Hyper Parameters in Reinforcement Learning

Here’s a brief explanation about the workings of hyper parameters in Reinforcement Learning.

Lunar_Lander-v2 by Oleg Klimov

Definition of the Hyper Parameters

Here's a breakdown of the parameters you've provided:

```
model = PPO( policy='MlpPolicy', env=env, n_steps=1024, batch
```

- `policy='MlpPolicy'`: Specifies the type of policy architecture to be used. In this case, it seems to be a multi-layer perceptron (MLP) policy, which is a type of neural network architecture.
- `env=env`: Specifies the environment for which the agent will be trained. The `env` variable likely holds an instance of the environment, such as those provided by OpenAI Gym.
- `n_steps=1024`: Defines the number of steps to collect for each update. During training, the agent collects experiences by interacting with the environment, and `n_steps` determines the number of steps before performing a policy update.
- `batch_size=64`: Sets the batch size used for training. The collected experiences are divided into batches, and each batch is used to update the policy.
- `n_epochs=4`: Specifies the number of optimization epochs. In each epoch, the collected experiences are used multiple times to update the policy.
- `gamma=0.999`: The discount factor, which influences the importance of future rewards in the agent's decision-making process. A value close to 1 gives more

weight to future rewards.

- `gae_lambda=0.98`: The Generalized Advantage Estimation (GAE) parameter, which controls the trade-off between bias and variance in estimating the advantages during the policy update.
- `ent_coef=0.01`: The entropy coefficient, which balances exploration and exploitation. Higher values encourage more exploration.
- `verbose=1`: Determines the verbosity level during training. Setting it to 1 typically results in the algorithm printing information about the training process.

Relationship between `n_steps` and `batch_size`:

- `n_steps`: This parameter defines the number of steps the agent takes in the environment before performing a policy update. It influences the length of the trajectories or episodes that the agent collects. A larger `n_steps` means longer trajectories.
- `batch_size`: This parameter determines how many steps are used in each mini-batch during the policy update. The collected experiences are divided into mini-batches of size `batch_size`, and each mini-batch is used to perform a policy update.

The relationship is essentially that the total number of steps collected (determined by `n_steps`) is divided into mini-batches of size `batch_size` for updating the policy. The actual number of policy updates during training is determined by the total number of steps collected divided by `batch_size`.

Verbosity (`verbose`):

- `verbose`: This parameter controls the amount of information printed to the console during the training process. It is typically used for debugging or monitoring the progress of the training.
  - `verbose=0`: No information is printed.
  - `verbose=1`: Prints information about the training process, such as the number of updates, the value of the loss function, etc.
  - `verbose=2`: Provides additional details, potentially including information about each policy update or other relevant statistics.

1. **Epochs and Batches:**
    - In one epoch, the model will perform several updates to its policy. The number of updates in one epoch is determined by the total number of steps collected divided by the batch size.
    - In this case, if `n_steps` (the number of steps collected for each update) is 1024 and `batch_size` is 64, then the number of updates per epoch would be 1024/64=16.
2. **Steps and Policy Update:**
    - During each of these 16 updates in one epoch, the model will use a batch of 64 steps to perform a policy update. The model collects experiences (observations, actions, rewards) from the environment for 1024 steps before using a batch of 64 steps to update the policy.
3. **Episode:**
    - In reinforcement learning, an "episode" is a sequence of interactions between the agent and the environment that starts from an initial state, involves a series of actions taken by the agent, and concludes when a termination condition is met (e.g., reaching a terminal state or a maximum time step). An episode represents one complete run of the environment from start to finish.

The `model.learn(total_timesteps=1000000)` line is initiating the training process for the reinforcement learning model. Let's break down what this line of code does:

```
pythonCopy code
model.learn(total_timesteps=1000000)
```

- `total_timesteps=1000000`:
    - This parameter specifies the total number of timesteps or steps the agent will take in the environment during the training process. The training will continue until the specified number of timesteps is reached.
- `model.learn()`:

- The `learn()` method is typically a function provided by reinforcement learning libraries, such as Stable Baselines or OpenAI Gym. This method is responsible for executing the training process, where the agent interacts with the environment, collects experiences, and updates its policy based on those experiences.

Training Workflow:

1. **Initialization:**
   - Before calling `model.learn()`, you should have already initialized the model with its architecture and hyperparameters using the `PPO` constructor or a similar method.
2. **Environment Interaction:**
   - The `learn()` method initiates the training loop. During each iteration of the loop, the agent interacts with the environment, collecting experiences (observations, actions, rewards) over a certain number of timesteps.
3. **Policy Update:**
   - After collecting a batch of experiences, the agent updates its policy based on those experiences. The frequency of policy updates is influenced by parameters such as `n_steps` and `batch_size`.
4. **Training Progress:**
   - The training process continues until the total number of timesteps specified by `total_timesteps` is reached. The agent adapts its policy over time to improve its performance in the given environment.

Purpose of `total_timesteps`:

- The `total_timesteps` parameter serves as a stopping criterion for training. Once the agent has interacted with the environment for the specified total number of timesteps, the `model.learn()` process will conclude, and the trained model will be ready for evaluation or use in applications.
- This parameter allows you to control the duration of training, especially when you have a specific amount of computational resources or time available for training.

In summary, `model.learn(total_timesteps=1000000)` initiates the training of the reinforcement learning model, and the training process continues until the agent

has interacted with the environment for a total of 1,000,000 timesteps.

The training loop output you provided is a common format used in reinforcement learning libraries, and it displays various metrics and information about the training process. Let's break down each section of the output:

First Block:

```
---------------------------------
| rollout/            |          |
|    ep_len_mean      | 91.5     |
|    ep_rew_mean      | -186     |
| time/               |          |
|    fps              | 3064     |
|    iterations       | 1        |
|    time_elapsed     | 10       |
|    total_timesteps  | 32768    |
---------------------------------
```

- `rollout/ep_len_mean`: Average length (number of timesteps) of episodes in the current rollout. In this case, it's 91.5.
- `rollout/ep_rew_mean`: Average cumulative reward of episodes in the current rollout. In this case, it's -186, which suggests that, on average, the agent is receiving negative rewards during episodes.
- `time/fps`: Frames per second, indicating the speed of training. In this case, it's 3064, indicating that the training process is running at a high speed.
- `time/iterations`: Number of training iterations. In this case, it's 1, indicating that this is the first iteration of the training loop.
- `time/time_elapsed`: Time elapsed since the beginning of training. In this case, it's 10 seconds.
- `total_timesteps`: Total number of timesteps the agent has interacted with the environment so far. In this case, it's 32768.

Second Block:

```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 90.2        |
|    ep_rew_mean          | -134        |
| time/                   |             |
|    fps                  | 1693        |
|    iterations           | 2           |
|    time_elapsed         | 38          |
|    total_timesteps      | 65536       |
| train/                  |             |
|    approx_kl            | 0.008383102 |
|    clip_fraction        | 0.0829      |
|    clip_range           | 0.2         |
|    entropy_loss         | -1.38       |
|    explained_variance   | 0.00136     |
|    learning_rate        | 0.0003      |
|    loss                 | 4.44e+03    |
|    n_updates            | 8           |
|    policy_gradient_loss | -0.00603    |
|    value_loss           | 1.38e+04    |
-----------------------------------------
```

- `rollout/ep_len_mean` and `rollout/ep_rew_mean`: Similar to the first block, these metrics represent the average episode length and average cumulative reward for the current rollout.
- `time/fps`, `time/iterations`, `time/time_elapsed`, and `total_timesteps`: Similar to the first block, these metrics provide information about the training progress.
- `train/approx_kl`: Approximate Kullback-Leibler divergence, a measure of how much the new policy differs from the old policy.
- `train/clip_fraction`: Fraction of policy updates where the policy is clipped. Clipping is a regularization technique used to prevent large policy updates.
- `train/clip_range`: Range within which the policy is clipped.
- `train/entropy_loss`: Entropy loss, a measure of the policy's uncertainty.
- `train/explained_variance`: Explained variance of the value function.
- `train/learning_rate`: The learning rate used for the optimization.

- `train/loss`: The total loss, which is a combination of policy gradient loss and value function loss.
- `train/n_updates`: Number of policy updates performed.
- `train/policy_gradient_loss` and `train/value_loss`: Separate components of the total loss representing the policy gradient loss and value function loss, respectively.

This output provides a comprehensive snapshot of the training process, including information about episode lengths, rewards, timing, and various training metrics. Monitoring these metrics is essential for understanding how well the agent is learning and whether adjustments to hyperparameters or the training process are necessary.

Expected Trends:

After training a reinforcement learning model, the trends and behaviors observed in the training metrics can provide insights into how well the training is progressing and the performance of the trained agent. Here are some typical trends and expectations based on the provided blocks:

Episode Length and Reward Trends:

1. **Episode Length (`rollout/ep_len_mean`):**
    - Expectation: The episode length may initially vary, but you would hope to see a trend towards longer episode lengths. This suggests that the agent is learning to perform more complex and extended sequences of actions in the environment.
2. **Episode Reward (`rollout/ep_rew_mean`):**
    - Expectation: You would typically want to see an increasing trend in the average episode reward. Positive rewards indicate that the agent is achieving its goals in the environment. A negative reward may be expected in the beginning as the agent explores and learns.

Training Metrics Trends:

1. **Approximate Kullback-Leibler Divergence (`train/approx_kl`):**
    - Expectation: It's desirable to see the KL divergence stabilize or decrease. Large divergences may indicate that the policy is changing too

quickly, potentially leading to instability.

2. **Clip Fraction (`train/clip_fraction`):**
    - Expectation: You might see the clip fraction stabilize or decrease. A higher clip fraction indicates that a significant portion of policy updates involves clipping, which might be a regularization strategy.
3. **Entropy Loss (`train/entropy_loss`):**
    - Expectation: You would want to monitor entropy loss. A decrease in entropy loss suggests that the policy becomes more deterministic over time. Maintaining some level of entropy is often desirable for exploration.
4. **Value Function Trends (`train/explained_variance` and `train/value_loss`):**
    - Expectation: You would like to see the explained variance increase, indicating that the value function is accurately predicting returns. A decreasing value loss suggests that the value function is converging.

Learning Rate and Update Trends:

1. **Learning Rate (`train/learning_rate`):**
    - Expectation: A stable or decreasing learning rate is generally preferred. A too-large learning rate might cause instability, and a too-small learning rate may slow down learning.
2. **Number of Updates (`train/n_updates`):**
    - Expectation: An increasing number of updates over time is expected as the model learns. However, the rate of updates may decrease as training progresses and the policy stabilizes.

Loss Trends:

1. **Total Loss (`train/loss`), Policy Gradient Loss (`train/policy_gradient_loss`), and Value Loss (`train/value_loss`):**
    - Expectation: A decreasing total loss is desirable, indicating that the policy and value function are improving. Monitoring the individual components (policy gradient loss and value loss) helps diagnose specific issues.

Timing and Timesteps:

1. **Frames per Second (`time/fps`), Total Timesteps (`total_timesteps`), and Time Elapsed (`time/time_elapsed`):**
    - Expectation: You may observe an increase in frames per second, a steady increase in total timesteps, and a corresponding increase in the time elapsed. The speed of training may vary based on the computational resources and complexity of the environment.

It's important to note that trends can vary based on the specific problem, environment, and algorithm used. It's recommended to monitor these trends closely and make adjustments to hyperparameters or training strategies if needed. Additionally, evaluating the agent's performance on a validation set or testing environment is crucial to ensure that the learned policy generalizes well.