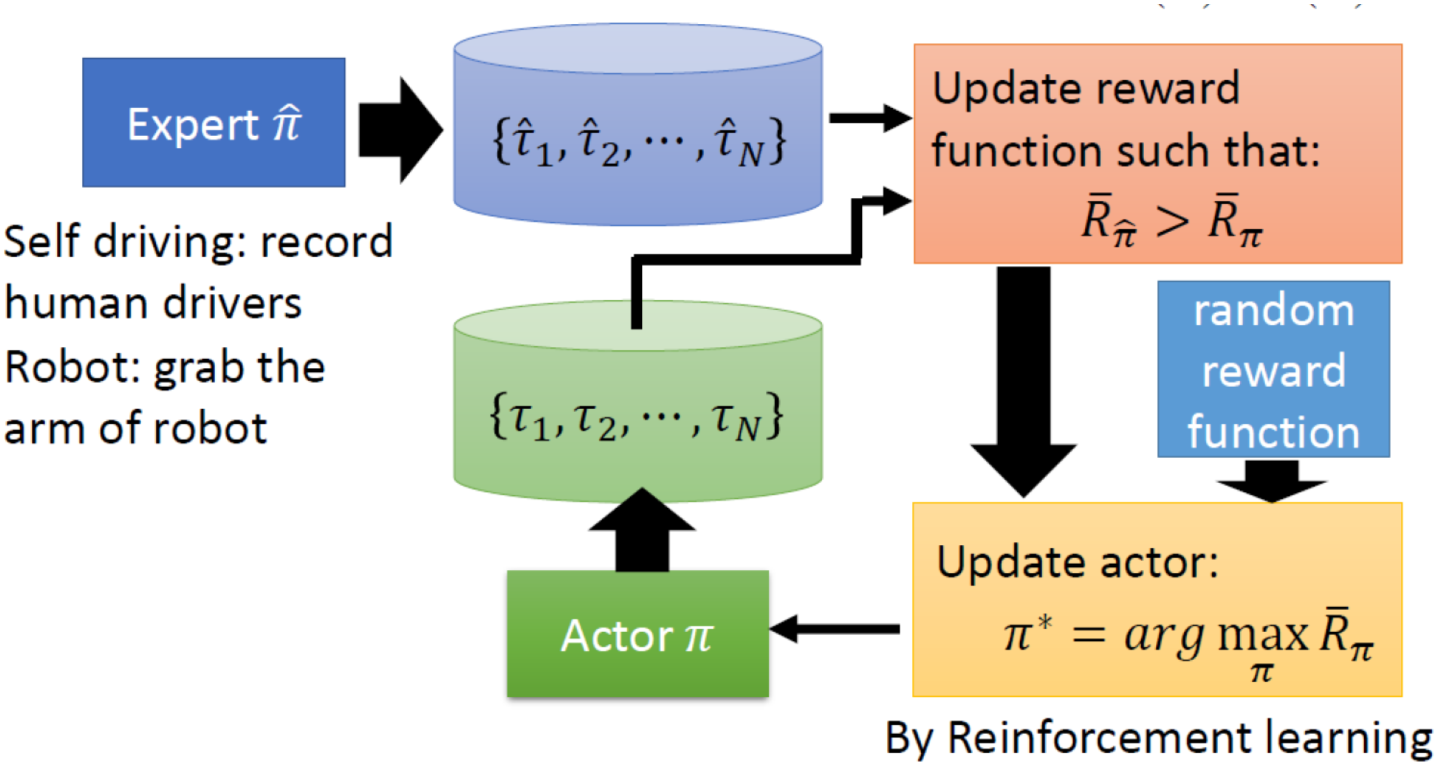


- Train and test time mismatch

Algorithms: Imitation Learning

Inverse Reinforcement Learning

- Learning experts' reward!



Expert $\hat{\pi}$

$\{\hat{\tau}_1, \hat{\tau}_2, \cdots, \hat{\tau}_N\}$

Self driving: record human drivers

Robot: grab the arm of robot

$\{\tau_1, \tau_2, \cdots, \tau_N\}$

Actor π

Update reward function such that:

$$\bar{R}_{\hat{\pi}} > \bar{R}_{\pi}$$

random reward function

Update actor:

$$\pi^* = arg \max_{\pi} \bar{R}_{\pi}$$

By Reinforcement learning

Algorithms: Imitation Learning

Third-Person Imitation Learning

- (I)RL+GAN

- Ref: Bradly C. Stadie, Pieter Abbeel, Ilya Sutskever, “Third-Person Imitation Learning”, arXiv preprint, 2017

First Person

http://lasa.epfl.ch/research_new/ML/index.php

Third Person

https://kknews.cc/sports/q5kbb8.html

http://sc.chinaz.com/Files/pic/icons/1913/%E6%9C%BA%E5%99%A8%E4%BA%BA%E5%9B%BE%E6%A0%87%E4%B8%8B%E8%BD%BD34.png

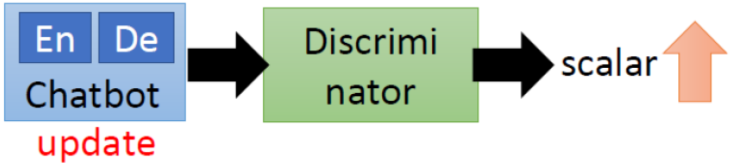
Agenda

- Introduction
- Algorithms
  - Deep Q Learning
  - REINFOCE
  - Actor-Critic
  - Imitation Learning
- Applications
  - SeqGAN
  - Hard Attention

Applications

SeqGAN

- How to sample in discrete space?
- Generate words as sequence of actions
- Consider the output of discriminator as reward
- Update generator to get maximum reward



$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{i=1}^{N}\left(\underset{\text{Discriminator Score}}{\overset{\text{reward}}{\boxed{D\left(h^i,x^i\right)}}} - b\right)\nabla log P_\theta\left(x^i|h^i\right)$$

Applications

Hard Attention

Figure 2. Attention over time. As the model generates each word, its attention changes to reflect the relevant parts of the image. "soft" (top row) vs "hard" (bottom row) attention. (Note that both models generated the same captions in this example.)

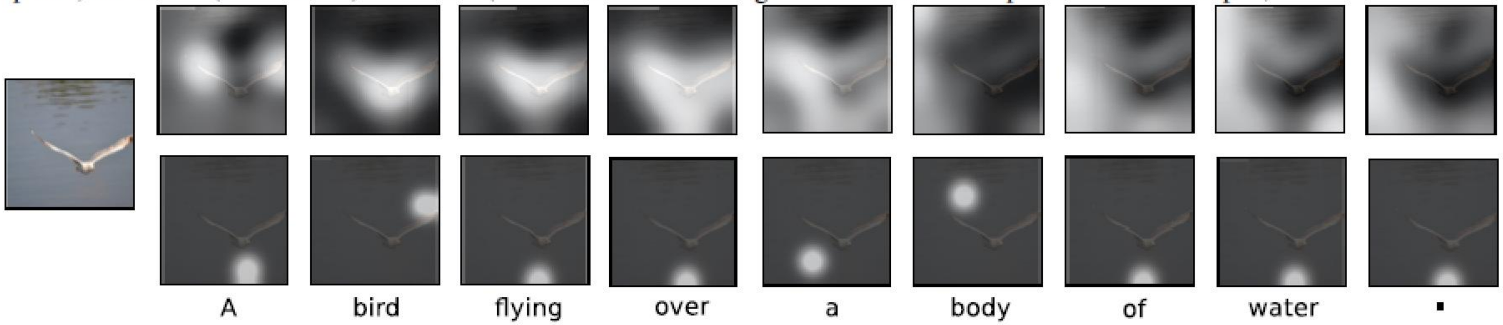
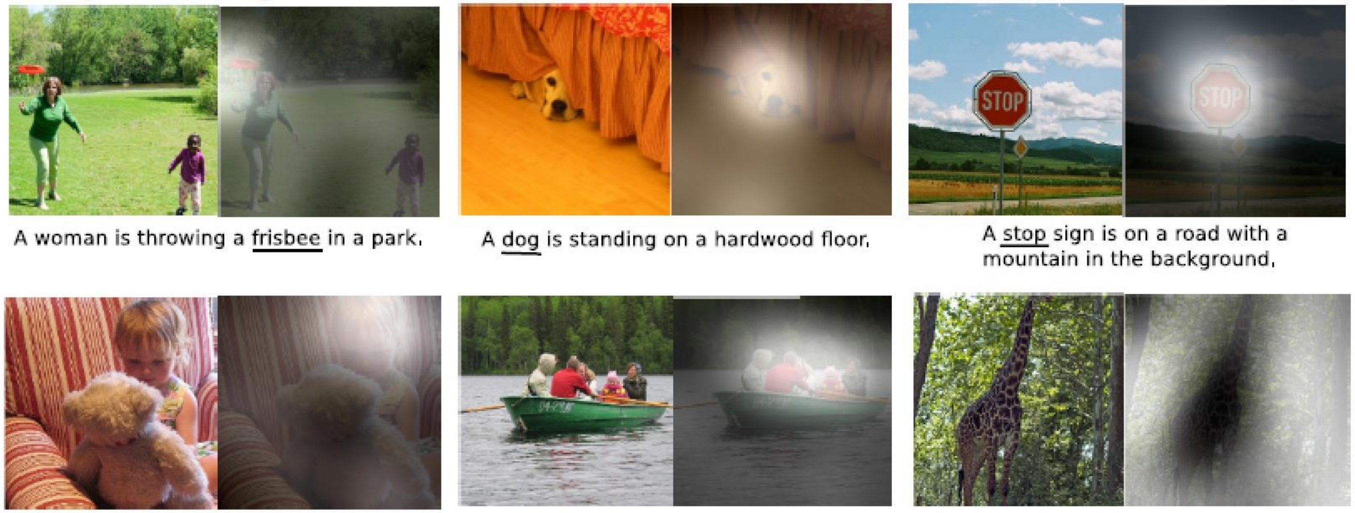


Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)



A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

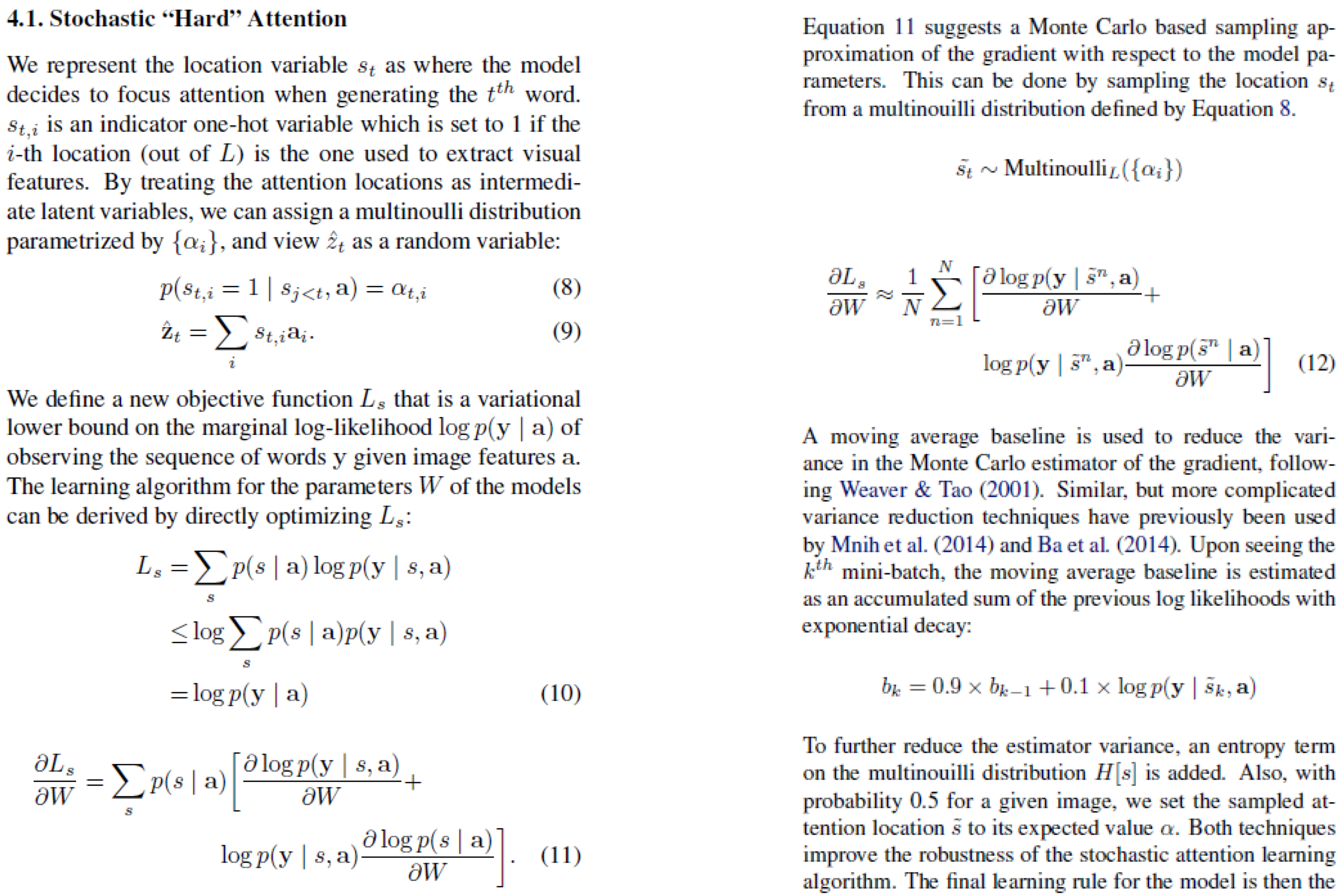
A little girl sitting on a bed with a teddy bear.

A group of people sitting on a boat in the water.

A giraffe standing in a forest with trees in the background.

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention

Applications
Hard Attention



4.1. Stochastic "Hard" Attention

We represent the location variable s_t as where the model decides to focus attention when generating the t^{th} word. $s_{t,i}$ is an indicator one-hot variable which is set to 1 if the i-th location (out of L) is the one used to extract visual features. By treating the attention locations as intermediate latent variables, we can assign a multinoulli distribution parametrized by $\{\alpha_i\}$, and view $\hat{z}_t$ as a random variable:

$$p(s_{t,i} = 1 \mid s_{j<t}, \mathrm{a}) = \alpha_{t,i} \tag{8}$$

$$\hat{\mathbf{z}}_t = \sum_i s_{t,i}\mathbf{a}_i. \tag{9}$$

We define a new objective function L_s that is a variational lower bound on the marginal log-likelihood $\log p(\mathrm{y} \mid \mathrm{a})$ of observing the sequence of words y given image features a. The learning algorithm for the parameters W of the models can be derived by directly optimizing L_s:

$$\begin{aligned} L_s &= \sum_s p(s \mid \mathrm{a}) \log p(\mathrm{y} \mid s, \mathrm{a}) \\ &\leq \log \sum_s p(s \mid \mathrm{a}) p(\mathrm{y} \mid s, \mathrm{a}) \\ &= \log p(\mathrm{y} \mid \mathrm{a}) \end{aligned} \tag{10}$$

$$\frac{\partial L_s}{\partial W} = \sum_s p(s \mid \mathrm{a}) \left[\frac{\partial \log p(\mathrm{y} \mid s, \mathrm{a})}{\partial W} + \log p(\mathrm{y} \mid s, \mathrm{a}) \frac{\partial \log p(s \mid \mathrm{a})}{\partial W} \right]. \tag{11}$$

Equation 11 suggests a Monte Carlo based sampling approximation of the gradient with respect to the model parameters. This can be done by sampling the location s_t from a multinouilli distribution defined by Equation 8.

$$\tilde{s}_t \sim \text{Multinoulli}_L(\{\alpha_i\})$$

$$\frac{\partial L_s}{\partial W} \approx \frac{1}{N} \sum_{n=1}^{N} \left[\frac{\partial \log p(\mathbf{y} \mid \tilde{s}^n, \mathbf{a})}{\partial W} + \log p(\mathbf{y} \mid \tilde{s}^n, \mathbf{a}) \frac{\partial \log p(\tilde{s}^n \mid \mathbf{a})}{\partial W} \right] \tag{12}$$

A moving average baseline is used to reduce the variance in the Monte Carlo estimator of the gradient, following Weaver & Tao (2001). Similar, but more complicated variance reduction techniques have previously been used by Mnih et al. (2014) and Ba et al. (2014). Upon seeing the k^{th} mini-batch, the moving average baseline is estimated as an accumulated sum of the previous log likelihoods with exponential decay:

$$b_k = 0.9 \times b_{k-1} + 0.1 \times \log p(\mathbf{y} \mid \tilde{s}_k, \mathbf{a})$$

To further reduce the estimator variance, an entropy term on the multinouilli distribution $H[s]$ is added. Also, with probability 0.5 for a given image, we set the sampled attention location $\tilde{s}$ to its expected value α. Both techniques improve the robustness of the stochastic attention learning algorithm. The final learning rule for the model is then the

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention

Reference

- NTU ADLxMLDS course
- Berkeley Deep RL Bootcamp
- ICML 2016 Tutorial
- My GitHub Repository

Thank You