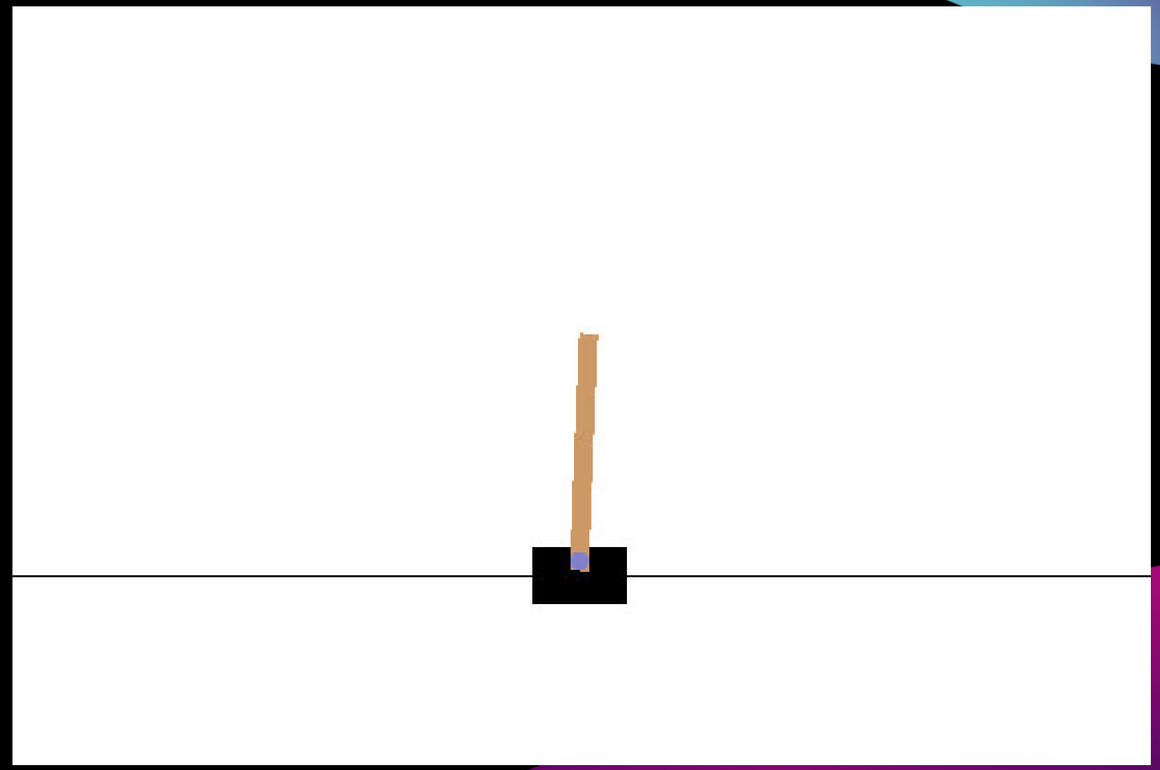
RL CHALLENGE

CartPole - Thomas Loux

CARTPOLE ENVIRONMENT

- Purpose : controle the system to make it stable
- Actions : push left, right or do nothing
- State : position, velocity, angle and angular speed
- Position and angular limits



TIMELINE

Get familiar with gym

Try two solutions

Q Table and Deep Q-Learning

Compare with non RL solutions

Random, do-nothing policies

Try improvement

Aiming at improving stability, convergence, training time

Two main approaches

It's a continuous state problem !

We can either discretise the problem or try to approach the expected reward function using machine learning

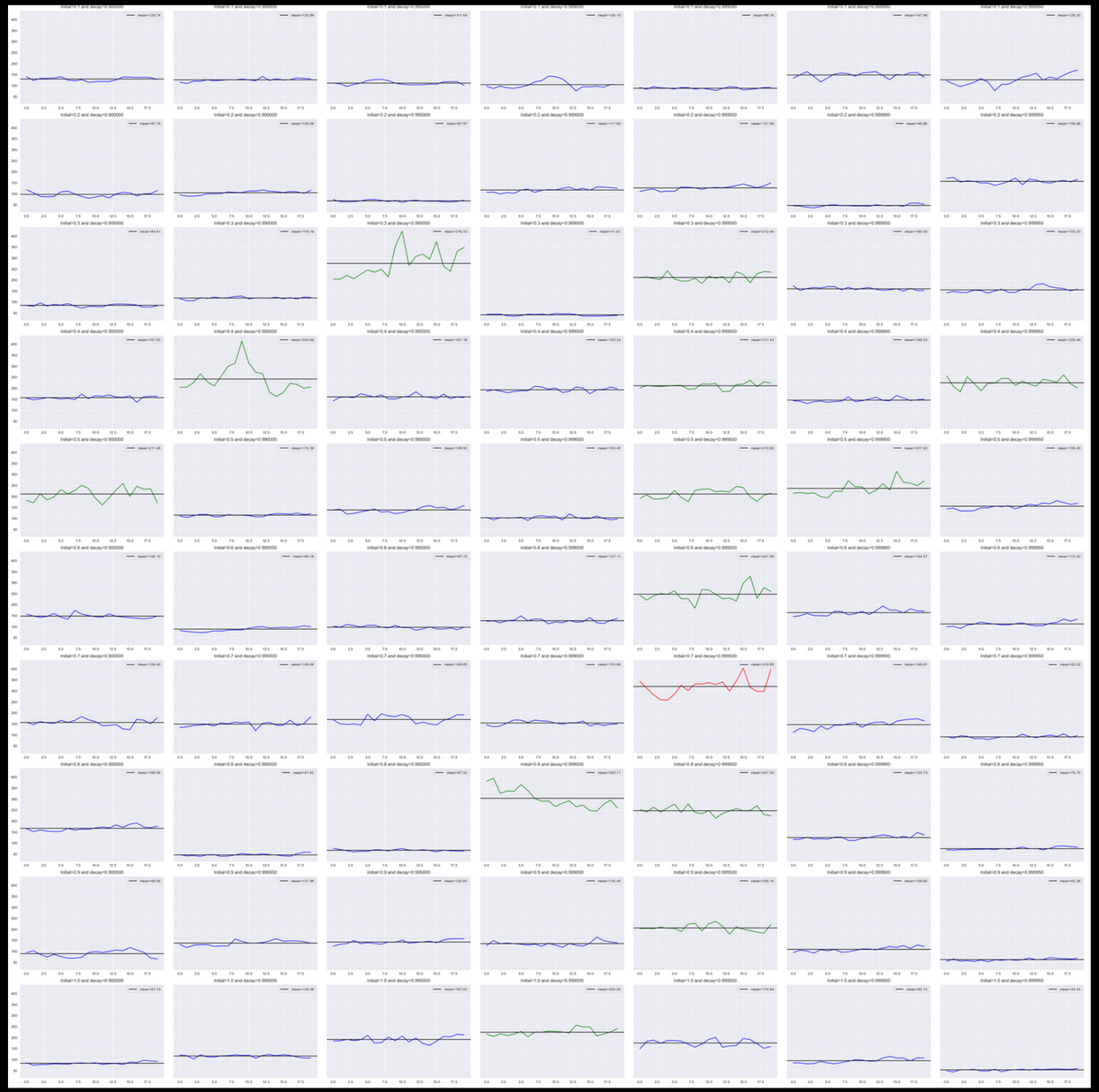
Q-Table - principles

- **Discretization**
- **Update using the Bellman-equation**

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^{\pi}(s').$$

Q-Table - Result

Find the best hyperparameters



```
# As a reminder : evaluate_model outputs a tuple (q_table, scores, steps_history, exploration_list)
# Furthermore, MIN_EXPLORE is set to 0.05
PATH_SAVE = "save_models/"
initials = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
decays = np.array([0.9,0.99,0.995,0.999,0.9995,0.9999,0.99995])
NB_EPISODES = 10_000
results = np.zeros(shape=(len(initials), len(decays))).tolist()
timeBegin = time.time()
for i in tqdm(range(len(initials))):
    for j in range(len(decays)):
        rslt = evaluate_model(3, NB_EPISODES, exploration=initials[i], exploration_decay=decays[j], tqdm_disable=True)
        results[i][j] = rslt[2][-20:]
timeEnd = time.time()
print(f"Processed time is {timeEnd - timeBegin} s")
```

- **Fast training**
- **Easy to implement**
- **Need to the wisely choose hyperparameters (more generally true in RL)**

Deep Q-Learning - principles

- Use of machine learning to estimate Q-function
- Recall learning from past experience
- Bellman equation

The difficult part :

- **Ensure the Q-value convergence to the optimal policy**
- **While using only an approximation of the Q value, which I also want to convergence**

Risk of instability

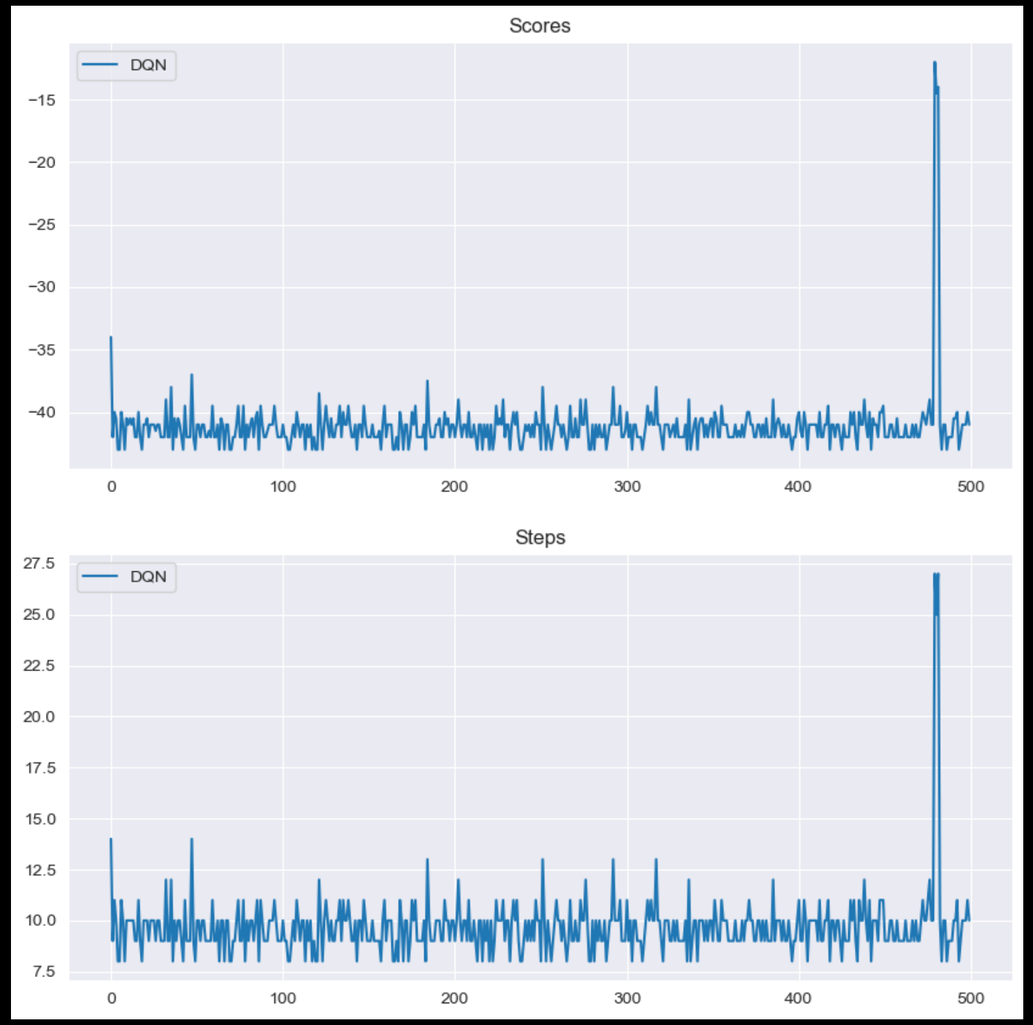
Main idea : experience replay to fit the neural network

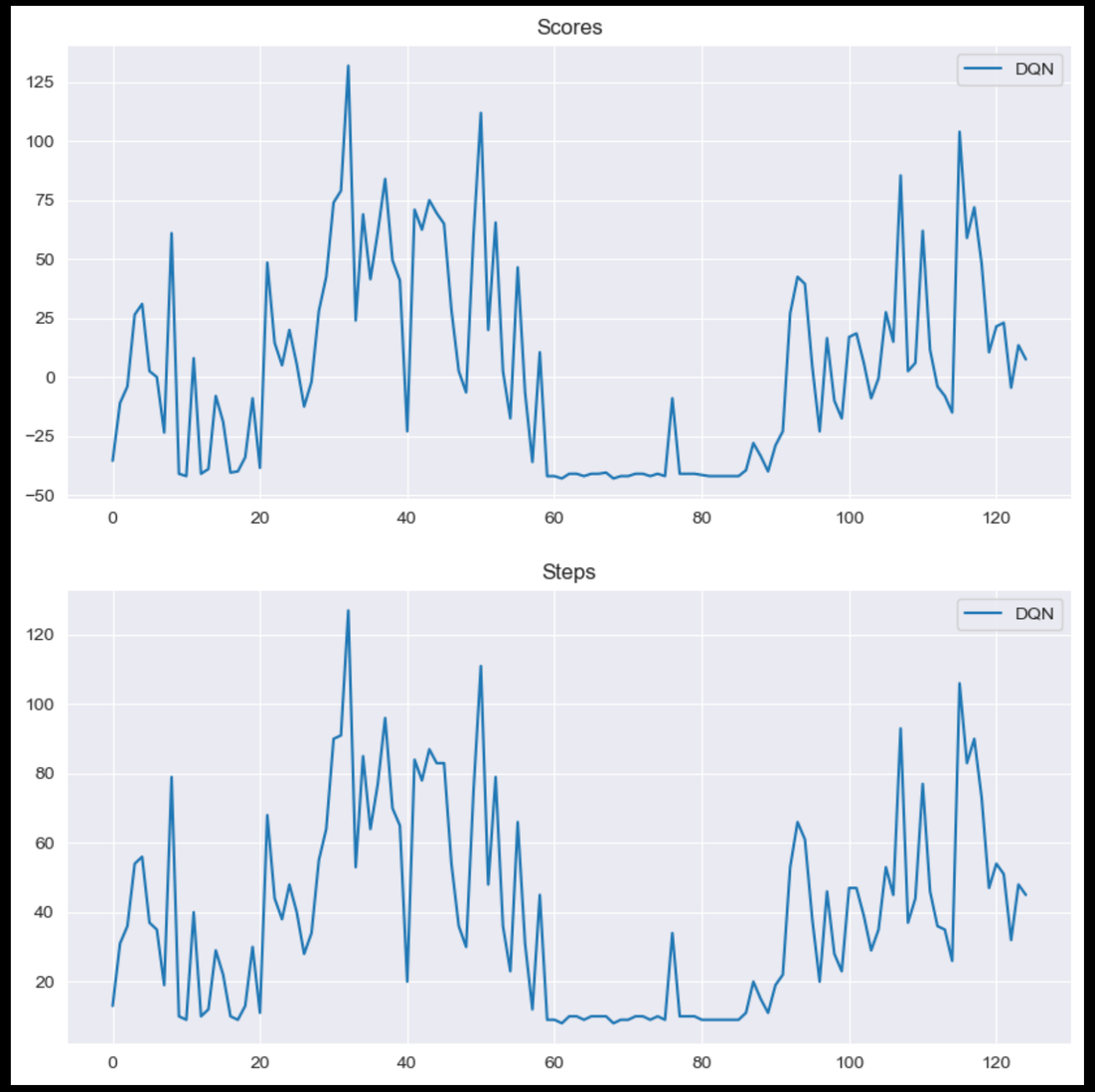
```python
def replay(self):
    #Get a batch of experiences
    batch_size = 20
    if len(self.memory) < batch_size:
        return
    else:
        batch = random.choices(self.memory, k=batch_size) #state, action, reward, next_state, done
        state, action, reward, next_state, done = map(np.array, zip(*batch))
            #Compute the target
                #Applying the Bellman equation
        argmax = np.argmax(self.model_predict.predict(np.concatenate(next_state, axis=0), verbose=0), axis=1)
        target = reward + self.gamma * self.model_learn.predict(np.concatenate(next_state, axis=0), verbose=0)[np.arange(batch_size),argmax] * (1 - done)
        #Compute the target for the action
        target_f = self.model_learn.predict(np.concatenate(state), verbose=0)
        #update for each target on the action coordinate
        target_f[np.arange(len(action)), action] = target
        #Train the model
        state = np.concatenate(state)

        self.model_learn.fit(state, target_f, epochs=1, verbose=0, batch_size=1)

        self.epsilon_exploration *= self.exploration_decay
        self.epsilon_exploration = max(0.05, self.epsilon_exploration)
```

Deep Q-Learning - Result



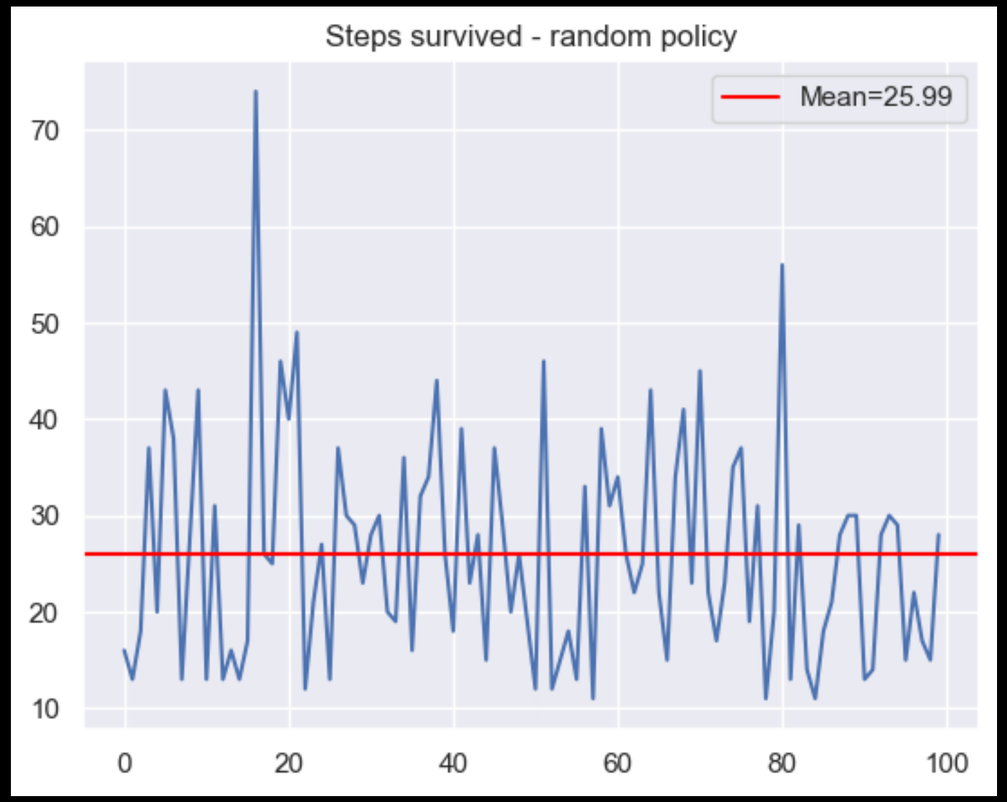


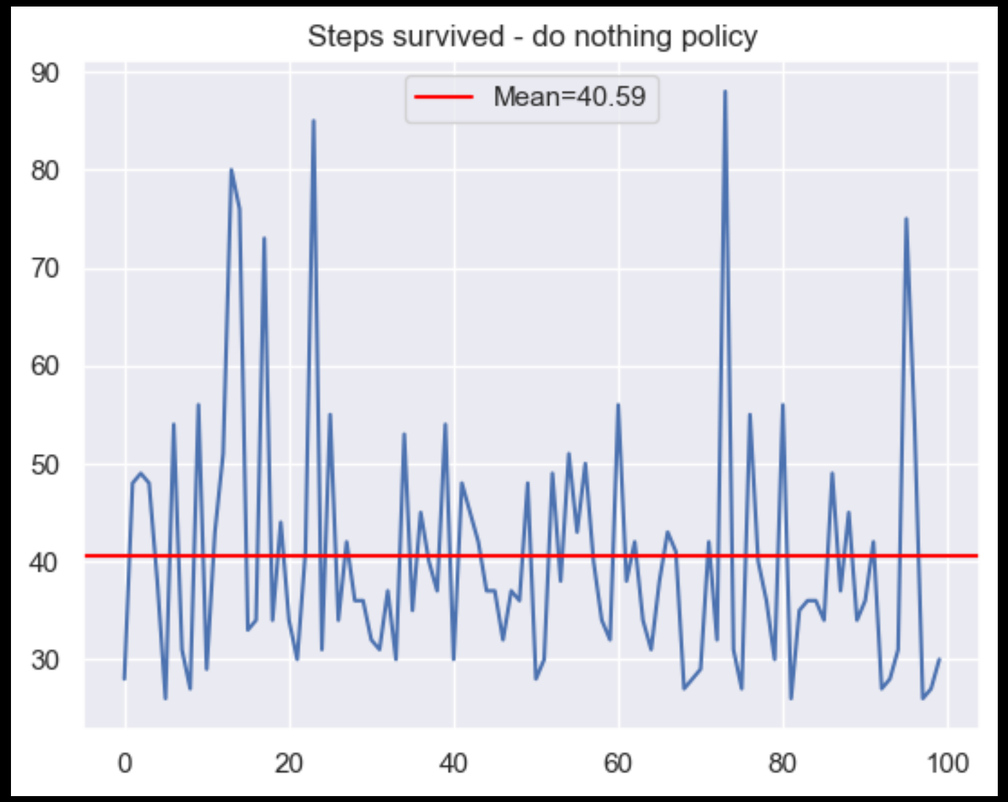
Catastrophic forgetting
Long training time (2 hours)

Why it works

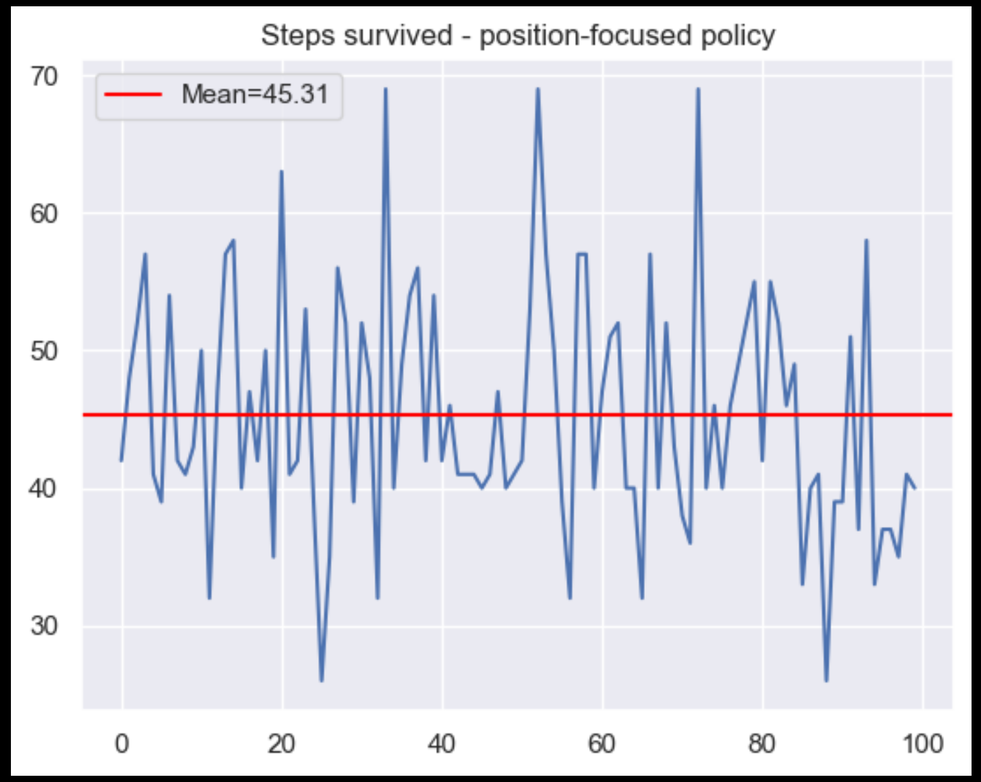
- **The problem can be approximated by a linear function when the angle is small**
- **The angular momentum is a concave function with a small derivative (when angle is small). The discrete approach is then possible.**

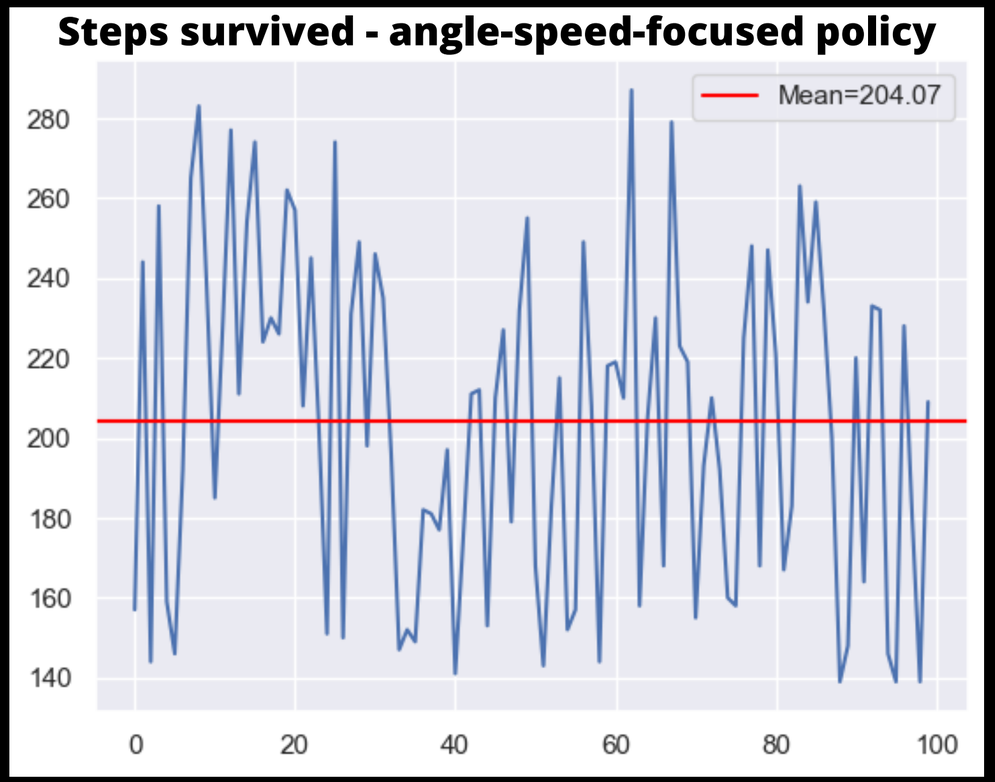
Other non-RL policies



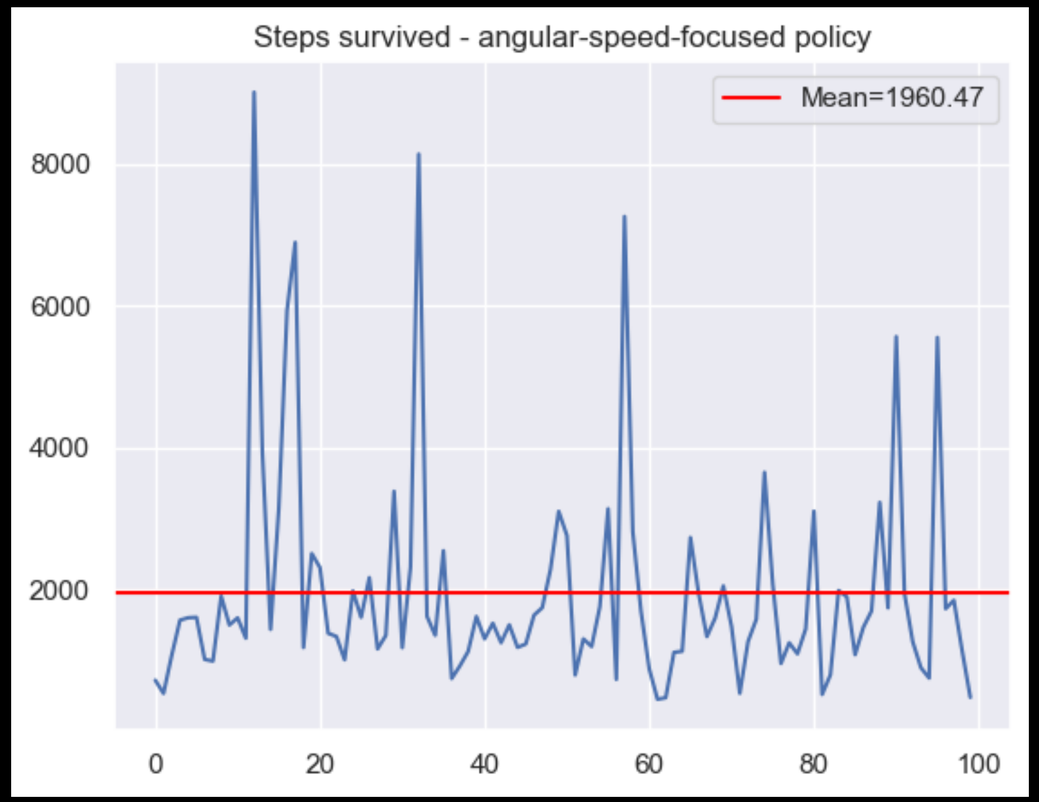


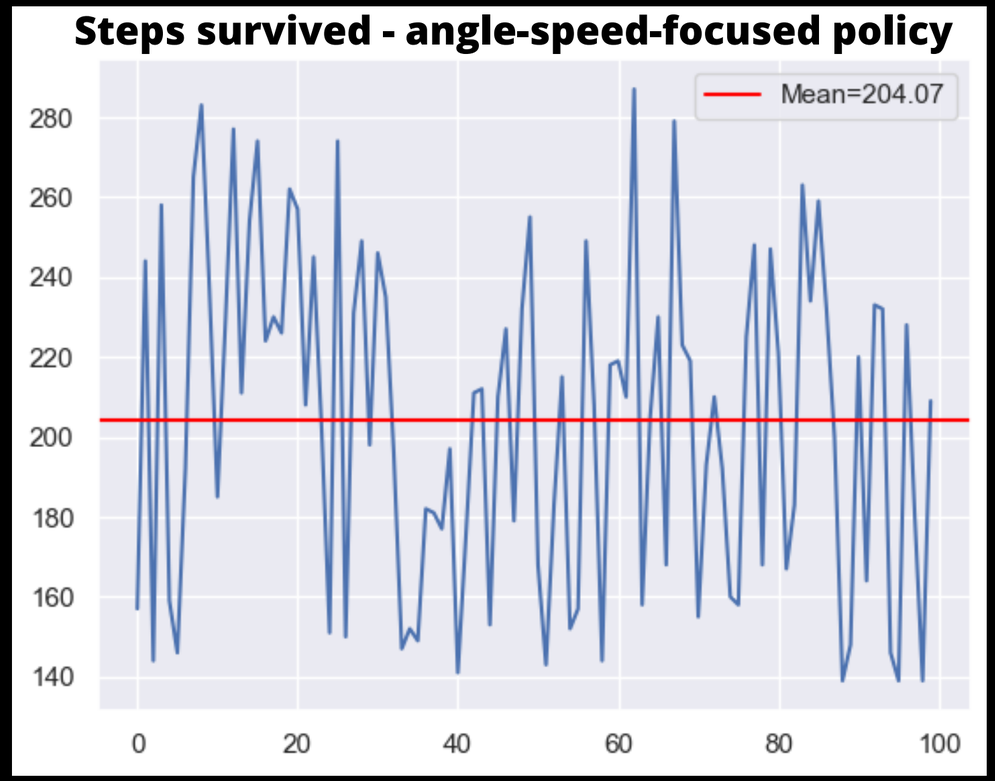
Other non-RL policies





Other non-RL policies





Further improvements

- **Asynchronious episodes**
- **Better models (Actor-critic)**
- **Provide efficient policy to learn in the first place**