PyData London

Hands-on workshop on developing Reinforcement Learning solutions with financial domain domain example use cases.

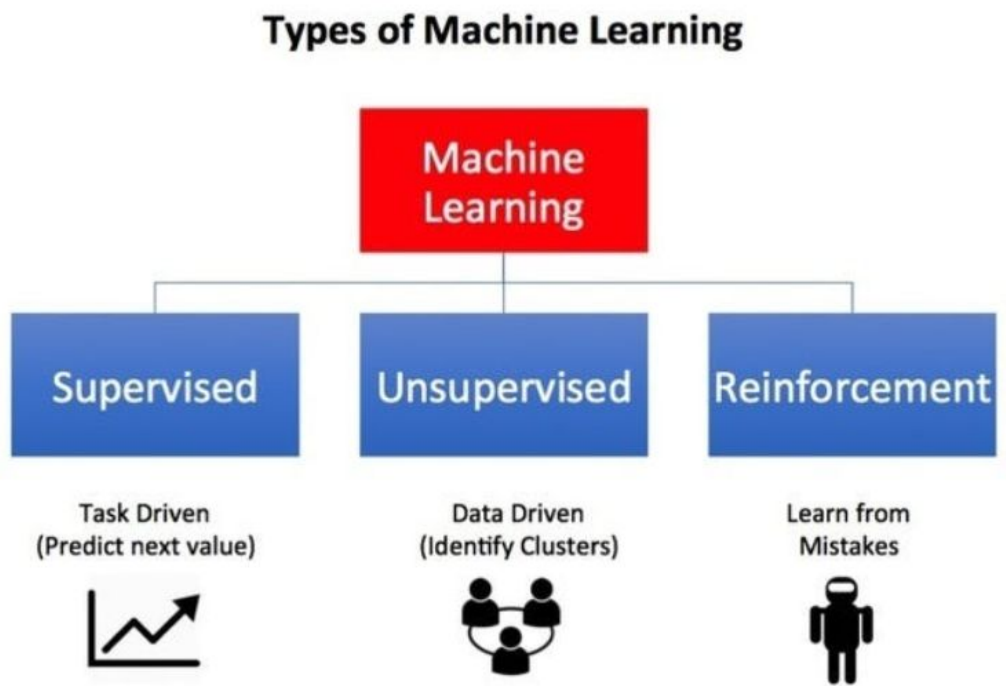
Who am I

- Ade Idowu
- Lead Data Scientist @ UBS
- Experience: Over 15 years working as a software/ML engineer
- Email: adeidowu@hotmail.com
- Github: https://github.com/aidowu1
- LinkedIn: https://www.linkedin.com/in/ade-idowu-ph-d-854b88/

Agenda

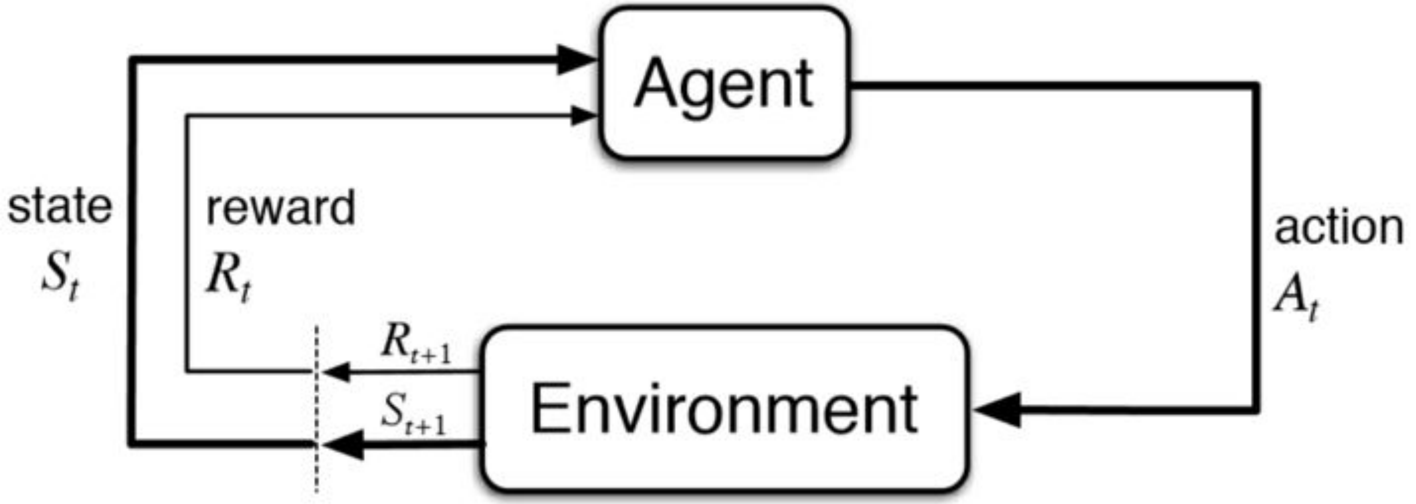
- Structure of the workshop
- A brief intro to Reinforcement Learning
- A close look at model-free RL models
- An overview some popular of RL frameworks
- Applications of RL in Finance
- Future work & conclusions
- Demos
- Q & A

Taxonomy of Machine Learning



Chirag Karia, "Categories of Machine Learning" Kychdev, 2020

What is Reinforcement Learning



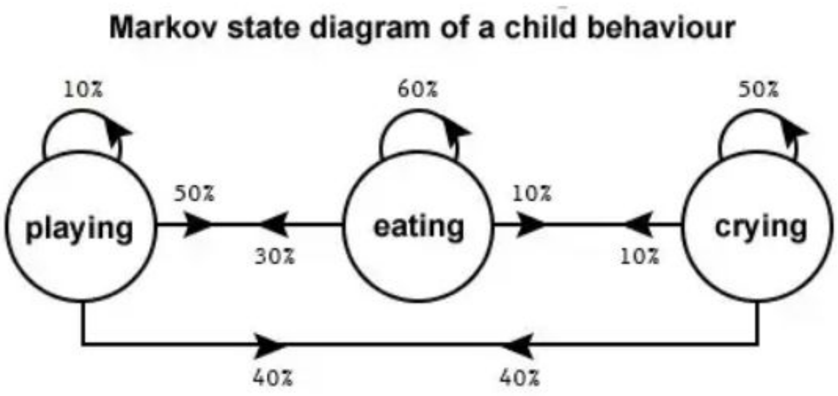
- [1] Sutton, R. S., & Barto, A. G. (2018). [“Reinforcement Learning: An Introduction”]. MIT Press.

Markov Decision Process

- S: a set of states
- A: a set of actions
- P: transition probability
- R: Reward function
- γ: Discount factor for future rewards

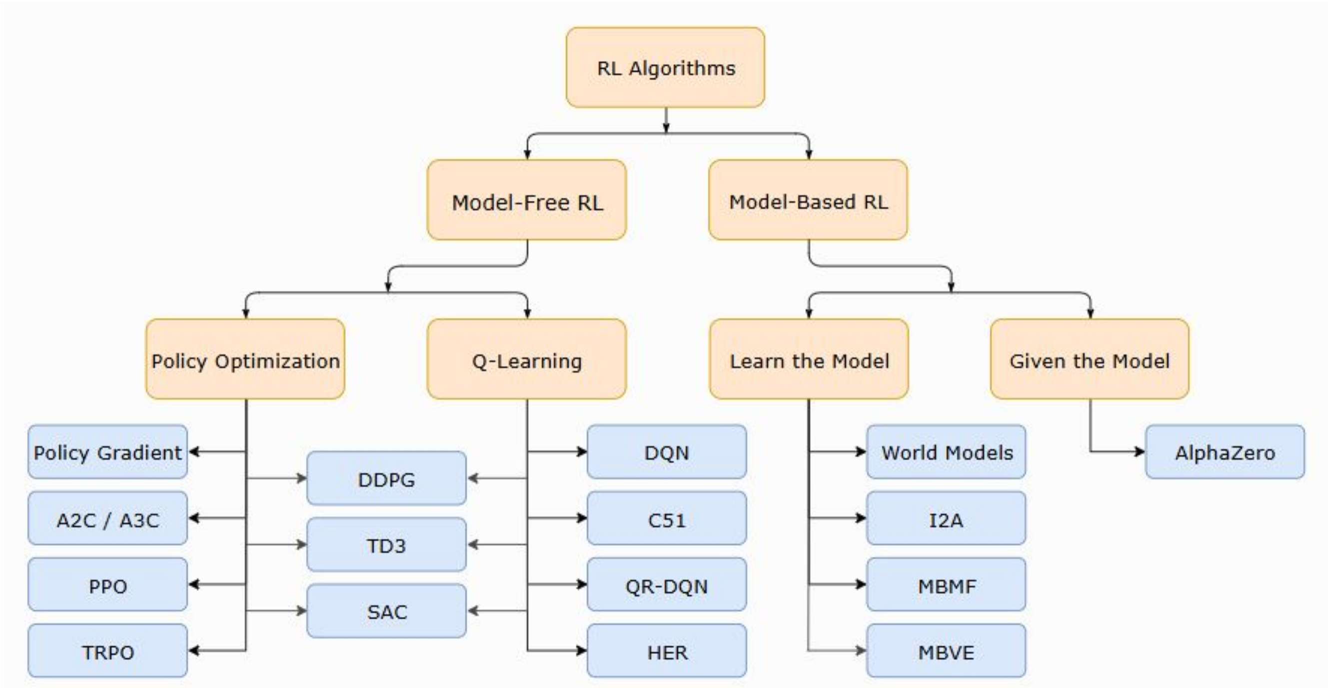
Making sequential decisions in an uncertain environment

“Next state and reward only depend on the current state and action taken”



Suraj Bansal, ["Markov Decision Processes — Learning Some Math"](), Medium 2020

Taxonomy of Reinforcement Learning



Courtesy of [OpenAI Spinning Up]

Model-based vs Model-free RL

- **Model-based**:
  - Model tries to understand the environment dynamics
  - Model typically will have well defined transition probabilities
  - If problem domain action/state space is small it can be solved using Dynamic Programming

Model-based vs Model-free RL

- Model-free:
  - Maximizes the expected reward without a model or prior knowledge
  - Normally used when we have incomplete info about environment or model
  - The agent’s policy provides insight on the optimal action to take in a certain state to maximize the rewards
  - Each state is associated with a value function V(s) or action-value function Q(s, a)
  - V(s) and Q(s,a) quantifies how good a state is
  - Model-free can either be value-based or policy-based

Value-based RL - Q-learning

- Q - learning:
  - Q-learning is the adaptation of Temporal Difference (TD) learning
  - This algorithm computes which action to take based on V(s) or Q (s,a)
  - Q-learning is an **off-policy approach** meaning is does not need to select actions based on the policy implied by the value function alone
  - To encourage a balance between exploration and exploitation, an epsilon-greedy strategy is used to select a random action with probability **ε**, else action is selected based on max Q(s,a)

Value-based RL - Q-learning

- Q - learning algorithm steps:

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q
        Take action a, observe r, s'
        Update
            $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$;
    Until s is terminal

1. Sutton, R. S., & Barto, A. G. (2018). “Reinforcement Learning: An Introduction”. MIT Press.

Value-based RL - SARSA

- Q - learning:
    - SARSA is also based on Temporal Difference (TD) learning
    - It updates Q(s,a) by the sequence of S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}
    - SARSA is an **on-policy approach** meaning it finds the optimal policy and uses it to invoke an action
    - Unlike Q-learning, SARSA policies used for updating and for acting are the same

Value-based RL - SARSA

- SARSA algorithm steps are:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

[1] Sutton, R. S., & Barto, A. G. (2018). “Reinforcement Learning: An Introduction”. MIT Press.

Deep RL DRL

- Deep Reinforcement Learning is a sub-field of ML that combines deep learning with RL
- As the dimensionality of action and state space for problems increase it becomes very ineffective and inefficient to use traditional Q-learning and SARA approaches
- DRL uses deep neural networks to represent policies, value-functions or environment to solve RL problems at scale
- Some popular DRL algorithms are **Deep Q Network** (DQN), **Deep Deterministic Policy Gradient** (DDPG), **Proximal Policy Gradient** (PPO), **Soft Actor-Critic (SAC)** etc..

Value-based RL - DQN

- DQN is the DRL extension of Q-learning (Mnih et al, 2013)
- It uses neural network to approximate the calculation of Q-values by learning a set of weights θ of the deep neural network which maps states to actions
- It uses **experience replay**, which involves storing the history of state, action, reward and next state transitions in a large replay data structure.
- Experience replay improves data efficiency and remove correlations in observation sequences
- DQN also uses a target network, which is used for periodic updating of the network weights via the minimization of the loss using gradient descent

Value-based RL - DQN

- DQN algorithm steps:

```
Initialize network Q
Initialize target network Q̂
Initialize experience replay memory D
Initialize the Agent to interact with the Environment
while not converged do
    /* Sample phase
    ϵ ← setting new epsilon with ϵ-decay
    Choose an action a from state s using policy ϵ-greedy(Q)
    Agent takes action a, observe reward r, and next state s'
    Store transition (s, a, r, s', done) in the experience replay memory D

    if enough experiences in D then
        /* Learn phase
        Sample a random minibatch of N transitions from D
        for every transition (s_i, a_i, r_i, s'_i, done_i) in minibatch do
            if done_i then
                y_i = r_i
            else
                y_i = r_i + γ max_{a'∈A} Q̂(s'_i, a')
            end
        end
        Calculate the loss L = 1/N Σ_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2
        Update Q using the SGD algorithm by minimizing the loss L
        Every C steps, copy weights from Q to Q̂
    end
end
```

Jordi TORRES.AI, "Deep Q-Network (DQN)-II", Towards Data Science, 2020

Policy-based RL

- Policy-based RL typical involves learning the policy function, π which maps each state to the best corresponding action
- Policy based can occasionally be simpler than value-based methods
- Most policy based techniques include DRL algorithms such as DDPG, Twin Delayed DDPG (TD3), PPO etc.

DDPG - policy/value-based RL

- DDPG is a robust algorithm for solving RL problems in continuous action spaces (Lillicrap et al., 2015)
- It combines the strengths of policy gradient methods and Q-learning, enabling effective policy optimization for complex control tasks in high-dimensional environments.
- It is a hybrid value and policy based algorithm
- It uses an actor-critic architecture, where the actor learns a deterministic policy, and the critic evaluates the policy using a Q-value function

DDPG - policy/value-based RL

- DQN algorithm steps are:

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$

repeat

  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$

  Execute a in the environment

  Observe next state s', reward r, and done signal d to indicate whether s' is terminal

  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

  If s' is terminal, reset environment state.

  if it's time to update **then**

    for however many updates **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for

  end if

until convergence

- Achiam, J. (2018). Spinning Up in Deep Reinforcement Learning. OpenAI

PPO - Policy-based RL

- PPO algorithm is a policy-based technique introduced by Schulman et al. (2017)
- PPO is designed to enhance the training stability and computational efficiency
- PPO can be used for environments with either discrete or continuous action spaces.
- In some open source RL libraries like Stable-baselines, Machin etc, PPO can be parallelized

PPO - Policy-based RL

- PPO algorithm steps:

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, ...$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

Achiam, J. (2018). [Spinning Up in Reinforcement Learning - PPO](). OpenAI

SAC - policy/value-based RL

- SAC is an off-policy algorithm developed by Haarnoja et al, (2018)
- It can theoretically be used for both discrete and continuous action space problems.
- In the Demo I used it to solve the Gymnasium Pendulum continuous action space problem
- SAC addresses these issues by maximizing both the expected reward and the entropy of the policy
- The entropy term incentivises stochasticity in action selection, ensuring that the policy remains exploratory throughout training

SAC - policy/value-based RL

- SAC algorithm steps:

Achiam, J. (2018). [Spinning Up in Deep Reinforcement learning - SAC](#) OpenAI

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Review of some RL libraries

OpenAI Baselines

Dopamine

rllib

Chainer RL

DeepMind TRFL

Review of some RL libraries

- Neptune.AI criteria for selecting these libraries include:
    - Number of SOTA algorithms implemented
    - Documentation and availability of tutorials
    - Code readability
    - Number of supported environments
    - Logging and tracking
    - Vectorization and parallelization
    - Regular updates
- 10 top libraries in descending order are: KerasRL, PyQlearning, Tensorforce, RL_Coach, TFAgent, Stable Baselines, MushroomRL, RLlib, Dopamine
- I regularly use Stable Baselines and Machin (not rated in the top 10, but I like it!!!)

A quick review of Gymnasium

- Gymnasium formerly known as Gym (under OpenAI) is a pythonic framework for simulating RL environments
- It allows developers to experiment with precreated RL environments or create customized ones
- For financial algo robo RL agent I used Gymnasium to develop the custom **TradingEnv** environment
- Basic steps for creating a custom environment include:
  - Create your new environment by deriving from the base Env class
  - Implement a number of key RL methods:
    - step()
    - reset()
    - render()
    - close

Gymnasium Documentation

Applications of RL in Finance

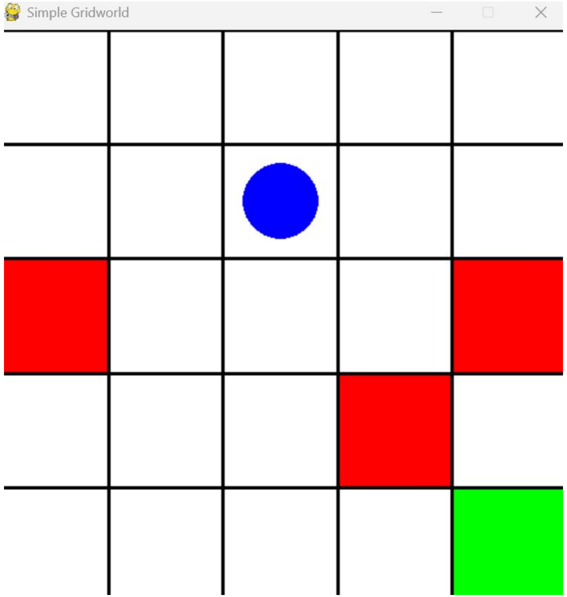
- RL and DRL is increasingly being used in financial domain uses case such as:
  - **Algorithmic trading of financial instruments**:
    - I will provide a simple demo of this use case but others include the works of Yves Hilpisch (2020)
  - **Hedging risk such as hedging the trading of derivatives/options**:
    - I recently did a project **RLDynamicHedger** on this, you can find it in this repo also see Cao et al, 2020
  - **Asset allocation and portfolio optimization**:
    - Such as the works of Sato (2019), Charpentier, Elie, and Remlinger (2020) and Mosavi et al. (2020)
  - **Order execution optimization**
    - See for instance the work of Joseph Jerome, et al (2022)

Demos

- There will be 5 demos:
    - The first 2 is an introduction to RL and uses Q-Learning, SARSA and DQN (these were coded natively in python with no RL packages)
    - The third one is an introduction to continuous action space problems and uses 3rd party open source libraries Machin and Stable Baselines
    - The fourth demo introduces us to hyper-parameter tuning using Stable Baseline library
    - The final demo is the main one which demonstrates a simple algo RL agent for trading the S & P index using price/bar data from 2010 to 2019

Grid World

- This is a classic computer science problem where an agent is trying to find the shortest path to its destination

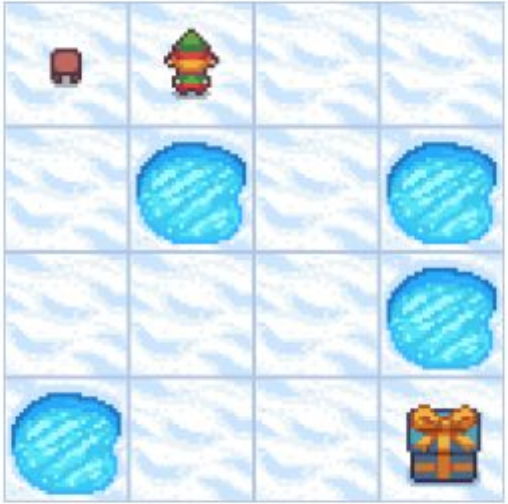


- Action space: Discrete(4)
- Observation space: Discrete(25)

- Rewards
  - Red squares are pits: 0 (terminate)
  - White squares: 0 (continue)
  - Green square: + 1 (target)
-

Frozen Lake

- Similar to grid world, but this time the agent navigating a frozen lake environment with pot holes



Action Space	`Discrete(4)`
Observation Space	`Discrete(16)`
import	`gymnasium.make("FrozenLake-v1")`

Rewards

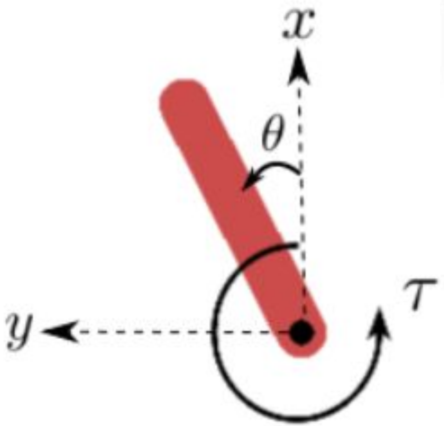
Reward schedule:

- Reach goal: +1
- Reach hole: 0
- Reach frozen: 0

Pendulum

- Inverted pendulum swingup problem is based on the classic problem in control theory.
- The pendulum starts in a random position and the goal is to apply torque on the free end to swing it into an upright position, with its center of gravity right above the fixed point.

Action Space	`Box(-2.0, 2.0, (1,), float32)`
Observation Space	`Box([-1. -1. -8.], [1. 1. 8.], (3,), float32)`
import	`gymnasium.make("Pendulum-v1")`

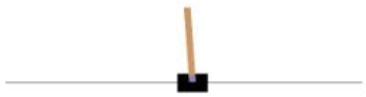


Num	Action	Min	Max
0	Torque	-2.0	2.0

- `x-y`: cartesian coordinates of the pendulum's end in meters.
- `theta` : angle in radians.
- `tau`: torque in `N m`. Defined as positive *counter-clockwise*.

Cart-pole

- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track.
- The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.



Action Space	`Discrete(2)`
Observation Space	`Box([-4.8 -inf -0.41887903 -inf], [4.8 inf 0.41887903 inf], (4,), float32)`
import	`gymnasium.make("CartPole-v1")`

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

- 0: Push cart to the left
- 1: Push cart to the right

RL robo algo trader

- The RL trading agent can invoke 3 trading actions Hold, Buy or Sell
- Profit is only made when it sells a unit of asset in its inventory (from a previous buy)
- The bought and sold amount is always a unit of one
- **There are no transaction costs or robust backtesting considered here or the consideration of risk management management!!**
- References:

  - Yves J Hilpisch, "Artificial Intelligence in Finance", page 268 - 276, O'Reilly, 2021
  - Hariom Tatsat, Sahil Puri & Brad Lookabaugh, "Machine Learning and Data Science Blueprints for Finance", page 298 - 316, O'Reilly, 2021
  - Mnih, V. et al., "Human-level control through deep reinforcement learning", Nature, 2015.
  - Moody, J., Saffell, M., "Learning to trade via direct reinforcement", IEEE, 2001.

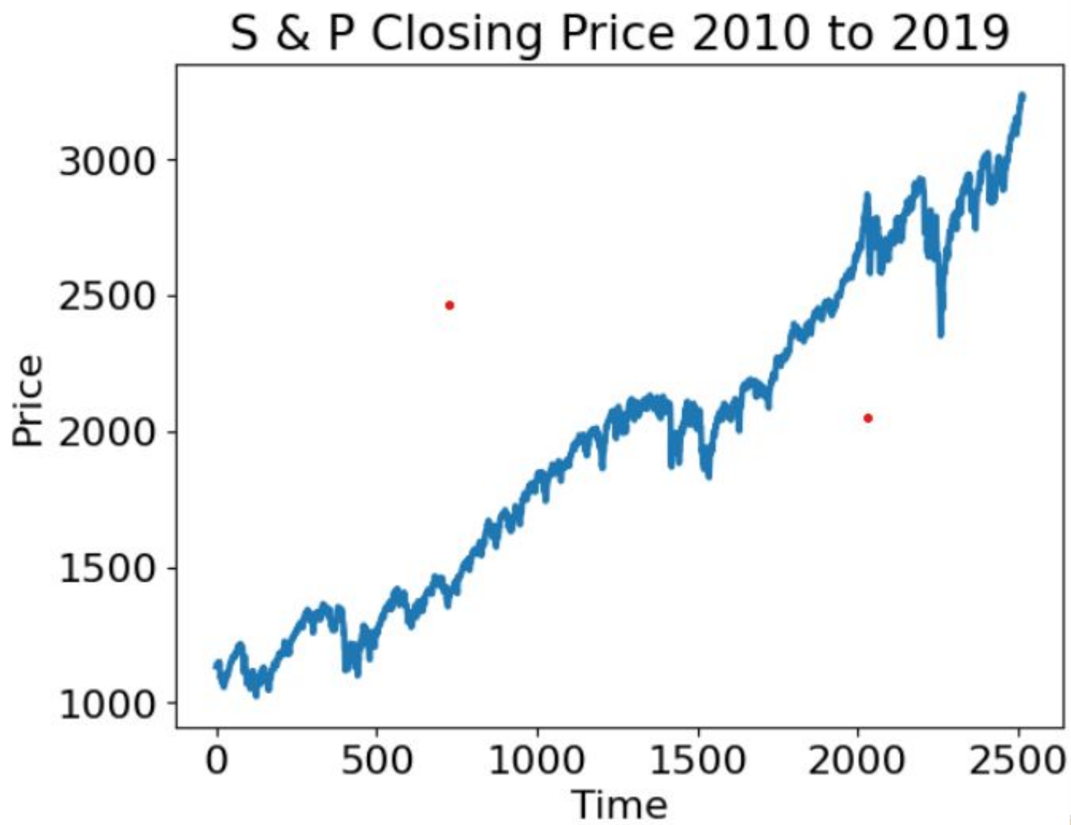
RL robo algo trader

- Action space is: Discrete(3)
- Observation space is: Box() with shape: (lags, n_features)
- Reward is the Profit and Loss

5 rows ⌄ 5 rows × 5 cols

	Close	High	Low	Open	Volume
0	1132.989990	1133.869995	1116.560059	1116.560059	3991400000
1	1136.520020	1136.630005	1129.660034	1132.660034	2491020000
2	1137.140015	1139.189941	1133.949951	1135.709961	4972660000
3	1141.689941	1142.459961	1131.319946	1136.270020	5270680000
4	1144.979980	1145.390015	1136.219971	1140.520020	4389590000

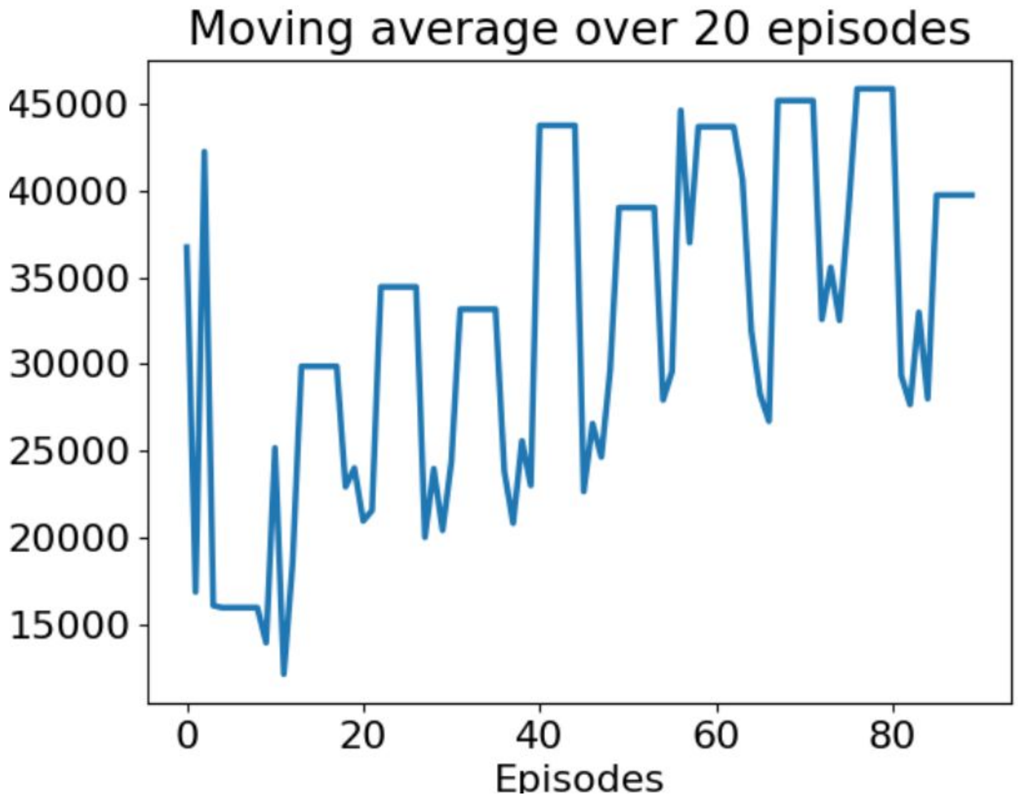
RL robo algo trader - the dataset



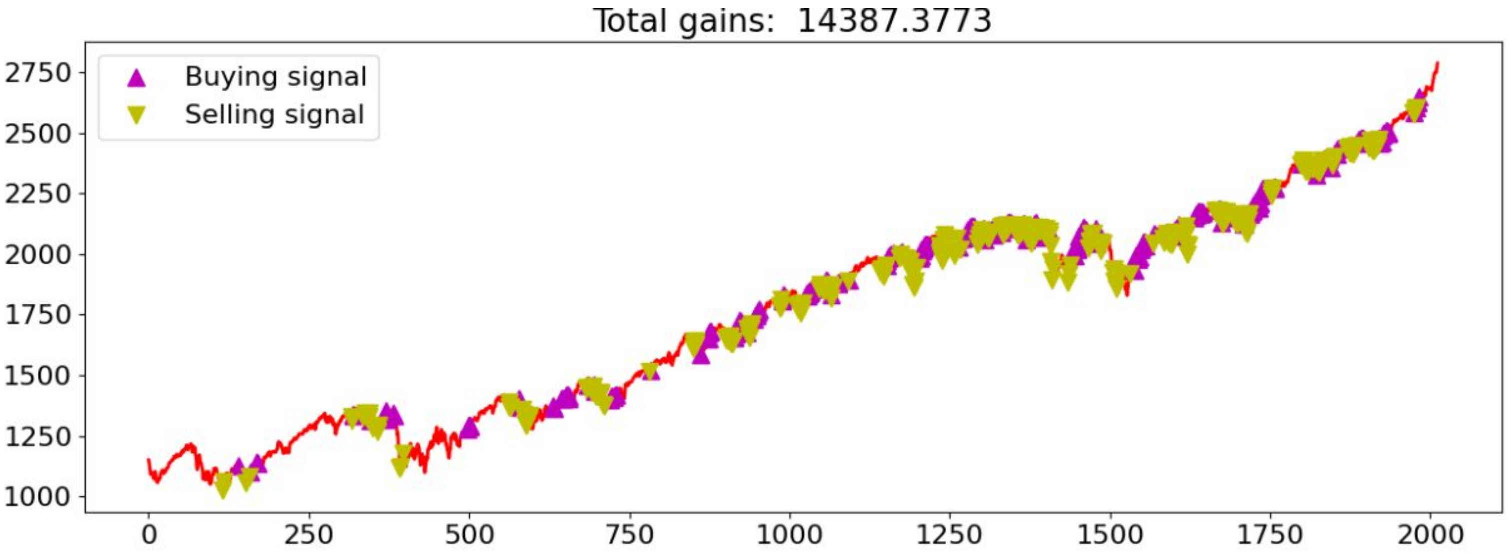
RL robo algo trader - Data exploration

- The RL data will be partitioned into:
  - 80% training (in-sample) data i.e. 2010-01-04 to 2017-12-26
  - 20% test (out-of-sample) data i.e. 2017-12-27 to 2019-12-31
- Feature set will be based on the Close price
- Data is scaled using normalization
- Close price was transformed into 4 features:
  - Log normal return
  - 10 window moving average of price
  - 10 window moving average of log-normal return
  - 10-window moving standard deviation of log-normal return
- These feature represent the state-space inputs of the RL agent

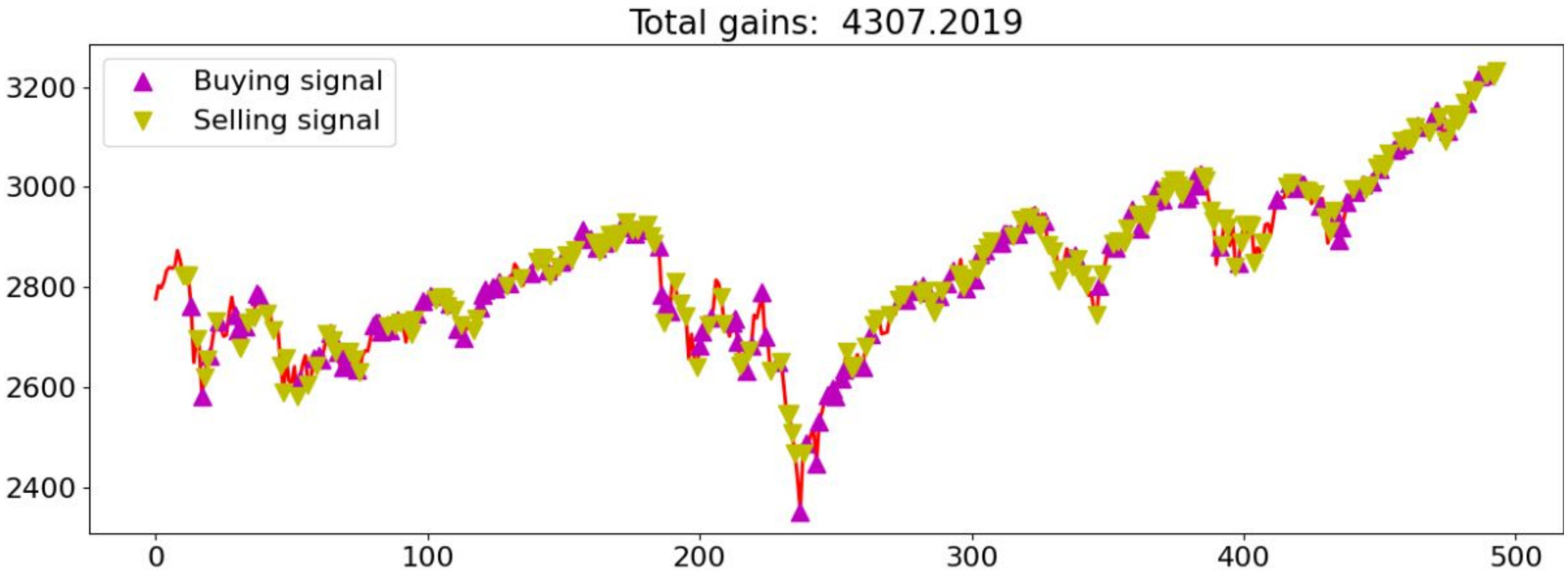
RL robo algo trader - PPO agent Performance



RL robo algo trader - RL training



RL robo algo trader - Testing



Conclusions

- RL is a robust learning paradigm in finance especially in circumstances where the environment dynamics (market, geo-politics etc) can not be easily modelled
- RL allows the trial-and-error exploration/exploitation of the problem domain enabling the model to episodically learn environment dynamics
- RL is very compute intensive
- A lot of thought is required when specifying the reward function, especially in the financial domain
- DRL requires a lot of hyper-parameter tuning in a very skillful way!!

Future Work

- Extend the current RL algo trade to include additional features such as new sentiment, additional technical analysis based trading signals
- To consider the impact of transaction costs on the observed profit and loss
- Provide risk management functionality
- Provide a comprehensive backtesting module

Q & A

Appendix