Name: Aishwarya Kulkarni
PSU ID: 924430258

# CSE 584: Homework #2

Reinforcement Learning:
In RL, an agent receives knowledge by interacting with an environment. Its objective is to maximize rewards on long-term decisions. Every interaction is one step and an interaction consists of the following components: observing the current state of the environment; taking an action based on its policy-the decision-making rule; getting a reward and observing the new state. Over time, it learns which actions yield a higher cumulative reward and refines its policy in order to be more successful.

Proximal Policy Optimization (PPO):
PPO is a type of policy gradient algorithm that directly tunes the agent's policy to maximize reward. PPO achieves stability in learning through limiting how much the policy is allowed to change at every update-the so-called "proximal" part-preventing big shifts that may make learning unstable.

For this homework, I am using this optimization algorithm code named ppo2.py ( stable-baselines/stable_baselines/ppo2/ppo2.py at master · hill-a/stable-baselines (github.com) ) from the hill-a: stable-baselines git repository. The repo can be found here: stable-baselines/stable_baselines at master · hill-a/stable-baselines (github.com).

## *ABSTRACT -*

1. *High-Level Overview of the Code:*
    - Agent's Goal: Maximize its cumulative rewards collected over a period of time.
    - Learning Mechanism: Updates to the policy network will be done by PPO's clipped objective, improving stability at making steady progress.
    - Data efficiency: PPO reuses the same experience many times, taking small, conservative policy updates.
    - Value Network: The agent, by estimating state values, gets an idea about actions leading to higher rewards, hence making better decisions.

In this PPO2 code, 'env' is a **'Gym' environment** from OpenAI's Gym library, commonly used for RL tasks due to its simplicity and versatility. In reinforcement learning, an environment is a place where the agent interacts to learn some particular task. The environment:

- Sets States: It sets the various observable conditions or properties at any given instance in time. An example could be position or velocity.
- Provides Actions: All the possible moves or decisions the agent can make, such as moving left or right.
- Returns Rewards: This is feedback given to the agent after each move taken. For example, the correct moves may be rewarded with +1 and -1 for wrong moves.

OpenAI's Gym environments have tasks that range from simple simulations, such as CartPole-balancing of a pole on a moving cart, to complex environments like Atari games. Each environment possesses a:

- State space: All the possible states the agent can observe.
- Action space: The set of all actions the agent can undertake.
- Reward structure: Rules regarding rewarding or estimating rewards for the agent's actions.

This PPO2 code is trained to solve a particular task by maximizing its cumulative rewards. The code implements Proximal Policy Optimization, one of the more powerful yet highly stable methods for learning complex decision-making policies. PPO2 can be applied to tasks requiring sequential decision-making, from robot control to playing games, including financial modeling.

2. *Overall Process in PPO2:*
    i. Initialization: The model first initializes itself with the environment, env, sets up the policy structure by creating placeholders and defines the parameter such as discount factor and learning rate. The agent is formed with all configurations in the '__init__' procedure, which includes:
    - In the __init__ method, the agent is created with all configurations, including:

- Environment: This refers to where the agent acts- for example, a game or simulation.
- Policy: The architecture of the neural network used-for example, MLP or CNN to make decisions.
- Hyper-parameters:
  - → Discount factor/ gamma: It indicates the value of placing future rewards from the present reward. As much as gamma values are closer to 1, the agent takes into consideration future rewards; with a smaller gamma, the agent will be more concerned about immediate rewards.
  - → Learning Rate: Step size for every update during training.

ii. Model Setup: The neural networks used in PPO2 are constructed through the function 'setup_model'.

- Policy Network: Acquires the ability to associate states (observations) with actions. This network directs the agent's choices in PPO.
- Value Network: Calculates a state's "goodness" based on anticipated future benefits. This aids PPO in striking a balance between exploitation (picking the most well-known action) and exploration (trying novel actions).

While this is being put up: Placeholders are made for network inputs like rewards, actions, and observations. The chosen policy (such as CNN or MLP) determines the neural network's structure.

iii. Run Episodes and Collect Data: The '_run' method gathers data by interacting with the environment. The agent goes step by step through the environment, observing the states, executing the actions and gathering the rewards on each step. Such data is called rollout and means a lot for training since it can provide real examples of the consequences of certain actions. So, on every step:

- There is some current state provided by the environment.
- An action is performed by the agent through the policy.
- The environment responds with a reward and a new state.

Rollouts are batched up and then used to update the agent's policy during training.

iv. Policy Training: '_train_step' is where PPO actually updates the policy with an aim of improving the performance of an agent:

- Clipping: PPO will bound each update to lie within a "safe" range using some form of clipping. That would prevent abrupt, large changes in the policy that might destabilize learning.
- Computation of Loss: The PPO objective or loss function balances two aspects:
  - → Policy Loss: It quantifies how much better the selected actions are.
  - → Value Loss: the difference in how well the value network predicts the state values to understand the long-term consequences of actions.

PPO does a backpropagation using this loss and updates the policy and value networks; hence, the agent's decision will get better step by step.

v. Learning Loop: The 'learn' method manages the entire training loop by orchestrating the above steps:

- The agent collects rollouts by calling _run.
- Then, _train_step is called once after each rollout in order to modify the policy.
- This loop flows for a number of provided iterations or until performance objectives met.

It keeps track of the agent's progress by logging some very useful metrics-rewards and losses over time-and helps us judge how much the agent is learning.

## *CORE SECTIONS OF RL IMPLEMENTATION –*

### 1. PPO2 Class: __init__() method:

```
def __init__(self, policy, env, gamma=0.99, n_steps=128, ent_coef=0.01, learning_rate=2.5e-4,
 vf_coef=0.5, max_grad_norm=0.5, lam=0.95, nminibatches=4, noptepochs=4, cliprange=0.2,
 cliprange_vf=None,verbose=0, tensorboard_log=None, _init_setup_model=True,
 policy_kwargs=None,full_tensorboard_log=False, seed=None, n_cpu_tf_sess=None):
        # defining and setting few fixed parameters that work for training
        self.learning_rate = learning_rate # Learning rate for optimizer
        self.cliprange = cliprange # Range for clipping the probability ratio in PPO
        self.cliprange_vf = cliprange_vf # Clip range for value function
        self.n_steps = n_steps # Number of steps the agent takes before updating the policy
        self.ent_coef = ent_coef # Coefficient for the entropy term
        self.vf_coef = vf_coef # Coefficient for the value function loss term
        self.max_grad_norm = max_grad_norm # Maximum norm for gradients
        self.gamma = gamma # discount factor
        self.lam = lam # lambda GAE
        self.nminibatches = nminibatches # Number of mini-batches
        self.noptepochs = noptepochs #no. of epochs, optimize policy using the same batch of data
        self.tensorboard_log = tensorboard_log # Path to log in TensorBoard, if needed
        self.full_tensorboard_log = full_tensorboard_log # Boolean variable to log all TensorBoard
 variables, if set to True

        # these steps are to assign some parameters to instance variables (like placeholder
 variables), set to None
        self.action_ph = None # placeholder for actions
        self.advs_ph = None # placeholder for advantages
        self.rewards_ph = None # placeholder for rewards
        self.old_neglog_pac_ph = None # placeholder for old policy's negative log probability
        self.old_vpred_ph = None # placeholder for old policy's value prediction
        self.learning_rate_ph = None # placeholder for learning rate, dynamic
        self.clip_range_ph = None # placeholder for clipping range, dynami
        self.entropy = None # placeholder for entropy
        self.vf_loss = None # placeholder for value function loss
        self.pg_loss = None # placeholder for policy gradient loss
        self.approxkl = None # placeholder for approx KL divergence
        self.clipfrac = None # placeholder for clipping fraction, to moniter ppolicy stability
        self._train = None
        self.loss_names = None
        self.train_model = None # model for training
        self.act_model = None # model for acting
        self.value = None #placeholder for value estimates
        self.n_batch = None # placeholder for batch size
        self.summary = None # placeholder for Tensorboard summary

        # super class
        super().__init__(policy=policy, env=env, verbose=verbose, requires_vec_env=True,
                         _init_setup_model=_init_setup_model, policy_kwargs=policy_kwargs,
                         seed=seed, n_cpu_tf_sess=n_cpu_tf_sess)

        #make a call to the parent class constructor
        if _init_setup_model:
            self.setup_model()
```

### 2. PPO2 Class: setup_model() method

```
    def setup_model(self):
        with SetVerbosity(self.verbose):  # setting the logging verbosity
                assert issubclass(self.policy, ActorCriticPolicy), "Error: the input policy for
                the PPO2 model must be an instance of common.policies.ActorCriticPolicy."
            self.n_batch = self.n_envs * self.n_steps # gives the total batch size
            # set a Tensorflow graph for the model
            self.graph = tf.Graph()
            with self.graph.as_default():
                self.set_random_seed(self.seed)#place a random seed value for reproducability
                # start a Tensorflow session
                self.sess = tf_util.make_session(num_cpu=self.n_cpu_tf_sess, graph=self.graph)
                # handle the bacth sizes
```

```
            n_batch_step = None
            n_batch_train = None
            if issubclass(self.policy, RecurrentActorCriticPolicy):
                assert self.n_envs % self.nminibatches == 0, "For recurrent policies, the
number of environments run in parallel should be a multiple of nminibatches."
                n_batch_step = self.n_envs
                n_batch_train = self.n_batch // self.nminibatches

            # set an actor-critic model for actions and the value estimates
            act_model = self.policy(self.sess, self.observation_space, self.action_space,
self.n_envs, 1, n_batch_step, reuse=False, **self.policy_kwargs)
            with tf.variable_scope("train_model", reuse=True,
            custom_getter=tf_util.outer_scope_getter("train_model")):
                train_model = self.policy(self.sess, self.observation_space,
self.action_space, self.n_envs // self.nminibatches, self.n_steps, n_batch_train, reuse=True,
**self.policy_kwargs)

            # these are placeholders for actions, rewards, ...
            with tf.variable_scope("loss", reuse=False):
                self.action_ph = train_model.pdtype.sample_placeholder([None],
name="action_ph")
                self.advs_ph = tf.placeholder(tf.float32, [None], name="advs_ph")
                self.rewards_ph = tf.placeholder(tf.float32, [None], name="rewards_ph")
                self.old_neglog_pac_ph = tf.placeholder(tf.float32, [None],
name="old_neglog_pac_ph")
                self.old_vpred_ph = tf.placeholder(tf.float32, [None], name="old_vpred_ph")
                self.learning_rate_ph = tf.placeholder(tf.float32, [],
name="learning_rate_ph")
                self.clip_range_ph = tf.placeholder(tf.float32, [], name="clip_range_ph")

                # this calculates the negative probablity and entropy
                neglogpac = train_model.proba_distribution.neglogp(self.action_ph)
                self.entropy = tf.reduce_mean(train_model.proba_distribution.entropy())

                vpred = train_model.value_flat

                # Value function clipping: not present in the original PPO
                if self.cliprange_vf is None:
                    # Default behavior (legacy from OpenAI baselines):
                    # use the same clipping as for the policy
                    self.clip_range_vf_ph = self.clip_range_ph
                    self.cliprange_vf = self.cliprange
                elif isinstance(self.cliprange_vf, (float, int)) and self.cliprange_vf < 0:
                    # Original PPO implementation: no value function clipping
                    self.clip_range_vf_ph = None
                else:
                    # Last possible behavior: clipping range
                    # specific to the value function
                    self.clip_range_vf_ph = tf.placeholder(tf.float32, [],
name="clip_range_vf_ph")

                if self.clip_range_vf_ph is None:
                    # No clipping
                    vpred_clipped = train_model.value_flat
                else:
                    # Clip the different between old and new value
                    # NOTE: this depends on the reward scaling
                    vpred_clipped = self.old_vpred_ph + \
                        tf.clip_by_value(train_model.value_flat - self.old_vpred_ph,
                                         - self.clip_range_vf_ph, self.clip_range_vf_ph)


                # calculate losses
                vf_losses1 = tf.square(vpred - self.rewards_ph)
                vf_losses2 = tf.square(vpred_clipped - self.rewards_ph)
                self.vf_loss = .5 * tf.reduce_mean(tf.maximum(vf_losses1, vf_losses2))
                ratio = tf.exp(self.old_neglog_pac_ph - neglogpac)
                pg_losses = -self.advs_ph * ratio
                pg_losses2 = -self.advs_ph * tf.clip_by_value(ratio, 1.0 - self.clip_range_ph,
1.0 + self.clip_range_ph)
```

```
                self.pg_loss = tf.reduce_mean(tf.maximum(pg_losses, pg_losses2))
                self.approxkl = .5 * tf.reduce_mean(tf.square(neglogpac-
self.old_neglog_pac_ph))
                self.clipfrac = tf.reduce_mean(tf.cast(tf.greater(tf.abs(ratio - 1.0),
self.clip_range_ph), tf.float32))
                # ppo clipped loss
                loss = self.pg_loss-self.entropy*self.ent_coef+self.vf_loss*self.vf_coef

                # summarize
                tf.summary.scalar('entropy_loss', self.entropy)
                tf.summary.scalar('policy_gradient_loss', self.pg_loss)
                tf.summary.scalar('value_function_loss', self.vf_loss)
                tf.summary.scalar('approximate_kullback-leibler', self.approxkl)
                tf.summary.scalar('clip_factor', self.clipfrac)
                tf.summary.scalar('loss', loss)

                # setup gradient optimization and apply clipping
                with tf.variable_scope('model'):
                    self.params = tf.trainable_variables()
                    if self.full_tensorboard_log:
                        for var in self.params:
                            tf.summary.histogram(var.name, var)
                grads = tf.gradients(loss, self.params)
                if self.max_grad_norm is not None:
                    grads, _grad_norm = tf.clip_by_global_norm(grads, self.max_grad_norm)
                grads = list(zip(grads, self.params))
            trainer = tf.train.AdamOptimizer(learning_rate=self.learning_rate_ph,epsilon=1e-5)
            self._train = trainer.apply_gradients(grads)
            self.loss_names = ['policy_loss', 'value_loss', 'policy_entropy', 'approxkl',
'clipfrac']

            with tf.variable_scope("input_info", reuse=False):
                tf.summary.scalar('discounted_rewards', tf.reduce_mean(self.rewards_ph))
                tf.summary.scalar('learning_rate', tf.reduce_mean(self.learning_rate_ph))
                tf.summary.scalar('advantage', tf.reduce_mean(self.advs_ph))
                tf.summary.scalar('clip_range', tf.reduce_mean(self.clip_range_ph))
                if self.clip_range_vf_ph is not None:
                    tf.summary.scalar('clip_range_vf', tf.reduce_mean(self.clip_range_vf_ph))

                tf.summary.scalar('old_neglog_action_probability',
tf.reduce_mean(self.old_neglog_pac_ph))
                tf.summary.scalar('old_value_pred', tf.reduce_mean(self.old_vpred_ph))

                if self.full_tensorboard_log:
                    tf.summary.histogram('discounted_rewards', self.rewards_ph)
                    tf.summary.histogram('learning_rate', self.learning_rate_ph)
                    tf.summary.histogram('advantage', self.advs_ph)
                    tf.summary.histogram('clip_range', self.clip_range_ph)
                    tf.summary.histogram('old_neglog_action_probability',
self.old_neglog_pac_ph)
                    tf.summary.histogram('old_value_pred', self.old_vpred_ph)
                    if tf_util.is_image(self.observation_space):
                        tf.summary.image('observation', train_model.obs_ph)
                    else:
                        tf.summary.histogram('observation', train_model.obs_ph)

            # set some parameters for the model to use at training
            self.train_model = train_model
            self.act_model = act_model
            self.step = act_model.step
            self.proba_step = act_model.proba_step
            self.value = act_model.value
            self.initial_state = act_model.initial_state
            tf.global_variables_initializer().run(session=self.sess)  # pylint: disable=E1101
            self.summary = tf.summary.merge_all()
```

### 3. PPO2 Class: _train_step() method

```
    def _train_step(self, learning_rate, cliprange, obs, returns, masks, actions, values,
neglogpacs, update, writer, states=None, cliprange_vf=None):
```

```
        """
        Training of PPO2 Algorithm

        :param learning_rate: (float) learning rate
        :param cliprange: (float) Clipping factor
        :param obs: (np.ndarray) The current observation of the environment
        :param returns: (np.ndarray) the rewards
        :param masks: (np.ndarray) The last masks for done episodes (used in recurent policies)
        :param actions: (np.ndarray) the actions
        :param values: (np.ndarray) the values
        :param neglogpacs: (np.ndarray) Negative Log-likelihood probability of Actions
        :param update: (int) the current step iteration
        :param writer: (TensorFlow Summary.writer) the writer for tensorboard
        :param states: (np.ndarray) For recurrent policies, the internal state of the recurrent
model
        :return: policy gradient loss, value function loss, policy entropy,
                approximation of kl divergence, updated clipping range, training update operation
        :param cliprange_vf: (float) Clipping factor for the value function
        """
        # get the advantages by subtracting value (estimates) from returns
        advs = returns - values
        # Standardize advantages to have mean 0 and std 1, stabilizes training
        advs = (advs - advs.mean()) / (advs.std() + 1e-8)

        # make a dictionary named td_map and use it to map input placeholders --> actual data
        td_map = {self.train_model.obs_ph: obs, self.action_ph: actions,
                  self.advs_ph: advs, self.rewards_ph: returns,
                  self.learning_rate_ph: learning_rate, self.clip_range_ph: cliprange,
                  self.old_neglog_pac_ph: neglogpacs, self.old_vpred_ph: values}
        # to check if states were provided, if yes - feed them to the placeholder
        if states is not None:
            td_map[self.train_model.states_ph] = states
            td_map[self.train_model.dones_ph] = masks
        # if value function is provided and valid, add an extra clipping range
        if cliprange_vf is not None and cliprange_vf >= 0:
            td_map[self.clip_range_vf_ph] = cliprange_vf

        # get the frequency factor for update depending on policy recurrence
        if states is None:
            update_fac = max(self.n_batch // self.nminibatches // self.noptepochs, 1)
        else:
            update_fac = max(self.n_batch//self.nminibatches//self.noptepochs//self.n_steps, 1)

        # TensorBoard Logging if writer is provided, log full metadata every 10th update
        if writer is not None:
            # run with the metadata there every 10 steps (if full logging is enabled)
            if self.full_tensorboard_log and (1 + update) % 10 == 0:
                run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
                run_metadata = tf.RunMetadata()
                summary, policy_loss, value_loss, policy_entropy, approxkl, clipfrac, _ =
self.sess.run([self.summary, self.pg_loss, self.vf_loss, self.entropy, self.approxkl,
self.clipfrac, self._train], td_map, options=run_options, run_metadata=run_metadata)
                writer.add_run_metadata(run_metadata, 'step%d' % (update * update_fac))
            else:
                # if not, make a normal run without the metadata
                summary, policy_loss, value_loss, policy_entropy, approxkl, clipfrac, _ =
self.sess.run([self.summary, self.pg_loss, self.vf_loss, self.entropy, self.approxkl,
self.clipfrac, self._train], td_map)

            # after, note the summary for TensorBoard visualization
            writer.add_summary(summary, (update * update_fac))

        else:
            # if no writer, run the training without metadata
            policy_loss, value_loss, policy_entropy, approxkl, clipfrac, _ = self.sess.run(
                [self.pg_loss, self.vf_loss, self.entropy, self.approxkl, self.clipfrac,
self._train], td_map)

        # function returns the key metrics: losses, entropy, KL-divergence, and clipping fraction
        return policy_loss, value_loss, policy_entropy, approxkl, clipfrac
```

## 4. PPO2 Class: learn() method

```
    def learn(self, total_timesteps, callback=None, log_interval=1, tb_log_name="PPO2",
              reset_num_timesteps=True):
        # Transform to callable if needed
        # Make the learning rate, clipping range and value function clipping range get updated
        self.learning_rate = get_schedule_fn(self.learning_rate)
        self.cliprange = get_schedule_fn(self.cliprange)
        cliprange_vf = get_schedule_fn(self.cliprange_vf)

        # Reset or initialize the timesteps counter
        new_tb_log = self._init_num_timesteps(reset_num_timesteps)

        # Initialize the callback if provided
        callback = self._init_callback(callback)

        # Set verbosity and init tensorboard writer
        with SetVerbosity(self.verbose), TensorboardWriter(self.graph, self.tensorboard_log,
tb_log_name, new_tb_log)
                as writer:
            # Prepare for training: setup model, alarms and others
            self._setup_learn()

            # Start timing for training session
            t_first_start = time.time()
            # Calculate the total number of updates required to reach the desired timesteps
            n_updates = total_timesteps // self.n_batch
            # Inform callback that training is starting
            callback.on_training_start(locals(), globals())

            # Main training loop, going through each update
            for update in range(1, n_updates + 1):
                # Assert that minibatch size is valid (checks if it divides batch size equally)
                assert self.n_batch % self.nminibatches == 0, ("The number of minibatches
(`nminibatches`) is not a factor of the total number of samples collected per rollout
(`n_batch`), some samples won't be used.)
                # Calculate the minibatch size
                batch_size = self.n_batch // self.nminibatches

                # Save the start time for each update
                t_start = time.time()

                # Get fraction of timesteps remaining
                frac = 1.0 - (update - 1.0) / n_updates

                # Adjust learning rate and clipping range based on progress
                lr_now = self.learning_rate(frac)
                cliprange_now = self.cliprange(frac)
                cliprange_vf_now = cliprange_vf(frac)

                # Callback should indicate to self that rollout collection is about to start
                callback.on_rollout_start()

                # true_reward is the reward without discount
                # Rollout environment, collect experience data
                rollout = self.runner.run(callback)

                # Unpack rollout data
                obs, returns, masks, actions, values, neglogpacs, states, ep_infos, true_reward =
rollout

                # Signal end of rollout collection to callback
                callback.on_rollout_end()

                # Early stopping due to the callback
                # Check whether early stopping was requested via callback
                if not self.runner.continue_training:
                    break

                # Collect statistics of all episodes
```

```
            self.ep_info_buf.extend(ep_infos)
            mb_loss_vals = [] # Create list to store loss values

            # Update model either by non-recurrent or recurrent approach, respectively
            if states is None:  # nonrecurrent version
         # Compute update frequency factor based on the no. of processed minibatches andepoch
                update_fac = max(self.n_batch // self.nminibatches // self.noptepochs, 1)
                inds = np.arange(self.n_batch)  # An array for randomized indices
                for epoch_num in range(self.noptepochs):
                    np.random.shuffle(inds)  # Shuffle indices in every epoch
                    for start in range(0, self.n_batch, batch_size):
                        # current timestep for logging purposes
                        timestep = self.num_timesteps // update_fac + ((epoch_num *
self.n_batch + start) // batch_size)
                        # define end of minibatch
                        end = start + batch_size
                        mbinds = inds[start:end] # get the indices for the minibatch
                        # create slices for minibatch data
                        slices = (arr[mbinds] for arr in (obs, returns, masks, actions,
values, neglogpacs))
                        # do training step, append loss values
                        mb_loss_vals.append(self._train_step(lr_now, cliprange_now,
*slices,writer=writer, update=timestep, cliprange_vf=cliprange_vf_now))
            else:  # recurrent version
                # adjust update frequency factor for recurrent case
                update_fac = max(self.n_batch // self.nminibatches // self.noptepochs //
self.n_steps, 1)
                # Make sure number of environments is divisible by number of mini batches
                assert self.n_envs % self.nminibatches == 0
                env_indices = np.arange(self.n_envs)  # get environment indices
                flat_indices = np.arange(self.n_envs * self.n_steps).reshape(self.n_envs,
self.n_steps)
                envs_per_batch = batch_size // self.n_steps   # get environments per minibatch
                for epoch_num in range(self.noptepochs):
                    np.random.shuffle(env_indices) # shuffle the environments at each epoch
                    for start in range(0, self.n_envs, envs_per_batch):
                        # Compute current timestep used for logging
                        timestep = self.num_timesteps // update_fac + ((epoch_num *
self.n_envs + start) // envs_per_batch)
                        # calculate the end of minibatch
                        end = start + envs_per_batch
                        mb_env_inds = env_indices[start:end] # the environment indices for
minibatch
                        mb_flat_inds = flat_indices[mb_env_inds].ravel() # flatten indices for
batching
                        # Create slices and states for recurrent model
                        slices = (arr[mb_flat_inds] for arr in (obs, returns, masks, actions,
values, neglogpacs))
                        mb_states = states[mb_env_inds]
                        # Train step, appending loss values
                        mb_loss_vals.append(self._train_step(lr_now,cliprange_now,*slices,
update=timestep,writer=writer, states=mb_states,cliprange_vf=cliprange_vf_now))

            # get the mean loss across minibatches
            loss_vals = np.mean(mb_loss_vals, axis=0)
            # get the frames per second (FPS) (for update performance)
            t_now = time.time()
            fps = int(self.n_batch / (t_now - t_start))

            # log the rewards and statistics in Tensorboard
            if writer is not None:
                total_episode_reward_logger(self.episode_reward,
                                            true_reward.reshape((self.n_envs, self.n_steps)),
                                            masks.reshape((self.n_envs, self.n_steps)),
                                            writer, self.num_timesteps)

            # log the training info on the first update or at every `log_interval` update
            if self.verbose >= 1 and (update % log_interval == 0 or update == 1):
                explained_var = explained_variance(values, returns) # get explained variance
for values
```

```
                logger.logkv("serial_timesteps", update * self.n_steps)
                logger.logkv("n_updates", update)
                logger.logkv("total_timesteps", self.num_timesteps)
                logger.logkv("fps", fps)
                logger.logkv("explained_variance", float(explained_var))
                if len(self.ep_info_buf) > 0 and len(self.ep_info_buf[0]) > 0:
                    # log the episode mean reward and lengths (if available)
                    logger.logkv('ep_reward_mean', safe_mean([ep_info['r'] for ep_info in
 self.ep_info_buf]))
                    logger.logkv('ep_len_mean', safe_mean([ep_info['l'] for ep_info in
 self.ep_info_buf]))
                logger.logkv('time_elapsed', t_start - t_first_start)

                # log each component of the loss
                for (loss_val, loss_name) in zip(loss_vals, self.loss_names):
                    logger.logkv(loss_name, loss_val)
                logger.dumpkvs()

        # tell the callback that the training has ended
        callback.on_training_end()
        return self  # return the self instance
```

**5. Runner Class: __init__() and _run() method**

```
class Runner(AbstractEnvRunner):
    def __init__(self, *, env, model, n_steps, gamma, lam):
        """
        A runner to learn the policy of an environment for a model
        :param env: (Gym environment) The environment to learn from
        :param model: (Model) The model to learn
        :param n_steps: (int) The number of steps to run for each environment
        :param gamma: (float) Discount factor
        :param lam: (float) Factor for trade-off of bias vs variance for Generalized Advantage
 Estimator
        """
        # super class iitialization
        super().__init__(env=env, model=model, n_steps=n_steps)

        # GAE parameters
        self.lam = lam
        self.gamma = gamma

    def _run(self):
        """
        Run a learning step of the model
        :return:
            - observations: (np.ndarray) the observations
            - rewards: (np.ndarray) the rewards
            - masks: (numpy bool) whether an episode is over or not
            - actions: (np.ndarray) the actions
            - values: (np.ndarray) the value function output
            - negative log probabilities: (np.ndarray)
            - states: (np.ndarray) the internal states of the recurrent policies
            - infos: (dict) the extra information of the model
        """
        # mb stands for minibatch
        # Initializations of empty lists for storing data from the minibatches at each step
        mb_obs, mb_rewards, mb_actions, mb_values, mb_dones, mb_neglogpacs = [],[],[],[],[],[]
        mb_states = self.states # Store the current internal state of the model
        ep_infos = [] # Initialize list to store information about each episode
         # Iterate over each step in the current batch of steps.
        for _ in range(self.n_steps):
            # Get action, value, updated state, and negative log probability from model.
            actions, values, self.states, neglogpacs = self.model.step(self.obs, self.states,
 self.dones)  # pytype: disable=attribute-error
            # Save current observation, action, value, and log probability into minibatch lists
            mb_obs.append(self.obs.copy())
            mb_actions.append(actions)
            mb_values.append(values)
            mb_neglogpacs.append(neglogpacs)
```

```
            mb_dones.append(self.dones) # Check if episode has finished for each env
            clipped_actions = actions
            # Clip actions if action space is continuous (Box) to stay within bounds
            # Clip the actions to avoid out of bound error
            if isinstance(self.env.action_space, gym.spaces.Box):
                clipped_actions = np.clip(actions, self.env.action_space.low,
self.env.action_space.high)
            # Take the clipped step in the environment
            self.obs[:], rewards, self.dones, infos = self.env.step(clipped_actions)
            # Increase total number of steps taken by model
            self.model.num_timesteps += self.n_envs
            # If callback is given, possibly early stop training
            if self.callback is not None:
                # Abort training early
                self.callback.update_locals(locals()) # Update callback with locals
                if self.callback.on_step() is False: # If callback returns False, stop training
                    self.continue_training = False
                    # Return dummy values
                    return [None] * 9 # Return dummy values to signal stop early
            # Add episode info if it exists
            for info in infos:
                maybe_ep_info = info.get('episode')
                if maybe_ep_info is not None:
                    ep_infos.append(maybe_ep_info)
            # Save rewards of current step in minibatch
            mb_rewards.append(rewards)
        # batch of steps to batch of rollouts
        # Turn minibatch lists into numpy arrays for further processing
        mb_obs = np.asarray(mb_obs, dtype=self.obs.dtype)
        mb_rewards = np.asarray(mb_rewards, dtype=np.float32)
        mb_actions = np.asarray(mb_actions)
        mb_values = np.asarray(mb_values, dtype=np.float32)
        mb_neglogpacs = np.asarray(mb_neglogpacs, dtype=np.float32)
        mb_dones = np.asarray(mb_dones, dtype=np.bool)
        # Calculate final value function values to complete advantage calculation
        last_values = self.model.value(self.obs, self.states, self.dones)  # pytype:
disable=attribute-error
        # discount/bootstrap off value fn
        # Initialize advantage array for GAE
        mb_advs = np.zeros_like(mb_rewards)
        true_reward = np.copy(mb_rewards) # Copy rewards for computing true rewards
        last_gae_lam = 0
        # Compute advantages in reverse order for GAE
        for step in reversed(range(self.n_steps)):
            # Terminal and next value
            # Check if next step is terminal and get the next value
            if step == self.n_steps - 1:
                nextnonterminal = 1.0 - self.dones
                nextvalues = last_values
            else:
                nextnonterminal = 1.0 - mb_dones[step + 1]
                nextvalues = mb_values[step + 1]
            # Temporal difference delta for GAE
            # Calculate the temporal difference delta for GAE
            delta = mb_rewards[step] + self.gamma * nextvalues * nextnonterminal - mb_values[step]
            # Update advantage using GAE
            # Update the advantage using the GAE formula
            mb_advs[step] = last_gae_lam = delta + self.gamma * self.lam * nextnonterminal *
last_gae_lam
        # Compute the returns by adding advantages to value estimates
        mb_returns = mb_advs + mb_values
        # Reshape and flatten each minibatch array for further processing in training
        mb_obs, mb_returns, mb_dones, mb_actions, mb_values, mb_neglogpacs, true_reward = \
            map(swap_and_flatten, (mb_obs, mb_returns, mb_dones, mb_actions, mb_values,
mb_neglogpacs, true_reward))

        # Return all minibatch data needed for training
        return mb_obs, mb_returns, mb_dones, mb_actions, mb_values, mb_neglogpacs, mb_states,
ep_infos, true_reward
```