**REINFORCE algorithm report**

Ericka Bermudez. 202230093

The following report describes the algorithm “REINFORCE”. The algorithm was invented by Ronald J. Williams in 1992, and the main idea is that learning that resulted in good outcomes is positively reinforced and should become more probable, while the ones with bad outcomes become less probable. The algorithm needs three components: (1) a parametrized policy (2) an objective to be maximized (3) a method for updating the policy parameters.

*# importing packages/libraries*

from torch.distributions import Categorical

import gym

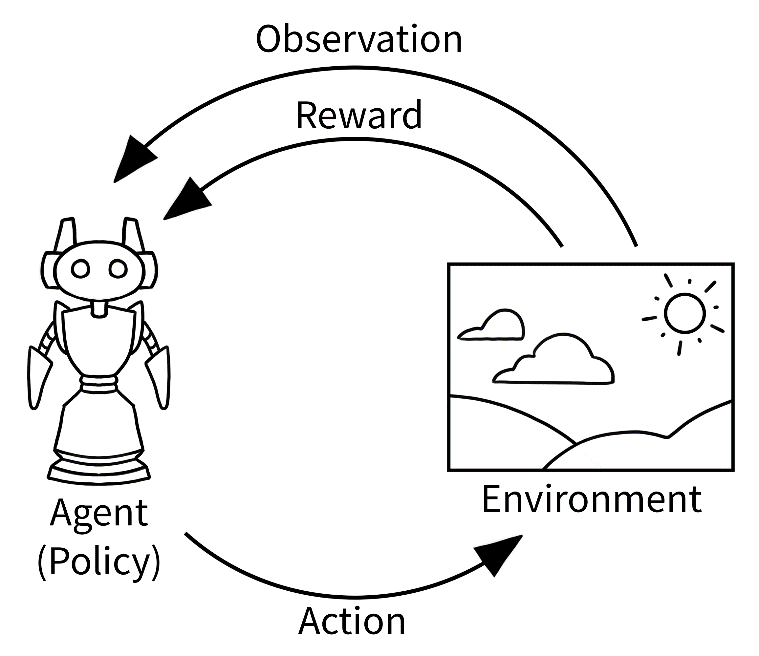
import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

These lines import all the necessary packages to use. Categorical “Creates a categorical distribution parameterized by either probs or logits (but not both)”, where the samples are integers from 0 to K-1. Gym is an API for reinforcement learning and a diverse collection of reference environments, and it implements an agent-environment loop as seen in the following picture.



Source: <https://www.gymlibrary.ml/content/api/>

Numpy is a well-known package for scientific computing, including diverse mathematical functions and linear algebra routines. Torch package “contains data structures for multi-dimensional tensors and defines mathematical operations over these tensors”. Torch nn refers to neural networks and optim to optimization algorithms.

gamma = 0.99 *# learning rate*

Gamma α is the learning rate. It controls the size of the parameters update.

class Pi(nn.Module): *# class to define the policy: its models and methods.*

This class is where the policy network is. Policy π maps states to action probabilities, how an agent produces actions in the environment to maximize the objective. In REINFORCE algorithm, an agent learns a policy and uses this to act in an environment. The policy is in charge of maximizing the cumulative discounted rewards.

def \_\_init\_\_(self, in\_dim, out\_dim):

        super(Pi, self).\_\_init\_\_()

        layers = [ *# setting up the model we will use in the policy*

            nn.Linear(in\_dim, 64),

            nn.ReLU(),

            nn.Linear(64, out\_dim)

        ]

        self.model = nn.Sequential(\*layers)

        self.onpolicy\_reset() *# giving the reset method to the policy*

        self.train() *# giving the train method to the policy*

Along these lines, we are constructing the policy network. In this case, it is a one-layer MLP with 64 hidden units, but we could substitute the model. We use the package nn to build it and give it the methods to reset the policy and train.

    def onpolicy\_reset(self):

        self.log\_probs = []

        self.rewards = []

At the time of policy reset, the logs of the probabilities and the rewards are cleared to prepare for the next state after training, as we cannot reuse the trajectory. **log\_probs** records action probabilities while **rewards** keeps the total rewards.

    def forward(self, x):

        pdparam = self.model(x)

        return pdparam

The model uses a neural network, which learns from a function. In these lines is where we make happen the sequential computation happens as a forward pass. It will take the input x to produce the predicted outputs.

def act(self, state): *#state into action*

        x = torch.from\_numpy(state.astype(np.float32)) *# to sensor*

        pdparam = self.forward(x) *# forward pass*

        pd = Categorical(logits = pdparam) *# probability distribution*

        action = pd.sample() *# pi (a | s ) in action via pd*

        log\_prob = pd.log\_prob(action) *# log\_prob of pi(a|s)*

        self.log\_probs.append(log\_prob) *# store for training*

        return action.item()

The act method defines the method to map the state to action probability; in REINFORCE algorithm, an agent lerns a policy and uses this to act in an environment. With Categorical, we construct an instance of an action probability distribution. The action is decided with a sample of the probability distribution of the params, and then on base to this action the probabilities will be calculated. After saving the probabilities, the method returns the action.

def train(pi, optimizer): *# method for training that takes as parameters the policy and optimizer*

*# inner gradient-ascent loop of REINFORCE algorithm*

    T = len(pi.rewards)

    rets = np.empty(T, dtype = np.float32)

    future\_ret = 0.0

*# compute the returns efficiently*

    for t in reversed(range(T)): *# for return of trajectory*

        future\_ret = pi.rewards[t] + gamma \* future\_ret *# gamma will help to see how much update*

        rets[t] = future\_ret

    rets = torch.tensor(rets) *# making the returns into a tensor*

    log\_probs = torch.stack(pi.log\_probs)

    loss = - log\_probs \* rets *# gradient term; negative for maximizing*

    loss = torch.sum(loss) *# get the total loss*

    optimizer.zero\_grad() *# set the gradients of all the optimized tensors to 0*

    loss.backward() *# backpropagate, compute gradients*

    optimizer.step() *# gradient-ascent, update the weights*

    return loss

The policy is a very important part of the lines of code. It is given as a parameter along with the optimizer, and during the control loop, it generates the actions to make it run. Then the return of the trajectory (discounted sum of rewards from time step *t* to the end of the trajectory) is computed.

The training step is repeated until the network has converged (outputs stop changing or the loss has minimized).

We have to increase the expected reward. After the time steps the loss will be calculated, and from this loss the gradient is calculated with respect to the parameters of the network.

We perform gradient ascent on the policy of the parameters to maximize the objective, by computing the gradient and using it to update the parameters. The optimizer is used to update the network parameters using the gradient.

def main(): *# this is where our program will start*

    env = gym.make('CartPole-v0') *# we create our environment taking it from the gym package*

    in\_dim = env.observation\_space.shape[0] *# input 4 dimensions*

    out\_dim = env.action\_space.n *# output 2 dimensions (action left and action right)*

    pi = Pi(in\_dim, out\_dim) *# policy pi\_theta for REINFORCE*

    optimizer = optim.Adam(pi.parameters(), lr = 0.01) *# selection of adam as optimizer with learning rate of 0.01*

This is the first part of the main() method. The environment we are working with currently is the CartPole-v0. We obtain the environment Cart-Pole in the library gym, and it consists of simulating a pole attached to a cart as an unstable system. The goal is to move the cart left and right to keep the pole from falling.

There are two four dimensions for the input (state): cart position, cart velocity, pole angle, and pole velocity at the tip. For the output, there are two, the probability of action left, and the probability of action left. We send these dimensions to our policy.

for epi in range (300): *# look for our episodes*

        state = env.reset() *# we clear the rewards and logs, as every episode must have its own training*

        for t in range(200): *# look for our timesteps*

            action = pi.act(state) *# the agent will observe the state and select an action*

            state, reward, done, \_ = env.step(action) *# getting the rewards and the state to follow*

            pi.rewards.append(reward) *#saving the rewards of current policy*

            env.render() *# show*

            if done: *# if the end condition is met*

                break *# finish current timestep*

Those lines are to execute the episodes from our reinforcement learning system. The agent and the environment are interacting and exchanging signals (state, action and reward; an experience). In the current algorithm there are 300 episodes and 200 time steps.

loss = train(pi, optimizer) *# train per episode*

        total\_reward = sum(pi.rewards) *# we save the rewards of current episode and policy*

        solved = total\_reward > 195.0 *# it will print solved when the rewards are more than 195. i*

        pi.onpolicy\_reset() *# clear memory after training*

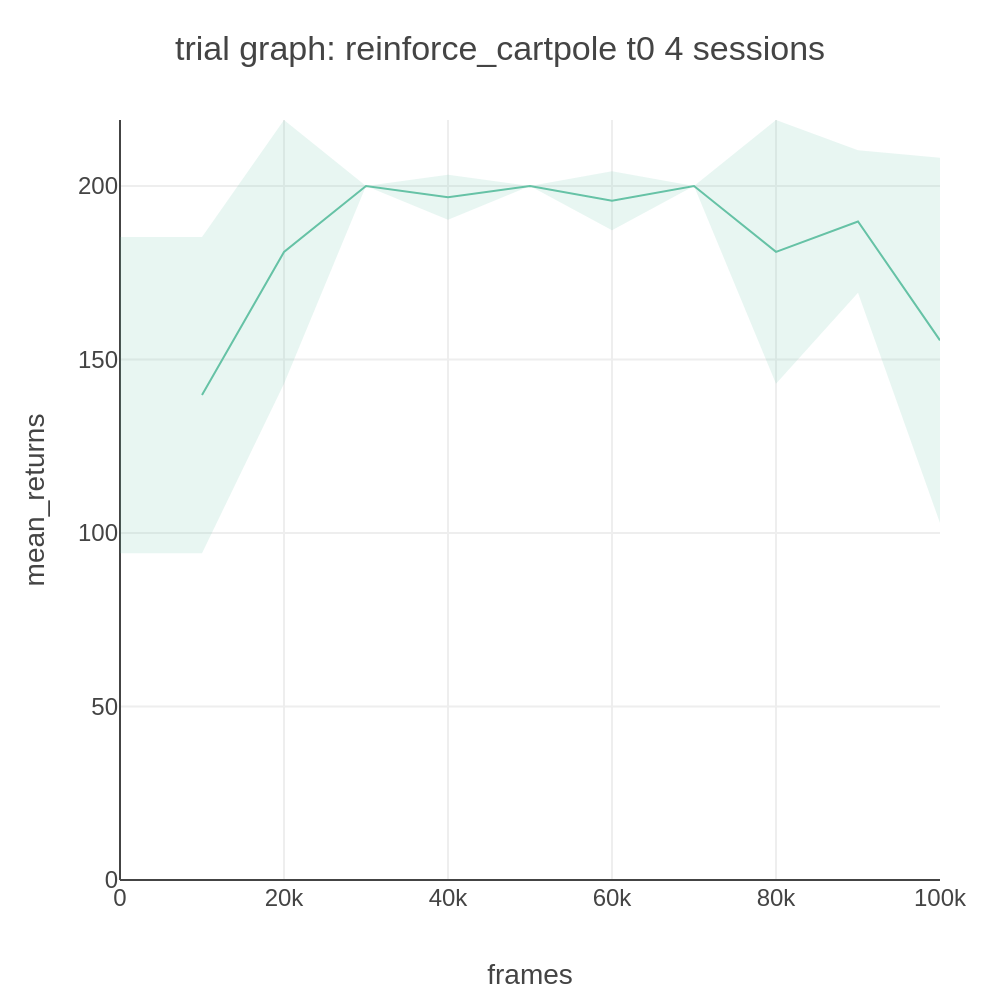
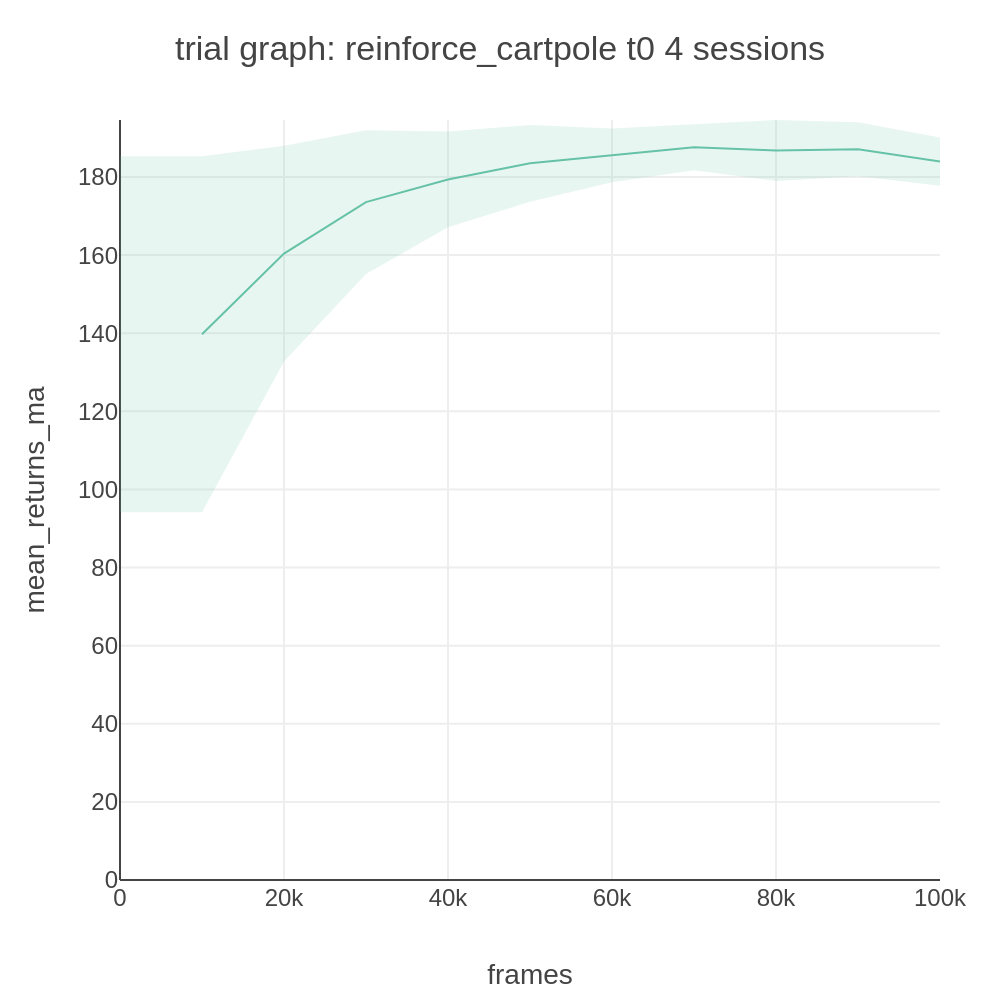
        print(f'Episode {epi}, loss: {loss}, \ *# printing results*

        total\_reward: {total\_reward}, solved: {solved}') *# printing results*

if \_\_name\_\_ == '\_\_main\_\_': *# starting command for python*

    main() *# execute main method*

By training, we obtain the loss per each episode. We save the sum of the total rewards from the episode and, if the total reward is more than our threeshold, we mark the episode as solved. After resetting the policy to prepare for the next episode, we print our results. THe last lines of code are python commands to run our code.

Figure 2.2

The graphs were obtained from SML Lab, and plotted by averaging four repeated sessions. On the left, we can see that the vertical axis shows the total rewards (denoted by mean\_returns) and the horizontal axis shows the total training frames. On the right side, we can observe the smoothed version where a window of 100 evaluation checkpoints is used. The code used to obtain this is the following:

conda activate lab

python run\_lab.py slm\_lab/spec/benchmark/reinforce/reinforce\_cartpole.json reinforce\_cartople train

Figure 2.3

In the following experiment, we will search over different gamma values. Gamma helps us to decide our learning rate by modifying the trajectoy, as seen in previos code: future\_ret = pi.rewards[t] + gamma \* future\_re. Using the following code:

conda activate lab

python run\_lab.py slm\_lab/spec/benchmark/reinforce/reinforce\_cartpole.json reinforce\_cartople search

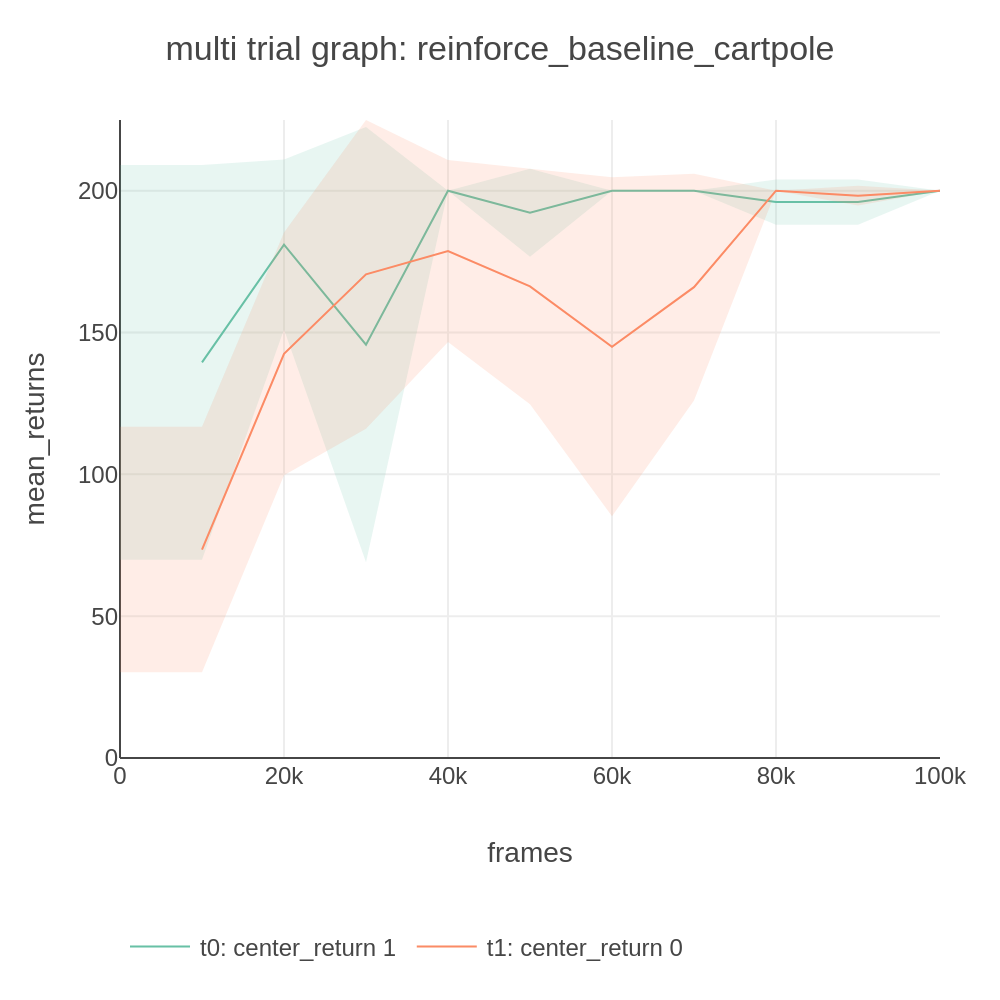
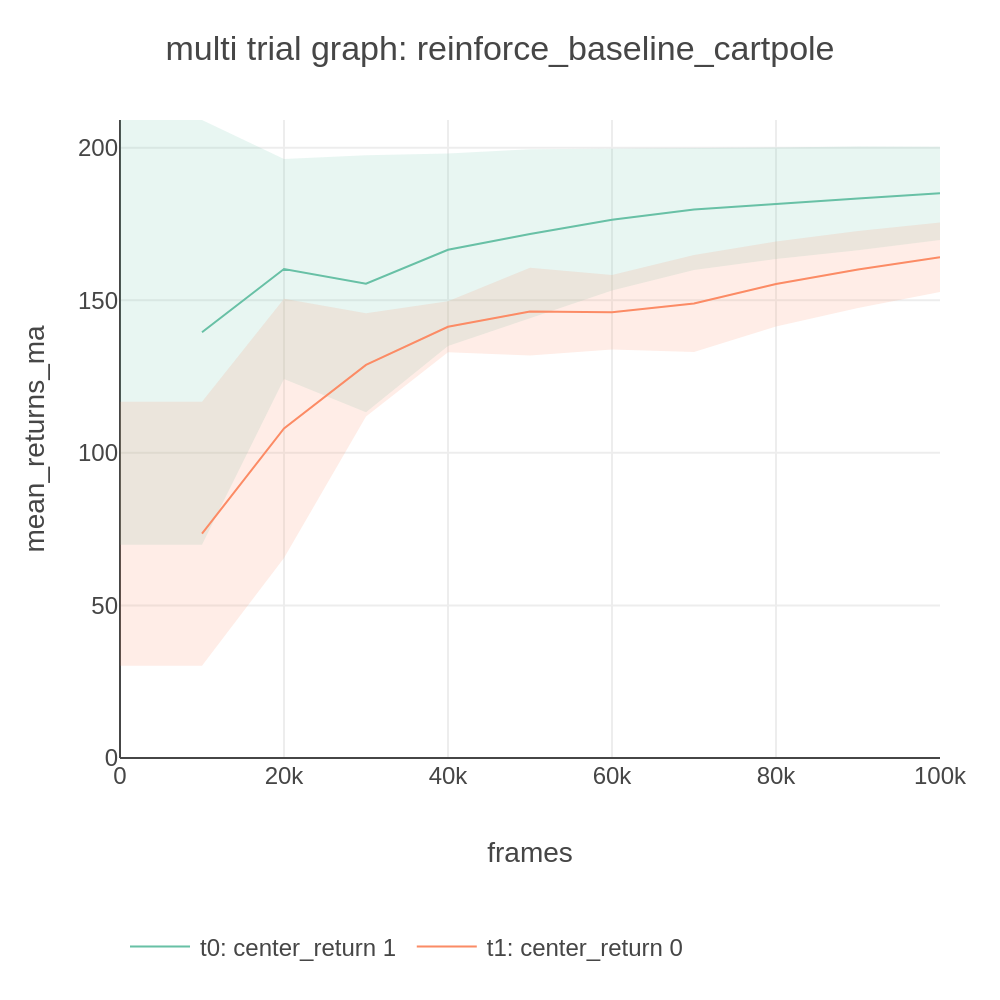
We will be using the following the option in the settings that uses “gamma\_grid\_search”, and going through different values of gamma to compare their behavior. In the case of CartPole, we can see how a lower gamma performs worse, while a higher gamma performs better. So in the case of CartPole, we want our gamma to be as closer to 0.9999 as possible.   


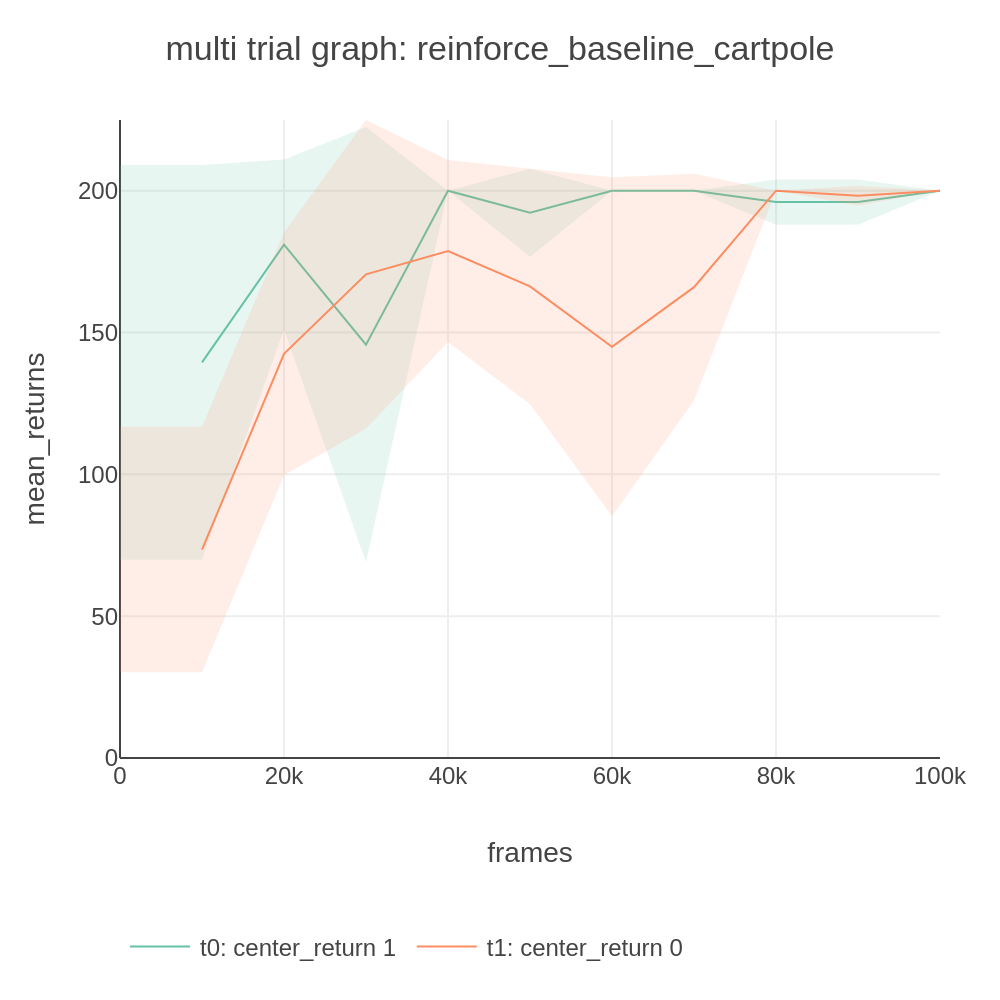
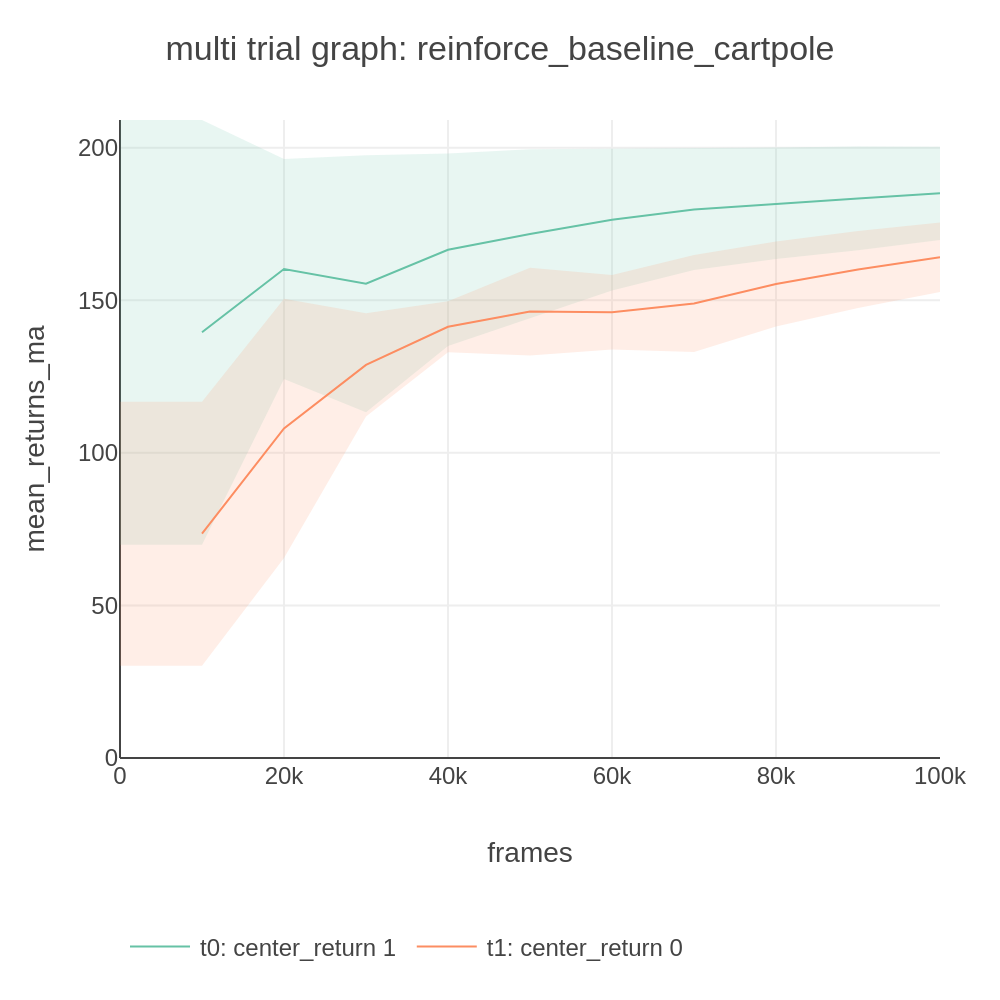
Figure 2.4 

Figure 2.4 graphs use a baseline. Baseline helps to reduce the variance of the Monte Carlo policy gradient estimate. As we can see compared to previous graphs, the performance is improved, as the rewards more rapidly get to the high levels with training and stay constant.

For this graph, in the settings we added a search spec to perform a grid search by toggling center\_return onf and off.

...

"search": {

"agent": [

{

"algorithm": {

"center\_return\_grid\_search": [true, false]

}

}

]

}

Then we run the code to execute it:

conda activate lab

python run\_lab.py slm\_lab/spec/benchmark/reinforce/reinforce\_cartpole.json reinforce\_baseline\_cartpole search