Aleksy Stocki

Projekt “Bipedal Walker”

Spis treści:

Wstępny opis projektu

Drugi projekt z inteligencji obliczeniowej polega na przetestowaniu trzech dobranych przeze mnie algorytmów uczenia przez wzmacnianie na środowisku Gymnasium z dwunożnym biegającym robotem i zbadaniu jak dobrze radzą sobie z rozwiązywaniem problemu nauki chodzenia przy różnych hiperparametrach.

Postanowiłem wykorzystać trzy podobne zamysłowo algorytmy:

- Deep Q Network (DQN)
- Deep Deterministic Policy Gradient (DDPG)
- Twin Delayed DDPG (TD3)

Podczas pracy nad tym projektem nauczyłem się wielu cennych informacji na temat sposobu w jaki działają algorytmy Reinforcement Learning, a także napotkałem wiele ciekawych zagadnień i przeszkód do pokonania związanych z tą dziedziną.

DQN

Deep Q Network to najprostszy w swoim działaniu i skomplikowaniu ze wszystkich trzech algorytmów.

Jak wszystkie trzy wybrane algorytmy, łączy on ze sobą zamysł Q-Learning i nagradzania podjętych decyzji z sieciami neuronowymi, które są do zdobywania tych nagród dostosowywane i uczone oraz podejmuje decyzje na podstawie procesu decyzyjnego Markova, zależnie od poprzedniego stanu.

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

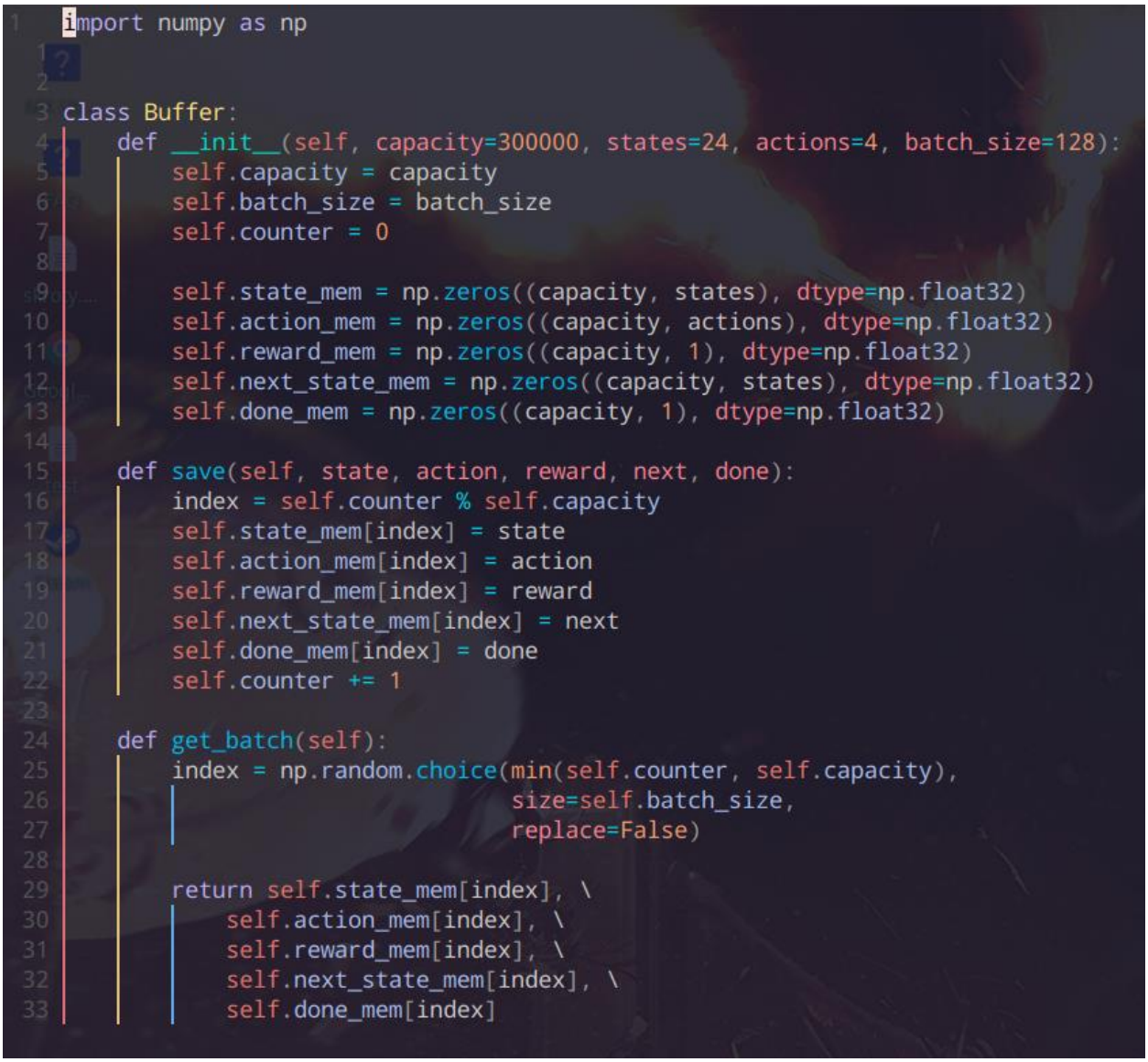
Every C steps reset $\hat{Q} = Q$

End For

End For

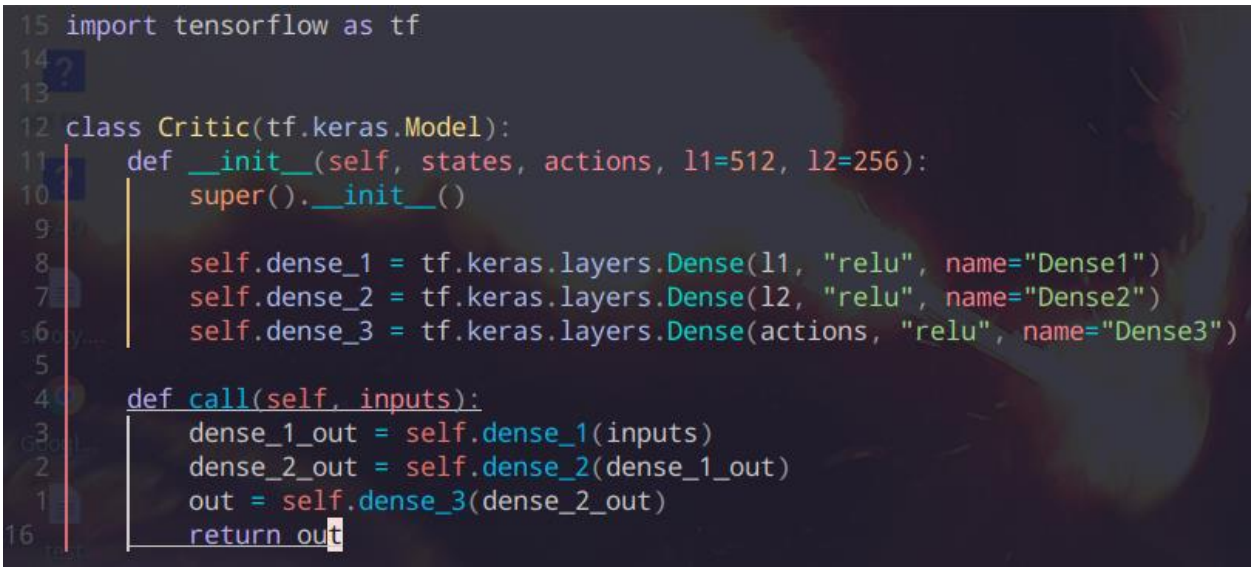
Zasada działania tego algorytmu, nieco różni się od dwóch kolejnych, ponieważ nie mamy tutaj żadnej ”pomocniczej” sieci, wszystko dzieje się na jednej, co z kolei prowadzi do pewnych ograniczeń, ponieważ ta jedna sieć musi na własną rękę zadbać o wykonywane czynności i ich jakość.

Cechą wspólną wszystkich trzech algorytmów jest bufor pamięci, który odpowiada za uczenie modelu sieci neuronowej na podstawie jej wspomnień z poprzednich podejść.



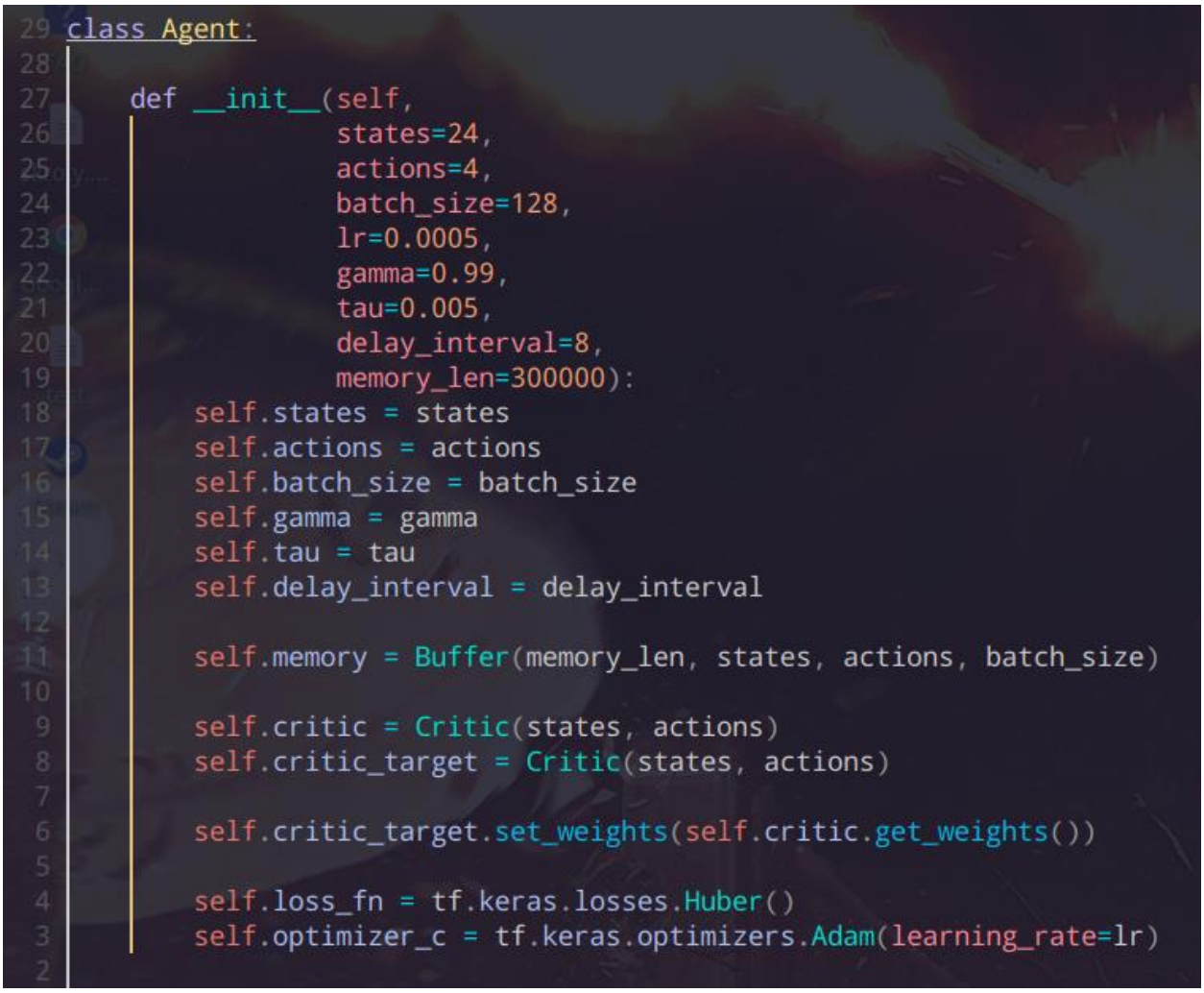
```python
import numpy as np


class Buffer:
    def __init__(self, capacity=300000, states=24, actions=4, batch_size=128):
        self.capacity = capacity
        self.batch_size = batch_size
        self.counter = 0

        self.state_mem = np.zeros((capacity, states), dtype=np.float32)
        self.action_mem = np.zeros((capacity, actions), dtype=np.float32)
        self.reward_mem = np.zeros((capacity, 1), dtype=np.float32)
        self.next_state_mem = np.zeros((capacity, states), dtype=np.float32)
        self.done_mem = np.zeros((capacity, 1), dtype=np.float32)

    def save(self, state, action, reward, next, done):
        index = self.counter % self.capacity
        self.state_mem[index] = state
        self.action_mem[index] = action
        self.reward_mem[index] = reward
        self.next_state_mem[index] = next
        self.done_mem[index] = done
        self.counter += 1

    def get_batch(self):
        index = np.random.choice(min(self.counter, self.capacity),
                                 size=self.batch_size,
                                 replace=False)

        return self.state_mem[index], \
            self.action_mem[index], \
            self.reward_mem[index], \
            self.next_state_mem[index], \
            self.done_mem[index]
```

Implementacja pamięci algorytmu jest z założenia prosta, ma ustalony rozmiar, po którego przekroczeniu nadpisywane są najstarsze (pierwsze) wspomnienia, a wspomnieniami jest zestaw (s a r s’), czyli aktualny stan, podjęta akcja, nagroda za akcję i kolejny stan. Opcjonalnie zapamiętywane jest jeszcze to, czy środowisko w danym momencie było zakończone (done), czy nie.



```python
import tensorflow as tf


class Critic(tf.keras.Model):
    def __init__(self, states, actions, l1=512, l2=256):
        super().__init__()

        self.dense_1 = tf.keras.layers.Dense(l1, "relu", name="Dense1")
        self.dense_2 = tf.keras.layers.Dense(l2, "relu", name="Dense2")
        self.dense_3 = tf.keras.layers.Dense(actions, "relu", name="Dense3")

    def call(self, inputs):
        dense_1_out = self.dense_1(inputs)
        dense_2_out = self.dense_2(dense_1_out)
        out = self.dense_3(dense_2_out)
        return out
```

Inicjalizacja sieci to szeroki i pozostawiony do interpretacji temat, sieć nie może być zbyt złożona, ani zbyt prosta w odniesieniu do rozwiązywanego problemu, w moim przypadku sieci o rozmiarach (400, 300), (512, 256), (256, 256) były wystarczająco optymalne i nie zaobserwowałem między nimi znaczących różnic w wydajności i jakości.



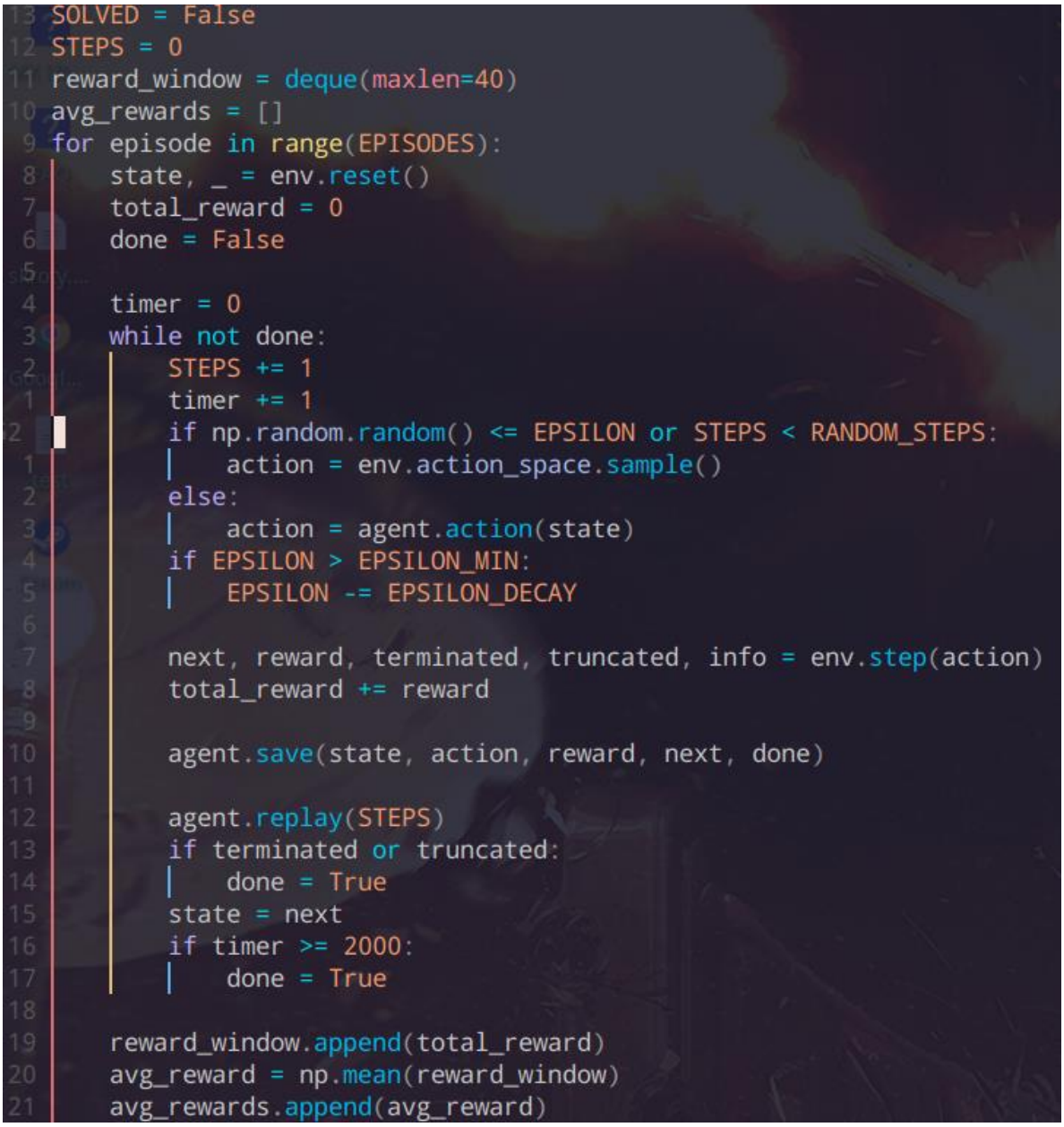
```
class Agent:

    def __init__(self,
                 states=24,
                 actions=4,
                 batch_size=128,
                 lr=0.0005,
                 gamma=0.99,
                 tau=0.005,
                 delay_interval=8,
                 memory_len=300000):
        self.states = states
        self.actions = actions
        self.batch_size = batch_size
        self.gamma = gamma
        self.tau = tau
        self.delay_interval = delay_interval

        self.memory = Buffer(memory_len, states, actions, batch_size)

        self.critic = Critic(states, actions)
        self.critic_target = Critic(states, actions)

        self.critic_target.set_weights(self.critic.get_weights())

        self.loss_fn = tf.keras.losses.Huber()
        self.optimizer_c = tf.keras.optimizers.Adam(learning_rate=lr)
```

Klasa Agent, to obiekt, w którym dzieje się cała “magia” naszej nauki, inicjalizuje ona sieci odpowiedzialne za naukę.

Sieć self.critic to sieć, która jest ciągle aktualizowana względem postępu w rozwoju, a sieć self.critic_target, to sieć “przechowująca” rozwiązanie.

Sieć target aktualizowana jest co jakiś odstęp czasowy i służy głównie zachowaniu poprzednich wag.

W prostych słowach, zamiast “Nie idzie mi, nauczę się znowu”, agent korzysta z “To rozwiązanie na razie wystarczy, jak znajdę jakieś lepsze, to je wezmę”.



```
SOLVED = False
STEPS = 0
reward_window = deque(maxlen=40)
avg_rewards = []
for episode in range(EPISODES):
    state, _ = env.reset()
    total_reward = 0
    done = False

    timer = 0
    while not done:
        STEPS += 1
        timer += 1
        if np.random.random() <= EPSILON or STEPS < RANDOM_STEPS:
            action = env.action_space.sample()
        else:
            action = agent.action(state)
        if EPSILON > EPSILON_MIN:
            EPSILON -= EPSILON_DECAY

        next, reward, terminated, truncated, info = env.step(action)
        total_reward += reward

        agent.save(state, action, reward, next, done)

        agent.replay(STEPS)
        if terminated or truncated:
            done = True
        state = next
        if timer >= 2000:
            done = True

    reward_window.append(total_reward)
    avg_reward = np.mean(reward_window)
    avg_rewards.append(avg_reward)
```

W pętli gry zapisany jest proces nauki agenta:

1. Dobór akcji, albo losowej, albo obliczonej przez naszą sieć
2. Zmniejszenie współczynnika epsilon (wraz z upływem czasu, dobierane będzie coraz mniej akcji losowych)
3. Zapisanie w pamięci wspomnienia z danego momentu
4. Nauka sieci neuronowej (Parametr steps służy do obliczenia, czy minęło wystarczająco iteracji do kolejnej aktualizacji wag w sieci)

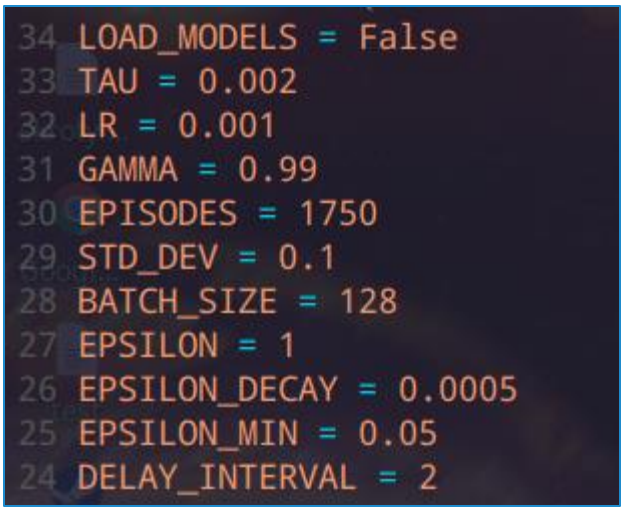
```
def replay(self, time):
    if self.memory.counter < self.batch_size:
        return
    states, actions, rewards, next_states, dones = self.memory.get_batch()
    self.train_critic(states, actions, rewards, next_states, dones)
    if self.memory.counter % self.delay_interval == 0:
        self.soft_update(self.critic, self.critic_target)

@tf.function
def train_critic(self, states, actions, rewards, next_states, dones):
    q_next = self.critic_target(next_states)
    y = rewards + (self.gamma * q_next) * (1 - dones)
    y = tf.reduce_max(y, axis=1)
    with tf.GradientTape() as tape:
        q = self.critic(states)
        q = tf.reduce_max(q, axis=1)
        loss = self.loss_fn(y, q)
    gradients = tape.gradient(loss, self.critic.trainable_variables)
    gradients = [tf.clip_by_value(gradient, -1, 1)
                 for gradient in gradients]
    self.optimizer_c.apply_gradients(
        zip(gradients, self.critic.trainable_variables))

def soft_update(self, evaluate_net, target_net):
    evaluate_weight = evaluate_net.get_weights()
    target_weight = target_net.get_weights()
    for i in range(len(evaluate_weight)):
        target_weight[i] = self.tau * evaluate_weight[i] + \
            (1 - self.tau) * target_weight[i]
    target_net.set_weights(target_weight)
```

Sieć co iterację uczona jest na próbce wspomnień, natomiast faktyczna aktualizacja sieci target następuje tylko co parę iteracji.

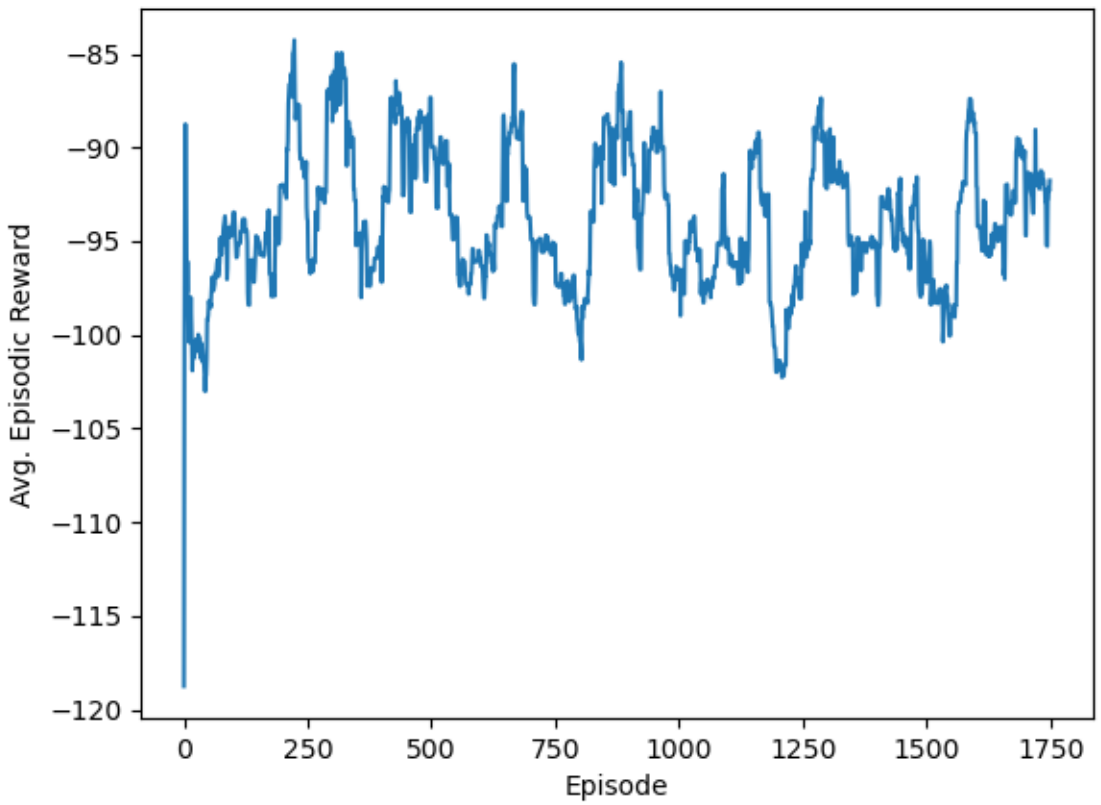
W trenowaniu modelu następuje zejście po gradiencie:

1. Obliczane są z próbki następne podjęte akcje (ich Q-Value z funkcji Bellmana), wartość kolejnych akcji przemnożonych przez współczynnik dyskontu.
2. Następnie obliczana jest Loss Function z Q-Value za poprzednie i przyszłe decyzje, oraz aktualizowany jest optimizer sieci.

Soft update to zwyczajnie przerzucenie wag z głównej sieci do sieci target, żeby aktualne zdolności modelu mogły być w nim przechowane. Są one aktualizowane w uzależnieniu od współczynnika tau, odpowiadającego za to, jak duże i gwałtowne są zmiany wag.



Moja implementacja wystartowała z takimi parametrami. Nie warto było przeprowadzać wielu badań i testów sprawności, ponieważ algorytm jest niedostosowany do ciągłych stanów żeby nauczyć się rozwiązywać środowisko dwunoga. Na wykresie nagród widać, że algorytm uczył się jakichś zasad, ale robił to zbyt słabo by osiągnąć wyższe wyniki.


−85
−90
−95
−100
−105
−110
−115
−120
Avg. Episodic Reward
0
250
500
750
1000
1250
1500
1750
Episode

DDPG

DDPG to opracowany przez Deepmind algorytm RL, który łączy w agencie cztery sieci neuronowe – aktora, krytyka oraz ich docelowe odpowiedniki.

DDPG różni się od DQN tym, że wyróżnione są dwie sieci:

Aktor odpowiadający za mapowanie stanu gry na kolejne akcje.

Krytyk oceniający (krytykujący) jak dobrze poradził sobie aktor.

Strona OpenAI mówi: DDPG można uznać za wersję DQN przystosowaną do stanów i akcji ciągłych.

Algorithm 1 DDPG algorithm

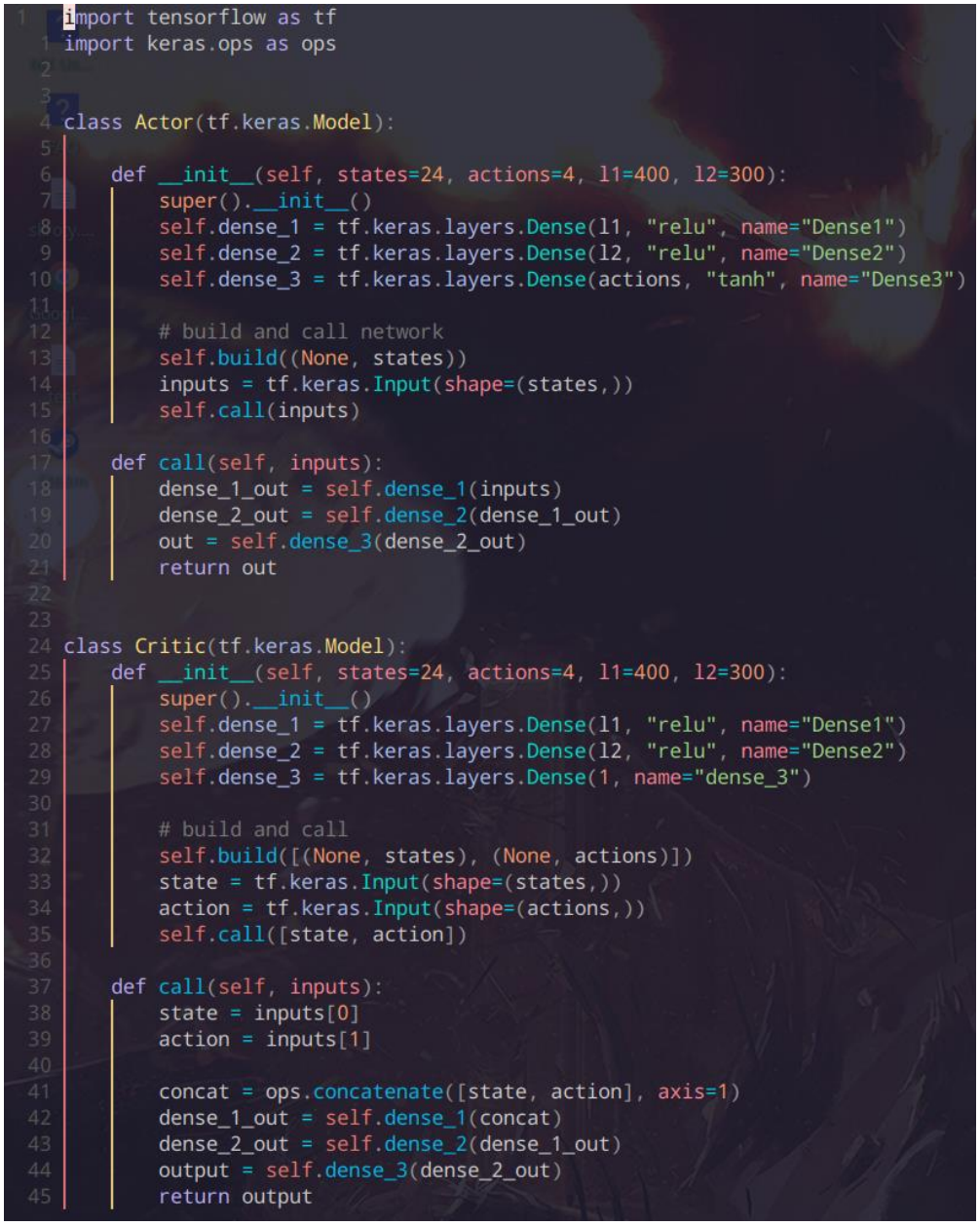
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

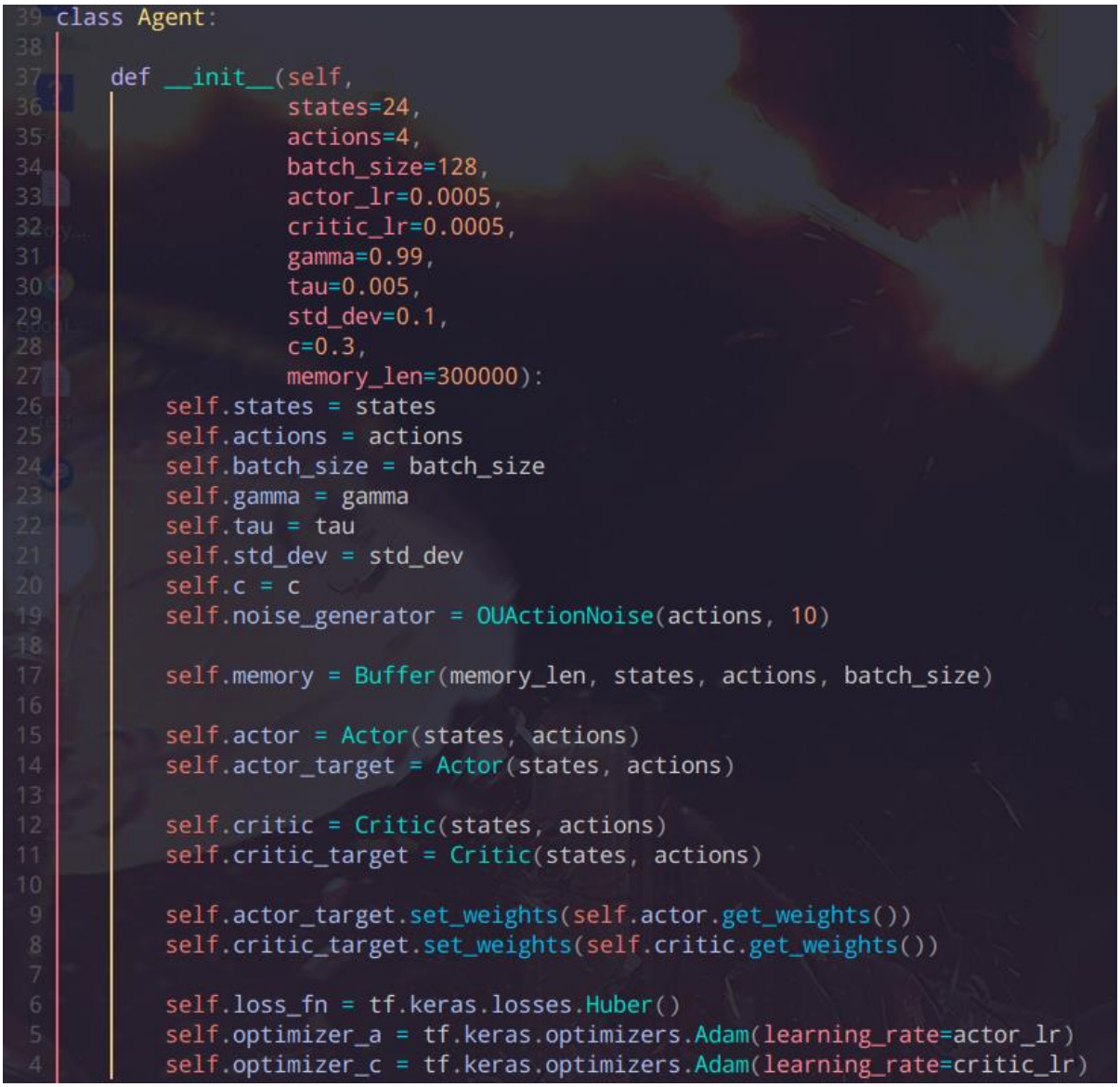
$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for



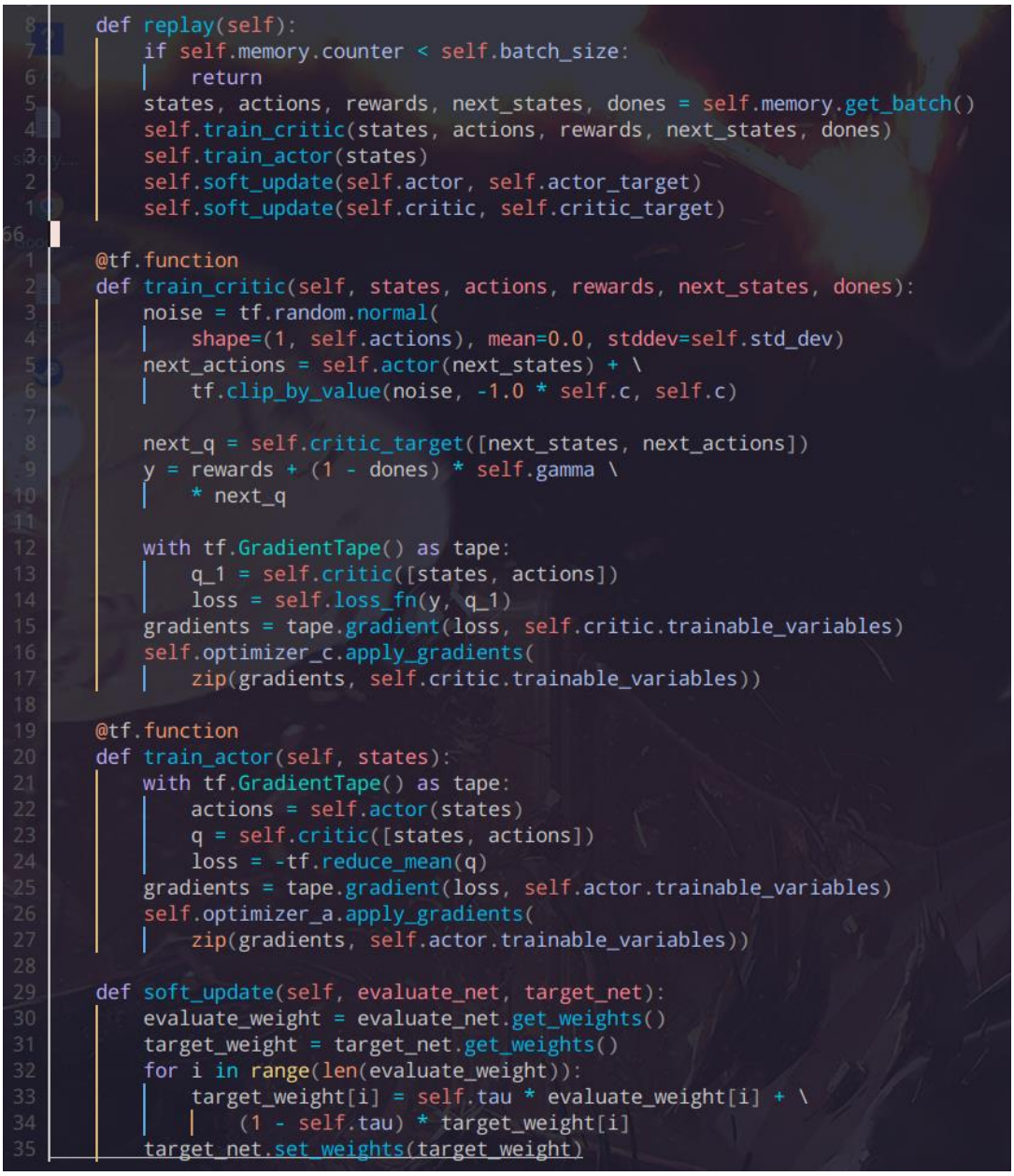
```python
import tensorflow as tf
import keras.ops as ops


class Actor(tf.keras.Model):

    def __init__(self, states=24, actions=4, l1=400, l2=300):
        super().__init__()
        self.dense_1 = tf.keras.layers.Dense(l1, "relu", name="Dense1")
        self.dense_2 = tf.keras.layers.Dense(l2, "relu", name="Dense2")
        self.dense_3 = tf.keras.layers.Dense(actions, "tanh", name="Dense3")

        # build and call network
        self.build((None, states))
        inputs = tf.keras.Input(shape=(states,))
        self.call(inputs)

    def call(self, inputs):
        dense_1_out = self.dense_1(inputs)
        dense_2_out = self.dense_2(dense_1_out)
        out = self.dense_3(dense_2_out)
        return out


class Critic(tf.keras.Model):
    def __init__(self, states=24, actions=4, l1=400, l2=300):
        super().__init__()
        self.dense_1 = tf.keras.layers.Dense(l1, "relu", name="Dense1")
        self.dense_2 = tf.keras.layers.Dense(l2, "relu", name="Dense2")
        self.dense_3 = tf.keras.layers.Dense(1, name="dense_3")

        # build and call
        self.build([(None, states), (None, actions)])
        state = tf.keras.Input(shape=(states,))
        action = tf.keras.Input(shape=(actions,))
        self.call([state, action])

    def call(self, inputs):
        state = inputs[0]
        action = inputs[1]

        concat = ops.concatenate([state, action], axis=1)
        dense_1_out = self.dense_1(concat)
        dense_2_out = self.dense_2(dense_1_out)
        output = self.dense_3(dense_2_out)
        return output
```

Sieć neuronowa krytyka mapuje zestaw stanów i podjętych akcji na wartości.



```python
class Agent:

    def __init__(self,
                 states=24,
                 actions=4,
                 batch_size=128,
                 actor_lr=0.0005,
                 critic_lr=0.0005,
                 gamma=0.99,
                 tau=0.005,
                 std_dev=0.1,
                 c=0.3,
                 memory_len=300000):
        self.states = states
        self.actions = actions
        self.batch_size = batch_size
        self.gamma = gamma
        self.tau = tau
        self.std_dev = std_dev
        self.c = c
        self.noise_generator = OUActionNoise(actions, 10)

        self.memory = Buffer(memory_len, states, actions, batch_size)

        self.actor = Actor(states, actions)
        self.actor_target = Actor(states, actions)

        self.critic = Critic(states, actions)
        self.critic_target = Critic(states, actions)

        self.actor_target.set_weights(self.actor.get_weights())
        self.critic_target.set_weights(self.critic.get_weights())

        self.loss_fn = tf.keras.losses.Huber()
        self.optimizer_a = tf.keras.optimizers.Adam(learning_rate=actor_lr)
        self.optimizer_c = tf.keras.optimizers.Adam(learning_rate=critic_lr)
```

Pojawia się w Agencie nowy parametr – c, jest to wartość, która służy do ograniczania dodawanych do podejmowanych przez aktora decyzji szumów.



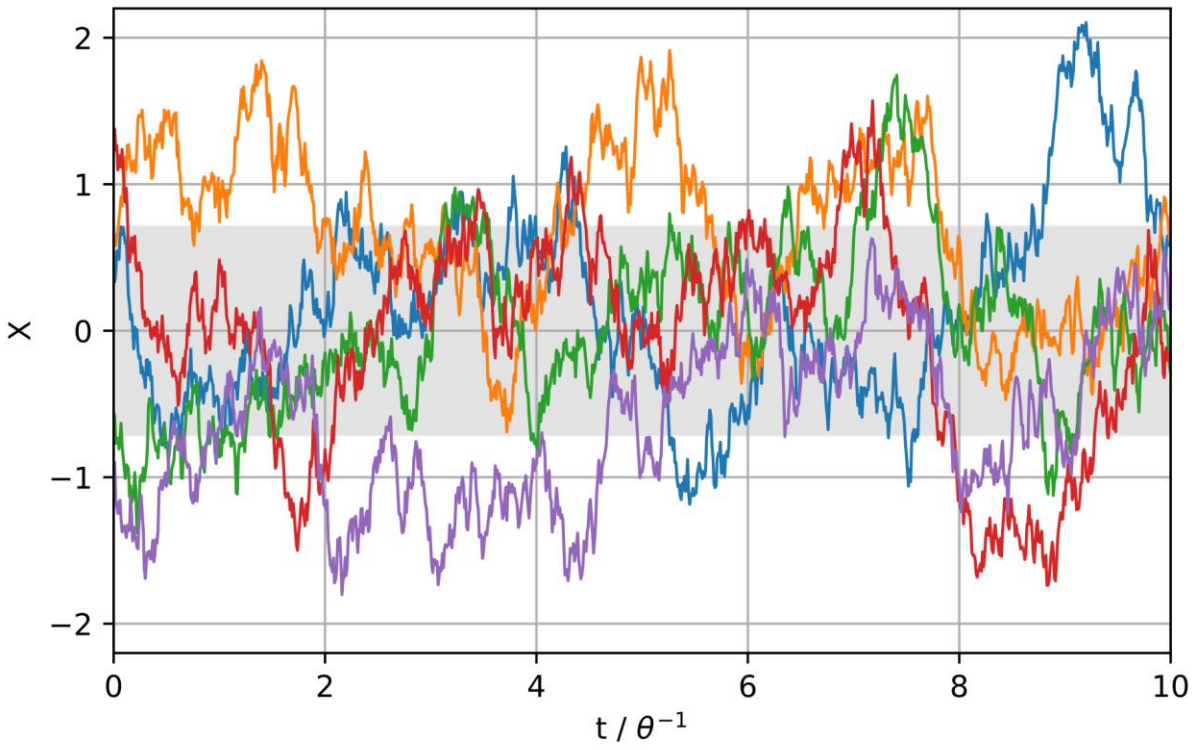
```
    def replay(self):
        if self.memory.counter < self.batch_size:
            return
        states, actions, rewards, next_states, dones = self.memory.get_batch()
        self.train_critic(states, actions, rewards, next_states, dones)
        self.train_actor(states)
        self.soft_update(self.actor, self.actor_target)
        self.soft_update(self.critic, self.critic_target)

    @tf.function
    def train_critic(self, states, actions, rewards, next_states, dones):
        noise = tf.random.normal(
            shape=(1, self.actions), mean=0.0, stddev=self.std_dev)
        next_actions = self.actor(next_states) + \
            tf.clip_by_value(noise, -1.0 * self.c, self.c)

        next_q = self.critic_target([next_states, next_actions])
        y = rewards + (1 - dones) * self.gamma \
            * next_q

        with tf.GradientTape() as tape:
            q_1 = self.critic([states, actions])
            loss = self.loss_fn(y, q_1)
        gradients = tape.gradient(loss, self.critic.trainable_variables)
        self.optimizer_c.apply_gradients(
            zip(gradients, self.critic.trainable_variables))

    @tf.function
    def train_actor(self, states):
        with tf.GradientTape() as tape:
            actions = self.actor(states)
            q = self.critic([states, actions])
            loss = -tf.reduce_mean(q)
        gradients = tape.gradient(loss, self.actor.trainable_variables)
        self.optimizer_a.apply_gradients(
            zip(gradients, self.actor.trainable_variables))

    def soft_update(self, evaluate_net, target_net):
        evaluate_weight = evaluate_net.get_weights()
        target_weight = target_net.get_weights()
        for i in range(len(evaluate_weight)):
            target_weight[i] = self.tau * evaluate_weight[i] + \
                (1 - self.tau) * target_weight[i]
        target_net.set_weights(target_weight)
```

Trenowanie modeli jest analogiczne do DQN, różni się tylko tym, że trenowane są dwa modele.

Aktor uczony jest poprzez oblicznony Loss z średniej Q-Value za swoje decyzje, a krytyk, poprzez Loss obliczony z Q-Value własnych, oraz docelowego krytyka.

```python
import numpy as np
import random
import copy


class OUActionNoise:

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        self.state = copy.copy(self.mu)

    def sample(self):
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.array([random.random() for i in range(len(x))])
        self.state = x + dx
        return self.state
```

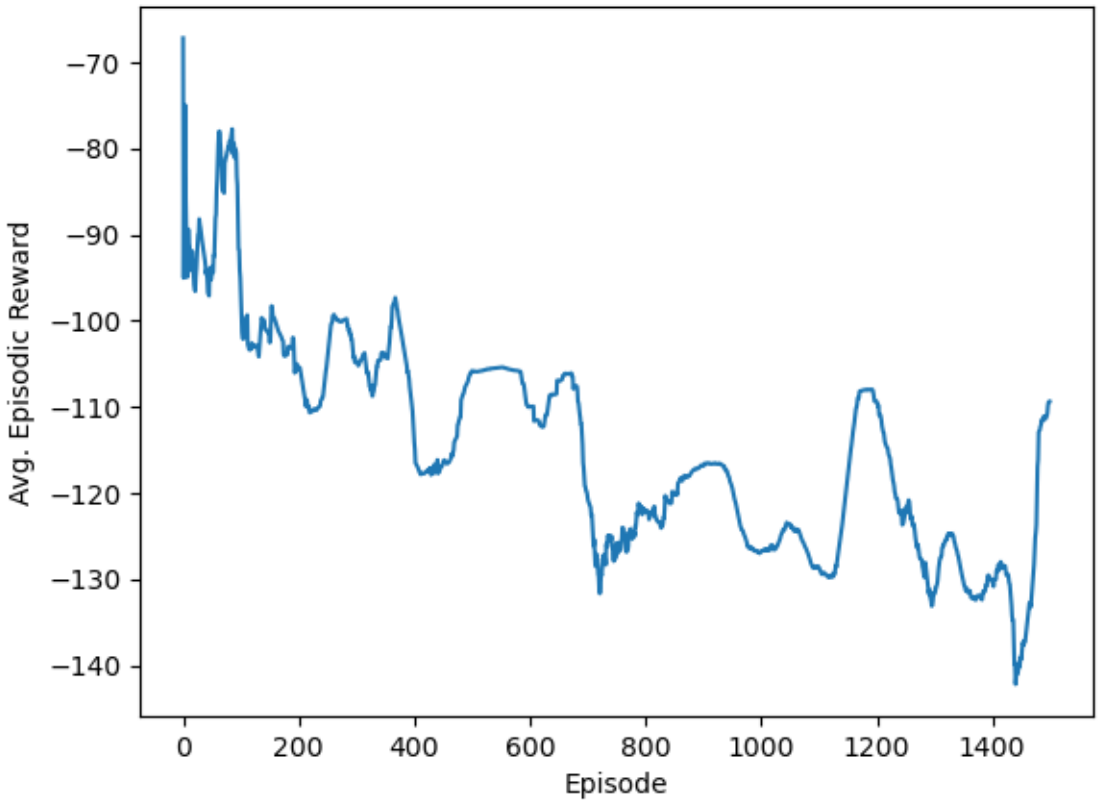
Proces Ornsteina-Uhlenbecka, to proces stochastyczny, generujący próbki tymczasowo skorelowane, w mojej implementacji DDPG używam go do generowania szumów potrzebnych do eksplorowania środowiska przez model.



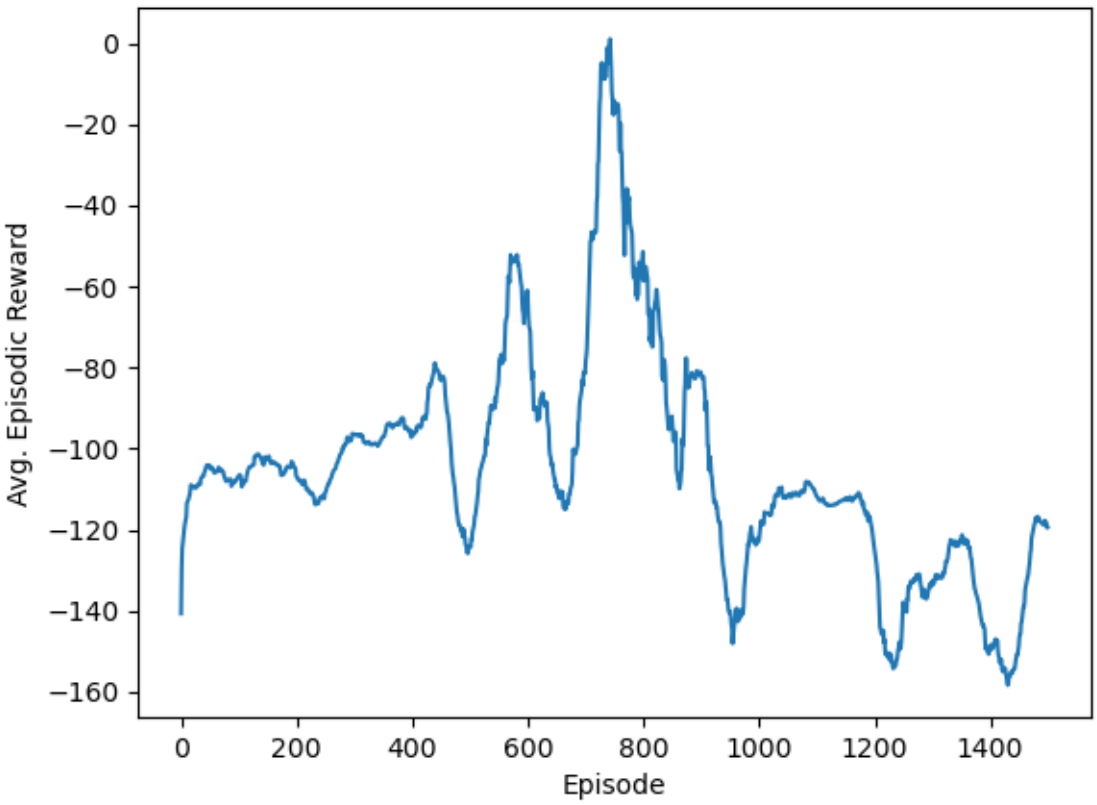
$\theta = 1, \sigma = 1, \mu = 0.$

```python
@tf.function
def noise_action(self, state):
    noise = self.noise_generator.sample()
    action = self.actor(state[None, :]) + noise
    return action[0]
```

Rozmiar bufora

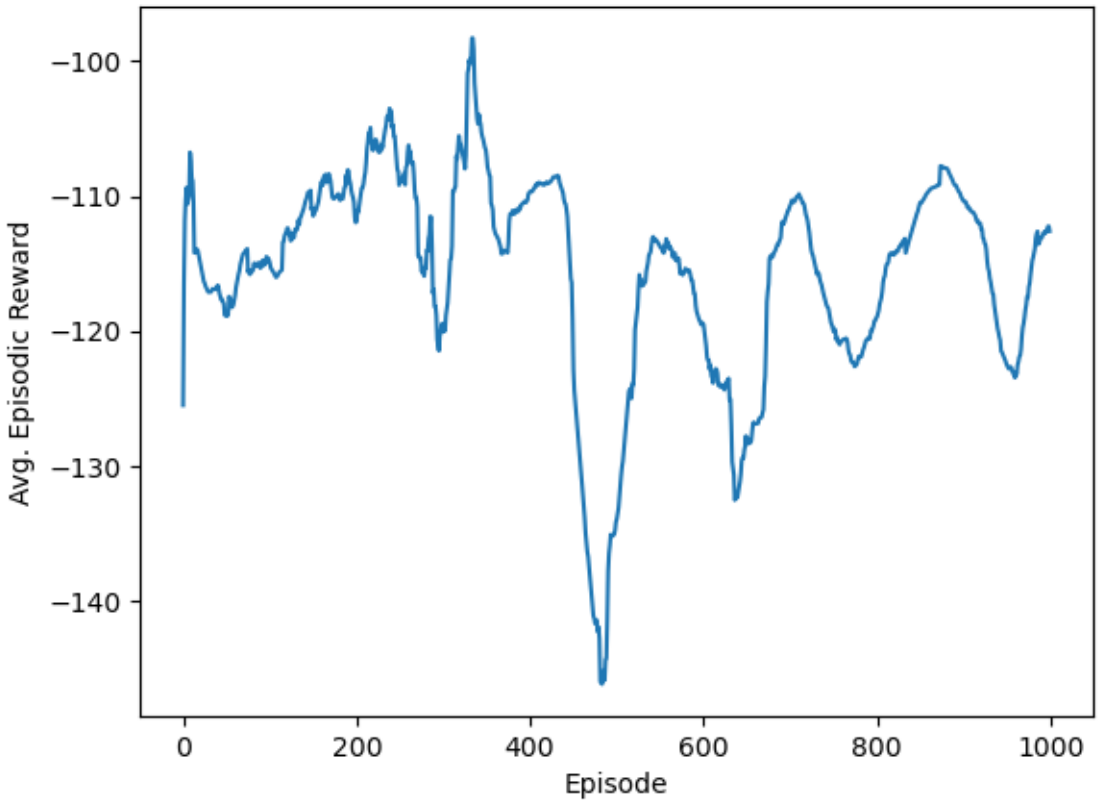


Bufor o rozmiarze 10000 był zdecydowanie za mały na przechowanie wartościowej ilości wspomnień, a model wraz z czasem stawał się coraz gorszy.

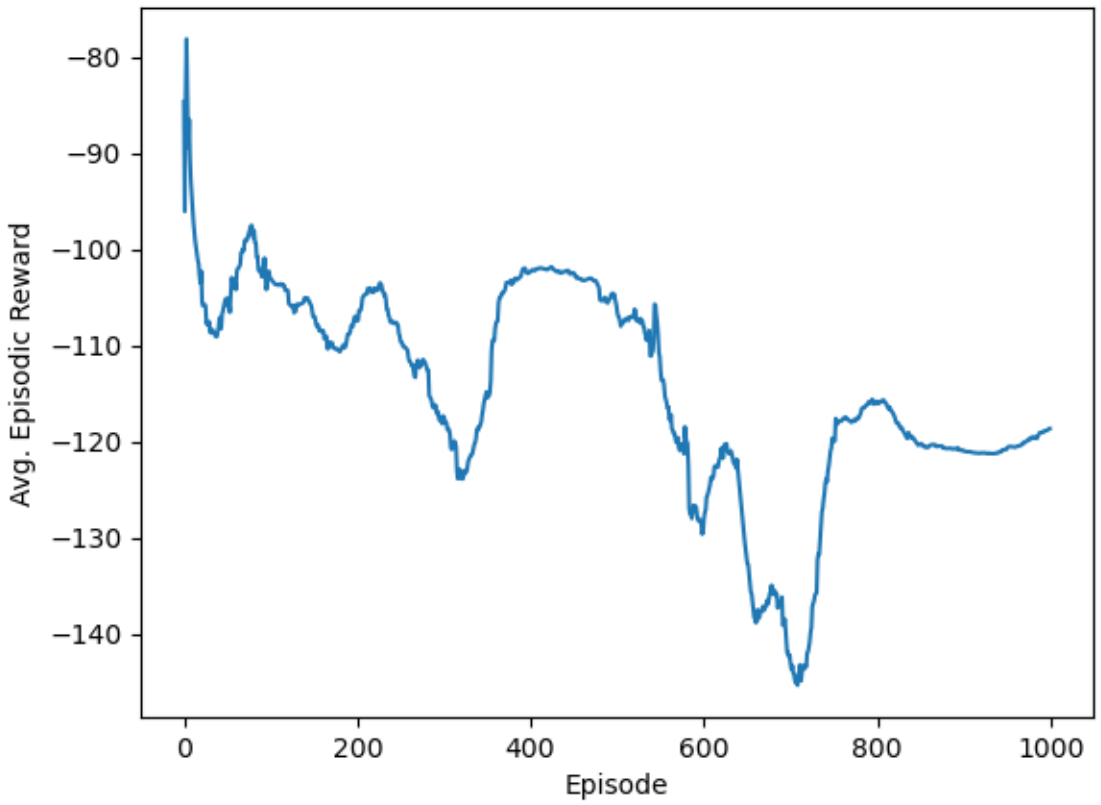


Bufor o rozmiarze 40000 to dalej za mało pamięci dla efektywnej nauki, nawet pomimo wzrostu średniej do wartości pozytywnych. Dopiero zakres od 100000 do 200000 stabilizował wyniki nauki i wprowadzał wystarczającą różnorodność w danych.

HYPERPARAMS



Przy ustawieniu wysokich wartości parametrów takich jak tau, lr i gamma zauważalna była gigantyczna różnica pomiędzy krótszymi odcinkami nauki. Robot zachowywał się gwałtownie i wykonywał bardzo agresywne, wyglądające na losowe ruchy, a wykres nagród odwzorowuje to nagłymi skokami pomiędzy wartościami oraz brakiem postępu w nauce. Bardzo szybko tracił swój postęp, a zejście po gradiencie wyskakiwało ze znalezionych minimum.



Przy odpowiednim skalibrowaniu tych współczynników, sytuacja była zupełnie inna. Samo tau o wartości 0.001 już pozwalało modelowi na spokojniejszy przebieg nauki, a dopasowanie i zmniejszenie learning rate modeli krytyka oraz aktora znacząco poprawiło jego postępy. Największą wadą tego podejścia jest diametralnie wydłużony czas uczenia modelu, z racji na powolniejsze dostosowywanie wag w sieci.

TD3

TD3 to algorytm, którego głównym założeniem jest wyeliminowanie największych problemów DDPG:

1. Niestabilności względem parametrów
2. Dramatycznego zawyżania wartości Q-Value, co skłania politykę do wykorzystywania tych błędów zamiast uczenia się.

Rozwiązania zaproponowane w TD3:

1. Wykorzystanie dwóch funkcji Q-Value i wybranie mniejszej z dwóch wartości
2. Opóźnienie aktualizacji polityki (aktualizacja następuje tylko co pewien odstęp czasowy)

Algorithm 1 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
  Store transition tuple (s, a, r, s') in $\mathcal{B}$

  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  if t mod d **then**
    Update ϕ by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  end if
end for

Moja implementacja TD3 jest analogiczna do DDPG, jedynie zmienia się ilość modeli sieci neuronowych.

```
        self.actor = Actor(states, actions)
        self.actor_target = Actor(states, actions)

        self.critic_1 = Critic(states, actions)
        self.critic_1_target = Critic(states, actions)
        self.critic_2 = Critic(states, actions)
        self.critic_2_target = Critic(states, actions)

        self.actor_target.set_weights(self.actor.get_weights())
        self.critic_1_target.set_weights(self.critic_1.get_weights())
        self.critic_2_target.set_weights(self.critic_2.get_weights())

        self.loss_fn = tf.keras.losses.Huber()
        self.optimizer_a = tf.keras.optimizers.Adam(learning_rate=actor_lr)
        self.optimizer_c1 = tf.keras.optimizers.Adam(learning_rate=critic_lr)
        self.optimizer_c2 = tf.keras.optimizers.Adam(learning_rate=critic_lr)
```

```
    @tf.function
    def noise_action(self, state):
        noise = tf.random.normal(
            shape=(1, self.actions), mean=0.0, stddev=self.std_dev)
        action = self.actor(state[None, :]) + tf.clip_by_value(noise,
                                                               -1.0 * self.c,
                                                               self.c)
        return action[0]
```

Tym razem skorzystałem z gaussowskiego szumu generowanego przez rozkład normalny, który przycinałem do ustalonych wymiarów.

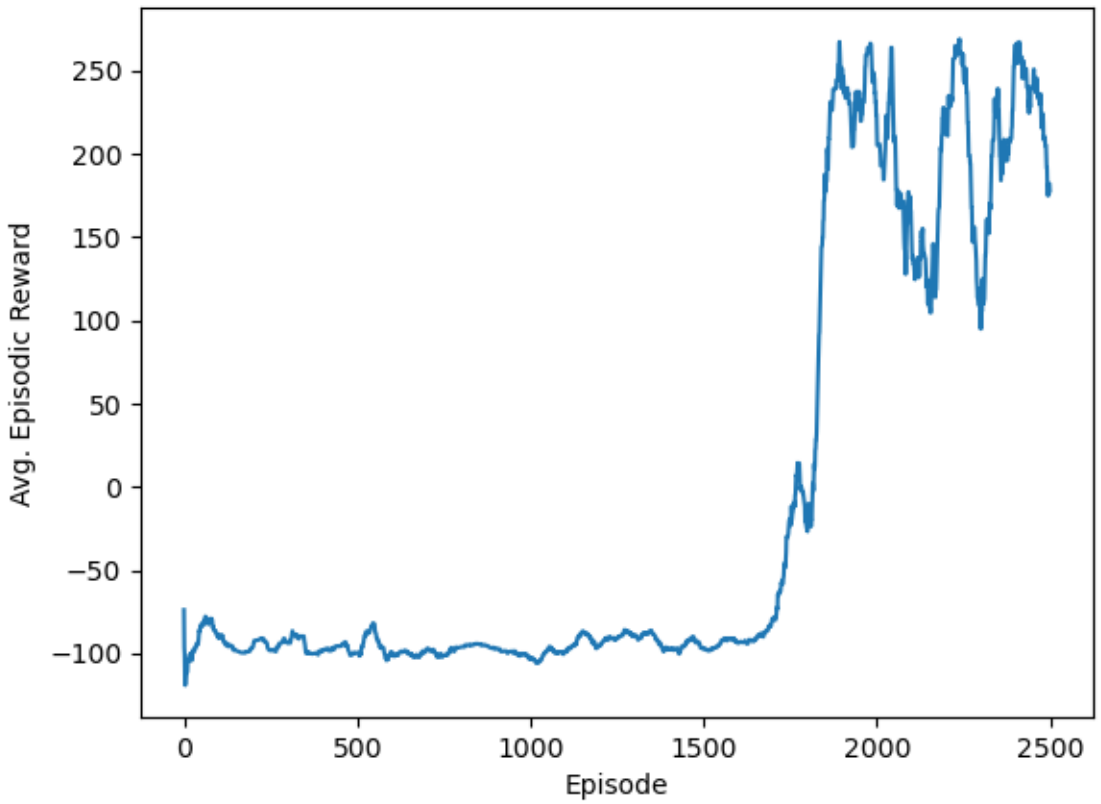
```
    @tf.function
    def train_critic(self, states, actions, rewards, next_states, dones):
        noise = tf.random.normal(
            shape=(1, self.actions), mean=0.0, stddev=self.std_dev)
        next_actions = self.actor(next_states) + \
            tf.clip_by_value(noise, -1.0 * self.c, self.c)

        next_q_1 = self.critic_1_target([next_states, next_actions])
        next_q_2 = self.critic_2_target([next_states, next_actions])
        y = rewards + (1 - dones) * self.gamma \
            * tf.math.minimum(next_q_1, next_q_2)

        with tf.GradientTape() as tape:
            q_1 = self.critic_1([states, actions])
            loss = self.loss_fn(y, q_1)
        gradients = tape.gradient(loss, self.critic_1.trainable_variables)
        self.optimizer_c1.apply_gradients(
            zip(gradients, self.critic_1.trainable_variables))

        # train critic network 2
        with tf.GradientTape() as tape:
            q_2 = self.critic_2([states, actions])
            loss = self.loss_fn(y, q_2)
        gradients = tape.gradient(loss, self.critic_2.trainable_variables)
        self.optimizer_c2.apply_gradients(
            zip(gradients, self.critic_2.trainable_variables))

    @tf.function
    def train_actor(self, states):
        with tf.GradientTape() as tape:
            actions = self.actor(states)
            q = self.critic_1([states, actions])
            loss = -tf.reduce_mean(q)
        gradients = tape.gradient(loss, self.actor.trainable_variables)
        self.optimizer_a.apply_gradients(
            zip(gradients, self.actor.trainable_variables))
```

Zainicjowałem agenta TD3 z następującym zestawem parametrów i buforem o rozmiarze 200.000

Było to najbardziej udane podejście do nauki modelu, pod koniec nauki robot świetnie sobie radził z pokonywaniem nierównej nawierzchni oraz otrzymywał nagrody większe lub równe 300.

Największym ograniczeniem w tym podejściu okazał się sprzęt, ponieważ do dalszej nauki modelu nie wystarczało pamięci ram na przechowywanie wszystkich wartości, przez co przedwcześnie się zakończyła.

Wahania w wykresie nagród otrzymywanych w ostatnich epizodach nauki wynikają głównie z tego, że pomimo bardzo optymalnej polityki, sieci neuronowe dalej próbowały się dostosować do środowiska oraz kładły duży nacisk na to, jak szybko dwunóg biega, co skutkowało potknięciami i przewracaniem się.


Avg. Episodic Reward
250
200
150
100
50
0
−50
−100
0
500
1000
1500
2000
2500
Episode

Wnioski

Reinforcement Learning to bardzo ciekawe oraz ciężkie zagadnienie, a głównym problemem, z którym się zmierzyłem, jest dostosowanie algorytmu, modeli oraz parametrów tak, aby zapewnić odpowiednie środowisko do nauki modelu.

Zbyt mały rozmiar bufora wiąże się ze zbyt małą ilością danych z poprzednich podejść, przez co model nie ma szans dostosować swoich wag do zależności między nimi, z kolei zbyt duży bufor, oprócz wymagań sprzętowych, potrafi spowolnić uczenie modelu poprzez większą ilość wspomnień, które stają się zbędne. Mój optymalny rozmiar to zakres ok. 100000 do 200000 wspomnień.

Wartość tau jest zależna od konkretnego problemu, w moim przypadku wartości 0.001 i 0.002 były o wiele efektywniejsze niż wysoka wartość typu 0.005, agresywniejsze aktualizowanie wag modelu wiąże się z ryzykiem ominięcia docelowych wag modelu.

Współczynnik dyskontu w celach dalekosiężnych zazwyczaj powinien przyjmować wartość prawie maksymalną, w innym wypadku model będzie szukał bliższych do osiągnięcia lokalnych maksimum Q-Value, w swoich badaniach zmiana współczynnika na 0.95 przyniosła odwrotne do zamierzonych skutków, a 0.98 nie miało bardzo dużego wpływu.

Learning rate modeli jest dosyć elastyczny, wiele wartości z niższych i nieco wyższych zakresów jest w stanie osiągnąć zamierzony cel, przy czym należy mieć na uwadze środowisko konkretnego problemu, spowolnienie bądź przyspieszenie nauki, oraz oddzielne wartości dla krytyka i aktora, te ostatnie mogą sprawić, że sieci będą zbyt rozbieżne względem siebie. Złotym środkiem w uczeniu TD3 sterowania dwunogiem okazało się być 0.001 i 0.0001.

Ważne jest dodawanie do wybieranych przez aktora wartości zakłóceń, wspomagają one eksploracje i proces nauki sieci.

Źródła

https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-reinforcement-learning/#:~:text=off-policy%20types.-,Reinforcement%20Learning%20Algorithms,action%20data%20for%20future%20reference.

https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-reinforcement-learning/#:~:text=off%2Dpolicy%20types.-,Reinforcement%20Learning%20Algorithms,action%20data%20for%20future%20reference.

https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc

https://keras.io/examples/rl/ddpg_pendulum/

https://towardsdatascience.com/deep-deterministic-and-twin-delayed-deep-deterministic-policy-gradient-with-tensorflow-2-x-43517b0e0185

https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-learning-with-weight-regularization/

https://towardsdatascience.com/deep-deterministic-policy-gradient-ddpg-theory-and-implementation-747a3010e82f

https://github.com/hmomin/TD3-Bipedal-Walker

https://github.com/FranciscoHu17/BipedalWalker?tab=readme-ov-file

https://github.com/ChienTeLee/td3_bipedal_walker

https://www.reddit.com/r/reinforcementlearning/comments/ewocbu/how_to_solve_bipedalwalker_using_ddpg/

https://www.reddit.com/r/reinforcementlearning/comments/nesywv/choosing_number_of_layers_for_ddpg_algorithm/

https://medium.com/@sthanikamsanthosh1994/reinforcement-learning-part-6-deep-deterministic-policy-gradient-ddpg-using-tesorflow2-fcdccf8f1172

https://www.youtube.com/watch?v=6Yd5WnYls_Y

https://github.com/QasimWani/policy-value-methods

https://github.com/udacity/deep-reinforcement-learning

https://pypi.org/project/nevopy/

https://github.com/standa42/bipedal-walker-deep-ga

https://www.mathworks.com/help/reinforcement-learning/ug/dqn-agents.html

https://keras.io/examples/rl/deep_q_network_breakout/

https://awjuliani.medium.com/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df

https://spinningup.openai.com/en/latest/algorithms/td3.html

https://www.sciencedirect.com/topics/computer-science/deep-q-network#:~:text=DQN%3A%20DQN%20is%20a%20value,near%20the%20end%20%5B45%5D.

https://www.turing.com/kb/how-are-neural-networks-used-in-deep-q-learning

https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

https://spinningup.openai.com/en/latest/algorithms/ddpg.html

https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93

https://towardsdatascience.com/how-does-a-dqn-approximate-bellmans-equation-53e591d5e33

https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b

https://arxiv.org/pdf/1804.08617