Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Avance Tarea 3

Seminario de Robótica y Sistemas Autónomos

Alumno: Sebastián Tinoco
Profesor: Javier Ruiz del Solar
Auxiliar: Francisco Leiva
Ayudante: Rodrigo Salas
Fecha de entrega: 8 de mayo de 2023
Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Introducción

El aprendizaje reforzado es una rama de la inteligencia artificial que se enfoca en enseñar a las máquinas a tomar decisiones óptimas basadas en la retroalimentación que reciben del entorno en el que se desenvuelven. Este tipo de aprendizaje ha demostrado ser muy útil en diversas aplicaciones de la vida real, como por ejemplo en la robótica, los videojuegos, la publicidad personalizada, la toma de decisiones en finanzas y la automatización industrial.

Con tal de dar continuidad al aprendizaje y entendimiento de esta disciplina, el presente trabajo tiene por objetivo implementar el algoritmo *Policy Gradients* y de esta forma obtener una política óptima en la resolución de problemas de control clásicos como "CartPole" y "Pendulum".

Para cumplir esto de forma efectiva, se divide el trabajo en diferentes apartados. La sección 1 introduce de forma general el objetivo del informe y da una idea de lo que se espera en este. La sección 2 contiene la formulación y programación de los métodos necesarios para implementar el algoritmo *Policy Gradients*, en conjunto con los primeros resultados de haber aplicado este algoritmo para obtener una política óptima en los ambientes "CartPole" y "Pendulum". Finalmente, la sección 3 finaliza el documento con los principales hallazgos del trabajo realizado, denotando los principales aprendizajes y dificultades.

2. Parte I

2.1. Pregunta 1: Parametrización de la política

2.1.1. Policy

La clase `Policy` puede ser encontrada en el anexo A.1.

2.1.2. select_action

La función `select_action` puede ser encontrada en el anexo A.2.

2.2. Pregunta 2: Muestreo de trayectorias

2.2.1. perform_single_rollout

La función `perform_single_rollout` puede ser encontrada en el anexo A.3.

2.2.2. sample_rollouts

La función `sample_rollouts` puede ser encontrada en el anexo A.4

2.2.3. Testing

Para testear la función `perform_single_rollout`, se realizaron pruebas usando los ambientes "CartPole" y "Pendulum". En concreto, se probó la función en los siguientes aspectos:

- Que la dimensión del arreglo de observaciones sea igual a $(time\_steps, nb\_obs)$
- Que la dimensión del arreglo de acciones sea igual a $(time\_steps, dim\_actions)$ en caso de que el ambiente tenga un espacio de acciones continuo, y $(time\_steps,)$ de tener un espacio de acciones discreto.
- Que la dimensión del arreglo de recompensas sea igual a $(time\_steps,)$

donde *time_steps* representa el número total de steps ejecutado en la trayectoria, *nb_obs* equivale a la dimensión de las observaciones emitidas por el ambiente y *dim_actions* es la dimensión de acciones.

Del mismo modo, se probó la función `sample_rollouts` en los ambientes "CartPole" y "Pendulum", comprobando que cada muestra tenga 3 elementos (observaciones, acciones, recompensas) y que su ejecución siempre retorne un batch de largo mayor o igual a *min_batch_steps*.

El código para ejecutar estas pruebas puede ser encontrado en las últimas líneas de los anexos A.3 y A.4, mientras que la ejecución de estas pruebas se realizó mediante el anexo A.7. Después de haber probado las funciones en 10 iteraciones distintas para cada ambiente, se concluye la implementación exitosa de las funciones `perform_single_rollout` y `sample_rollouts`.

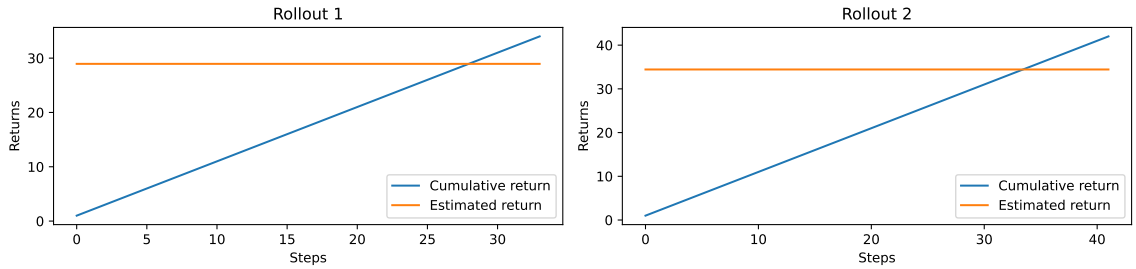
2.3. Pregunta 3: Estimación de retornos

2.3.1. estimate_returns

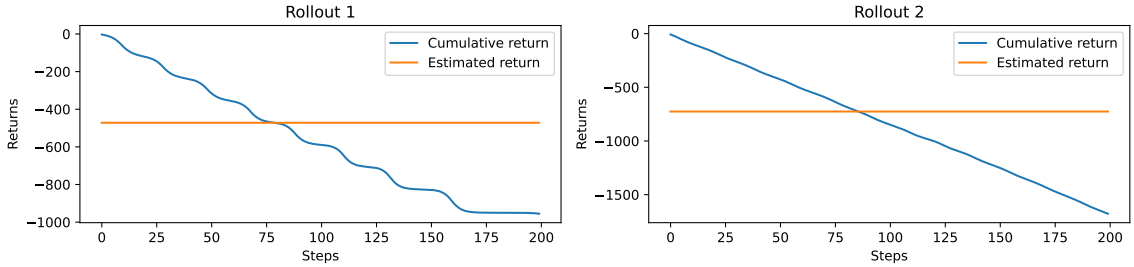
La función `estimate_returns` puede ser encontrada en el anexo A.5, la cual también se apoya de la función `discount_rewards` del anexo A.6.

2.3.2. Testing

Habiendo programado la función `estimate returns`, se procede a testear su funcionamiento en los ambientes "CartPole" y "Pendulum". En específico, para cada ambiente se muestrean dos trayectorias, para luego usar la función programada para estimar los retornos correspondientes. La Figura 1 contiene los resultados del ejercicio anterior.



(a) "CartPole"



(b) "Pendulum"

Figura 1: Test de *estimate_returns*

Para analizar la Figura anterior, es importante recordar que los retornos de cada trayectoria vienen determinados por:

$$\sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t) \tag{1}$$

donde $r(s_t, a_t)$ representa la recompensa de haber tomado la acción a_t en el estado s_t, γ alude a la tasa de descuento intertemporal (en este experimento se asume $\gamma = 0.99$) y el subíndice t hace referencia al *timestep* de la trayectoria. De esta manera, es posible interpretar la ecuación 1 como la suma descontada de las recompensas obtenidas a lo largo de toda la trayectoria, donde las recompensas más cercanas a $t = 1$ obtienen una mayor importancia en el resultado de la sumatoria.

Considerando la ecuación anterior y en vista de los resultados de la Figura 1, es posible observar

como el retorno estimado es una constante que es siempre menor a la suma total de recompensas sin descuento intertemporal, lo que se evidencia cuando la recta de retorno estimado "corta" la curva de recompensas acumuladas. Otro punto interesante a notar es que la "altura" en donde la recta de retorno estimado "corta" la curva de recompensas acumuladas está inversamente relacionada con el largo de la trayectoria T, donde una menor magnitud de T implica que el retorno estimado se sitúe en un valor más cercano a la suma total de recompensas descontadas. Lo anterior es consecuencia directa de que en casos donde la trayectoria tiene un largo reducido, el factor de descuento γ tiene un impacto reducido en las recompensas futuras.

2.4. Pregunta 4: Policy gradients

2.4.1. update

La función `update` puede ser encontrada en el anexo A.9.

2.5. Pregunta 5: Reducción de varianza

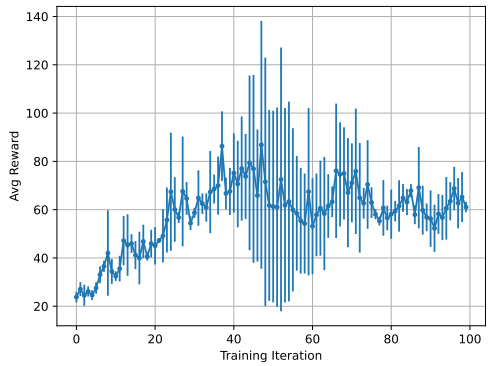
2.5.1. estimate_returns

La extensión de la función `estimate_returns` para implementar tanto *reward to go* como el uso de *baseline* puede ser encontrada en el anexo A.5.

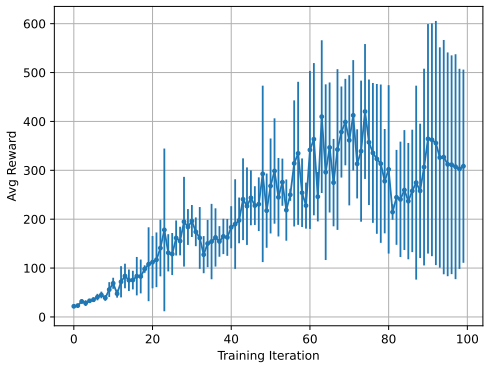
2.6. Pregunta 6: Evaluación del algoritmo

2.6.1. Experimentos

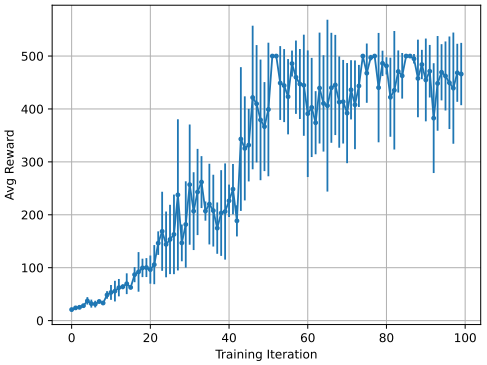
Habiendo programado el algoritmo, es posible ahora generar experimentos y así entender el impacto de técnicas e hiperparámetros en el desempeño del agente en los ambientes "CartPole" y "Pendulum". En específico, se evaluará el impacto del uso de *baseline*, *reward to go* y *batch size*, tomando como base los parámetros $lr = 0.005$ y $\gamma = 0.99$. Para conseguir resultados representativos, se busca evaluar las diferentes combinaciones posibles, repitiendo 3 veces cada experimento para obtener una mayor robustez en los resultados. Por último, se grafica la recompensa promedio usando el código A.10. Las Figuras 2, 3 y 4 presentan los resultados de este ejercicio.



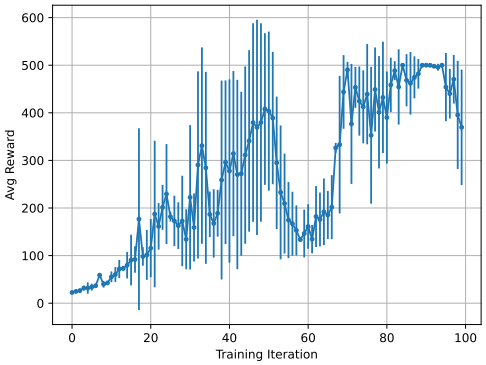
(a) *use_baseline* = False, *reward_to_go* = False



(b) *use_baseline* = False, *reward_to_go* = True

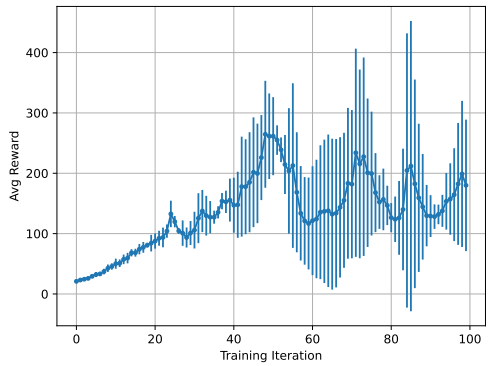


(c) *use_baseline* = True, *reward_to_go* = False

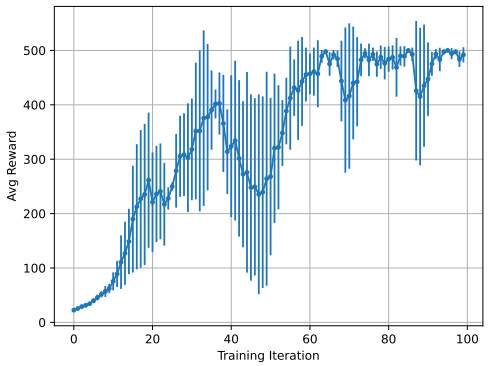


(d) *use_baseline* = True, *reward_to_go* = True

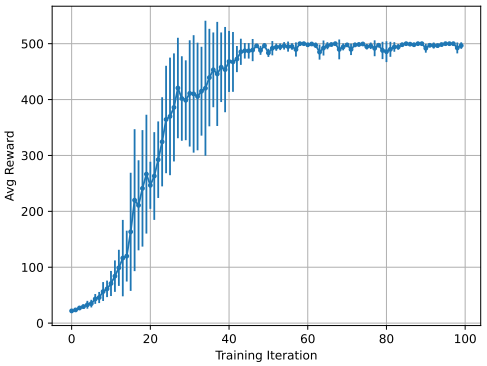
Figura 2: "CartPole" con *batch_size* = 500 y *training_iterations* = 100



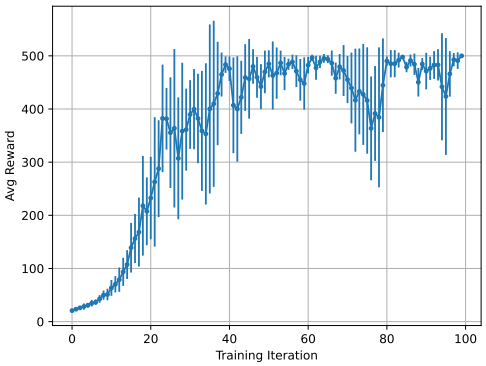
(a) *use_baseline* = False, *reward_to_go* = False



(b) *use_baseline* = False, *reward_to_go* = True

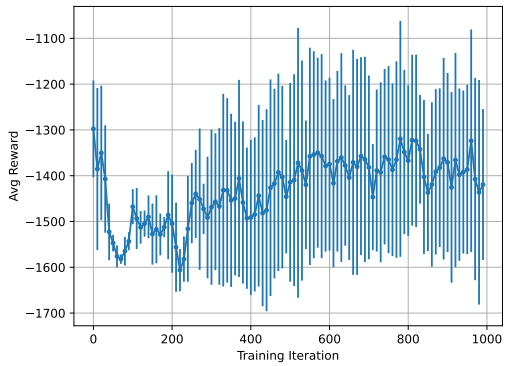


(c) *use_baseline* = True, *reward_to_go* = False

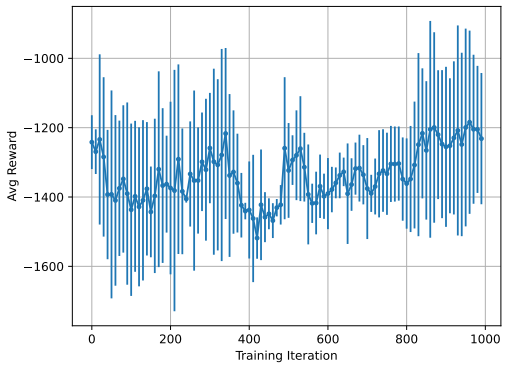


(d) *use_baseline* = True, *reward_to_go* = True

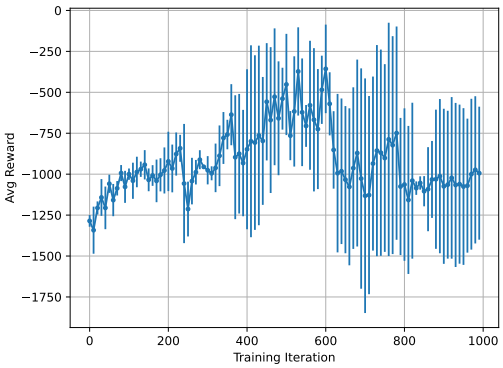
Figura 3: "CartPole" con *batch_size* = 5000 y *training_iterations* = 100



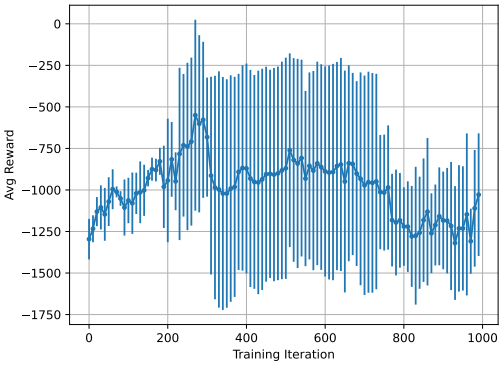
(a) *use_baseline* = False, *reward_to_go* = False



(b) *use_baseline* = False, *reward_to_go* = True



(c) *use_baseline* = True, *reward_to_go* = False



(d) *use_baseline* = True, *reward_to_go* = True

Figura 4: "Pendulum" con *batch_size* = 5000 y *training_iterations* = 1000

2.6.2. Análisis de resultados

A partir de la Figura 2, es posible observar como el uso de *reward to go* impacta de manera positiva en la recompensa promedio obtenida por el agente, a pesar de mantener los niveles de varianza en el entrenamiento. En paralelo, se observa como el uso de *baseline* genera el mismo efecto, pero siendo este levemente superior al obtener un mayor nivel de recompensa y con un menor nivel de varianza en el entrenamiento. Además, se observa como el uso de ambas técnicas replica el comportamiento anterior, alcanzando un mayor nivel de recompensa y con un nivel de varianza menor al caso inicial.

En cuanto a los resultados de la Figura 3, se observa el mismo comportamiento analizado en la Figura 2: la inclusión de las técnicas *reward to go* y el uso de *baseline* impactan de manera positiva sobre la recompensa promedio obtenida por el agente, donde se resalta el hecho de un menor nivel de varianza cuando se usa *baseline* en comparación a los casos con *reward to go*. El análisis se repite para cuando ambas técnicas se combinan, obteniendo recompensas promedio significativamente superiores al caso inicial y con un menor nivel de varianza. Además, se destaca el hecho de que aumentar el tamaño del *batch* de entrenamiento de 500 a 5000 genera un impacto significativo sobre el entrenamiento, obteniendo mejoras en las recompensas promedio obtenidas por el agente en todas las combinaciones.

Finalmente y analizando los resultados de la Figura 4, se observa que el uso de *reward to go* impacta de manera positiva en las recompensas promedio obtenidas por el agente, reduciendo levemente el nivel de varianza en el entrenamiento. De manera similar, la utilización de *baselines* tiene un efecto positivo en las recompensas promedio obtenidas por el agente, reduciendo de igual manera el nivel de varianza de entrenamiento. Es interesante notar que, a diferencia de los ambientes con espacio de acciones discretas, la reducción de varianza en el caso del *baseline* es bastante menor, lo que evidencia el mayor nivel de complejidad al interactuar con ambientes con espacio de acciones continuas. Además, se observa que la utilización simultánea de las técnicas *baseline* y *reward to go* genera un impacto positivo en las recompensas promedio obtenidas por el agente, pero apenas diferenciable del caso sin *reward to go* (ver Figura 4.c).

Por último, es importante recordar que aunque se repitió 3 veces cada experimento para una mayor robustez, el número de repeticiones sigue siendo extremadamente bajo para realizar conclusiones con fundamento estadístico, por lo que es posible que con un mayor número de experimentos algunas conclusiones cambien. Esto es particularmente importante en el entrenamiento de *Policy Gradients*, donde la "semilla" del experimento tiene un impacto mayor en el entrenamiento del agente. Lo anterior queda ilustrado con la Figura 5 donde el agente fue entrenado sin el uso de *baseline* ni *reward to go* para resolver "Pendulum", y sin embargo, obtiene resultados significativamente mejores que los presentados en la Figura 4.a.

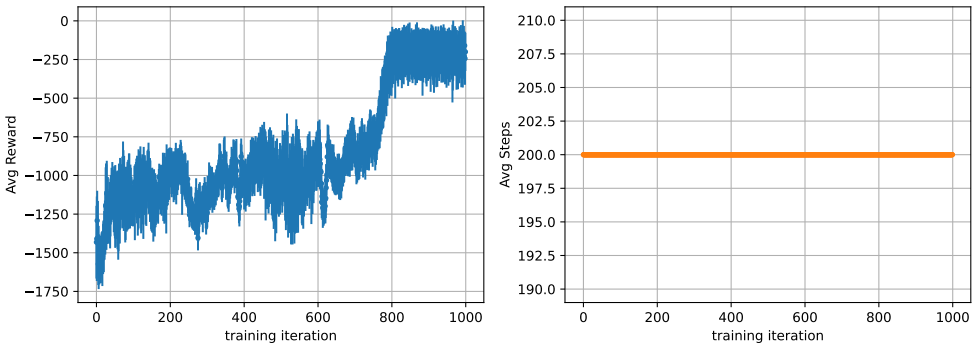


Figura 5: Ejemplo de entrenamiento con “buena” semilla

3. Conclusiones

El aprendizaje reforzado ha demostrado ser una parte fundamental en diversas aplicaciones de la vida real, como robótica, videojuegos, publicidad personalizada, toma de decisiones en finanzas y automatización industrial. En el trabajo expuesto fue posible dilucidar el diseño y entrenamiento del algoritmo *Policy Gradients* para obtener una política óptima en los ambientes "CartPole" y "Pendulum".

En la sección 1 se da una explicación preliminar del objetivo del informe y la metodología a implementar. En la sección 2 se programan diferentes métodos y funcionalidades para implementar el algoritmo *Policy Gradients*. Por último, se hace uso del algoritmo programado para encontrar una política óptima y entender el impacto de los hiperparámetros del algoritmo.

Tras analizar los resultados presentados en este informe, se puede concluir que el algoritmo *Policy Gradients* alcanza resultados aceptables para desempeñarse en los ambientes "CartPole" y "Pendulum". Este resultado coincide con la teoría y demuestra que utilizar algoritmos *Policy-Based* es una estrategia adecuada para determinar una política óptima en los Procesos de Decisión de Markov, especialmente cuando se tiene un espacio de acciones continuo y no se cuenta con información completa sobre la dinámica del entorno. Adicionalmente, los resultados apoyan la teoría que implementar técnicas como *baseline* y *rewards to go* apoyan en la estabilización del gradiente y la reducción de la varianza de entrenamiento, logrando de esta manera mejoras significativas en la política obtenida por el agente.

En consideración del ajuste del algoritmo en donde se logra generar una política convergente para resolver los ambientes antes nombrados, se concluye la obtención de resultados aceptables en la resolución del problema.

A pesar de haber implementado con éxito el algoritmo, el trabajo expuesto no estuvo exento de dificultades. La principal dificultad estuvo en el entendimiento del algoritmo, y de qué forma hace uso de las técnicas *baseline* y *reward to go*. Otra complejidad encontrada fue la alta varianza en los resultados a través de las distintas repeticiones de un mismo experimento, lo que complejizó el análisis del impacto de los diferentes hiperparámetros.

En retrospectiva del trabajo realizado, se consolida el aprendizaje de el diseño y ajuste del algoritmo *Policy Gradients* mediante código Python y las librerías Pytorch, Numpy y Gym.

Finalmente, queda como trabajo futuro profundizar el impacto de los hiperparámetros en la política obtenida, así como de buscar también la mejor combinación de hiperparámetros para resolver el problema de manera con mayor eficiencia.

Anexo A. Código implementado

Código A.1: Policy

```

class Policy(nn.Module):

    def __init__(self, dim_states, dim_actions, continuous_control):
        super(Policy, self).__init__()
        # MLP, fully connected layers, ReLU activations, linear ouput activation
        # dim_states -> 64 -> 64 -> dim_actions

        self._continuous_control = continuous_control

        self.fc1 = nn.Linear(dim_states, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, dim_actions)

        if continuous_control:
            # trainable parameter
            self._log_std = torch.zeros(dim_actions)
            self._log_std = nn.Parameter(self._log_std)

    def forward(self, input):

        input = F.relu(self.fc1(input))
        input = F.relu(self.fc2(input))

        return self.fc3(input) # logits or means
```

Código A.2: select_action

```

def select_action(self, observation):
    return self._select_action(observation)


def _select_action_discrete(self, observation):
    # sample from categorical distribution

    observation = torch.from_numpy(observation).float().unsqueeze(0).to(device)

    with torch.no_grad():
        logits = self._policy(observation).to(device)
        distr = Categorical(logits = logits)

    action = distr.sample().item()

    return action


def _select_action_continuous(self, observation):
    # sample from normal distribution
    # use the log std trainable parameter

    observation = torch.from_numpy(observation).float().unsqueeze(0).to(device)

    with torch.no_grad():
        mean = self._policy(observation).to(device)
```

```
        std = torch.exp(self.__policy.__log_std).to(device)
        distr = Normal(mean, std)

    action = distr.sample().squeeze(0).cpu().numpy()

    return action
```

Código A.3: perform_single_rollout

```

def perform_single_rollout(env, agent, episode_nb, render=False):

    # Modify this function to return a tuple of numpy arrays containing (observations, actions, rewards).
    # (np.array(obs), np.array(acs), np.array(rws))
    # np.array(obs) -> shape: (time_steps, nb_obs)
    # np.array(acs) -> shape: (time_steps, nb_acs) if actions are continuous, (time_steps,) if actions are discrete
    # np.array(rws) -> shape: (time_steps,)

    dim_states = env.observation_space.shape[0]
    continuous_control = isinstance(env.action_space, gym.spaces.Box)
    dim_actions = env.action_space.shape[0] if continuous_control else env.action_space.n

    ob_t = env.reset()

    done = False
    episode_reward = 0
    nb_steps = 0

    obs, acs, rws = [], [], []

    while not done:

        if render:
            env.render()
            time.sleep(1. / 60)

        action = agent.select_action(ob_t)

        ob_t1, reward, done, _ = env.step(action)

        obs.append(ob_t)
        acs.append(action)
        rws.append(reward)

        ob_t = np.squeeze(ob_t1) # <-- may not be needed depending on gym version
        episode_reward += reward

        nb_steps += 1

        if done:

            obs = np.array(obs)
            acs = np.array(acs)
            rws = np.array(rws)

            # Testing
            assert obs.shape == (nb_steps, dim_states), 'shape of np.array(obs) is not (time_steps, nb_obs)'
            if continuous_control:
                assert acs.shape == (nb_steps, dim_actions), 'shape of np.array(acs) is not (time_steps, nb_acs)'
            else:
```

```
            assert acs.shape == (nb_steps,), 'shape of np.array(acs) is not (time_steps,)'
        assert rws.shape == (nb_steps,), 'shape of np.array(rws) is not (time_steps,)'

        return obs, acs, rws
```

Código A.4: sample_rollouts

```

def sample_rollouts(env, agent, training_iter, min_batch_steps):

    sampled_rollouts = []
    total_nb_steps = 0
    episode_nb = 0

    while total_nb_steps < min_batch_steps:

        episode_nb += 1
        #render = training_iter%10 == 0 and len(sampled_rollouts) == 0 # Change training_iter%10 to any number you want
        render = False

        # Use perform_single_rollout to get data
        # Uncomment once perform_single_rollout works.
        # Return sampled_rollouts

        sample_rollout = perform_single_rollout(env, agent, episode_nb, render=render)
        total_nb_steps += len(sample_rollout[0])

        sampled_rollouts.append(sample_rollout)

    # Testing
    assert sum([len(rollout[0]) for rollout in sampled_rollouts]) >= min_batch_steps, 'number of total steps is not equal or greater than min_batch_steps'
    assert all([len(rollout) == 3 for rollout in sampled_rollouts]), 'dimension of sampled rollouts are not equal to 3: (obs, acs, rws)'

    return sampled_rollouts
```

Código A.5: estimate_returns

```

def estimate_returns(self, rollouts_rew):
    estimated_returns = []
    for rollout_rew in rollouts_rew:

        if self._use_reward_to_go:
            # only for part 2
            estimated_return = deque() # for efficiency
            n_steps = len(rollout_rew) # steps of rollout

            for t in range(n_steps)[::-1]:
                disc_return_t = estimated_return[0] if len(estimated_return) > 0 else 0
                estimated_return.appendleft(disc_return_t * self._gamma + rollout_rew[t]) # pi_t = pi_{t+1} + r_t

            estimated_return = list(estimated_return)

        else:
            #estimated_return = [sum(rollout_rew) * (self._gamma ** t) for t in range(len(rollout_rew))]
            estimated_return = [self._discount_rewards(rollout_rew)] * len(rollout_rew)

```

```
        estimated_returns = np.concatenate([estimated_returns, estimated_return])

    if self._use_baseline:
        # only for part 2
        average_return_baseline = np.mean(estimated_returns)
        # Use the baseline:
        estimated_returns -= average_return_baseline

    return np.array(estimated_returns, dtype=np.float32)
```

Código A.6: discount_rewards

```

def _discount_rewards(self, rewards: list):

    # discount future rewards using gamma
    return sum([rewards[t] * (self._gamma ** t) for t in range(len(rewards))])
```

Código A.7: test_methods

```

import gym
from policy_gradients import PolicyGradients
from train_agent import perform_single_rollout, sample_rollouts
from utils.plot_test import plot_test

if __name__ == '__main__':

    environments = ['CartPole-v1', 'Pendulum-v1']
    for environment in environments:

        for _ in range(10):

            env = gym.make(environment) # initiate environment
            dim_states = env.observation_space.shape[0] # compute state space
            continuous_control = isinstance(env.action_space, gym.spaces.Box) # True if action space is continuous, False if not
            dim_actions = env.action_space.shape[0] if continuous_control else env.action_space.n # compute action space

            # initiate agent
            policy_gradients_agent = PolicyGradients(dim_states=dim_states,
                                        dim_actions=dim_actions,
                                        continuous_control=continuous_control,
                                        lr = 5e-3,
                                        gamma = .99,
                                        )

            # test for perform_single_rollout
            single_rollout = perform_single_rollout(env = env, agent = policy_gradients_agent, episode_nb = 1)

            # test for sample_rollouts
            sampled_rollouts = sample_rollouts(env = env, agent = policy_gradients_agent, training_iter = 1,
        ↪ min_batch_steps = 250)
            sampled_rew = [sampled_rollouts[i][2] for i in range(len(sampled_rollouts))]

            estimated_returns = policy_gradients_agent.estimate_returns(sampled_rew)

            # test for estimated returns

            if _ == 0: # plot just for the first iteration

```

```
            sampled_rew = [list(perform_single_rollout(env = env, agent = policy_gradients_agent, episode_nb = 1)[2]) for
    ↪ _ in range(2)] # sample 2 rollouts, obtain rewards
            returns = policy_gradients_agent.estimate_returns(sampled_rew) # compute returns

            plot_test(sampled_rew, returns, environment = environment) # plot rewards vs returns
```

Código A.8: plot_test

```

import matplotlib.pyplot as plt

def plot_test(sampled_rew, returns, environment):

    fig, axes = plt.subplots(1, 2, figsize = (12, 3))

    rewards = sampled_rew[0]
    axes[0].plot([sum(rewards[:t+1]) for t in range(len(rewards))], label = 'Cumulative return')
    axes[0].plot(returns[:len(sampled_rew[0])], label = 'Estimated return')
    axes[0].set(xlabel = 'Steps', ylabel = 'Returns', title = 'Rollout 1')
    axes[0].legend()

    rewards = sampled_rew[1]
    axes[1].plot([sum(rewards[:t+1]) for t in range(len(rewards))], label = 'Cumulative return')
    axes[1].plot(returns[-len(sampled_rew[1]):], label = 'Estimated return')
    axes[1].set(xlabel = 'Steps', ylabel = 'Returns', title = 'Rollout 2')
    axes[1].legend()

    plt.tight_layout()
    plt.savefig(f'figures/test_experiments/returns_{environment}.pdf')
    plt.close()
```

Código A.9: update

```

def update(self, observation_batch, action_batch, advantage_batch):
    # update the policy here
    # you should use self._compute_loss

    loss = self._compute_loss(observation_batch, action_batch, advantage_batch)

    self._optimizer.zero_grad()
    loss.backward()
    self._optimizer.step()


def _compute_loss_discrete(self, observation_batch, action_batch, advantage_batch):
    # use negative logprobs * advantages

    observation_batch = torch.from_numpy(observation_batch).to(device)
    action_batch = torch.from_numpy(action_batch).to(device)

    logits = self._policy(observation_batch).to(device)
    distr = Categorical(logits = logits)

    log_probs = distr.log_prob(action_batch) # compute log_prob for each pair mean-action
    log_probs = log_probs.squeeze().to(device)

    advantage_batch = torch.from_numpy(advantage_batch).to(device)
    assert log_probs.shape == advantage_batch.shape

```

```
    loss = torch.multiply(-log_probs, advantage_batch)

    return torch.mean(loss)

def _compute_loss_continuous(self, observation_batch, action_batch, advantage_batch):
    # use negative logprobs * advantages

    observation_batch = torch.from_numpy(observation_batch).to(device)
    action_batch = torch.from_numpy(action_batch).to(device)

    mean = self._policy(observation_batch).to(device)
    std = torch.exp(self._policy._log_std).to(device)

    distr = Normal(mean, std) # n mean, 1 std

    log_probs = distr.log_prob(action_batch) # compute log_prob for each pair mean-action
    log_probs = log_probs.squeeze().to(device)

    advantage_batch = torch.from_numpy(advantage_batch).to(device)
    assert log_probs.shape == advantage_batch.shape

    loss = torch.multiply(-log_probs, advantage_batch)

    return torch.mean(loss)
```

Código A.10: Graficar experimentos

```

import matplotlib.pyplot as plt
import pandas as pd
import os

def plot_performance_metrics(tr_steps_vec, avg_reward_vec, std_reward_vec, exp_id, n_points = 100):

    plot_steps = len(tr_steps_vec) // n_points if len(tr_steps_vec) > 1 else 1

    tr_steps_vec = tr_steps_vec[::plot_steps]
    avg_reward_vec = avg_reward_vec[::plot_steps]
    std_reward_vec = std_reward_vec[::plot_steps]

    plt.errorbar(tr_steps_vec, avg_reward_vec, yerr = std_reward_vec, marker = '.', color = 'C0')
    plt.xlabel('Training Iteration')
    plt.ylabel('Avg Reward')
    plt.grid('on')
    plt.savefig(f'figures/agg_experiments/{exp_id}.pdf')
    plt.close()

def plot_experiment(exp_id):

    files = os.listdir('metrics')

    exp_files = [file for file in files if exp_id in file]

    exp_data = pd.DataFrame()

    for file in exp_files:

        file_data = pd.read_csv(f'metrics/{file}', sep = '\t')
        exp_data = pd.concat((exp_data, file_data), axis = 1)

```

```
    steps = exp_data['steps'].mean(axis = 1)
    avg_reward = exp_data['avg_reward'].mean(axis = 1)
    std_reward = exp_data['avg_reward'].std(axis = 1)

    plot_performance_metrics(steps, avg_reward, std_reward, exp_id)

if __name__ == '__main__':

    # experiment ids
    ids = ['CartPole_exp_11', 'CartPole_exp_21', 'CartPole_exp_31', 'CartPole_exp_41',
           'CartPole_exp_12', 'CartPole_exp_22', 'CartPole_exp_32', 'CartPole_exp_42',
           'Pendulum_exp_12', 'Pendulum_exp_22', 'Pendulum_exp_32', 'Pendulum_exp_42'
           ]

    # plot for each experiment
    for exp_id in ids:
        plot_experiment(exp_id)
```