Christian-Albrechts-Universität zu Kiel

Technische Fakultät

Autonomous Learning

Assignment 4

Simon Reichhuber

July 6, 2020

University of Kiel, Summer Term 2020

Table of Contents

C A U
Christian-Albrechts-Universität zu Kiel
Technische Fakultät

Policy Gradient vs. DQN

The decision which action to chose in a state is driven by the highest Q-Value (the maximum future reward in every state).

$\Downarrow$

The policy is only determined by these action value assessments.

- no learning of the value function $\rightarrow$ expected sum of future rewards for given state and action
- direct learning of the policy $\rightarrow$ mapping of state to an action
- optimisation of the policy function π (without a value function)
- direct parametrisation of π

Deterministic Policy

- maps a state directly to an action

$$\pi(s) = a$$

- applied in deterministic environments
  - The chosen actions determine the result.
  - no uncertainty
  - example: Chess

Stochastic Policy

- returns a random distribution of all actions

$$\pi(s) = P(a_t | s_t)$$

- When an action *a* is chosen it will only be executed with a certain probability.
- applied in uncertain environments
  - Partially observable Markov decision process (POMDP)
  - example: robots

ADVANTAGES

Deep Q Learning works well!

So, why Policy-based Reinforcement Learning?

Advantages: Convergence

- better convergence properties
- **value-based:**
  - Strong oscillation during training are possible.
  - A small change of the prediction (action values) can lead to a proportionally dramatic change when selection the action.
- policy gradient:
  - Follow the gradient to find the best parameters.
    $\rightarrow$ smooth policy updates with each step
  - Convergence in local maximum (worst case) or global maximum (best case) is guaranteed.

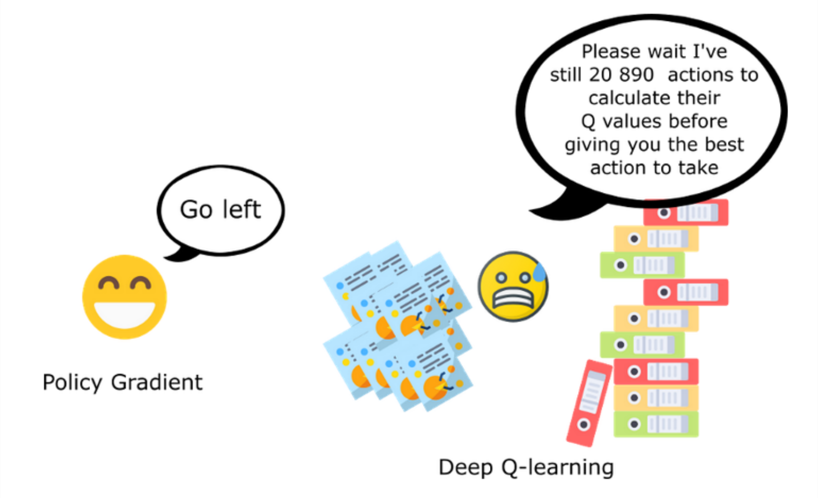

best parameters
local maximum
the policy
θ

Advantages: High-dimensional Action Space

C|A|U
Christian-Albrechts-Universität zu Kiel
Technische Fakultät

- more efficient for continuous actions or for an high-dimensional action space
- **value-based:**
  - For every time step a value is assigned to *every* possible action.
    $\rightarrow$ curse of dimensionality
  - How to deal with an *infinitive* number of actions?
    $\rightarrow$ e. g. self-driving car
- policy based:
  - direct adaptation of the parameters
  - More comprehensible what the maximum actually means.

High-dimensional Action Space (2)



ADVANTAGES: STOCHASTIC POLICIES

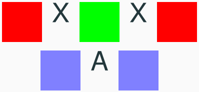
- policy gradient can learn stochastic policies
- no more trade-off between exploration and exploitation
- The output is a probability distribution of the actions.

  $\rightarrow$ The search space is explored without always choosing the same action.
- no problem of perceptual aliasing:

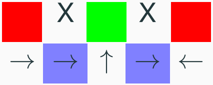
  The states are the same but require different actions.

Stochastic Policies (2)

Blue fields are aliased states.



deterministische policy:

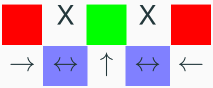


problem: The agent is stuck.

Stochastic Policies (3)

stochastic policy:

The policy will move randomly to the left or right. $\rightarrow$ no getting stuck



$$\pi = (\textit{wall UP and DOWN} \mid \textit{Go LEFT}) = 0.5$$

$$\pi = (\textit{wall UP and DOWN} \mid \textit{Go RIGHT}) = 0.5$$

DISADVANTAGES

A huge disadvantage:
It often converges to a local maximum instead of a global maximum.

It converges slowly step by step $\rightarrow$ can take longer than DQN

Policy Search

Policy as Optimisation Problem

- Policy π has parameter θ and probability distribution of the actions:

$$\pi_\theta(a|s) = P[a|s]$$

- When is a policy good?

  $\rightarrow$ policy as an optimisation problem:

$$J(\theta) = E_{\pi\theta}\left[\sum \gamma r\right]$$

- 2 steps:
  - Measure the quality of π with a policy score function $J(\theta)$.
  - Find the bester parameter θ with policy gradient ascent.

Depending on environment and goals there are 3 different methods.

1. *Episodic environment*:
   - start value → calculate the mean reward from the first step G_1
   - cumulated discounted reward of the whole episode:

$$J_1(\theta) = E_\pi[G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + \dots] = E_\pi(V(s_1))$$

   - When repeatedly starting in s_1:
     What is the total reward for this state up to the final state?
   - A policy that maximises G_1 is an optimal policy.

Example: Breakout

- The game always starts in the same state.
- calculation of the score using $J_1(\theta) \rightarrow$ number of destroyed bricks

Policy Score Function (2)

2. *Continuous environment*:
   - mean value $\rightarrow$ no fixed starting state
   - Every state value is weighted using the probability of that state:

$$J_{avgV}(\theta) = E_\pi(V(s)) = \sum d(s)V(s)$$

   with:

$$d(s) = \frac{N(s)}{\sum_{s'} N(s')}$$

   - $N(s)$ – occurrence of state s
   - $\sum_{s'} N(s')$ – occurrence of all states

3. *Environments with average reward per time step*:
   - highest reward for every time step

$$J_{avgR}(\theta) = E_{\pi}(r) = \sum_{s} d(s) \sum_{a} \pi_{\theta}(s, a) R_s^a$$

   where:
   - $\sum_s d(s)$ – probability of being in state s
   - $\sum_a \pi_\theta(s, a)$ – probability of choosing action a
   - R_s^a – immediate reward

Step 2: Policy Gradient Ascent

- maximising $J(\theta) \rightarrow$ Gradient Ascent on policy parameters
- Gradient Ascent is the inverse of Gradient Descent (steepest ascent).
- No Gradient Descent because it is not about minimising something (error function vs. score function).
- Find the gradient that updates the current policy towards the highest ascent and iterate:
  1. policy: π_θ
  2. objective function: $J(\theta)$
  3. gradient: $\nabla_\theta J(\theta)$
  4. update: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

CAU
Christian-Albrechts-Universität zu Kiel
Technische Fakultät

Goal: Find the best parameter θ^* that maximises the score:

$$\theta^* = \operatorname*{argmax}_\theta \underbrace{E_{\pi\theta}\left[\sum_t R(s_t, a_t)\right]}_{J(\theta)}$$

Score function:

$$J(\theta) = \overbrace{E_\pi}^{\text{expexted given policy}} \overbrace{[R(\underbrace{\tau}_{\substack{s_0, a_0, r_1, \\ s_1, a_1, r_2, \\ \dots}})]}^{\text{expected future reward}}$$

$\Rightarrow$ total sum of all expected rewards given policy π

Differentiating the score function:

$$J_1(\theta) = V_{\pi\theta}(s_1) = E_{\pi\theta}[v_1] = \underbrace{\sum_{s \in S} d(s)}_{\text{state distribution}} \underbrace{\sum_{a \in A} \pi_\theta(s, a)}_{\text{action distribution}} R_s^a$$

Problem:

- The policy parameters influence how the actions are chosen.

  → rewards and which state is visited how often
- The performance depends on the *action choice* and the *distribution of the respective states*.

  → The challenge is to find changes that ensure improvement.
- How do the parameters influence the state distribution?

  → The environment function is unknown.
- How is the gradient determined regarding π when it depends on an unknown relation between changes in the state distribution?

Solution: Policy Gradient Theorem

It offers an analytic expression for the gradient ∇ von $J(\theta)$ regarding π that does not include a differentiation of the state distribution:

$$J(\theta) = E_\pi\left[R(\tau)\right]$$

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau \pi(\tau;\theta) R(\tau)$$

$$= \sum_\tau \nabla_\theta \pi(\tau;\theta) R(\tau)$$

Likelihood ratio trick:

$$\pi(\tau;\theta)\frac{\nabla_\theta \pi(\tau;\theta)}{\pi(\tau;\theta)} \quad \nabla \log x = \frac{\nabla x}{x}$$

$$= \sum_\tau \pi(\tau;\theta) \nabla_\theta (\log \pi(\tau;\theta)) R(\tau)$$

$$\Downarrow$$

$$\nabla_\theta J(\theta) = E_\pi\left[\nabla_\theta (\log \underbrace{\pi(\tau|\theta)}_{\textit{policy function}}) \underbrace{R(\tau)}_{\textit{score function}}\right]$$

Policy Gradient Ascent (5)

- policy gradient: $E_\pi\left[\nabla_\theta(\log\pi(s,a,\theta))R(\tau)\right]$
- update rule: $\Delta\theta = \alpha \times \nabla_\theta(\log\pi(s,a,\theta))R(\tau)$
- $R(\tau)$ is a scalar:
  - total discounted future reward
  - large $R(\tau) \rightarrow$ On average actions are chosen that lead to a high reward. The probability of seen actions increases.
  - small $R(\tau) \rightarrow$ The probability of seen actions decreases.

Algorithm 1: REINFORCE: Monte-Carlo Policy Gradient Control (episodic) for π

Initialize θ

for *each episode* **do**

  Generate an epsiode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following π_θ **for** *t=0 to T-1* **do**

    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

    $\theta \leftarrow \theta + \alpha\gamma^t G \nabla \log \pi(A_t|S_t, \theta)$

REINFORCE IN KERAS

- $G_t = r_t + \gamma * r_{t+1} + \gamma^2 * r_{t+2} + ...$
- Keras minimises its error function using Gradient Descent.
- Trick: negative sign $\rightarrow$ Gradient Descent
- $J(\theta)$ as error function:

$$loss_t = -\log \pi(A_t, S_t) * G_t$$

$$loss = \frac{\sum_t loss_t}{T}$$

Loss Funtions in Keras


```
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import RMSProp
import keras.backend as K

inL = Input(self.state_size)
h1 = Dense(16, activation='relu')(inL)
out = Dense(self.action_size, activation='softmax')(h1)
self.model = Model(input=inL, output=out)
self._build_train()

def _build_train():
  # placholder Tensor for target values
  target = K.placeholder()
  prediction = self.model.output
  # compute error
  error = K.mean(K.sqrt(1 + K.square(prediction - target)) - 1, axis=-1)
  # create optimizer
  opimizer = RMSProp(lr=self.learningrate)
  update = optimizer.get_updates(loss=error, params=self.model.trainable_weights)
  # create fit function with inputs and outputs
  self.fit = K.function(inputs=[self.model.input, target], outputs=[error], updates=update)

# train model
err = self.fit([states, targets])
```

SOFTMAX

- normalised exponential function
- It "squeezes" a vector of dimension K into a vector of dimension K *and* a value range of $(0, 1)$.
- The components sum up to 1.
- probability distribution over K different events

Bibliography

Pictures

- https://medium.freecodecamp.org/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f