Dynamic Programming and Reinforcement Learning

Week 5b: Further Additions to Deep Q-Networks

Rainer Schlosser und Alexander Kastius

Enterprise Platform and Integration Concepts

19.05.22

Current State of DQN

- Last session, we introduced **Deep Q-Networks**
- Replacing the table based estimate with an ANN resolved the state space issues, but required us to become as efficient on our training data as possible.
- **Double Deep Q-Networks** helps us overcome the maximization bias.
- **Dueling Deep Q-Networks** can improve the estimation by enforcing a structure of computation that resembles the internal structure of the Q-values.
- **Experience Replay** and especially **Prioritized Experience Replay** do highly increase our efficiency with regard to data usage.

- Are there more extensions available? Yes, and they are aggregated in **RAINBOW** which combines **DQN** and the three mentioned above with: **distributional Q-learning, (n)-step returns, and noisy nets.**

Distributional RL - Discretization

We just mentioned RAINBOW is estimating **distributions of rewards.**

This is very different to previous approaches. All previous methods assumed that we are learning the **expected value** of a state or state-action pair.

Intro: To simplify the algorithm, we discretize the space of values. Instead of assuming a value to be constant, we assume that it belongs to a set of possible choices.

Necessary hyperparameters: **A minimum value $\boldsymbol{v_{min}}$** and **a maximum value $\boldsymbol{v_{max}}$**. We then **discretize** the range between v_{min} and v_{max} into $\boldsymbol{N_{atoms}}$ elements. The values of those three variables are chosen manually and have to be configured correctly. (Example: Rainbow uses $v_{min} = -10, v_{max} = 10$ and $N_{atoms} = 51$ for Atari games with rewards clipped to -1 and 1).

$$z_i = v_{min} + (i-1)\frac{v_{max} - v_{min}}{N_{atoms}} \; for \; i \in \{1, \dots, N_{atoms}\}$$

Distributional RL - Discretization

Given this vector z consisting of our atoms:

$$z_i = v_{min} + (i-1)\frac{v_{max} - v_{min}}{N_{atoms}} \; for\ i \in \{1, \dots, N_{atoms}\}$$

We can create a prob. distribution by assigning each value in z a probability.

This can be achieved by letting our network Q output N_{atoms} values per action and normalizing those ($Q(s,a)_i$ is the i-th output of Q for *s* and *a*, p_i the probability that choosing *a* in *s* achieves value z_i):

$$p_i = \frac{e^{Q(s,a)_i}}{\sum_{j=1}^{N_{atoms}} e^{Q(s,a)_j}}$$

Now our network outputs a distribution of values instead of expected values!

Distributional RL – Target Distribution

Okay got it, but how to learn those distributions?

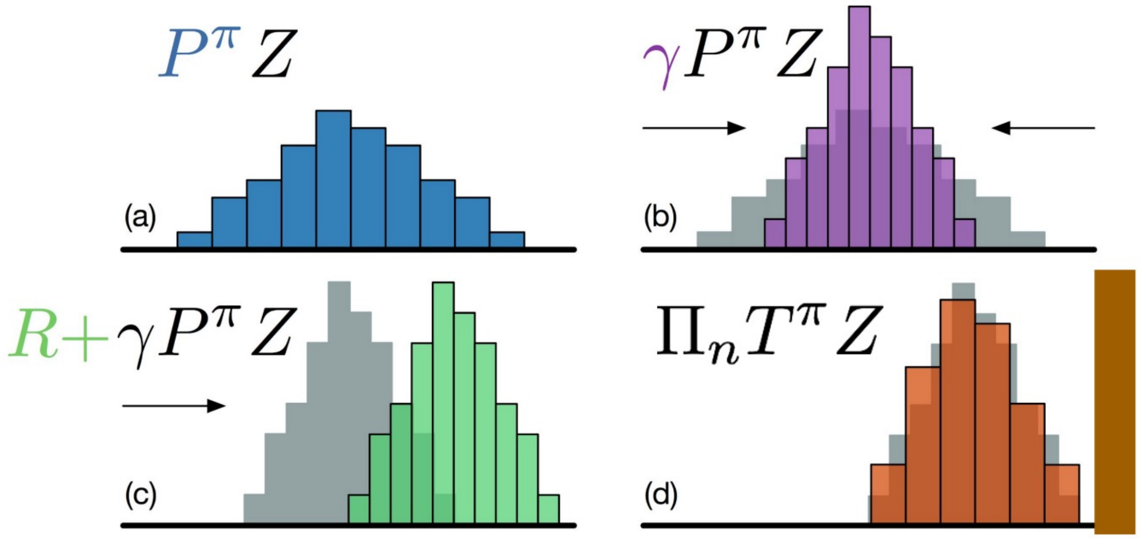
Idea:

Estimate a target distribution and use a measure for the difference between to distributions as optimization loss.

1. We have a realized reward.
2. We have a distribution of Q-values for all possible actions over a range of values in the state after this step.

If we now assume that every time we start from s_t and follow our policy, we would yield the same reward this means that the distribution of values from s_t, a_t has the same shape as the distribution of values for the following s,a combination, but is shifted towards the realized reward in that step and is discounted according to γ.

Distributional RL



https://physai.sciencesconf.org/data/pages/distributional_RL_Remi_Munos.pdf

Distributional RL – Shifting and Compressing

Shifting towards a target distribution:

Given for this step: (s_t, a_t, r_t, s_{t+1}) as usual. z as defined by our hyper parameters. p_i, the probabilities computed from our Q-network for $Q(s_{t+1}, a')$.

a' is chosen by selecting the action that has the maximum expected value according to the output distribution: $a' = \mathrm{argmax}_{a \in A} E(Q(s_{t+1}, a))$.

We then can shift the output of the probability distribution for state s_{t+1} towards the reward achieved in r_t to achieve a valid distribution for s_t.

$$z'_i = r_t + \gamma z$$
$$p'_i = p_i$$

This distribution has different values in z!

Distributional RL - Projection

Fitting the atoms:

It is necessary, that our distributions use the same set of atoms, to compute a loss function that we can minimize.

We have to project the target distribution back onto the values available in our output system.

Example Projection:

1. Initialize another probability vector p'' with 0.
2. Compute $b = \frac{z_i' - v_{min}}{\Delta_z}$ and $b_u = \lceil b \rceil$ and $b_l = \lfloor b \rfloor$ for all i (b_l and b_u contain the respective next lower and larger index in original z for z_i')
3. Then $p''_{b_l} \leftarrow p''_{b_l} + p_i' * (b - b_l)$ and $p''_{b_u} \leftarrow p''_{b_u} + p_i' * (b_u - b)$

If the z' is too large or too small to fulfill 2. (The shift was so large, that there are values with positive probability that exceed our allowed range), account them to p_1'' or $p''_{N_{atoms}}$.

Distributional RL - Loss

We can compute the cross-entropy or Kullback-Leibler-divergence of both distributions and use this a loss function. Both do provide us with a measure of how similar to prob. distributions are.

p_i = Prob. of choosing z_i with the orig. dist., p_i'' = Prob. of choosing z_i in the target. dist.

Cross-entropy:

$$-\sum_{i=1}^{N_{atoms}} p_i'' \log(p_i)$$

Kullback-Leibler-Divergence:

$$\sum_{i=1}^{N_{atoms}} p_i'' \log(\frac{p_i''}{p_i})$$

RAINBOW uses KL, original distributional QL paper uses cross-entropy.

Distributional RL – Extensions

- The double network approach can be taken into account by using it to predict the values for the target distribution.
- The dueling network can still be implemented, as there is no limitation on the internal structure of the network predicting the values used to compute the probabilities.
- Experience replay can still be applied as well.

Two extensions left!

(n)-Step returns

- We introduced (n)-step returns and importance sampling for Q-learning some weeks ago.
- The same principles can still be applied when using an estimator.
- Interestingly: **RAINBOW** omits the use of importance sampling factors even though it operates on a different policy. Why? Because we can replace ϵ-greedy with a different mechanism.

- Remember:

$$G_{t:t+n} = \sum_{k=t}^{t+n} \gamma^{k-t} r_k$$

$$target = G_{t:t+n-1} + \gamma^n \max_{a \in A} Q(s_{t+n}, a)$$

Overcoming ϵ-greedy

For now we implemented exploration with ϵ-greedy. New idea: Add noise to the networks output.

A linear layer:

$$y = Wx + b$$

A noisy linear layer:

$$y = Wx + b + (W_{noisy} \odot e_w)x + (b_{noisy} \odot e_b)$$

$\odot$ is the element wise product. e_W and e_b are randomly generated every time we need to compute something. W_{noisy} and b_{noisy} have the same shape as W and b.

By adjusting W_{noisy} and b_{noisy}, the training process can reduce the noise, if it causes too much harm in the divergence! If the values are far off only due to noise → reduce their weights. But this can happen depending on the state under assessment. Also manually tuning exploration rates is not necessary anymore.

RAINBOW

Everything together:

- We use capable ANNs for the estimation of our values.
- **We estimate distributions of values instead of expected values.**
- We use double learning to overcome the maximization bias.
- We use dueling networks to take the structure of Q into account.
- **We use (n)-step returns to learn from delayed rewards.**
- **We use noise in the network for exploration, the policy is greedy.**
- We use prioritized experience replay, to increase our data efficiency.
- This is perfect, isn’t it?

No. We still can’t do anything when the action space is continuous.
Our perfect DQN system only works on discrete action spaces.

→ Policy Gradients!

Schedule

Week	Dates	Topic
1	April 21	Introduction
2	April 25/28	Finite + Infinite Time MDPs
3	May 2/5	Approximate Dynamic Programming (ADP) + DP Exercise
4	May 12	Q-Learning (QL) (not Mon May 9)
5	May 16/19	**Q-Learning Extensions and Deep Q-Networks**
6	May 23	DQN Extensions (not Thu May 26 “Himmelfahrt”)
7	May 30/June 2	Policy Gradient Algorithms
8	June 9	Project Assignments (not Mon June 6 “Pfingstmontag”)
…		