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`.

   (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.

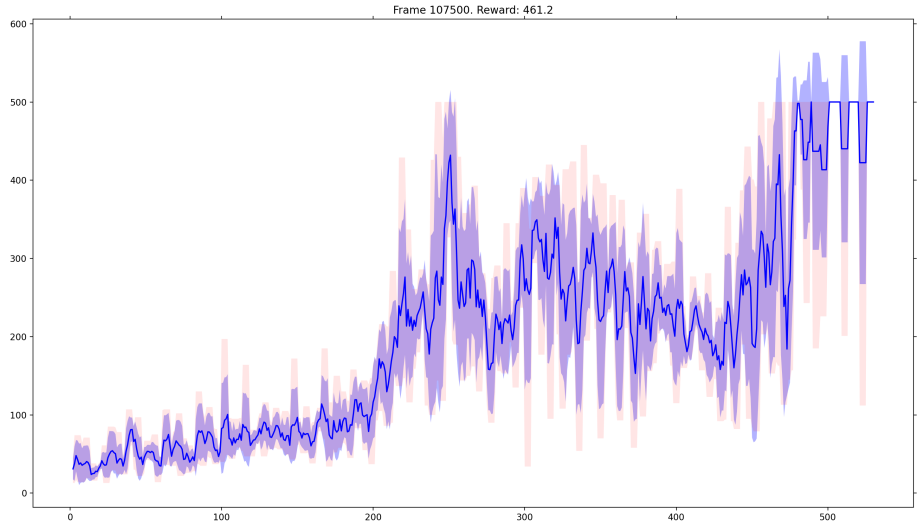

Figure 1: Sample reward plot we were able to achieve with our implementation on CartPole (hyperparameters as provided in skeleton, Linear(128) → ReLU → Linear → ReLU NN architecture). The results are strongly hyperparameter-related.