Un mdp è un framework per definire l’ambiente (stati, azioni, funzioni,ecc…) con il quale il nostro agente (IA o programmazione dinamica(algoritmo VI)) andra ad interagire per apprendere e risolvere il task.

E’ quadrupla S,A,P,R,gamma dove S è l’insieme dei possibili stati, ed ogni stato ha delle variabili che lo caratterizzano, e queste variabili possono assumere un valore all’interno di un certo dominio (anche semplicemente booleane) e altre quindi un vettore di tutti gli attacchi classificabili dall’intrucion detection system (ho gli attacchi nello stato tanti quanti ne conosce l’IDS e per ognuno dia ha una probabilità che indica se sono sotto attacco da quella tipologia di attacco).

A sono le azioni che possono essere intraprese all’istante t per andare su un nuovo stato t+1 con una certa probabilità P, con cui un’’azione venga scelta ed è data da una matrice StxAtxSt+1 e questa stessa matrice mi da o la ricompensa o il costo (asseconda del segno) che mi rappresenta la bonta di quell’azione in quello stato che mi portato sul nuovo (se l’azione ha senso reward alta, se non centra nulla bassa, quindi costo per segno negativo). Nel caso dei sistemi stazionari probabilità e reward non cambiano anche se cambia l’istante temporale, perche se mi trovo in St e vado in St+1 con sempre le stesse azioni, avro sempre stessa bonta. Cambia nel caso di sistemi non stazionari ovvero St puo variare al variare di t (non solo aggiungo nuova azione, rimuovo un’attacco, aggiungo variabile,ecc…)

L’obiettivo è di massimizzare la ricompensa, ed il comportamento dell’agente è descritto da una policy pigreco: SxA ovvero la probabilità con cui l’agente scelga l’azione At in St.

Infine in S ovvero gli stati trovo grazie a T, una funzione target, un sottoinsieme di S che sono gli stati di non attacco, stati target sicuri dove il mio agente si arresta, così riduco la dimensione. Infine ho anche una funzione PC ovvero pre condizioni che mi determina un sottoinsieme delle azioni applicabile in un determinato stato (se ho un certo attacco alcune contromisure sono inutili quindi non faccio determinate azioni se non mi aiutano a patchare; sto subendo sql injection ma vado a petchare cross site scripting, inutile).

DQN: deep Q network, uno dei primi se non il primo algoritmo per la risoluzione, ovvero l'approssimazione del comportamento di Q\*, ovvero la policy dell'agente, con reti deep. E' basato su convolutional neural network,multilayer perceptron il tutto addestrato con l'algoritmo Q learning con una variante ovvero lo stocastic gradient discent.

APEX-DQN: un'evoluzione del DQN in quanto, risolve alcuni problemi di efficacia e instabilità dell'algoritmo dqn. Basato sull'architettura apex per ridurre i tempi di convergenza nella fase di training ed il delta del valore di approssimazione della policy.

Fa uso di un'architettura distribuita per scalare DQN, distribuendo la generazione e la selezione dei dati piuttosto che usare un approccio standard di parallelizzazione della risoluzione del gradient (le tecniche possono essere combinate).

La raccolta delle esperienze può essere estesa a centinaia di lavoratori della CPU grazie ad un valore di priorità dell'esperienza che viene assegnata prima dell'archiviazione nei buffer di riproduzione.

C'è un buffer di riproduzione che viene utilizzato per le esperienze ovvero coppie di stato-azione-reward (interazioni passate dell'agente con l'ambiente) che vengono utilizzati per il training.

ciò permette di eliminare la correlazione temporale dell'azione, aumenta il riuso dell'esperienza con priorità maggiore dato che ha un carico informativo maggiore, campionandoli più volte ed evitare di rigenerare sempre nuovi dati dall'interazione dell'agente con l'ambiente.

IMPALA: basato sullo stocasthic gradient discent progettato per ambienti paralleli e distribuiti che fa uso di V-TrACE ovvero un algoritmo di apprendimento.

ovvero ci sono più attori che generano esperienze (azione-observation-reward) e che le mandano ad un learner che apprende, quindi c'è un disaccoppiamento tra l'apprendimento mediante i dati e la generazione dei dati, con la possibilità di una generazione parallela con più attori.

C'è anche un lag tra la policy usata per la generazione dei dati e quella per l'apprendimento del learner(la stima del gradiente), e qui entra in gioco l'algoritmo v-trace per ridurre questo lag.

PG: Policy gradient, basato su SGD per massimizzare la cumuative reward.

PPO: Proximal Policy Optimization Basato sul PG con l'aggiunta di tecniche di clipping per evitare aggiornamenti di policy troppo grandi.

Proximal Policy Optimization Basato sul PG con l'aggiunta di tecniche di clipping per evitare aggiornamenti di policy troppo grandi (+ stabilita, + convergenza)

COSE FATTE FIN’ORA:

creato un ambiente custom con pettingzoo con 2 agenti, un attaccante ed un difensore.

Ho definito un ambiente relativo al modello iniziale dove:

7 attacchi (pscan,pvsftpd,psmbd,pphpcgi,pircd,pdistccd,prmi) hanno una probabilità con tui il difensore lo valuta

# 0 < T1 < T2 < 1 e p < T1 rumore, T1 < p < T2 possibile attacco (prevenzione), p > T2 attacco by IDS (contromisure),

# p=1 attacco noto e strategia da attuare

# Lo stato del difensore è composto da: 7 p (una per ogni attacco) + 14 variabili di sistema,

# presumibilmente l'attaccante ora vede tutto per un taining più efficace

# il difensore ha 18 azioni, 21 componenti nello stato

# l'attaccante ne ha 7, lo stato esamina quello del difensore

# considererei lo stato clean quello di partenza e lo stato target una configurazione delle anomalie innocua (DA DEFINIRE BENE)

# per l'attaccante qual'è lo stato target di vittoria??

# STATO

# [ firewall([True/False])(0), blockedip([])(1), flowlimit\_ips([])(2), alert([True/False])(3), honeypot\_ips([])(4),

# log\_verb([0-5])(5),

# active([True/False])(6), quarantined([True/False])(7), rebooted([True/False])(8), backup([True/False])(9),

# updated([True/False])(10),

# manuallySolved([True/False])(11), everQuarantined([True/False])(12), everShutDown([True/False])(13),

# +

# pscan([0-1])(14), pvsftpd([0-1])(15), psmbd([0-1])(16), pphpcgi([0-1])(17), pircd([0-1])(18), pdistccd([0-1])(19), prmi([0-1])(20),]

# DIFENSORE: 18 azioni= [GenerateAlert(0), FirewallActivation(1), BlockSourceIp(2), UnblockSourceIp(3),

# FlowRateLimit(4), UnlimitFlowRate(5), RedirectToHoneypot(6), UnHoneypot(7), IncreaseLog(8),

# DecreaseLog(9), QuarantineHost(10), UnQuarantineHost(11), ManualResolution(12), SystemReboot(13),

# SystemShutdown(14), SystemStart(15), BackupHost(16), SoftwareUpdate(17), noOp(18)]

# ATTACCANTE: attacchi=[Pscan(0), Pvsftpd(1), Psmbd(2), Pphpcgi(3), Pircd(4), Pdistccd(5), Prmi(6), noOp(7)]

spiegare tutte le pre e post condizioni del 1 modello !

Value iteration (VI) and policy iteration (PI) are both dynamic programming algorithms used to solve Markov decision processes (MDPs). MDPs are a type of reinforcement learning problem in which an agent interacts with an environment in order to maximize its expected reward.

Value iteration is an iterative algorithm that updates the value function of each state until it converges to the optimal value function. The value function is a mapping from states to expected rewards, and it represents the long-run average reward that an agent can expect to receive if it starts in a given state and follows an optimal policy.

Policy iteration is also an iterative algorithm, but it updates the policy and value function simultaneously. The policy is a mapping from states to actions, and it represents the action that an agent should take in each state. The value function is updated using the policy, and the policy is updated using the value function.

The main difference between VI and PI is that VI only updates the value function, while PI updates both the value function and the policy. This means that PI is more efficient than VI, but it can also be more difficult to implement.

Here is a table summarizing the key differences between VI and PI:

|  |  |  |
| --- | --- | --- |
| Feature | Value iteration | Policy iteration |
| Update rule | Only updates the value function | Updates both the value function and the policy |
| Convergence | Converges to the optimal value function | Converges to the optimal policy |
| Efficiency | Less efficient | More efficient |
| Implementation difficulty | Easier to implement | More difficult to implement |

In general, VI is a good choice for problems that are small or have a simple structure. PI is a good choice for problems that are large or have a complex structure.

Proximal policy optimization (PPO) and policy gradient (PG) algorithms are both policy-based reinforcement learning algorithms. This means that they optimize the policy directly, rather than the value function. This can be more efficient than value-based algorithms, which need to learn the value function first.

PPO and PG algorithms are both iterative algorithms, but they update the policy differently. PPO uses a technique called policy clipping to stabilize the update process. Policy clipping limits the change in the policy between updates, which can help to prevent the algorithm from overfitting to the training data. PG algorithms do not use policy clipping, and they can sometimes overfit to the training data.

Here is a table summarizing the key differences between PPO, PG, VI, and PI:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Type | Update rule | Convergence | Efficiency | Implementation difficulty |
| Value iteration (VI) | Value-based | Updates the value function | Converges to the optimal value function | Less efficient | Easier to implement |
| Policy iteration (PI) | Policy-based | Updates the value function and the policy | Converges to the optimal policy | More efficient | More difficult to implement |
| Proximal policy optimization (PPO) | Policy-based | Updates the policy with policy clipping | Converges to an approximately optimal policy | More efficient | More difficult to implement |
| Policy gradient (PG) | Policy-based | Updates the policy directly | Converges to an approximately optimal policy | More efficient | More difficult to implement |

The value function and the current policy are two key concepts in reinforcement learning (RL). They are both used to represent the agent's knowledge about the environment and its actions. However, they differ in their purpose and representation.

Value function

The value function is a mapping from states to values, where the value of a state represents the expected cumulative reward that the agent can achieve from that state. It is a measure of how good a state is for the agent.

The value function is calculated using dynamic programming. The agent starts with an initial value function, and then it repeatedly updates the value function until it converges to the optimal value function. The optimal value function is the value function that maximizes the expected cumulative reward.

Current policy

The current policy is a mapping from states to actions, where the action for a state represents the action that the agent will take in that state. It is a plan of action for the agent.

The current policy is updated using the value function. The agent uses the value function to identify the actions that are most likely to lead to good outcomes. It then updates the current policy to favor those actions.

Nel contesto di Reinforcement Learning, spesso si fa riferimento a "iterazioni" o "passi di addestramento" che indicano quanti passaggi di addestramento (aggiornamenti di rete, esperienze raccolte, ecc.) vengono eseguiti. Questo potrebbe non coincidere direttamente con il concetto tradizionale di "epoche" in cui un'epoca potrebbe rappresentare un passaggio completo attraverso l'intero set di dati di addestramento.

Solitamente, un'epoca nel contesto del machine learning classico significa attraversare l'intero dataset di addestramento una volta. Tuttavia, in Reinforcement Learning, il concetto di epoche potrebbe non essere così direttamente applicabile, poiché il modello apprende dall'interazione con l'ambiente e non da un set statico di dati.

Training iteration

Raccolta dei dati: L'agente interagisce con l'ambiente, raccoglie esperienze (coppie di stato, azione, ricompensa, nuovo stato) e le conserva nel memory buffer.

1. Campionamento dei dati: Vengono estratte casualmente delle esperienze dal buffer di memoria per addestrare il modello.
2. Aggiornamento del modello: Il modello (la rete neurale, nel caso di DQN) viene addestrato su questi dati estratti, regolando i pesi della rete per migliorare le sue previsioni.
3. Valutazione del modello: Il modello addestrato viene testato o valutato per vedere quanto bene si comporta nell'ambiente, valutando le azioni che prende in base agli stati ricevuti.

Questi passaggi possono essere ripetuti in cicli successivi. L'impostazione di una condizione di stop dopo un numero specifico di iterazioni di addestramento in RLlib determina quanti di questi cicli completare prima di interrompere il processo di addestramento.

Implementazione del tempo:

voglio implementare la gestione del tempo in questa maniera:

ho una interfaccia azione che viene implementata da due classi una azioni sincrone ed un’altra azioni asincrone.

Poi le azioni dell’attaccante saranno azioniSincrone ed azioniAsincrone dove le azioni sincrone dell’attaccante e del difensore andranno ad interagire con una variabile Tempo che avrà un delta iniziale di 120 ovvero un dominio intero di [+60;-60].

Quando il difensore deve eseguire una mossa andrà a verificare che il tempo sia <=0, se ciò è vero lui può eseguire una mossa e fare tempo + X, dove X sarà positivo.

L’attaccante quando andrà ad eseguire una mossa verificherà che il tempo sia >=0, se ciò è vero lui potrà eseguire la sua mossa e fare tempo +Y con Y negativo.

Ciò cosa rappresenta nella gestione del tempo, ovvero se il difensore trova un tempo negativo vuol dire che il tempo sta andando avanti e l’attaccante sta facendo, e lui può agire.

Viceversa se l’attaccante trova un tempo positivo vuol dire che il tempo va avanti ed il difensore sta facendo un’azione.

Ovvero per esempio: se l’attaccante trova il tempo negativo vuol dire che lui stesso sta facendo un’azione che ancora non termina e che se non termina non può fare altro (sia ipotizzando un fattore di risorse o più semplicemente perché posso fare un exploit solo dopo aver concluso un portscan e scoperto i servizi attaccabili).

Invece le azioni asincrone, sono azioni che non vincolano le altre ma permettono la ‘parallelizzazione’ temporale ma non istantanea. Ovvero il difensore può sempre fare un’azione per turno, in un turno non può scegliere più azioni asincrone da fare insieme, ma al suo turno successivo può eseguire un’altra azione senza attendere che termini l’altra precedentemente attivata.

L’intenzione è quella di dare vita ad un nuovo agente che attende la terminazione del tempo per quell’azione, mentre l’agente principale continua ad operare nelle operazioni successive.

Solo allo scadere del tempo l’agente asincrono modifica lo stato e si distrugge.

Capire se è proprio necessario definire un vero agente nell’ambiente o basta lanciare un sottoprocesso e tutto funziona (capire bene quelle asincrone come implementarle)

Ho diverse mosse asincrone attive, come faccio a non riselezionarle? L’attacante potrebbe avere una cosa di azione in corso.

Siccome anche l’attaccante le avrà mettiamola in Agente