Policy-based Reinforcement Learning

1 Vanilla Policy Gradient (VPG)

In this exercise we will implement the VPG method. The idea of VPG in deep reinforcement learning is to represent the policy using a deep neural network which is parameterized by its internal weights. Remember that it is an on-policy method. The policy gradient w.r.t to the model parameters is defined as:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_\pi[Q^\pi(s,a) \ln \pi_\theta(a|s)] \\ &= \nabla_\theta \mathbb{E}_\pi[G_t \ln \pi_\theta(a|s)]\end{aligned}$$

Programming Tasks:

1. **Network Architecture** Similar to the last exercise about DQN it makes sense to begin with defining the neural network architecture inside `ActorNetwork`. You can either decide to use a `torch.nn.softmax` output layer and treat outputs directly as action probabilites, or work on logit outputs.

   Don't get too fancy with your network. Even using two `torch.nn.Linear` layers with ReLU/Tanh activation should converge quite nicely on easy environments like *CartPole*.

2. **Transition Memory** Even though it is not strictly necessary to do so, we will use a transition memory to store experience episodes of the actor. This will make things cleaner and allow us to expand the framework to more sophisticated methods in the next exercise.

   (a) `put` expects an observation, action, reward and log-probability of the action taken and should store them for later processing.

   (b) `finish_trajectory` will be called once an episode is over. It should calculate the return at all the timesteps of the episode using `compute_returns` and store them for later.

   (c) `get` should return stored data in the form of *five* lists (observations, actions, rewards, log-probabilities and returns).

   (d) `clear` should delete all entries inside the transition memory.

3. **Return Calculation** Next up you will have to implement the `compute_returns` function, which should yield returns for every timestep of an episode based on a list of rewards and the discount factor γ. Remember that the return at timestep t of an episode is defined as $R_t(\tau) = \sum_{k=t}^{T} \gamma^{k-t} r_k$

4. **Time for Action** Continue with implementing the actors `predict` function.

   First feed your observation through the actor network, yielding action probabilities or preference logits.

   Sample an action according to the probability distribution outputted by the actor network and calculate the log probability for that action (*Hint:* PyTorch provides prebuilt distribution classes, e.g. `torch.distributions.Categorical` for discrete actions). Return the action (as well as the log probability if `train_returns == True`).

5. **Loss Function** Write code for the function `calc_actor_loss` which calculates the "loss" function (more objective function) as described above. (*Hint:* Because you can use the autograd capabilities of PyTorch to compute your gradient you only have to calculate the expectation term. To compute the gradient you can then use `loss.backward()`).

6. **Training Loop** Continue with implementing the training loop inside the `VPG.learn` function.

   (a) Start by resetting the environment and saving the first observation into a variable.

   (b) For every iteration, sample an action from your actor, take a step inside the environment with that action and save the transition inside the `TransitionMemory`.

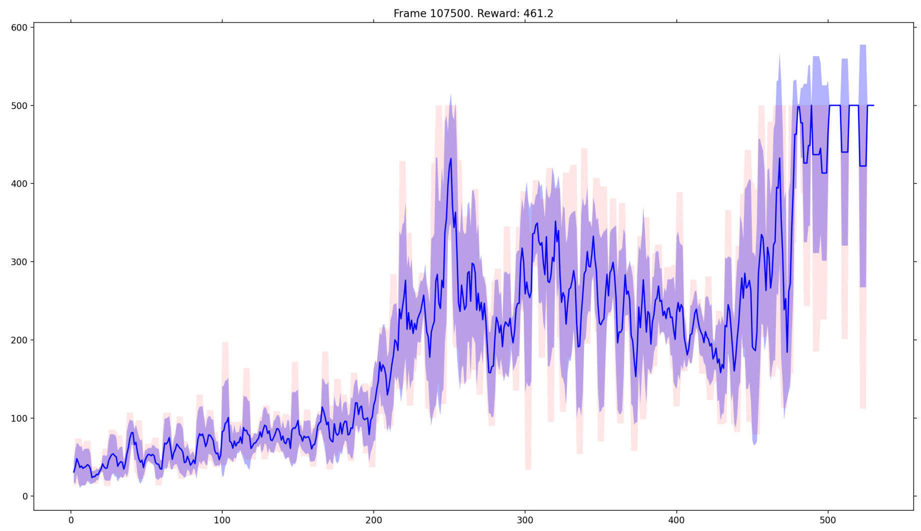

Figure 1: Sample reward plot we were able to achieve with our implementation on CartPole (hyperparameters as provided in skeleton, Linear(128) → ReLU → Linear → ReLU NN architecture). The results are strongly hyperparameter-related.

(c) Once you reach a terminal state reset your environment and finish the trajectory using the `finish_trajectory` function of the `TransitionMemory` you previously implemented.

(d) Once you recorded enough episodes (see `episodes_update` attribute) calculate the loss and optimize the `ActorNetwork` using PyTorch autograd. *Note:* Because VPG is an on-policy method, transitions inside the memory can only be used for one optimization step.

2 Advantage Actor Critic (A2C)

Remember that the policy gradient is defined as:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(a|s)\, Q^\pi(s,a)].$$

The most basic **actor-critic** framework uses a critic to approximate the action-value function $Q_\pi(s,a)$ and applies its action-value estimates for calculating the policy gradient. In this exercise we will go one step further.

Remember that, in order to reduce variance in the estimates of the gradient, it is mathematically sound to subtract a baseline term from the policy gradient. It turns out that a reasonable choice for the baseline term is the state-value function $V^\pi(x)$, which leads to the policy gradient formulation:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(a|s)\,(Q^\pi(s,a) - V^\pi(s))]\\ &= \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(a|s)\, A^\pi(s,a)],\end{aligned}$$

where $A^\pi(s,a)$ is the so-called *advantage function.* This function can be interpreted as the difference in expected return when taking action a in state s compared to the expected return when following the policy π in state s. Policy gradient methods that use this advantage function in its gradient calculations belong to the family of **advantage actor-critic** methods.

Instead of approximating the two functions $Q^\pi(s,a)$ and $V^\pi(s)$ or the advantage function $A^\pi(s,a)$ directly, common approaches use the critic to approximate the state-value function $V^\pi(s)$ and estimate the advantage in one of the following ways:

1. $A^\pi(s,a) = R(s,a) - V^\pi(x)$: MC advantage estimate, where $R(s,a)$ is the return received in state s
2. $A^\pi(s,a) = r + \gamma V^\pi(s') - V^\pi(s)$: TD advantage estimate, often also in the form of a n-step TD formulation

3. $A^\pi(s,a) = \sum_{l=0}^{T}(\gamma\lambda)^l \delta_{t+l}$: Generalized advantage estimate (GAE) [1], where $\delta_t = r_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$ is the TD error. GAE is the average of all n-step TD advantages weighted by a discount parameter λ.

In this exercise we will implement the advantage actor-critic method and focus on the 1. and 3. estimation approach. The VPG implementation of last exercise can be naturally extended for this.

Another benefit when using a critic is that we can change the training loop to optimize the network parameters after a fixed amount of steps performed in the environment (as compared to a fixed amount of episodes in the VPG exercise), i.e. on partial trajectories. We can do this, because we can bootstrap the value of the state the trajectory was terminated in using the critic.

Programming Tasks:

1. **Network Architecture** Similar to the last exercises it makes sense to begin with defining the neural network architecture inside `CriticNetwork`. As state values can have positive or negative sign the output layer has to be `torch.nn.Linear`.
2. **Transition Memory** Think about what additional information we need here.
3. **Predict Function** Expand the function to also calculate and return the value of the current observation to the training loop.
4. **MC Advantage** Write the function `compute_advantages` which estimates the MC advantage based on a list of returns and state values.
5. **Critic Loss** Write the function `calc_critic_loss`, which computes the MC error (e.g. `F.mse_loss`) between predicted state values and returns. *Note:* This is just for simplicity. You could also use 1-step or n-step TD error here.
6. **Actor Loss** Adjust the `calc_actor_loss` function to work on advantages instead of action-value functions. Do you have to change anything?
7. **Training Loop**
   (a) As in the last exercise, sample an action from your actor, take a step inside the environment and save the transition inside the `TransitionMemory`
   (b) Write code for the case that a terminal state has been reached (*Hint:* We don't only optimize when an episode is finished in this exercise)
   (c) Write the optimization code, which should be executed when enough samples were collected to fill the training batch (compare with `self.batch_size`). You will have to call `finish_trajectory`, but this time provide the termination state value using the critic (don't forget to call `item()`, we don't want to backpropagate from these outputs). Next, calculate actor and critic loss using the function you wrote, and do the optimization step using the actor and critic optimizers.
8. **GAE** At this point your implementation should already work and be able to learn useful policies. Now as a last step, write the `compute_generalized_advantages` function and use it instead of the old MC method to estimate the state advantage function.

3 BONUS: Proximal Policy Optimizaton (PPO)

Last but not least, we want to extend the previous implementation by using a simple version of PPO with advantage clipping, i.e. PPO-Clip. We only have to make some small changes:

Programming Tasks:

1. **Transition Memory** and **Training Loop** Only some minor changes are necessary.
2. **Actor Loss** Adjust the `calc_actor_loss` function to accommodate the *advantage clipping for conservative policy updates* introduced in the lecture.
3. **Advanced** Think about additional modification from the lecture that can be introduced to make training more stable.

[1] Schulman et al., https://arxiv.org/abs/1506.02438