ISIA 2025/2026 - PL 2
Stable-Baselines3 tutorial

1 Reinforcement Learning in Python with Stable Baselines 3

from https://pythonprogramming.net/introduction-reinforcement-learning-stable-baselines-3-tutorial/

updated for Gymnasium

Welcome to a tutorial series covering how to do reinforcement learning with the Stable-Baselines3 (SB3) package. The objective of the SB3 library is to be for reinforcement learning like what sklearn is for general machine learning. Stable-Baselines3 enables you to try things like PPO, then maybe some TD3 and then why not try some SAC?!

The SB3 website contains links and resources to the original papers for algorithms and even some followup information in many cases. There's really no need for you to know every single RL algorithm, just to try them, and it's completely reasonable to try some algorithms on some problem, then do more research into the ones that seem promising to you. To start, here are some quick words and definitions that you're likely to come across:

- The Environment: What are you trying to solve? (cartpole, lunar lander, some other custom environment). If you're trying to make some AI play a game, the game is the environment.
- The Model: What algorithm are you using (PPO, SAC, TRPO, TD3...etc).
- The Agent: this is the thing that interacts with the environment using an algorithm/model.

Then, looking closer at the environments, you have two major elements:

- The Observation (or "state"): What is the state of the environment? This could be imagery/visuals, or just vector information. For example, your observation in cartpole is the angle and velocity of the pole. In the bipedal walker environment, the observation contains readings from lidar, the hull's angle, leg positions...etc. An observation is all of this information at some point in time. The observation space is a description, mainly the shape, of those observations.
- The Action: What are the options for your agent in this environment? For example, in cartpole, you can either push left or right. The action space is a description of these possible actions, both in terms of the shape and type (discrete or continuous).
- Step: Take a step in the environment. In general, you pass your action to the step method, the environment performs the step and returns a new observation and reward. You can think of this like frames per second. If you're playing 30 frames per second, then you could have 30 steps per second, but it can get far more complicated than this. A step is just simply progressing in the environment.

What's discrete and continuous?

- Discrete: Think of discrete like classifications. Cartpole has a discrete action space. You can go left, or right. Nothing in between, there's no concept of sort of left or half left. It's left, or right.
- Continuous: Think of continuous like regression. It's a range of nearly infinite possibilities. The bidpedal walker environement is a continuous action space, because you set the servo torque anywhere in a range between -1 and positive 1.

Typically, a continuous environment is harder to learn, but sometimes this is a requirement. In robotics, servos/motors of decent quality tend to be naturally largely continuous. Even servos, however, are *actually* discrete, with something like 32,768 positions, but this large of a discrete space is, typically, far too big for a discrete space. That said, there are ways to convert continuous spaces to discrete, in effort to make training faster and easier, and maybe more on that later on. For now, it's just important to understand the two major types of action spaces, as well as the general description of environments, agents, and models.

Once you find some algorithm that seems to be maybe working and learning something, you should dive more deeply into that model, how it works, and the various hyperparameter options that you can tweak.

With all of that out of the way, let's play! To start, you will need Pytorch and stable-baselines3. For Pytorch, just follow the instructions here: Pytorch getting started (https://pytorch.org/get-started/locally/). For stable-baselines3: `pip3 install stable-baselines3[extra]`. Finally, we'll need some environments to learn on, for this we'll use Gymnasium, which you can get with `pip install gymnasium[box2d]`. On Linux for Gymnasium and the box2d environments, you also needed to do the following:

```
apt install xvfb ffmpeg xorg-dev libsdl2-dev swig cmake
pip install gymnasium[box2d]
```

For this tutorial, we'll start with the lunar lander environment. First, let's get a grasp of the fundamentals of our environment. When choosing algorithms to try, or creating your own environment, you will need to start thinking in terms of observations and actions, per step. The structure of a Farama Gymnasium problem is the standard by which basically everyone does reinforcement learning. Let's take a peak at the lunar lander environment:

[ ]:
```python
import gymnasium as gym

# Create the environment
env = gym.make('LunarLander-v3', render_mode='human')  # continuous: LunarLanderContinuous-v3

# required before you can step the environment
env.reset(seed=42)

# sample action:
print("sample action:", env.action_space.sample())

# observation space shape:
print("observation space shape:", env.observation_space.shape)

# sample observation:
print("sample observation:", env.observation_space.sample())

env.close()
```

Running that, you should see something like:

```
sample action: 2
observation space shape: (8,)
sample observation: [-0.51052296  0.00194223  1.4957197  -0.3037317  -0.20905018 -0.1737924
    1.8414629   0.09498857]
```

Our actions are discrete, and just 1 discrete action of 0, 1, 2, or 3. 0 means do nothing, 1 means fire the left engine, 2 means fire the bottom engine, and 3 means fire the right engine. The observation space is a vector of 8 values. Looking at the gymnasium code for this environment, we can find out what the values in the observation mean https://github.com/Farama-Foundation/Gymnasium/blob/main/gymnasium/envs/box2d/lunar_lander.py#L800:

```
     env: The environment
s (list): The state. Attributes:
            s[0] is the horizontal coordinate
            s[1] is the vertical coordinate
            s[2] is the horizontal speed
            s[3] is the vertical speed
            s[4] is the angle
            s[5] is the angular speed
            s[6] 1 if first leg has contact, else 0
            s[7] 1 if second leg has contact, else 0
```

We can see a sample of the environment by running:

[ ]:
```python
import gymnasium as gym


env = gym.make('LunarLander-v3', render_mode="human")  # continuous: LunarLanderContinuous-v3
env.reset(seed=42)

for step in range(200):
    env.render()
    # take random action
    env.step(env.action_space.sample())

env.close()
```

That doesn't look too good, but it's some an agent acting randomly, using `env.action_space.sample()`. Each time we step in the environment, the step method also returns some information to us. We can collect it with:

```
observation, reward, terminated, truncated, info = env.step(env.action_space.sample())
```

Here, we're gathering the observation, reward, whether or not the environment has reported that it's done, and any other extra info. The observation will be those 8 values listed above, the reward is a value reported by the environment, meant to be some sort of signal of how well the agent is accomplishing the desired objective. In the case of the lunar lander, the goal is to land between the two flags. Let's check out the reward and done values:

[ ]:
```python
import gymnasium as gym


env = gym.make('LunarLander-v3', render_mode="human")  # continuous: LunarLanderContinuous-v3
env.reset(seed=42)

for step in range(200):
    env.render()

    # take random action
    observation, reward, terminated, truncated, info = env.step(env.action_space.sample())
    done = terminated or truncated

    print(reward, done)

env.close()
```

```
-8.966428059751639 False
-3.2056261702008144 False
-7.918002269808301 False
-5.045565371482126 False
-4.492306794371302 False
-12.056824418777229 False
-8.002752408838138 False
-11.950438693580214 False
-10.814724683236523 False
-4.34849509508271 False
4.965781267653142 False
-5.928775063421142 False
-100 True
-100 True
-100 True
-100 True
-100 True
-100 True
-100 True
```

As you can see, we got some varying rewards, and then eventually got a bunch of negative -100s and True for the environment being done. This is the environment knowing we crashed.

The objective of reinforcement learning is essentially to keep playing in some environment with the goal of seeking out better and better rewards. So now, we need an algorithm that can solve this environment. This tends to boil down to the action space. Which algorithms can support our action space? What is our action space? It's a single discrete value.

From https://stable-baselines3.readthedocs.io/en/master/guide/algos.html:

Name	Box	Discrete	MultiDiscrete	MultiBinary	Multi Processing
ARS [1]	✔	✔	✖	✖	✔
A2C	✔	✔	✔	✔	✔
CrossQ [1]	✔	✖	✖	✖	✔
DDPG	✔	✖	✖	✖	✔
DQN	✖	✔	✖	✖	✔
HER	✔	✔	✖	✖	✔
PPO	✔	✔	✔	✔	✔
QR-DQN [1]	✖	✔	✖	✖	✔
RecurrentPPO [1]	✔	✔	✔	✔	✔
SAC	✔	✖	✖	✖	✔
TD3	✔	✖	✖	✖	✔
TQC [1]	✔	✖	✖	✖	✔
TRPO [1]	✔	✔	✔	✔	✔
Maskable PPO [1]	✖	✔	✔	✔	✔

[1] Implemented in *SB3 Contrib* (https://github.com/Stable-Baselines-Team/stable-baselines3-contrib)

This table gives the algorithms available in SB3, along with the action spaces that they support, and multiprocessing. When you have some problem, you can start by consulting this table to see which algorithms you can start off by trying. You should recognize discrete, but then there are things like "MultiDiscrete" and "Box" and "MultiBinary." What are those? Where's continuous?

- Box: For now, you can think of this as your continuous space support.
- MultiDiscrete: some environments are just one discrete action, but some environments may have multiple discrete actions to take.
- MultiBinary: Similar to MultiDiscrete, but for instances where there are only 2 options for each action

It looks like we have quite a few options to try: A2C, DQN, HER, PPO, QRDQN, and maskable PPO. There may be even more algorithms available later, so be sure to check out the SB3 algorithms (https://stable-baselines3.readthedocs.io/en/master/guide/algos.html) page later when working on your own problems. Let's try out the first one on the list: A2C. To start, we'll need to import it:

```
from stable_baselines3 import A2C
```

Then, after we've defined the environment, we'll define the model to run in this environment, and then have it learn for 10,000 timesteps.

```
model = A2C('MlpPolicy', env, verbose=1)
model.learn(total_timesteps=10000)
```

After we've got a trained model, we probably are keen to see it, so let's check out the results:

```
episodes = 10

for ep in range(episodes):
    observation = env.reset()
    done = False
    while not done:
        # pass observation to model to get predicted action
        action, _states = model.predict(observation)

        # pass action to env and get info back
        observation, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated


        # show the environment on the screen
        env.render()
```

This is a very simple example, but it's a good starting point. We'll see more examples later. Full code up to this point:

[ ]:
```
import gymnasium as gym
from stable_baselines3 import A2C


env = gym.make('LunarLander-v3', render_mode='human')  # continuous: LunarLanderContinuous-v3
env.reset(seed=42)

model = A2C('MlpPolicy', env, verbose=1, device='cpu')
model.learn(total_timesteps=10000)

episodes = 10

for ep in range(episodes):
    observation, info = env.reset()
    done = False
    while not done:
        action, _states = model.predict(observation)
        observation, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        env.render()
        print(reward)
```

Outputs in the terminal will look something like:

```
-----------------------------------
| rollout/              |          |
|    ep_len_mean        | 190      |
|    ep_rew_mean        | -195     |
| time/                 |          |
|    fps                | 506      |
|    iterations         | 1300     |
|    time_elapsed       | 12       |
|    total_timesteps    | 6500     |
| train/                |          |
|    entropy_loss       | -0.575   |
|    explained_variance | 0.0149   |
|    learning_rate      | 0.0007   |
|    n_updates          | 1299     |
|    policy_loss        | -3.38    |
|    value_loss         | 93       |
-----------------------------------
```

This contains a few statistics like how many steps your model has taken as well as probably the thing you care about most, the episode's reward mean: `ep_rew_mean`.

Okay, not terrible, but this wasn't enough time apparently to train the agent! Let's try 100,000 steps instead.

[ ]:
```python
import gymnasium as gym
from stable_baselines3 import A2C



env = gym.make('LunarLander-v3', render_mode='human')  # continuous: LunarLanderContinuous-v3
env.reset()

model = A2C('MlpPolicy', env, verbose=1, device='cpu')
model.learn(total_timesteps=100000)

episodes = 5

for ep in range(episodes):
    observation, info = env.reset()
    done = False
    while not done:
        action, _states = model.predict(observation)
        observation, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        env.render()
        print(reward)
```

Hmm, well, at least the lander isn't crashing, but it also is pretty rarely actually landing at a reasonable pace. On a realistic problem, you might start thinking about tweaking the reward a bit to maybe

disincentivise floating in place, or maybe you just need to be more patient and do more steps. A2C is a fairly old (in terms of reinforcement learning) algorithm, maybe we'll try something else instead? Let's try PPO. All we need to do is import it:

```
from stable_baselines3 import PPO
```

Then change our model from A2C to PPO:

```
model = PPO('MlpPolicy', env, verbose=1)
```

[ ]:
```
import gymnasium as gym
from stable_baselines3 import PPO



env = gym.make('LunarLander-v3', render_mode='human')  # continuous: LunarLanderContinuous-v3
env.reset()

model = PPO('MlpPolicy', env, verbose=1, device='cpu')
model.learn(total_timesteps=100000)

episodes = 5

for ep in range(episodes):
    observation, info = env.reset()
    done = False
    while not done:
        action, _states = model.predict(observation)
        observation, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        env.render()
        print(reward)
```

It's that simple to try PPO instead! After 100K steps with PPO:

```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 575         |
|    ep_rew_mean          | -0.463      |
| time/                   |             |
|    fps                  | 468         |
|    iterations           | 49          |
|    time_elapsed         | 214         |
|    total_timesteps      | 100352      |
| train/                  |             |
|    approx_kl            | 0.013918768 |
|    clip_fraction        | 0.134       |
|    clip_range           | 0.2         |
|    entropy_loss         | -0.92       |
|    explained_variance   | 0.305       |
|    learning_rate        | 0.0003      |
```

```
|    loss                 | 73.1        |
|    n_updates            | 480         |
|    policy_gradient_loss | -0.00609    |
|    value_loss           | 76.8        |
-----------------------------------------
```

This agent looks much better, and it seems to be better overall.

At this point, you should have a very general idea of how to use Stable Baslines 3 and some of how reinforcement learning works.

Now try DQN, with a Neural Network and a CUDA device, if available. All we need to do is import it:

```
from stable_baselines3 import DQN
import torch
```

Then change our model to DQN:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = DQN('MlpPolicy', env, verbose=1, device=device)
```

[ ]:
```
import gymnasium as gym
from stable_baselines3 import DQN
import torch



env = gym.make('LunarLander-v3', render_mode='human')  # continuous: LunarLanderContinuous-v3
env.reset(seed=42)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = DQN('MlpPolicy', env, verbose=1, device=device)
model.learn(total_timesteps=10000)

episodes = 10

for ep in range(episodes):
    observation, info = env.reset()
    done = False
    while not done:
        action, _states = model.predict(observation)
        observation, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        env.render()
        print(reward)
```

2 How to save and load models

This is the part 2 of the reinforcement learning with Stable Baselines 3 tutorials. We left off with training a few models in the lunar lander environment. While this was beginning to work, it seemed like maybe even more training would help. How much more? When we trained for 10,000 steps, and decided we wanted to try 100,000 steps, we had to start all over. If we want to try 1,000,000 steps, we'll also need to start over. It makes a lot more sense to save models along the way, and to probably just train until you're happy with the model or want to change it in some way, which will require starting over. With this in mind, how might we save and load models?

Our training code up to this point:

[ ]:
```
import gymnasium as gym
from stable_baselines3 import PPO

env = gym.make('LunarLander-v3', render_mode='human')
env.reset()

model = PPO('MlpPolicy', env, verbose=1, device='cpu')
model.learn(total_timesteps=100000)
```

Let's decrease the timesteps to 10,000 instead, as well as create a models directory:

```
import os



models_dir = "models/PPO"


if not os.path.exists(models_dir):
    os.makedirs(models_dir)
```

Then, we'll encase our training in a while loop:

```
TIMESTEPS = 10000
while True:
    model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False)
```

Note the `reset_num_timesteps=False`. This allows us to see the actual total number of timesteps for the model rather than resetting every iteration. We're also setting a constant for however many timesteps we want to do per iteration. Now we can save with `mode.save(PATH)`. Let's track however many iterations we've made, and then calculate what timestep that model is, saving that as the name:

```
TIMESTEPS = 10000
iters = 0
while True:
    iters += 1
    model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False)
    model.save(f"{models_dir}/{TIMESTEPS*iters}")
```

Full code up to this point:

[ ]:
```python
import gymnasium as gym
from stable_baselines3 import PPO
import os



models_dir = "models/PPO"

if not os.path.exists(models_dir):
    os.makedirs(models_dir)



env = gym.make('LunarLander-v3', render_mode=None)
env.reset()

model = PPO('MlpPolicy', env, verbose=1, device='cpu')

TIMESTEPS = 10000
iters = 0
while True:
    iters += 1

    model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False)
    model.save(f"{models_dir}/{TIMESTEPS*iters}")
```

Let's go ahead and run that for a bit, you should see every ~10,000 timesteps a model will be saved. **Keep in mind that this cell runs with an infinite loop. Interrupt it whenever you want.** You can just keep an eye on the `ep_rew_mean`, waiting for something positive, or just go make some coffee or something while you wait for the model to hopefully do something. Around 110,000 steps, the agent began scoring an average in the positives:

```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 628         |
|    ep_rew_mean          | 4.46        |
| time/                   |             |
|    fps                  | 6744        |
|    iterations           | 4           |
|    time_elapsed         | 16          |
|    total_timesteps      | 110592      |
| train/                  |             |
|    approx_kl            | 0.0025332319 |
|    clip_fraction        | 0.0189      |
|    clip_range           | 0.2         |
|    entropy_loss         | -0.975      |
|    explained_variance   | 0.759       |
|    learning_rate        | 0.0003      |
|    loss                 | 2           |
|    n_updates            | 530         |
```

```
|    policy_gradient_loss | -0.00157     |
|    value_loss           | 29.1         |
-----------------------------------------
```

By 150,000 steps, the model appears to be on to something:

```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 783         |
|    ep_rew_mean          | 80.6        |
| time/                   |             |
|    fps                  | 9801        |
|    iterations           | 4           |
|    time_elapsed         | 15          |
|    total_timesteps      | 151552      |
| train/                  |             |
|    approx_kl            | 0.013495723 |
|    clip_fraction        | 0.0808      |
|    clip_range           | 0.2         |
|    entropy_loss         | -0.723      |
|    explained_variance   | 0.719       |
|    learning_rate        | 0.0003      |
|    loss                 | 4.06        |
|    n_updates            | 730         |
|    policy_gradient_loss | -0.00426    |
|    value_loss           | 48.3        |
-----------------------------------------
```

We have these models saved, so we might as well let things continue to train as long as the model is improving quickly. While we wait, we can start programming the code required to load and actually run the model, so we can see it visually. In a separate script, the main bit of code for loading a model is:

```
env = gym.make('LunarLander-v3', render_mode='human')  # continuous: LunarLanderContinuous-v3
env.reset()


model_path = f"{models_dir}/250000.zip"
model = PPO.load(model_path, env=env, device='cpu')
```

Then, we can use the same code as we used before to play the model from part one, making our full code:

[ ]:
```
import gymnasium as gym
from stable_baselines3 import PPO


models_dir = "models/PPO"


env = gym.make('LunarLander-v3', render_mode='human')  # continuous: LunarLanderContinuous-v3
env.reset()


model_path = f"{models_dir}/250000.zip"
```

```
model = PPO.load(model_path, env=env, device='cpu')

episodes = 5

for ep in range(episodes):
    observation, info = env.reset()
    done = False
    while not done:
        action, _states = model.predict(observation)
        observation, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated
        env.render()
        print(reward)
```

This appears to be "solved" at this point, and it only took a few minutes! It wont always be this easy, but you can probably stop training this model now. On a more realistic problem, you will likely need to go for millions...or billions... of steps, and that's if things work at all. You'll also probably need to tweak things like the reward function. But, for now, let's take the win!

While we can see our model's stats in the console, it can often be hard to really visualize where you are in the training process, and it can be even harder when you're going to be comparing multiple models. The next thing we can do to make this easier is to log model performance with Tensorboard. To do this, we just need to specify a name and location for the Tensorboard logs. First, we'll make sure the log dir exists:

```
logdir = "logs"

if not os.path.exists(logdir):
    os.makedirs(logdir)
```

Next, when specifying the model, we can pass the log directory:

```
model = PPO('MlpPolicy', env, verbose=1, tensorboard_log=logdir)
```

As we train, we can name the individual log:

```
model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False, tb_log_name="PPO")
```

Finally, let's limit the number of steps to 300K. Making the full code now:

[ ]:
```
import gymnasium as gym
from stable_baselines3 import PPO
import os



models_dir = "models/PPO"
logdir = "logs"

if not os.path.exists(models_dir):
    os.makedirs(models_dir)
```

```
if not os.path.exists(logdir):
    os.makedirs(logdir)

env = gym.make('LunarLander-v3', render_mode=None)
env.reset()

model = PPO('MlpPolicy', env, verbose=1, device='cpu', tensorboard_log=logdir)

TIMESTEPS = 10000
iters = 0
for i in range(30):
    model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False, tb_log_name="PPO")
    model.save(f"{models_dir}/{TIMESTEPS*i}")
```

Now, while the model trains, we can view the results over time by opening a new terminal and doing: `tensorboard --logdir=logs`, and point your browser to http://localhost:6006

Now we can compare to other algorithms to see which one performs better. Let's try A2C again by changing the import, model definition, and log name:

[ ]:
```
import gymnasium as gym
from stable_baselines3 import A2C
import os



models_dir = "models/A2C"
logdir = "logs"

if not os.path.exists(models_dir):
    os.makedirs(models_dir)

if not os.path.exists(logdir):
    os.makedirs(logdir)

env = gym.make('LunarLander-v3', render_mode=None)
env.reset()

model = A2C('MlpPolicy', env, verbose=1, device='cpu', tensorboard_log=logdir)

TIMESTEPS = 10000
iters = 0
for i in range(30):
    model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False, tb_log_name="A2C")
    model.save(f"{models_dir}/{TIMESTEPS*i}")
```

Now, it should be much more clear the differences between PPO and A2C. The most obvious difference is A2C seems far more volatile in the reward over time.