29.2.2024

Project

Model learning for Reinforcement Learning

In reinforcement learning (RL), an agent acts in an environment and receives rewards. At each discrete time step $t = 1, \dots, T$, the agent receives from the environment a state $\boldsymbol{s}_t$ and chooses an action $\boldsymbol{a}_t$ according to his policy π. Typically, the policy is a probability distribution over actions for the given state $\pi(\boldsymbol{a}|\boldsymbol{s})$, so actions are chosen stochastically.

After choosing an action, the environment transitions into a new state $\boldsymbol{s}_{t+1}$ and the agent receives a reward $r_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$ which indicates how good the action $\boldsymbol{a}_t$ was in state $\boldsymbol{s}_t$. The goal of the agent is to maximize the return R which is in the simplest case just the sum of received rewards

$$R = \sum_{t=1}^{T} r_t \, .$$

He does this by adapting his policy π in order to choose actions that lead to high reward.

There are two main classes of RL algorithms: model-based and model-free algorithms. In model-free algorithms, the agent is agnostic to the environmental dynamics and tries to maximize rewards directly. In model-based algorithm, the agent learns a model of the environment which he tries to utilize e.g. by planning in this model.

In this project, we will investigate model learning for reinforcement learning in particular for simple robot-like tasks. The model consists of two approximators which will be implemented by neural networks. The *state model* f_s approximates the state-dynamics of the environment, and the reward-model f_r approximates the reward function.

Reward model

The reward model predicts the reward for the given state and action. The predicted reward is thus

$$\hat{r}_t = f_r(\boldsymbol{s}_t, \boldsymbol{a}_t).$$

We will train a simple feed forward neural network for this.

State model

The state model gets actions and states and predicts the next state. We will use a recurrent neural networks (RNN) for this, but a feed forward network might be tried as well for simple cases.

There are several options for how to choose the input to the state model. In option 1, the network receives just the initial state $\boldsymbol{s}_1$, then all actions $\boldsymbol{a}_1, \dots, \boldsymbol{a}_t$ as input, and outputs state $\boldsymbol{s}_{t+1}$. Here, we need an RNN as the network has to keep track of the state. Note that here, the network receives in computation step 0 the initial state $\boldsymbol{s}_1$. Then in computation step $t = 1 \dots T$ it receives as input action $\boldsymbol{a}_t$ and outputs state $\boldsymbol{s}_{t+1}$.

In option 2, the network receives at each computation step $t = 1 \dots T$ state $\boldsymbol{s}_t$ and action $\boldsymbol{a}_t$ and outputs state $\boldsymbol{s}_{t+1}$. This might be the simpler option and here one can also try a feed forward network.

Generation of training data

To generate training data, we define the environmental dynamics and a reward function and generate trajectories from them. Note that in practice these dynamics are of course not known. Anyways, we use well-defined dynamics here to test the principle. We then generate sample trajectories using an exploration policy π_{explore}. Since we have not yet learned a policy during model learning, we have to choose π_{explore}, i.e., how to explore the state space. A simple first choice is a random policy, but there might be better options.

We then generate M trajectories with π_{explore} and train the models in a supervised manner.

First toy environment

As a first toy environment, we choose a simple 2D target reaching problem. The agent is a point in 2D space which can accelerate. The goal is to reach a target position $\boldsymbol{x}^* \in \mathbb{R}^2$. The state at each time is given by the position vector $\boldsymbol{x}_t$ and the velocity vector $\boldsymbol{v}_t$ of the agent. The action is the acceleration vector $\boldsymbol{a}_t \in \mathbb{R}^2$.

The linear dynamics are

$$\boldsymbol{v}_{t+1} = \boldsymbol{v}_t + \alpha_v \boldsymbol{a}_t$$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \alpha_x \boldsymbol{v}_t$$

With constants α_v and α_x. Note that the system can easily be generalized to higher dimensions. Friction could be included. For the reward function, one has again some options. The simplest one measures at each time step the distance to the goal position:

$$r_t = -(\boldsymbol{x}_t - \boldsymbol{x}^*)^T(\boldsymbol{x}_t - \boldsymbol{x}^*).$$

A more challenging (for the RL algorithm) reward function would only give reward close to the target position.

Task

We work in Python 3 with Tensorflow or PyTorch.

- Implement the environmental dynamics
- Draw sample trajectories
- Train neural networks for the state and reward models.
- Show training progress and evaluate model performances.

Model-Gradient Reinforcement Learning

After we have learned a model of the environment, we add a policy and optimize the policy in the loop with the environment.

The policy network has parameters W_π, it gets as input a state vector and outputs an action vector. We denote the function it computes as f_π. The system that we consider now thus consists of the policy network, which feeds its action to the state model and the reward model. The output of the state model also feeds to the reward model and in addition back to the policy network.

Let’s say we start in initial state $\boldsymbol{s}_1$. We set $\hat{\boldsymbol{s}}_1 = \boldsymbol{s}_1$. Note: We use a hat for predicted states to distinguish them from observed states. The system will however evolve according to such predicted or “imagined” states. The system evolves according to (for $t = 1, \dots, T$)

$$\boldsymbol{a}_t = f_\pi(\hat{\boldsymbol{s}}_t),$$

$$\hat{\boldsymbol{s}}_{t+1} = f_s(\hat{\boldsymbol{s}}_t, \boldsymbol{a}_t)$$

$$\hat{r}_t = f_r(\hat{\boldsymbol{s}}_t, \boldsymbol{a}_t)$$

$$R = \sum_{t=1}^{T} \hat{r}_t.$$

In other words, we produce an imagined trajectory using actions from the policy network. We can now compute the gradient of the reward w.r.t. the policy parameters $\frac{\partial R}{\partial W_\pi}$ and update the policy network weights

$$\Delta W_\pi = \varepsilon \frac{\partial R}{\partial W_\pi},$$

where $\varepsilon > 0$ is some small learning rate. Note that, since we want to maximize the reward, we have to go in the direction of the gradient (not in the negative direction as usually in gradient descent). Also note that during this optimization the parameters of the state- and reward-model are fixed.

After training has converged, we test the policy in the “real” environment.