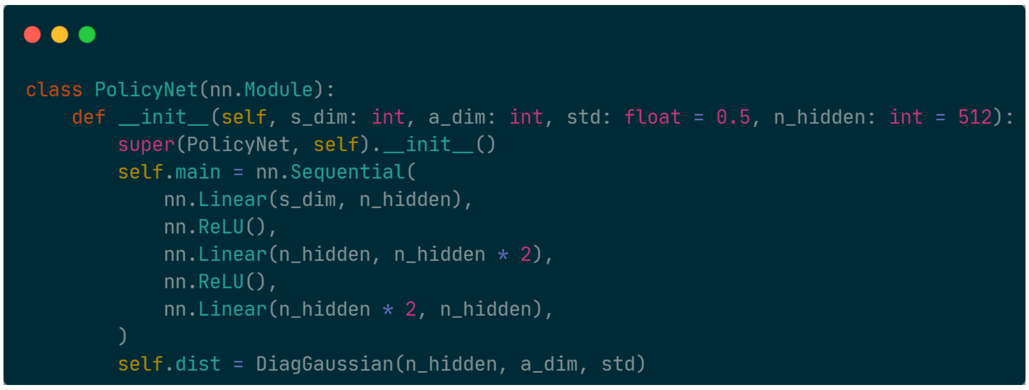
Robotic Navigation and Exploration
Homework #2

1 Code

1.1 Policy Network

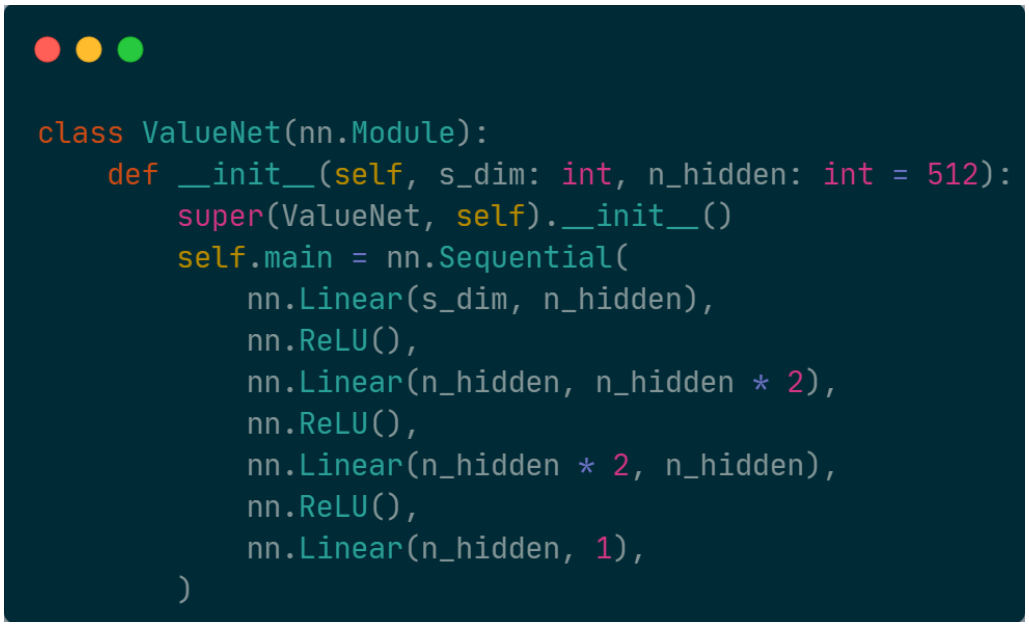
The policy network contains three linear layers with ReLU activations, expanding the feature representation progressively. The final layer ’ s output serves as input to the diagonal Gaussian distribution module.



```
class PolicyNet(nn.Module):
    def __init__(self, s_dim: int, a_dim: int, std: float = 0.5, n_hidden: int = 512):
        super(PolicyNet, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(s_dim, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden * 2),
            nn.ReLU(),
            nn.Linear(n_hidden * 2, n_hidden),
        )
        self.dist = DiagGaussian(n_hidden, a_dim, std)
```

1.2 Value Network

Similar to the policy network, it uses 3 layers with ReLU activations to process the input state into a feature representation, concluding with a single output unit representing the state value.

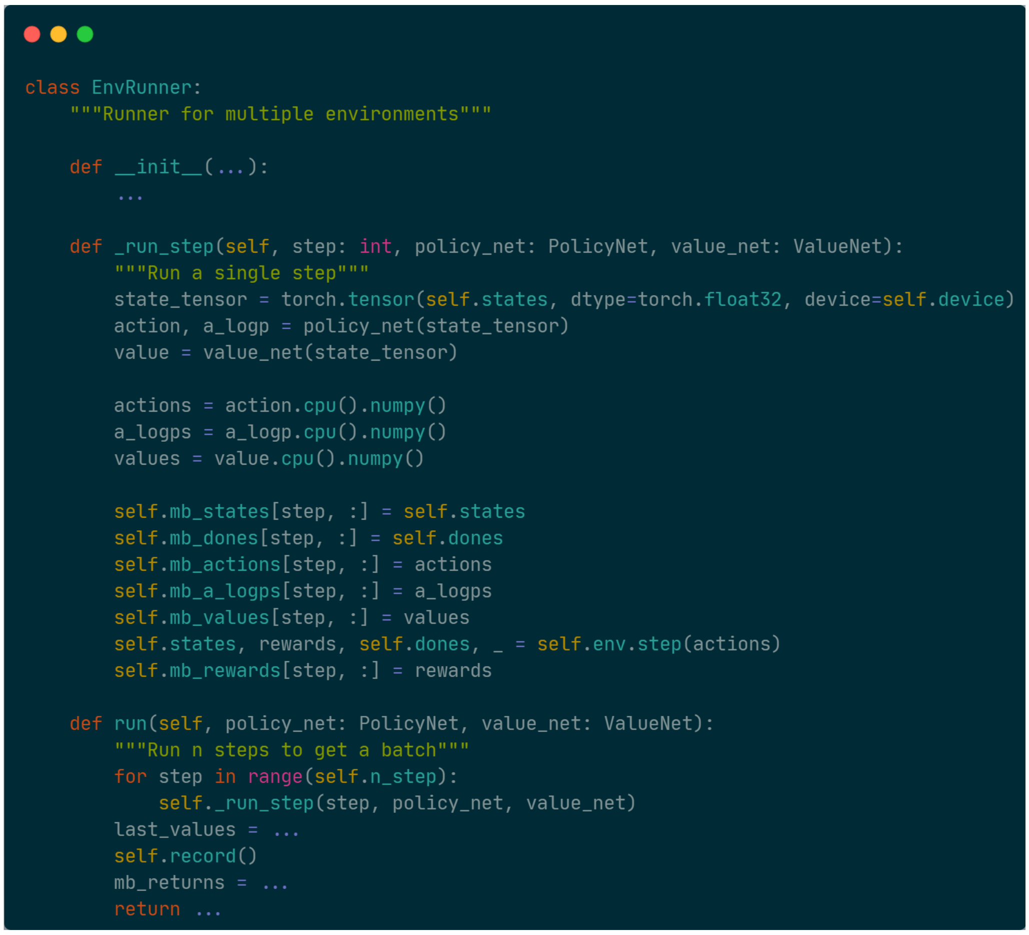


```
class ValueNet(nn.Module):
    def __init__(self, s_dim: int, n_hidden: int = 512):
        super(ValueNet, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(s_dim, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden * 2),
            nn.ReLU(),
            nn.Linear(n_hidden * 2, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, 1),
        )
```

1.3 Environment Runner

We modify the original template code to make it more readable, the TODO part is extracted as a standalone private method `_run_step` in the `EnvRunner` class. In this method, the core functionality is as follows:

1. **Pre-processing:** The states of the environments are converted into a pytorch tensor, which serves as the input to the neural networks.
2. **Network Forwarding:**
   - The policy network is invoked with the tensor to generate actions and their corresponding log probabilities.
   - The value network provides an estimate of the expected reward from each state.
3. **Post-processing:** One crucial part not to overlook is that those tensors need to be moved from the device (e.g., GPU) to the CPU and then converted to NumPy arrays for further processing.
4. **Perform Actions:** The agent performs the actions in the environment, which returns the new states, rewards, and done flags.



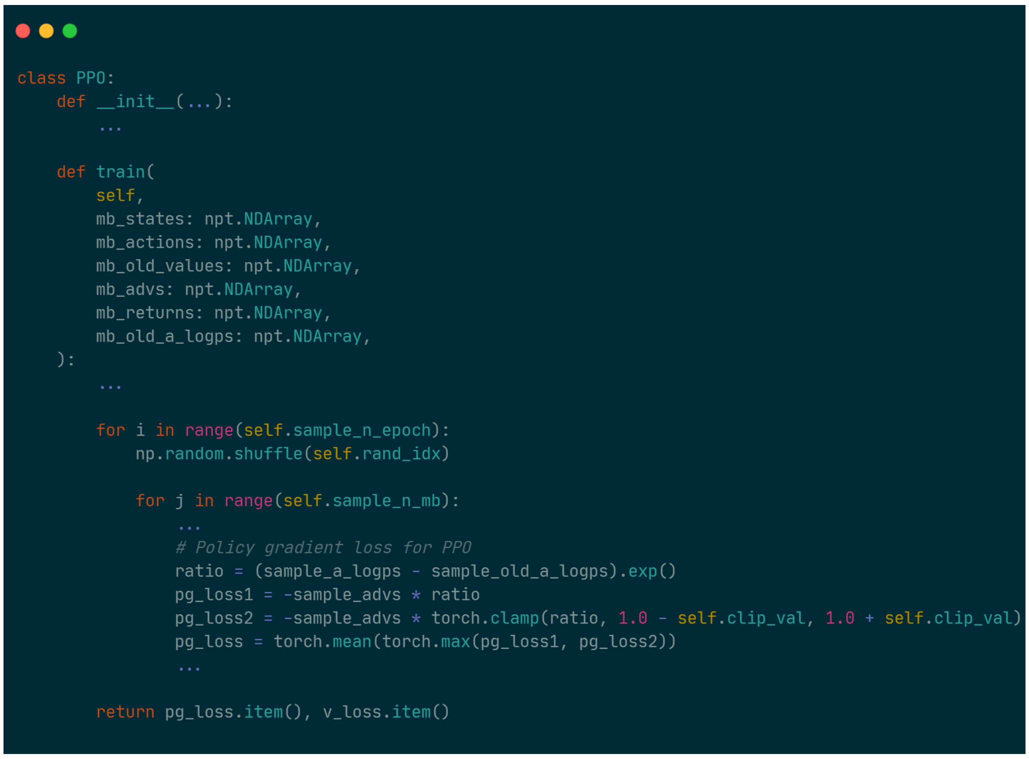
```python
class EnvRunner:
    """Runner for multiple environments"""

    def __init__(...):
        ...

    def _run_step(self, step: int, policy_net: PolicyNet, value_net: ValueNet):
        """Run a single step"""
        state_tensor = torch.tensor(self.states, dtype=torch.float32, device=self.device)
        action, a_logp = policy_net(state_tensor)
        value = value_net(state_tensor)

        actions = action.cpu().numpy()
        a_logps = a_logp.cpu().numpy()
        values = value.cpu().numpy()

        self.mb_states[step, :] = self.states
        self.mb_dones[step, :] = self.dones
        self.mb_actions[step, :] = actions
        self.mb_a_logps[step, :] = a_logps
        self.mb_values[step, :] = values
        self.states, rewards, self.dones, _ = self.env.step(actions)
        self.mb_rewards[step, :] = rewards

    def run(self, policy_net: PolicyNet, value_net: ValueNet):
        """Run n steps to get a batch"""
        for step in range(self.n_step):
            self._run_step(step, policy_net, value_net)
        last_values = ...
        self.record()
        mb_returns = ...
        return ...
```

1.4 PPO Loss Calculation

The policy gradient loss in PPO is designed to prevent the policy from changing too drastically during each update. It achieves this by using a clipped ratio of the new and old probabilities, combined with the advantage estimates. Calculation steps are as follow:

1. **Log Probability Difference:** The difference gives the logarithm of the ratio of the new probabilities to the old probabilities: $\log prob_{new} - \log prob_{old} = \log\left(\frac{prob_{new}}{prob_{old}}\right)$.
2. **Exponentiation:** Taking the exponential of the above difference yields the actual ratio of the new probabilities to the old probabilities: $ratio = \exp\left(\log\left(\frac{prob_{new}}{prob_{old}}\right)\right) = \frac{prob_{new}}{prob_{old}}$.
3. **Policy Gradient Loss Components:**
   - $loss$: This is the negative of the product of the advantages ($\hat{A}$) and the ratio. It represents the part of the policy gradient that encourages taking actions that have higher advantages than expected.
   - $loss_{clipped}$: This applies the clipping ($\text{clip}(ratio, 1-\epsilon, 1+\epsilon)$) to the ratio before multiplying by the advantages ($\hat{A}$). It represents the constrained part of the policy gradient.
4. **PPO's Clipped Loss:**

$$loss^{CLIP} = \min\left(ratio \cdot \hat{A}, \text{clip}(ratio, 1-\epsilon, 1+\epsilon) \cdot \hat{A}\right) = \min\left(loss, loss_{clipped}\right)$$

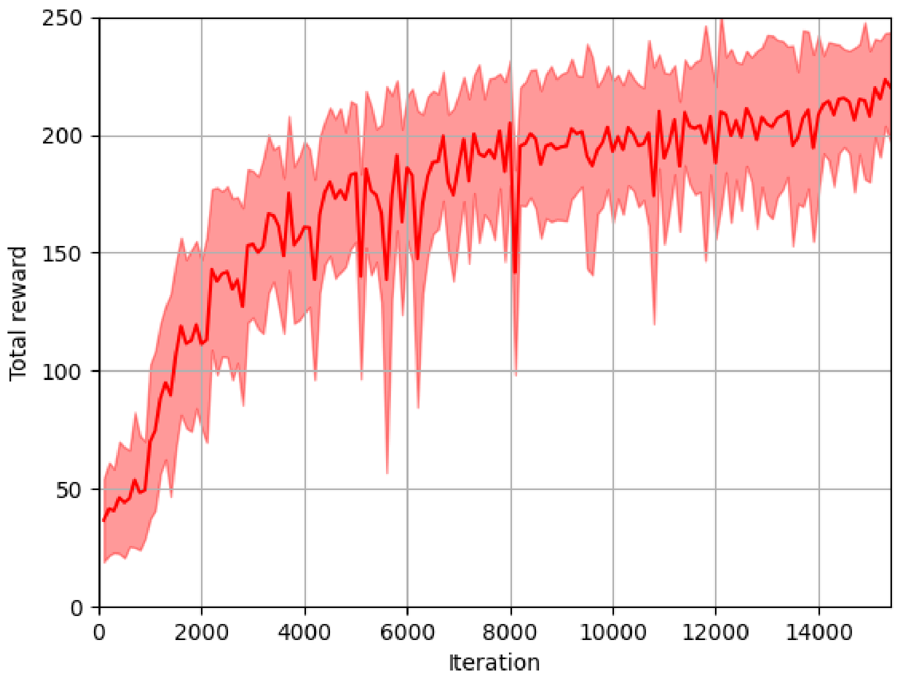


```python
class PPO:
    def __init__(...):
        ...

    def train(
        self,
        mb_states: npt.NDArray,
        mb_actions: npt.NDArray,
        mb_old_values: npt.NDArray,
        mb_advs: npt.NDArray,
        mb_returns: npt.NDArray,
        mb_old_a_logps: npt.NDArray,
    ):
        ...

        for i in range(self.sample_n_epoch):
            np.random.shuffle(self.rand_idx)

            for j in range(self.sample_n_mb):
                ...
                # Policy gradient loss for PPO
                ratio = (sample_a_logps - sample_old_a_logps).exp()
                pg_loss1 = -sample_advs * ratio
                pg_loss2 = -sample_advs * torch.clamp(ratio, 1.0 - self.clip_val, 1.0 + self.clip_val)
                pg_loss = torch.mean(torch.max(pg_loss1, pg_loss2))
                ...

        return pg_loss.item(), v_loss.item()
```

2 Evaluation

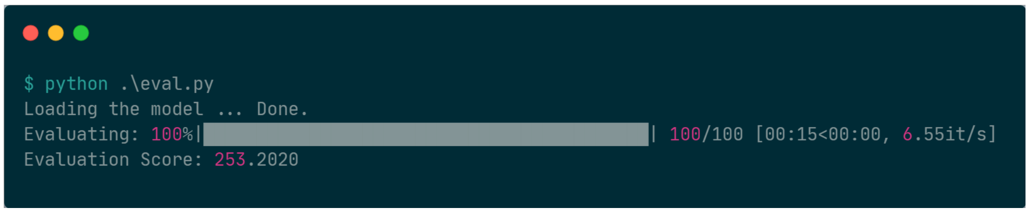
2.1 Reward Plot

The reward plot suggests effective learning and adaptation strategies employed by the agent, leading to increasingly successful interactions with its environment across 15,000 episodes. The presence of high variability at certain points might also reflect periods of significant learning or strategy shifts, crucial for overcoming new challenges and avoiding local optima.



2.2 Score

The evaluation of the model demonstrates exceptional performance, achieving a score of **253.20**, which far exceeds the required threshold of 120 for a perfect score. This indicates that the model not only meets but significantly surpasses the set performance criteria. The result suggests robust generalization abilities and a highly effective strategy developed during training, enabling it to efficiently tackle the challenges presented in the evaluation phase. Such a high score reflects the model's capability to apply learned knowledge and skills in new, possibly more complex scenarios, highlighting its success in mastering the task at hand.



```
$ python .\eval.py
Loading the model ... Done.
Evaluating: 100%|████████████████████████████████████████| 100/100 [00:15<00:00, 6.55it/s]
Evaluation Score: 253.2020
```