Introduction to DRL

Introduction to DRL

P. Durdevic[1]

[1]Department of Energy Technology
Aalborg University

Wednesday 13th July, 2022

Table of Contents

Introduction to Deep Reinforcement Learning

Introduction to Deep Reinforcement Learning (DRL)

Why DRL Scaling up RL to high-dimensional problems

How is it possible Due to neural Networks (NN) being powerful function approximators

Another benefit DRL can deal with the curse of dimensionality (where e.g. tabular methods suffer)

Example Deep Reinforcement Learning (DRL) allows for control of robotic systems using inputs from a camera in the real world , Arulkumaran et al. (2017)

HOW in general? Train Deep NN (DNN) to approximate the optimal policy π^* and/or the optimal value functions V^*, Q^*, A^*, Arulkumaran et al. (2017)

Summary

Value-based and policy-based methods

Value-based Methods We focus on the value of the *state* (V), or value of the *state-action* (Q)

- Central topic in **Value Iteration** and **Q-learning**
- To obtain these values, we used the **Bellman equation** in the **previous lecture**
  - which expresses the value on the current step via the values on the next step *(it makes a prediction from a prediction)*

Ultimate goal of Reinforcement learning:

- We wish to learn the **optimal policy** $\pi*$, through an interaction with the **Environment**
- **So far, we've been learning at value-based methods, where we first find an estimate of the optimal action-value function q* from which we obtain the optimal policy** $\pi*$**.**

Rewrite

Policy-Based Methods

TODO

On-Policy and Off-Policy Algorithms

On-Policy and Off-Policy Algorithms

insert a good illustration of RL

How training iterations make use of data

On-Policy Trains on the data which is generated while using the current policy π. i.e. each training iteration uses only on the current policy π_1 to generate the data.

- **Consequence** Data is discarded after training as it has become unusable
- **Efficiency** Sample-inefficient, and require more training data.
- **Examples** SARSA, REINFORCE, Actor-Critic methods, PPO

On-Policy Any data colleted can be use for training

- **Efficiency** More sample-efficient
- **Consequence** Might require more storage
- **Examples** DQN

Introduction To Deep Learning (DL) for RL

THE GOOD DNNs are good at complex nonlinear function approximation , a powerful function approximator.

Their Structure Alternating layers of parameters and non-linear/linear activation functions.

DL History Practical application started with Yann Lecun's work on convolutional neural networks (CNN) in 1989 LeCun et al. (1989), their usefulness has exploded after Alex Krizhevsky's work with deep convolutional neural network (DCNN) and classification of 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest Krizhevsky et al. (2012).

DL RL History 1991 NN trained using RL to play backgammon Tesauro et al. (1995), in 2015 Google Deepmind achieved human-level performance on Atari games Mnih et al. (2015) using a deep Q-network (DQN), which positioned Deep learning in the center of RL research Graesser and Keng (2019).

Graesser and Keng (2019)

Deep Learning (DL) for RL

Short Recap on Deep Learning

- In the *forward pass* they can compute an output from the input
- The network consists of parameters (weights), i.e. it is parametrized by θ
- Generate a data-set of inputs and outputs
- Define a loss function which represents the error between network-predicted output and the output from the data-set
- We wish to minimize the loss by adjusting the parameters (weights)
- We use gradient descent *("go in direction of steepest descent on the loss surface in search of the global minimum")*

Graesser and Keng (2019)

Deep Learning (DL) for RL

Short Recap on Deep Learning

Example
How to structure and design a DNN, see pages 17-18 in Graesser and Keng (2019). **TODO**

Deep Learning (DL) for RL

- Training of the network is done ad hoc
- The input and output data are generated through the agents interactions with the environment [states,rewards]
- Network training tightly coupled with the MDP loop
- Issues with Gradient Descent, discussed later bottom page 18

Introduction to DRL: Policy-Gradient Methods: REINFORCE Algorithm

Policy-Gradient Methods: REINFORCE Algorithm

Introduction

REINFORCE

- Introduced in paper *"Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning"*
- learns a **parametrized policy** which produces **action probabilities** from **states**. **Agents** use this **policy directly** to **act** in an **environment**
- **Action probabilities** are *changed* by following the **policy gradient**, therefore REINFORCE is known as a **policy gradient algorithm**.

Three main components:

1. A parametrized policy
2. An objective to be maximized
3. A method for updating the policy parameters

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

The Policy

Policy π A **function** that maps the *state* (s) to *action* (a) *probabilities*

Purpose Used to sample an *action* $a \sim \pi(s)$

Goal of good policy *Maximize* the **cumulative discounted rewards**

We can use FUNCTION APROXIMATIONS to represent the policy:

Learnable parameters Using a DNN, we can represent the policy by learnable parameters θ, called the **Policy Network** π_θ, i.e. the policy is parametrized by θ

Learning the Policy The process of learning a good policy corresponds to searching for a good set of values for θ

Differentiable network As we wish to optimize the network, i.e. search for optimal values of θ, the policy network must be **Differentiable**

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

The Objective Function

The objective — The objective that is minimized by the agent (agents goal)

- The goal: e.g. **highest score**

- An agent acting in a **environment** generates a trajectory
- Result is a sequence of rewards along with the states and actions, i.e.:

$$\tau = s_0, a_0, r_0, \ldots, s_T, a_T, r_T \tag{1}$$

Discounted sum of rewards *(from time-step t to the end of a trajectory)*, is called the **return** $R_t(\tau)$

$$R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t \tag{2}$$

- The **OBJECTIVE** is the expected return over all complete trajectories generated by an agent

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r\right] \tag{3}$$

- The expectation is calculated over many trajectories sampled from a policy $(\tau \sim \pi_\theta)$.
- This expectation approaches the true value as more samples are gathered

Based on Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Policy Gradient

The agents acts through the policy π_θ and the target i maximized through the objective $J(\pi_\theta)$
The policy gradient algorithm solves the following problem:

$$\max_{\theta} J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} [R(\tau)] \tag{4}$$

- To **maximize** the **objective** we perform **gradient ascent** on the **policy parameters**, as the **gradient** points in the direction of **steepest ascent**.
- To improve on the objective $J(\pi_\theta)$ compute the gradient and use it to update the parameters

text from Graesser and Keng (2019)

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta) \tag{5}$$

With the learning rate α, and $\nabla_\theta J(\pi_\theta)$ is the policy gradient:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} R_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \tag{6}$$

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Policy Gradient

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} R_t(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{7}$$

where:

- The action is sampled from the policy; $a_t \sim \pi_\theta(s_t)$
- **probability of an action taken by the agent at time step** t is given by $\pi_\theta(a_t|s_t)$, i.e. it depends on the state a time$t$
- In the *rhs.* the **log probability** of the **action** wrt. θ is multiplied by the **return** of a **trajectory** $R_t(\tau)$ [1]

Equation (7) explain this based on following commented text, it is from Graesser and Keng (2019) page 27-28

[1] log probability is a **logarithm of a probability, The use of log probabilities means representing probabilities on a logarithmic scale, instead of the standard $[0, 1][0, 1]$ unit interval. Since the probabilities of independent events multiply, and logarithms convert multiplication to addition, log probabilities of independent events add.**

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Policy Gradient Derivation

Task Derive the policy gradient, equation (7), from the gradient of the objective:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{8}$$

Issue $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$ cannot be differentiated with respect to θ, as the rewards r_t are generated by an unknown function $\mathcal{R}(s_t, a_t, s_{t+1})$

"The only way for the policy variables θ to influence $R(\tau)$ is by changing the state and action distributions which, in turn, change the rewards received by an agent. We therefore need to transform Equation (8) into a form where we can take a gradient with respect to θ." Graesser and Keng (2019)

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Policy Gradient Derivation

"Given a function $f(x)$, a parametrized probability distribution $p(x|\theta)$, and its expectation $\mathbb{E}_{x\sim p(x\ \theta)}[f(x)]$, the gradient of the expectation can be rewritten as follows:" Graesser and Keng (2019)

$$
\begin{aligned}
&\nabla_\theta \mathbb{E}_{x\sim p(x\ \theta)}[f(x)] \\
&= \nabla_\theta \int dx\, f(x) p(x|\theta) && (\textit{definition of expectation}) \\
&= \int dx \nabla_\theta \left(p(x|\theta) f(x)\right) && (\textit{bring in } \nabla_\theta) \\
&= \int dx \left(f(x) \nabla_\theta p(x|\theta) + p(x|\theta) \nabla_\theta f(x)\right) && (\textit{chain rule}) \\
&= \int dx\, f(x) \nabla_\theta p(x|\theta) && (\nabla_\theta f(x) = 0) \\
&= \int dx\, f(x) p(x|\theta) \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)} && \left(\textit{multiply} \frac{p(x|\theta)}{p(x|\theta)}\right) \\
&= \int dx\, f(x) p(x|\theta) \nabla_\theta \log p(x|\theta) && \left(\nabla_\theta \log p(x|\theta) = \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)}\right) \\
&= \mathbb{E}_x\left[f(x) p(x|\theta) \nabla_\theta \log p(x|\theta)\right] && (\textit{definition of expectation})
\end{aligned}
\tag{9}
$$

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Policy Gradient Derivation

finish page 29 top till eq. 2.15

$$\nabla_\theta \mathbb{E}_{x\sim p(x\ \ \theta)}[f(x)] = \mathbb{E}_x[f(x)p(x|\theta)\nabla_\theta \log p(x|\theta)] \tag{10}$$

Now substituting $x = \tau$, $f(x) = R(\tau), p(x|\theta) = p(\tau|\theta)$ we have:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)\nabla_\theta \log p(\tau|\theta)] \tag{11}$$

finish from page 29 eq. 2.15, (marked green in the text)

After rewriting we can with a equtaion that can be estimated using a policy network π_θ, and we can compute the gradient. *(note: this can be done automatically using NN libraires such as PyTorch)*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} R_t(\tau)\nabla_\theta \log \pi_\theta(a_{|}\, s_t)\right] \tag{12}$$

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Monte Carlo Sampling

- The REINFORCE algorithm numerically estimates the policy gradient using Monte Carlo sampling
- Monte Carlo sampling, generates data through random sampling and uses this data to approximate a function.
- An example of this for estimating π is shown in Graesser and Keng (2019),

$$\frac{\textit{area of circle}}{\textit{area of square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4} \tag{13}$$

- How many of the sampled dots are in the area of the circle

$$\textit{circle\_dots} = \sqrt{(x-0)^2(y-0)^2} \leq 1 \tag{14}$$

- Ratio of dots in circle and the total amount of dots, multiplied by 4 to get π from equation 13

$$\pi = \frac{\textit{circle\_dots}}{\textit{total\_dots}} \times 4 \tag{15}$$

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Monte Carlo Sampling

```python
import torch
from matplotlib import pyplot as plt

randNum = 10000
x = torch.rand(1,randNum)
y = torch.rand(1,randNum)
x_in = torch.zeros(1,randNum)
y_in = torch.zeros(1,randNum)
exp = torch.tensor(2)
origin = torch.tensor(0)

count = torch.tensor(0)
for i in range(randNum):
    if (torch.sqrt(torch.pow((x[0,i]-origin),exp) + torch.pow((y[0,i]-origin),exp)) < 1) or (torch.sqrt(torch.
        pow((x[0,i]-origin),exp) + torch.pow((y[0,i]-origin),exp)) == 1):
        x_in[0,i] = x[0,i]
        y_in[0,i] = y[0,i]
        count += 1
ratio = count/randNum
pi = ratio*4
print(pi)
plt.plot(x,y,'b.')
plt.plot(x_in,y_in,'r.')
plt.show()
```

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Monte Carlo Sampling

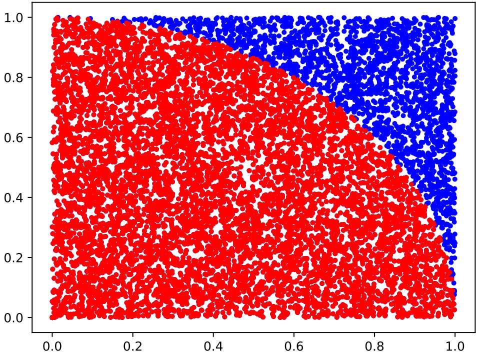

Figure: Monte Carlo

Policy-Gradient Methods: REINFORCE Algorithm

Monte Carlo for **REINFORCE**

TODO: Finish explaining

Numerically estimates the policy gradient (13) using Monte Carlo sampling?

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} R_t(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t) \right] \tag{13}$$

'The expectation $\mathbb{E}_{\tau \sim \pi_\theta}$ implies that as more trajectories τ are sampled using a policy π_θ and averaged, it approaches the actual policy gradient $\nabla_\theta J(\pi_\theta)$. Instead of sampling many trajectories per policy, we can sample just one as shown in Equation **??** *Graesser and Keng (2019)*

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} R_t(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t) \tag{14}$$

'This is how policy gradient is implemented—as a Monte Carlo estimate over sampled trajectories...' Graesser and Keng (2019)

TODO explain this

Based on, Graesser and Keng (2019)

Policy-Gradient Methods: REINFORCE Algorithm

Monte Carlo Sampling

The *on-policy algorithm* **REINFORCE** *algorithm*.

Algorithm 1 Pseudocode for the REINFORCE algorithm

Initialize learning rate α
Choose a discount rate $0 < \gamma \leq 1$
Initialize weights θ of a policy network π_θ at random
Choose a max number of episodes N
for *episode* $n < N$ **do**
    Generate a trajectory $\tau = [s_0, a_0, r_1, s_1, a_1, \ldots, s_T, a_T, r_T]$ following policy π_θ
    Set $\nabla_\theta J(\pi_\theta) = 0$
    for $t = 0, \ldots, T$ **do**
        $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t$
        $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t)$
    end for
    $\theta = \theta + \alpha \nabla_\theta J(\pi_\theta)$
end for

Based on, Graesser and Keng (2019); Jordi TORRES (2020)

Policy-Gradient Methods: REINFORCE Algorithm

Monte Carlo Sampling

- *With comments*

Algorithm 2 Pseudocode for the REINFORCE algorithm

Initialize learning rate α
Choose a discount rate $0 < \gamma \leq 1$
Initialize weights θ of a policy network π_θ at random
Choose a max number of episodes N
for *episode* $n < N$ **do**
  Generate a trajectory $\tau = [s_0, a_0, r_1, s_1, a_1, \ldots, s_T, a_T, r_T]$ following policy π_θ
  Set $\nabla_\theta J(\pi_\theta) = 0$
  for $t = 0, \ldots, T$ **do**
    $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_t'$ {compute the return $R_t(\tau)$ for each t in τ}
    $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$ {Estimate the policy gradient $\nabla_\theta J(\pi_\theta)$ using $R_t(\tau)$ and Sum $\nabla_\theta J(\pi_\theta)$ for all time steps}
  end for
  $\theta = \theta + \alpha \nabla_\theta J(\pi_\theta)$ {update policy network parameters θ}
end for

Based on, Graesser and Keng (2019); Jordi TORRES (2020)

Policy-Gradient Methods: REINFORCE Algorithm

Monte Carlo Sampling

Algorithm 3 Pseudocode for the REINFORCE algorithm

1: Initialize learning rate α
2: Choose a discount rate $0 < \gamma \leq 1$
3: Initialize weights θ of a policy network π_θ at random
4: Choose a max number of episodes N
5: **for** *episode* $n < N$ **do**
6: Generate a trajectory $\tau = [s_0, a_0, r_1, s_1, a_1, \ldots, s_T, a_T, r_T]$ following policy π_θ
7: Set $\nabla_\theta J(\pi_\theta) = 0$
8: **for** $t = 0, \ldots, T$ **do**
9: $R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r_t'$
10: $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$
11: **end for**
12: $\theta = \theta + \alpha \nabla_\theta J(\pi_\theta)$
13: **end for**

In an on-policy algorithm the parameter update equation depends on the current policy

Line 6 A trajectory is discarded after each parameter update *[on-policy algorithm]*

Line 9 The return $R_t(\tau)$ is generated by the current policy π_θ, $[\tau \sim \pi_{theta}]$

Line 10 The policy gradient depends only on action probabilities $\pi_\theta(a_t|s_t)$ generated by the current policy π_θ, *but not the past policy* π_θ'.

Based on, Graesser and Keng (2019); Jordi TORRES (2020)

Policy-Gradient Methods: REINFORCE Algorithm

Comments and Improvement (TEMP SLIDE MUST BE REWRITTEN) Graesser and Keng (2019)

'Our formulation of the REINFORCE algorithm estimates the policy gradient using Monte Carlo sampling with a single trajectory. This is an unbiased estimate of the policy gradient, but one disadvantage of this approach is that it has a high variance. In this section, we introduce a baseline to reduce the variance of the estimate. Following this, we will also discuss reward normalization to address the issue of reward scaling'
'When using Monte Carlo sampling, the policy gradient estimate may have high variance because the returns can vary significantly from trajectory to trajectory. This is due to three factors. First, actions have some randomness because they are sampled from a probability distribution. Second, the starting state may vary per episode. Third, the environment transition function may be stochastic.'
Method One
One way to reduce the variance of the estimate is to modify the returns by subtracting a suitable action-independent baseline

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} \left(R_t(\tau) - b(s_t)\right) \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{15}$$

One option for the baseline is the value function V^π. This choice of baseline motivates the **Actor-Critic algorithm.**
Method Two
An alternative is to use the mean returns over the trajectory.

$$b = \frac{1}{T} \sum_{t=0}^{T} R_t(\tau) \tag{16}$$

Note that this is a constant baseline per trajectory that does not vary with state st. It has the effect of centering the returns for each trajectory around 0. For each trajectory, on average, the best 50% of the actions will be encouraged, and the others discouraged.
To see why this is useful, consider the case where all the rewards for an environment are negative. Without a baseline, even when an agent produces a very good action, it gets discouraged because the returns are always negative. Over time, this can still result in good policies since worse actions will get discouraged even more, thus indirectly increasing the probabilities of better actions. However, it can lead to slower learning because probability adjustments can only be made in one direction. The converse happens for environments where all the rewards are positive. Learning is more effective when we can both increase and decrease the action probabilities. This requires having both positive and negative returns.

Extra: Probability Distributions in PyTorch

Policy-Gradient Methods: Hill Climbing algorithm

Probability Distributions: `TORCH.DISTRIBUTIONS`

FINISH

ideas `https://towardsdatascience.com/policy-based-methods-8ae60927a78d`

Policy-Gradient Methods: REINFORCE Algorithm

Probability Distributions: TORCH.DISTRIBUTIONS

FINISH

'The `distributions` package contains parameterizable probability distributions and sampling functions. This allows the construction of stochastic computation graphs and stochastic gradient estimators for optimization.' PyTorch (2020)

It is not possible to directly backpropagate through random samples. However, there are two main methods for creating surrogate functions that can be backpropagated through. These are the score function estimator/likelihood ratio estimator/REINFORCE and the pathwise derivative estimator. REINFORCE is commonly seen as the basis for policy gradient methods in reinforcement learning, and the pathwise derivative estimator is commonly seen in the reparameterization trick in variational autoencoders.

Score function

When the probability density function is differentiable with respect to its parameters, we only need sample() and $\log_p rob() to implement REINFORCE$:

`probs (Number, Tensor)` the probability of sampling

`logits (Number, Tensor)` the log-odds of sampling

Policy-Gradient Methods: REINFORCE Algorithm

Log Probability

FINISH THIS FROM NOTES

`ttps://pytorch.org/docs/stable/distributions.html` PROBABILITY DISTRIBUTIONS - TORCH.DISTRIBUTIONS for REINFORCE

Code Example: REINFORCE

Code Example - REINFORCE

Intoduction

- Next consider the **REINFORCE** algorithm
- The following code is a modified version of Code 2.1 in Graesser and Keng (2019)
- We shall go through the code guided by the pseudo code and the theory we have introduced

Code Example - REINFORCE

<class ’__main__.Pi’>

Line 1-9 Import libraries

Line 13-23 Construct the neural network

Output of the network:

```
Pi(
  (model): Sequential(
    (0): Linear(in_features=4, out_features=64, bias=
        True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=2, bias=
        True)
  )
)
```

```
1  from torch.distributions import Categorical
2  import gym
3  import numpy as np
4  import torch
5  import torch.nn as nn
6  import torch.optim as optim
7  import numpy as np
8  from matplotlib import pyplot as plt
9  from IPython import display
10
11 gamma = 0.99
12
13 class Pi(nn.Module):
14     def __init__(self, in_dim, out_dim):
15         super(Pi, self).__init__()
16         layers = [
17             nn.Linear(in_dim, 64),
18             nn.ReLU(),
19             nn.Linear(64, out_dim),
20         ]
21         self.model = nn.Sequential(*layers)
22         self.onpolicy_reset()
23         self.train() # set training mode
24
25     def onpolicy_reset(self):
26         self.log_probs = []
27         self.rewards = []
28
29     def forward(self, x):
30         pdparam = self.model(x)
31         return pdparam
```

Code Example - REINFORCE

<function Pi.act>

Line 25-40 The method `act` produces an action

Line 36-37 The action is sampled from a distribution

- Creates a categorical distribution parametrized by `logits`
  - `logits (Tensor)`: event log probabilities (un-normalized)

```
33  def act(self, state):
34      x = torch.from_numpy(state.astype(np.float32)) # to
    tensor
35      pdparam = self.forward(x) # forward pass
36      pd = Categorical(logits=pdparam) # probability
    distribution
37      action = pd.sample() # pi(a|s) in action via pd
38      log_prob = pd.log_prob(action) # log_prob of pi(a|s)
39      self.log_probs.append(log_prob) # store for training
40      return action.item()
```

https://pytorch.org/docs/stable/distributions.html

Code Example - REINFORCE

<function train>

Algorithm 4 Pseudocode for the REINFORCE algorithm

1: **for** $t = 0, \ldots, T$ **do**
2: $\quad R_t(\tau) = \sum_{t'=t}^{T} \gamma^{t'-t} r'_t$
3: $\quad \nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + R_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$
4: **end for**

Line 47 computing the returns, line 2 in algorithm 4

Line 52-53 loss is computed: sum of negative log probabilities multiplied by the returns, line 3 in algorithm 4

Line 52 (-) is used to maximize the objective using the default PyTorch optimizer (minimizer)

Line 55 compute gradients of the loss *(=policy gradient)*

Line 56 Policy parameters are updated using optimizer.step()

```
41 def train(pi, optimizer):
42     # Inner gradient-ascent loop of REINFORCE algorithm
43     T = len(pi.rewards)
44     rets = np.empty(T, dtype=np.float32) # the returns
45     future_ret = 0.0
46     # compute the returns efficiently
47     for t in reversed(range(T)):
48         future_ret = pi.rewards[t] + gamma * future_ret # equation 2.1
49         rets[t] = future_ret
50     rets = torch.tensor(rets)
51     log_probs = torch.stack(pi.log_probs)
52     loss = - log_probs * rets # gradient term; Negative for maximizing
53     loss = torch.sum(loss)
54     optimizer.zero_grad()
55     loss.backward() # backpropagate, compute gradients
56     optimizer.step() # gradient-ascent, update the weights
57     return loss
```

Code Example - REINFORCE

<function main>

Line

Line

Line

Line

Line

```python
def main():
    env = gym.make('CartPole-v0')
    in_dim = env.observation_space.shape[0] # 4
    out_dim = env.action_space.n # 2
    pi = Pi(in_dim, out_dim) # policy pi_theta for REINFORCE
    optimizer = optim.Adam(pi.parameters(), lr=0.01)
    N=500 #Max Epiodes
    for epi in range(N): # org was 1000
        state = env.reset()
        for t in range(200): # cartpole max timestep is 200
            action = pi.act(state)
            state, reward, done, _ = env.step(action)
            pi.rewards.append(reward)
            #env.render() # remove for speed
            if done:
                break
        loss = train(pi, optimizer) # train per episode
        total_reward = sum(pi.rewards)
        solved = total_reward > 199.0
        pi.onpolicy_reset() # onpolicy: clear memory after training
        print(f'Episode {epi}, loss: {loss}, \
        total_reward: {total_reward}, solved: {solved}')
        if solved:
            break
```

Code Example - REINFORCE

<function run>

Line

Line

Line

Line

Line

```python
def run():
    env = gym.make('CartPole-v0')
    for trials in range(10):
        in_dim = env.observation_space.shape[0] # 4
        out_dim = env.action_space.n # 2
        pi = Pi(in_dim, out_dim) # policy pi_theta for REINFORCE
        state = env.reset()
        rewards = []
        img = plt.imshow(env.render(mode='rgb_array'))
        done = False
        while done==False:
            pred = pi(torch.from_numpy(state).float())
            action = pi.act(state)
            img.set_data(env.render(mode='rgb_array'))
            plt.axis('off')
            display.display(plt.gcf())
            display.clear_output(wait=True)
            state, reward, done, _ = env.step(action)
            rewards.append(reward)
        sum_rewards = sum(rewards)
    env.close()
run()
```

https://pytorch.org/docs/stable/distributions.html

Code Example - REINFORCE

Results

Update figure

Line

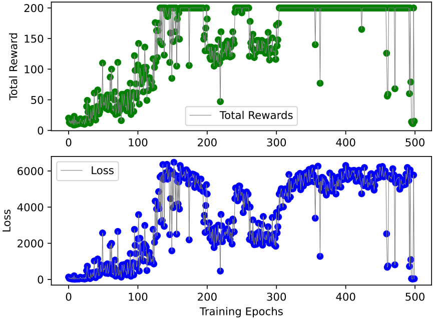

Figure: ...

https://pytorch.org/docs/stable/distributions.html

Code Example - REINFORCE

Interactive

References I

Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38.

Graesser, L. and Keng, W. L. (2019). *Foundations of deep reinforcement learning: theory and practice in Python*. Addison-Wesley Professional.

Jordi TORRES (2020). Policy-Gradient Methods REINFORCE algorithm. `https://towardsdatascience.com/policy-gradient-methods-104c783251e0`.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

PyTorch (2020). PROBABILITY DISTRIBUTIONS - TORCH DISTRIBUTIONS. `https://pytorch.org/docs/stable/distributions.html`.

Tesauro, G. et al. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.